TAO Developer's Guide

OCI TAO Version 2.2a
updated for patch 7
OCI Part Number 621-01

Summary Of Changes In Patch 7

Section 13.4, added description of new IOR "refresh" feature of the IORTable.

Section 17.13.54, explain new behavior for explicit `ORBPreferIPV6Interfaces 0`.

Sections 18.2.12 and 18.6.15 added description of new resource factory option for initialization of the IORTable refresh feature.

Section 27.6.2, Reordered and reformatted options list, added new options SSLCheckHost, SSLPassword, and SSLVersionList.

Sections 27.10.1 through 27.10.15, Freshened the option descriptions and added missing options.

Section D.1.5 updated the information about generating build files for supported Visual C++ versions.

Summary Of Changes In Patch 5

Section 3.3.2.1, added parameters to example tao_idl command line.

Section 3.3.6, modified example mpc file.

Summary Of Changes In Patch 4

Section 17.13.32, added ORBIIOPClientPortBase option description.

Section 17.13.33, added ORBIIOPClientPortSpan option description.

Section 22.9.1, added -l command line parameter description to table.

Section 22.9.3, added paragraph explaining the -l parameter.

Section 22.9.7, updated text of various examples to show "random" as a supported strategy along with round-robin.

Section 28.2, provide an overview of new ImR capabilities.

Section 28.3, clarify startup behavior description.

Section 28.5, added text noting the ping interval and timeout values are configurable.

Section 28.12.1, added new command line options for ping timeout and lockout.

Summary Of Changes In Patch 1

Section 15.3, dynamic thread pool added to the list of ORB threading strategies.

Sections 15.3.7, 15.3.10, refactored the discussion of dynamic thread pool usage to include a discussion of applying thread pools to ORBs as well as POAs.

Section 17.12, add link to ORBId option.

Section 17.13.14, add description of ORBDynamicThreadPoolName option.

Section 17.13.17, add description of ORBForwardDelay option.

Sections 17.13.18 - 17.13.27, make the section names consistent, fix example

Section 17.13.30, add description of ORBId option.

Section 17.13.55, clarified the description and added more examples.

Section 20.2.3, update table to include ORBDefaultSyncScope option.

Section 20.3.3, add description of ORBDefaultSyncScope strategy setting.

Section 28.2, provide an overview of new ImR capabilities.

Section 28.11.1, update command table.

Section 28.11.1.4 add detailed description of kill command

Section 28.11.1.5, add detailed description of link command

Section 28.11.1.6, expand description of list command

© 2015 Object Computing, Inc.
All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior, written permission of Object Computing, Inc.

Whereas every effort has been made to ensure the technical accuracy of this document, Object Computing, Inc. assumes no responsibility for errors or omissions. This document and technologies described herein are subject to change without notice.

OMG is a registered trademark and Object Request Broker, ORB, OMG IDL, and CORBA are trademarks of Object Management Group, Inc.

Sun, Sun Microsystems, Solaris, Sun Workshop Compiler, Forte, UltraSPARC, and Java are trademarks or registered trademarks of Oracle Corporation in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Oracle Corporation.

UNIX is a registered trademark of The Open Group.

SGI and IRIX are registered trademarks of Silicon Graphics, Inc.

AIX is a registered trademark of IBM.

HP-UX and Tru64 are registered trademarks of Hewlett-Packard Company.

Microsoft, Windows, Windows NT, Windows 2000, Windows 95, Windows 98, Windows XP, Win32, Windows Vist, and Windows 7 are trademarks of Microsoft Corporation in the U.S. and other countries.

Borland C++ Builder is a registered trademark of Borland Software Corporation.

VxWorks and Tornado are registered trademarks of WindRiver Systems.

LynxOS is a registered trademark of LynuxWorks.

OS-9 is the registered trademark of RadiSysCorporation.

All other products or services noted in this guide are covered by the registered trademarks, service marks, or product names of their respective holders.

Your feedback on, or your submission of content to, this documentation is appreciated. Please contact us at either office location via phone or fax, or e-mail your feedback to: techpubs@ociweb.com.

How to contact us:

Object Computing, Inc. (Corporate Office)
12140 Woodcrest Executive Drive, Suite 250
St. Louis, MO 63141
+1.314.579.0066 Voice
+1.314.579.0065 Fax

Support:	support@ociweb.com
Training:	training@ociweb.com
Sales:	sales@ociweb.com
Internet:	www.ociweb.com

Contents

Summary Of Changes In Patch 7 ii
Summary Of Changes In Patch 5 ii
Summary Of Changes In Patch 4 ii
Summary Of Changes In Patch 1 iii

Contents vii

Foreword xix

Preface xxv

Detailed Licensing Terms . . . . **xxxvii**

Part 1 Introduction to TAO Programming 1

Chapter 1 Introduction 3

Design Goals 4
Development History 6
Architecture of TAO 8
CORBA Compliance 13
High Performance and Real-Time Support 15
Relationship Between ACE and TAO 16

Chapter 2 Building ACE and TAO 19

Introduction 19
Where to Get ACE and TAO 20
System Requirements 20
Steps to Build ACE and TAO 21

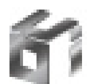
Chapter 3 Getting Started 25

Introduction 25
Setting Up Your Environment 26
A Simple Example 27
Summary 35

Part 2 Features of TAO 37

Chapter 4 TAO IDL Compiler 39

Introduction 39
Executables 40
Output Files Generated 40
Using TAO IDL Compiler Options 42
Preprocessing Options 42
Output File Options 45
Starter Implementation Files 46



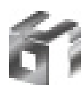
Additional Code Generation Options 48
OpenDDS-related Options 50
Operation Lookup Strategy Options 50
Collocation Strategy Options 51
Back End Options 52
Suppression Options 54
Options Used Internally by TAO 55
Output and Reporting Options 56

Chapter 5 Error Handling 59
Introduction 59
CORBA System Exceptions 61
CORBA User Exceptions 66
TAO Minor Codes 68
Summary 74

Chapter 6 CORBA Messaging 75
Introduction 75
AMI Callback Model 76
Quality of Service Policies 102
Bi-Directional GIOP 115
Endpoint Policy 119
Specifying Differentiated Services with TAO 121

Chapter 7 Asynchronous Method Handling 125
Introduction 125
Participants in an AMH Servant 128
Generating AMH Related Code 130
An AMH Example Program 131
AMH and Advanced CORBA Features 139
Combining AMH with AMI 142

Chapter 8 Real-Time CORBA 149
Introduction 149



Real-Time CORBA Overview 150
Real-Time CORBA Architecture 153
Dynamic Scheduling 173
TAO's Implementation of Real-Time CORBA 185
Client-Propagated Priority Model 205
Server-Declared Priority Model 208
Using the RTScheduling::Current 212
Real-Time CORBA Examples 217

Chapter 9 Portable Interceptors 219
Introduction 219
Using TAO Request Interceptors 220
Marshaling and the Service Context 237
IOR Interceptors 243
The PortableInterceptor::Current 249
Interceptor Policy 255
Summary 256

Chapter 10 Value Types 259
Introduction 259
Uses for Value Types 260
Defining Value Types in IDL 261
A Value Type Example 262
An Example using Value Types as Events 267
Value Types and Inheritance 269
Value Boxes 274
TAO Compliance 274

Chapter 11 Smart Proxies 277
Introduction 277
Smart Proxy Use Cases 279
TAO's Smart Proxy Framework 280
Writing and Using Smart Proxy Classes 286
Linking Your Application 289

A Smart Proxy Example 289

Chapter 12 Local Interfaces 299
Introduction 299
C++ Mapping for LocalObject 300
Changing Existing Interfaces to Local Interfaces 301
Example: ServantLocator 302

Chapter 13 IOR Table 311
Introduction 311
IOR Table 312
Locator 314
IOR Refresh 316

Chapter 14 Using Pluggable Protocols 319
Introduction 319
Protocol Introduction 320
Protocols Provided with TAO 321
Building the Protocol Libraries 323
Loading Pluggable Protocols 323
IIOP 324
UIOP 328
SHMIOP 330
DIOP 333
SSLIOP 335
MIOP/UIPMC 337
HTIOP 341
SCIOP 344
ZIOP 347
COIOP 348
Combining Protocols 349
TAO and IPv6 350
Developing Pluggable Protocols 351

Chapter 15 Multithreading with TAO 357
Introduction 357
Overview of Client/Server Roles in CORBA 359
Multithreading in the Server 363
Multithreading in the Client 412
Summary 429

Part 3 Run-time Configuration of TAO 431

Chapter 16 Configuring TAO Clients and Servers 433
Introduction 433
Patterns and Components for Configuring TAO Clients and Servers 434
The ACE Service Configurator 436
Service Configurator Control Options 439
The ACE Service Configurator Framework 440
XML Service Configurator 445
Service Objects 448
ACE Service Manager 457
Summary 458

Chapter 17 ORB Initialization Options 459
Introduction 459
Interface Definition 460
Controlling Service Configurator Behavior 463
Controlling Debugging Information 464
Optimizing Request Processing 465
Connection Management and Protocol Selection 466
Socket Configuration Options 467
Service Location Options 467
IPv6-Related Options 468
Multiple Invocation Retry Options 469
Implementation Repository Options 471
Miscellaneous Options 472
Option Descriptions 473

Chapter 18 **Resource Factory** 549
Introduction 549
Interface Definition 551
Resource Factory for Qt GUI Toolkit 568
Resource Factory for X Windowing Toolkit 569
Advanced Resource Factory 571
Resource Factory Options 574
Advanced Resource Factory Options 588

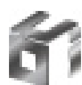
Chapter 19 **Server Strategy Factory** 595
Introduction 595
Interface Definition 596
Default Server Strategy Factory Options 604

Chapter 20 **Client Strategy Factory** 617
Introduction 617
Interface Definition 618
Client Strategy Factory Options 624

Part 4 **TAO Services** 633

Chapter 21 **TAO Services Overview** 635
Introduction 635
Customizing Access to the Services 636
TAO's ORB Services Libraries 637
Locating Service Objects 640

Chapter 22 **Naming Service** 645
Introduction 645
Resolving the Naming Service 647
Naming Service Example 648
Object URLs 655
The NamingContextExt Interface 660
TAO-Specific Naming Service Classes 663



Naming Service Utilities 667
Naming Service Command Line Options 672
Fault Tolerant Naming Service 678
Using the NT Naming Service 690

Chapter 23 Event Service 693
Introduction 693
Overview of the Event Service 694
TAO's Event Channel Implementation 695
How to Use the Event Service 695
tao_cosevent Command Line Options 713
Event Channel Resource Factory 715

Chapter 24 Real-Time Event Service 741
Introduction 741
Overview of the TAO Real-Time Event Service 741
Using the TAO Real-Time Event Service 744
tao_rtevent Command Line Options 783
Event Channel Resource Factory 784
The IIOP Gateway Factory 818

Chapter 25 Notification Service 827
Introduction 827
Notification Service Architecture 828
Notification Service Features 829
Using the Notification Service 864
Compatibility with the Event Service 909
tao_cosnotification Command Line Options 909
Notification Service Configuration Options 911

Chapter 26 Interface Repository 945
Introduction 945
Using the Interface Repository 946
TAO's Interface Repository Implementation 947

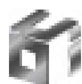
Example IFR Client 951

Chapter 27 TAO Security 961
Preface 961
Introduction 963
Introduction to CORBA Security 964
Secure Sockets Layer Protocol 983
Working with Certificates 994
Building ACE and TAO Security Libraries 1001
Security Unaware Application 1005
Security Policy Controlling Application 1012
Security Policy Enforcing Application 1018
Mixed Security Model Applications 1026
SSLIOP Factory Options 1029

Chapter 28 Implementation Repository 1037
Introduction 1037
New for patched OCI TAO 2.2a 1039
The Operation of the ImR 1040
Basic Indirection Example 1043
Server Start-up 1046
Activation Modes 1051
Using the ImR and the IOR Table 1053
ImR and IOR Table Example 1054
Advanced Examples 1060
Repository Persistence 1060
ImR Utility 1061
tao_imr_locator 1070
tao_imr_activator 1076
JacORB Interoperability 1079

Part 5 Appendices 1081

Appendix A Configuring ACE/TAO Builds 1083



System Requirements 1084
Generating Makefiles and Project Files 1085
GNU Make Build Flags 1090
Using the Build Flags 1094

Appendix B Choosing How To Build ACE and TAO 1103

Appendix C Building ACE and TAO on UNIX 1105
Building ACE and TAO on a UNIX System 1105
Customizing ACE and TAO Builds 1111

Appendix D Building ACE and TAO Using Visual C++ 1115
Building ACE and TAO 1115
Build Notes 1120

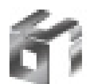
Appendix E Using ACE and TAO with VxWorks 1123
Kernel and System Configuration 1124
Environment Setup 1125

Appendix F Using ACE and TAO with Android 1129
Android Development Kits 1130
Setup ACE/TAO Workspaces 1130
Build The Host Tools 1131
Build The Target Libraries 1132

Appendix G Using ACE and TAO with LynxOS 1135
Cross-Compilation 1135

Appendix H Testing ACE and TAO on VxWorks and LynxOS 1139
Building the Tests 1139
Running the Tests 1140

Appendix I CORBA Compliance 1145
Introduction 1145



CORBA 3.1 1146
CORBA for Embedded 1149
Real-Time CORBA 1151
C++ Language Mapping 1152
Naming Service 1153
Notification Service 1154
Security Service 1154

References 1157

Index 1161

Foreword

Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, Tennessee
USA

August 2011

For the past two decades I've led many R&D groups and projects on distributed object computing (DOC), service-oriented architecture (SOA), and publish/subscribe (pub/sub) middleware in academia, industry, and government. Many middleware technologies have come and gone during this time. For example, ToolTalk, SOM, DCOM, or proprietary message-oriented middleware from the 90's have vanished from all but the most stubborn legacy systems.

An important technology advance over the past two decades has been the evolution and maturation of DOC, SOA, and pub/sub middleware based on open standards, such as the OMG Common Object Request Broker Architecture (CORBA) and the OMG Data Distribution Service (DDS). This middleware resides between applications and the underlying operating systems, network protocol stacks, and hardware. At the heart of DOC

and SOA middleware is the object request broker (ORB), whose primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure to

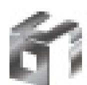
1. extend the scope of portable software via common industry-wide standards,
2. coordinate how parts of applications are connected and how they interoperate across networked nodes, and
3. ease the integration and interoperability of software artifacts developed by multiple technology suppliers and application developers..

When developed and deployed properly, DOC, SOA and pub/sub middleware can reduce the cost and risk of developing distributed applications and systems. The right middleware helps to simplify the development of distributed applications in several ways, including:

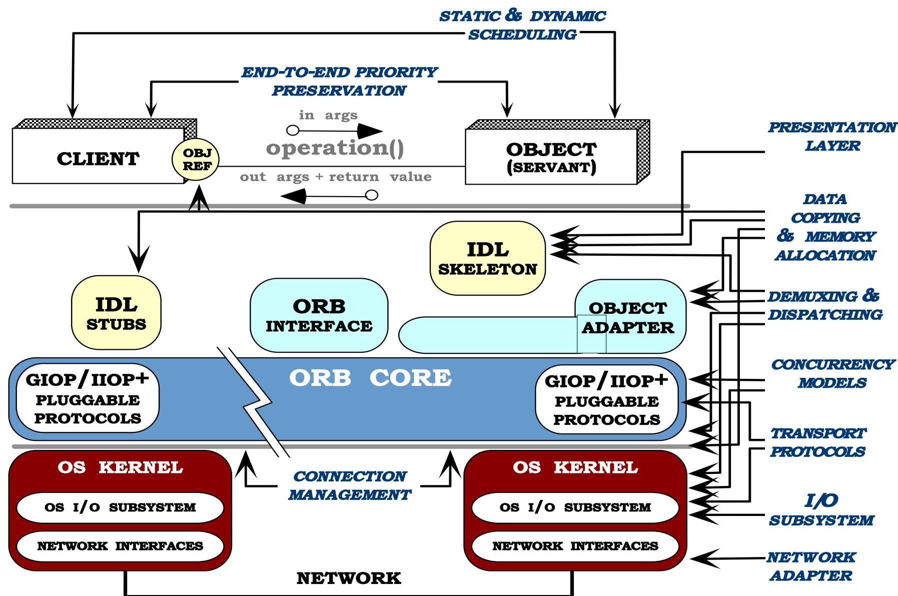
- Providing a consistent set of capabilities that are closer to application design-level abstractions than to the underlying mechanisms in the computing platforms and communication networks. These higher-level abstractions shield application developers from lower-level, tedious, and error-prone platform details (such as socket-level network programming and multithreading) and help application developers effectively manage system resources (such as memory, network, and CPU resources).
- Helping application developers amortize software lifecycle costs by (1) leveraging previous development expertise and capturing implementations of key patterns in reusable frameworks, rather than rebuilding them manually for each use and (2) providing a wide range of reusable application-oriented services (such as logging, event notification, naming, and security) that have proven necessary to operate effectively in distributed environments.

This documentation set from Object Computing, Inc. (OCI) is the seventh installment of an ever growing and continually improving body of knowledge describing the capabilities and effective usage of The ACE ORB (TAO), which is a highly portable, open-source, high-performance, and real-time implementation of the OMG CORBA specification using the C++ frameworks and wrapper facade classes provided by the popular ACE open-source toolkit.

In this seventh release from OCI, the TAO middleware---and the agile open-source community development process that drives it---show a remarkable ability to evolve steadily and continue to lead the field in implementing the latest OMG CORBA specifications and span of supported operating system platforms and C++ compilers. Crucial to the success and longevity of TAO is its robust design based on the patterns and frameworks in ACE that substantially improve its efficiency, predictability, and scalability. Much of the R&D activities over the past several years since the previous OCI release have focused on



optimizing the time and space overhead of ACE and TAO so they can meet more stringent application quality-of-service (QoS) requirements in a wider range of domains.



The figure above illustrates where optimizations and enhancements have been applied to the following TAO components:

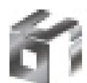
- TAO includes a highly optimized CORBA IIOP protocol engine and an IDL compiler that generates compiled stubs and skeletons that apply a wide range of time and space optimizations. TAO's IDL compiler supports OMG IDL 2.x and 3.x features, including CORBA's object-by-value features.
- TAO's ORB Core supports high-performance and real-time concurrency and dispatching strategies that minimize context switching, synchronization, dynamic memory allocation, and data movement.
- TAO's Portable Object Adapter (POA) implementation uses active demultiplexing and perfect hashing optimizations that associate client requests with target objects in constant time, regardless of the number of objects, operations, or nested POAs.

- TAO can be configured to use non-multiplexed connections that avoid priority inversion and behave predictably when used with multi-rate distributed real-time and embedded applications. It can also be configured to use multiplexed connections, which make it more scalable when run in large-scale Internet or enterprise application environments.
- TAO's pluggable protocols framework supports a wide range of transport mechanisms, such as standard TCP/IP protocols, UDP, IP multicast, UNIX-domain sockets, secure sockets (SSL), and shared memory. This framework also allows users to develop end-to-end fault tolerant systems, where network reliability may require special purpose protocols, such as the Stream Control Transmission Protocol (SCTP), which is also supported by TAO. TAO also supports the HTTP Tunneling Inter-ORB Protocol (HTIOP) that layers GIOP messages over HTTP packets and allows inter-ORB communication across firewalls.
- TAO's Real-time Event Service, Notification Service, and Naming Service integrate the capabilities of the TAO ORB described above to support performance-driven requirements for many application domains and projects.

The OCI TAO 2.2a release provides all these optimizations within the standard CORBA object model and API. The OCI TAO 2.2a documentation set contains over 1,300 pages of text, examples, tables, and figures that explain the strategies and tactics for applying these and many other TAO features and services to address your middleware and application needs. The size of the OCI TAO documentation set is testimony to the inherent value of their product, which will positively impact your project schedule and software development costs. The OCI TAO documentation set and the OCI's website dedicated to TAO (http://www.theaceorb.com) are key community repositories of best practices for developing effective distributed real-time and embedded applications.

We are fortunate that OCI has committed itself to supporting TAO using a truly open-source business model. Many commercial developers would not have received management support to use TAO without the assurance of commercial quality products and services. Now that open-source software has achieved critical mass, commercial users are not just accepting it, but are recognizing its importance in the mix of software development models. Moreover, many government agencies and programs are now mandating the use of standards-based open-source software to avoid proprietary vendor lock-in.

As a result of these trends, users are no longer restricted to choosing between one-size-fits-all commercial-off-the-shelf toolkits vs. custom development. There is now a third way in which users participate heavily in the open-source process and sponsor the fulfillment of their unmet needs as feature additions. Many of OCI's contributions to ACE and TAO during the past two decades have been at their clients' behest for features or ports in a timely manner. By adding JacORB (an open-source Java ORB), OpenDDS (an open-source implementation



of OMG DDS), CIAO (an open-source CORBA Component Model implementation), and JBoss (an open-source J2EE implementation) to its supported products, OCI has met the needs of many TAO users who require a broad and deep range of open-source middleware to support their mission-critical applications.

In closing, it is important to recognize the extent to which the success of TAO has benefited from OCI's open-source development model. I'm proud that so many bright students, staff, OCI engineers, and members of the ACE+TAO open-source community have worked together closely over the past two decades. As you work with TAO please feel free to experiment with, dissect, repair, and improve it. We accept bug reports, appreciate bug fixes/ enhancements, and strive to integrate correct bug fixes quickly using our online problem tracking system. We look forward to seeing your name in subsequent releases of our software!

Douglas C. Schmidt

Preface

What Is TAO?

TAO (The ACE ORB) is an open source, advanced, CORBA-compliant, real-time Object Request Broker (ORB). Its research-guided and industry-driven middleware architecture is designed to meet the stringent Quality of Service (QoS) requirements of real-time applications. This focus on QoS requirements has resulted in TAO's superior end-to-end predictability, efficiency, and scalable performance. TAO has been built with components from the ACE (ADAPTIVE Communication Environment) framework allowing for a highly extensible architecture. Although TAO was designed to meet the demanding requirements of real-time applications, it is also well-suited for general-purpose CORBA applications that do not have stringent QoS requirements.

Licensing Terms

TAO is made available under the *open source software* model. The source code may be freely downloaded and is open for inspection, review, comment,

and improvement. Copies may be freely installed across all your systems and those of your customers. There is no charge for development or run-time licenses. The source code is designed to be compiled, and used, across a wide variety of hardware and operating systems architectures. You may modify it for your own needs, within the terms of the license agreements. You must not copyright ACE/TAO software. For details of the licensing terms, as specified by the Center for Distributed Object Computing, please refer to *"Detailed Licensing Terms"* on page xxxvii.

TAO also utilizes, and is distributed with, two other open source software products; GPERF and MPC. The open source license for MPC is similar to that of ACE and TAO. GPERF is under the GNU Public License (GPL), Version 2. Detailed licensing terms for GPERF are found on page xxxix. Detailed licensing terms for MPC are found on page page xlii.

TAO utilizes two software products obtained/derived from Sun Microsystems. The first product implements the OMG's Internet Inter-ORB Protocol (IIOP). You may copy, modify, distribute, or sublicense the licensed product without charge, as part of a product or software program developed by you, so long as you preserve the interoperability specified by the OMG's IIOP.

The second Sun Microsystems product implements an OMG Interface Definition Language (IDL) compiler front-end. You may also include this product freely in any distribution, and may modify it, as long as you do not remove functionality.

In both cases, you must not use the Sun Microsystems name, logo, or copyrighted material in any subsequent distribution or promotion of your product. In addition, you must include the Sun Microsystems licensing terms, which can be found in their entirety on page xxxix and page xl.

TAO is open source and the development group welcomes code contributions. Active participation by users ensures a robust implementation. Before you send code, please refer to the terms and conditions relating to software submissions on the DOC group's TAO web site, accessible via `<http://www.theaceorb.com/references/>`. Incorporation of your code into TAO means that it is now "open." The ACE/TAO copyright and terms protect the user community from legal infringement or violation.

About This Guide

This Developer's Guide is the sixth edition and corresponds to *OCI's Distribution of TAO Version 2.2a*. It extends the previous edition which corresponded to *OCI's Distribution of TAO Version 2.0a*. The publication and release of this edition does not mean the previous edition is obsolete. Much of the information in the previous edition still applies to the TAO 2.2a release. However, some features/options described in the previous edition are deprecated in TAO 2.2a and are so noted in the text.

This guide focuses on the aspects of TAO that make it unique from other ORBs. It is not meant to be a comprehensive CORBA developer's guide. Please refer to *Advanced CORBA Programming with C++* by Michi Henning and Steve Vinoski or *Pure CORBA* by Fintan Bolton for a more complete treatment of general-purpose CORBA programming topics.

Highlights of the TAO 2.2a Release

OCI's Distribution of TAO Version 2.2a includes new features and improvements over the previous release. This section highlights some of the more important and visible changes and describes how they may impact your existing TAO applications.

Performance Enhancements

- *Dynamic Thread Pooling in POAs*—A new Dynamic Thread Pool and Queuing strategy was created for POA usage. It leverages the existing Custom Servant Dispatching framework for invocation and activation. The strategy dynamically adjusts the number of threads using configuration parameters similar to the ORB Dynamic Thread Pool. Using a thread pool for request processing is useful for avoiding nested upcalls and for ensuring efficient request dequeuing. However, static thread pools made too small may result in request processing back logs, or made too large waste resources. A dynamic thread pool can grow to accommodate timely execution of requests when a surge happens, but will taper off releasing resources when activity is reduced. See 15.3.10 for further details.

Reliability Enhancements

- *Fault Tolerant Implementation Repository*— The Implementation Repository Locator now supports a dual-redundant fault tolerant configuration which provides replication and seamless failover between the primary and backup locator servers. See 28.12.2 for further details.
- *High performance Implementation Repository*— The Implementation Repository Locator has been re-implemented using AMI/AMH to avoid the problem of nested upcalls under heavy load.
- *Fault Tolerant Naming Service*— A new Fault Tolerant Naming Service (`tao_ft_naming`), provides dual-redundant fault tolerant servers which utilize replication and seamless failover between the primary and backup server. The Fault Tolerant Naming Service can be used to provide load balancing capabilities through the use object groups. This feature is supported by a separate utility for managing the object groups (`tao_nsgroup`) as well as a programmatic interface via IDL. See 22.9 for further details.
- *Multiple Invocation Retry Support*—Extended TAO to support multiple retries in the presence of `COMM_FAILURE`, `INV_OBJREF`, `OBJECT_NOT_EXIST`, and `TRANSIENT` exceptions. This feature is used to support fault tolerant services (specifically the Fault Tolerant Naming and Implementation Repository services described earlier). The new invocation retry support allows configuration on how many times to try to connect to each server and the delay between tries. See 17.10 for further details.

Interoperability

- *JacORB*—TAO's Implementation Repository has been extended to allow it to manage JacORB application servers. See 28.14 for further details.

Important Bug Fixes

- Many other bug fixes or work-arounds appear in this release. See `$ACE_ROOT/OCIReleaseNotes.html` for details.

Structure of the Guide

Part 1, "Introduction to TAO Programming"

This section discusses the design goals, development history, and architecture of TAO. It also describes TAO's support for various aspects of the OMG CORBA specifications and discusses TAO's extensions to these specifications to improve predictability and performance. It addresses how to obtain, build, and install the TAO source code distribution. (Detailed instructions for building in specific environments are presented in the appendices.) This section will also help you quickly get started writing and building applications with TAO. Getting started with TAO on different platforms is simpler than in the past with the help of MPC, which is also described in this section. We also introduce a simple *Messenger* example that is used throughout the guide to illustrate features of TAO as they are discussed.

Part 2, "Features of TAO"

This section describes several features of TAO, including: TAO's IDL compiler; dealing with errors and exceptions; TAO's implementation of the CORBA Messaging specification, including Asynchronous Method Invocation (AMI); TAO's Asynchronous Method Handling (AMH) feature; Real-Time CORBA; Portable Interceptors, Value Types, and Smart Proxies; using local objects; using TAO's IOR Table feature; using Pluggable Protocols with TAO; and multithreading with TAO.

Part 3, "Run-time Configuration of TAO"

The role of the ACE Service Configurator in configuring TAO at run time is discussed in detail in this section. This section describes several initialization options and environment variables for configuring the ORB and fully describes configuration options for TAO's internal resource and strategy factories.

Part 4, "TAO Services"

This section describes the various services that TAO offers to CORBA applications. These services include some of the standard CORBA services (e.g., Naming, Events, Notification, Interface Repository, Implementation Repository, Security), as well as the TAO-specific Real-Time Event Service.

Part 5, “Appendices”
This section includes appendices describing how to configure, build, test, and use TAO, including detailed information for specific operating environments. Also included is an appendix that details TAO’s level of compliance with particular OMG specifications. Following the appendices is a list of important references.

Conventions

This guide uses the following conventions:

`Fixed pitch text`	Indicates example code or information a user would enter using a keyboard.
`Bold fixed pitch text`	Indicates example code that has been modified from a previous example or text appearing in a menu or dialog box.
Italic text	Indicates a point of emphasis.
...	A horizontal ellipsis indicates that the statement is omitting text.
. . .	A vertical ellipsis indicates that a segment of code is omitted from the example.

Coding Examples

Throughout this guide, we illustrate topics with coding examples. The examples in this guide are intended for illustration purposes and should not be considered to be “production-ready” code. In particular, error handling is sometimes kept to a minimum to help the reader focus on the particular feature or technique that is being presented in the example. The source code for all these examples is available as part of the ACE and TAO source code distribution in the `$TAO_ROOT/DevGuideExamples` and `$TAO_ROOT/orbsvcs/DevGuideExamples` directories. The example files are arranged in subdirectories by chapter name. MPC files are provided with all the examples for generating build-tool specific files, such as GNU Makefiles or Visual C++ project and solution files. See `$TAO_ROOT/DevGuideExamples/readme.txt` for instructions on building

the examples. A Perl script named `run_test.pl` is provided with each example so you can easily run it.

OMG Specification References

Throughout this guide, we refer to various specifications published by the Object Management Group (OMG). These references take the form *group/number* where *group* represents the OMG working group responsible for developing the specification, or the keyword `formal` if the specification has been formally adopted, and *number* represents the year, month, and serial number within the month the specification was released. For example, Part 1 of the OMG CORBA 3.1 specification is referenced as `formal/08-01-04`.

You can download any referenced OMG specification directly from the OMG web site by prepending `<http://www.omg.org/cgi-bin/doc?>` to the specification's reference. Thus, the specification `formal/08-01-04` becomes `<http://www.omg.org/cgi-bin/doc?formal/08-01-04>`. Providing this destination to a web browser should take you to a site from which you can download the referenced specification document.

Additional Documents

In several places throughout the text, we refer to information found in the following books:

> Michi Henning and Steve Vinoski. 1999. *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley.
>
> Fintan Bolton. 2002. *Pure CORBA: A Code-Intensive Premium Reference*. Sams Publishing.

The above books provide extensive coverage of general-purpose CORBA programming topics and techniques that are not covered in this book. We *strongly* recommend that you obtain copies of these books if you do not already have access to them.

OCI will continue to produce documentation for TAO. In addition, we will publish any corrections or errata to the existing documentation on the OCI web site at `<http://www.theaceorb.com/references/>` as necessary.

Finally, be sure to visit the TAO Frequently Asked Questions (FAQ) pages at `<http://www.theaceorb.com/faq/>`.

Product Version Numbering Scheme

Version numbers for *OCI's Distribution of TAO* are different from those of the code base maintained by the "DOC group," even though *OCI's Distribution of TAO* is derived from that code base. For example, TAO 2.2a is based on the TAO 2.2 micro release kit distributed by the DOC group with patches carefully applied to fix specific problems.

Also, note that neither distribution's version has any relationship to OMG specification version numbers. Neither group could ever hope to keep up with the other.

See *"What's the relationship between OCI's TAO and DOC's TAO?"* in the OCI TAO FAQ at `<http://www.theaceorb.com/faq/>` for more information about the relationship between *OCI's Distribution of TAO* and the DOC group's distribution.

Note *Check* `<http://www.theaceorb.com/references/>` *to find source code patches to OCI's distributions of TAO.*

Note *Mixing OCI patches and DOC group patches is untested and unlikely to work correctly. Neither group supports this configuration.*

Supported Platforms

TAO has been ported to a wide variety of platforms, operating systems, and C++ compilers. We continually update TAO to support additional platforms. Please visit `<http://www.theaceorb.com>` for the most recent platform support information.

Customer Support

Enterprises are discovering that it takes considerable experience, knowledge, and money to design and build a complex distributed application that is robust and scalable. OCI can help you successfully architect and deliver your solution by drawing on the experience of seasoned architects who have extensive experience in today's middleware technologies and who understand how to leverage the power of CORBA.

Our service areas include systems architecture, large-scale distributed application architecture, and object oriented design and development. We excel in technologies such as CORBA (ACE+TAO and JacORB), DDS (OpenDDS), J2EE, FIX (QuickFIX), and FAST (QuickFAST).

Support offerings for TAO include:

- Consulting services to aid in the design of extensible, scalable, and robust CORBA solutions, including the validation of domain-specific approaches, service selection, ORB customization and extension, and migrating your applications to TAO or JacORB from other ORBs.
- 24x7 support that guarantees the highest response level for your production-level systems.
- On-demand service agreement for identification and assessment of minor bugs and issues that may arise during the development and deployment of CORBA-based solutions.

Our architects have specific and extensive domain expertise in security, telecommunications, defense, financial, and other real-time distributed applications.

We can provide professionals who can assist you on short-term engagements, such as architecture and design review, rapid prototyping, troubleshooting, and debugging. Alternatively, for larger engagements, we can provide mentors, architects, and programmers to work alongside your team, providing assistance and thought leadership throughout the life cycle of the project.

Contact us at +1.314.579.0066 or `<sales@ociweb.com>` for more information.

Object Technology Training

OCI provides a rich program of more than 50 well-focused courses designed to give developers a solid foundation in a variety of technical topics, such as Object Oriented Analysis and Design, C++ Programming, Java Programming, Distributed Computing Technologies, Patterns, XML, and UNIX/Linux. Our courses clearly explain major concepts and techniques, and demonstrate, through hands-on exercises, how they map to real-world applications.

Note *Our training offerings are constantly changing to meet the latest needs of our clients and to reflect changes in technology. Be sure to check out our web site at* `<http://www.ociweb.com>` *for updates to our Educational Programs.*

On-Site Classes

We can provide the following courses at your company's facility, integrating them seamlessly with other employee development programs. For more information about these or other courses in the OCI curriculum, visit our course catalog on-line at `<http://www.ociweb.com/training/>`.

Introduction to CORBA

In this one-day course, you will learn the benefits of distributed object computing; the role CORBA plays in developing distributed applications; when and where to apply CORBA; and future development trends in CORBA.

CORBA Programming with C++

In this hands-on, four-day course, you will learn: the role CORBA plays in developing distributed applications; the OMG's Object Management Architecture; how to write CORBA clients and servers in C++; how to use CORBAservices such as Naming and Events; using CORBA exceptions; and basic and advanced features of the Portable Object Adapter (POA). This course also covers the specification of interfaces using OMG Interface Definition Language (IDL) and details of the OMG IDL-to-C++ language mapping, and provides hands-on practice in developing CORBA clients and servers in C++ (using TAO).

Advanced CORBA Programming Using TAO

In this intensive, hands-on, four-day course, you will learn: several advanced CORBA concepts and techniques and how they are supported by TAO; how to configure TAO components for performance and space optimizations; and how to use TAO's various concurrency models to meet your application's end-to-end QoS guarantees. The course covers recent additions to the CORBA specifications and to TAO to support real-time CORBA programming, including Real-Time CORBA. It also covers TAO's Real-Time Event Service, Notification Service, and Implementation Repository, and provides extensive hands-on practice in developing advanced TAO clients and servers in C++. This course is intended for experienced and serious CORBA/C++ programmers.

Using the ACE C++ Framework

In this hands-on, four-day course, you will learn how to implement Interprocess Communication (IPC) mechanisms using the ACE (ADAPTIVE Communication Environment) IPC Service Access Point (SAP) classes and the Acceptor/Connector pattern. The course will also show you how to use a Reactor in event demultiplexing and dispatching; how to implement thread-safe applications using the ACE thread encapsulation class categories; and how to identify appropriate ACE components to use for your specific application needs.

Object-Oriented Design Patterns and Frameworks

In this three-day course, you will learn the critical language and terminology relating to design patterns, gain an understanding of key design patterns, learn how to select the appropriate pattern to apply in a given situation, and learn how to apply patterns to construct robust applications and frameworks. The course is designed for software developers who wish to utilize advanced object oriented design techniques and managers with a strong programming background who will be involved in the design and implementation of object oriented software systems.

OpenDDS Programming with C++

In this three-day course, you will learn to build applications using OpenDDS, the open source implementation of the OMG's Data Distribution Service (DDS) for Real-Time Systems. You will learn how to build data-centric systems that share data via OpenDDS. You will also learn to configure

OpenDDS to meet your application's Quality of Service requirements.This course if intended for experienced C++ developers.

C++ Programming Using Boost

In this four-day course, you will learn about the most widely used and useful libraries that make up Boost. Students will learn how to easily apply these powerful libraries in their own development through detailed expert instructor-led training and by hands-on exercises. After finishing this course, class participants will be prepared to apply Boost to their project, enabling them to more quickly produce powerful, efficient, and platform independent applications.

For information about training dates, contact us by phone at +1.314.579.0066, via electronic mail at training@ociweb.com, or visit our web site at `<http://www.ociweb.com>` *to review the current course schedule.*

Detailed Licensing Terms

The ACE ORB source code is copyrighted by Dr. Douglas C. Schmidt and his research group at Washington University, University of California, Irvine, and Vanderbilt University. The actual terms are reproduced below. TAO is made available by means of an open source model. TAO may be used without the payment of development license or run-time fees. The TAO source may be made available along with any added value products that utilize TAO. The acknowledgement of the use of TAO should conform to its copyright terms. You may reference the OCI version number and OCI web site as a location of the source code. OCI is an authorized distributor of TAO products and services. The use of the ACE, The ACE ORB and TAO trade or service marks is by permission of Dr. Douglas C. Schmidt.

TAO, under certain circumstances, also uses a software program called GPERF. This software was also written by Dr. Schmidt and is licensed under the terms of the Free Software Foundation's GNU Public License (GPL). Details on this license may be found in this section.

TAO also includes software from Sun Microsystems. This software is related to the IDL compiler and IIOP. This software may also be freely distributed without fees. The licensing details are also published in this section.

Please read this section carefully to understand your obligations as a user.

The following are the terms and conditions of The ACE ORB source code:

Copyright and Licensing Information for ACE(TM), TAO(TM), CIAO(TM), DAnCE(TM), and CoSMIC(TM)

ACE(TM), TAO(TM), CIAO(TM), DAnCE(TM), and CoSMIC(TM) (henceforth referred to as "DOC software") are copyrighted by Douglas C. Schmidt and his research group at Washington University, University of California, Irvine, and Vanderbilt University Copyright (c) 1993-2013, all rights reserved. Since DOC software is open-source, freely available software, you are free to use, modify, copy, and distribute--perpetually and irrevocably--the DOC software source code and object code produced from the source, as well as copy and distribute modified versions of this software. You must, however, include this copyright statement along with any code built using DOC software that you release. No copyright statement needs to be provided if you just ship binary executables of your software products.

Usage

You can use DOC software in commercial and/or binary software releases and are under no obligation to redistribute any of your source code that is built using DOC software. Note, however, that you may not do anything to the DOC software code, such as copyrighting it yourself or claiming authorship of the DOC software code, that will prevent DOC software from being distributed freely using an open-source development model. You needn't inform anyone that you're using DOC software in your software, though we encourage you to let

<doc_group@cs.wustl.edu> us know so we can promote your project in the DOC software success stories <http://www.cs.wustl.edu/~schmidt/ACE-users.html>.

Warranty

DOC software is provided as is with no warranties of any kind, including the warranties of design, merchantability, and fitness for a particular purpose, noninfringement, or arising from a course of dealing, usage or trade practice.

Support

DOC software is provided with no support and without any obligation on the part of Washington University, UC Irvine, Vanderbilt University, their employees, or students to assist in its use, correction, modification, or enhancement. A number of companies around the world provide commercial support for DOC software, however.

Year 2000

DOC software is Y2K-compliant, as long as the underlying OS platform is Y2K-compliant. Likewise, DOC software is compliant with the new US daylight savings rule passed by Congress as “The Energy Policy Act of 2005,” which established new daylight savings times (DST) rules for the United States that expand DST as of March 2007. Since DOC software obtains time/date and calendaring information from operating systems users will not be affected by the new DST rules as long as they upgrade their operating systems accordingly.

Liability

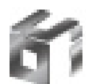
Washington University, UC Irvine, Vanderbilt University, their employees, and students shall have no liability with respect to the infringement of copyrights, trade secrets or any patents by DOC software or any part thereof. Moreover, in no event will Washington University, UC Irvine, or Vanderbilt University, their employees, or students be liable for any lost revenue or profits or other special, indirect and consequential damages.

Submissions

The ACE, TAO, CIAO, DAnCE, and CoSMIC web sites are maintained by the DOC Group at the Institute for Software Integrated Systems (ISIS) and the Center for Distributed Object Computing of Washington University, St. Louis for the development of open-source software as part of the open-source software community. Submissions are provided by the submitter ``as is'' with no warranties whatsoever, including any warranty of merchantability, noninfringement of third party intellectual property, or fitness for any particular purpose. In no event shall the submitter be liable for any direct, indirect, special, exemplary, punitive, or consequential damages, including without limitation, lost profits, even if advised of the possibility of such damages.

Trademarks

The names ACE(TM), TAO(TM), CIAO(TM), DAnCE(TM), CoSMIC(TM), Washington University, UC Irvine, and Vanderbilt University, may not be used to endorse or promote products or services derived from this source without express written permission from Washington University, UC Irvine, or Vanderbilt University. This license grants no



permission to call products or services derived from this source ACE(TM), TAO(TM), CIAO(TM), DAnCE(TM), or CoSMIC(TM), nor does it grant permission for the name Washington University, UC Irvine, or Vanderbilt University to appear in their names.

Contact

If you have any suggestions, additions, comments, or questions, please let me <d.schmidt@isis.vanderbilt.edu> know.

Douglas C. Schmidt <http://www.dre.vanderbilt.edu/~schmidt/>

Copyright and Licensing Information for GPERF

GPERF is a standalone software program. GPERF generates perfect hash functions for lookups based on a set of key words when the key words are known in advance. They are called perfect hash functions because only a single access into the data structure is needed to perform a lookup. When the set of IDL operations is known in advanced TAO uses the perfect hash functions generated by GPERF to perform the operation lookup in constant time.

GPERF was originally developed by Dr. Douglas C. Schmidt. Dr. Schmidt subsequently signed the copyright over to the Free Software Foundation, causing gperf to be licensed under the GPL (GNU General Public License) Version 2. The FSF still maintains that version of gperf. When perfect hashing was added as an option to TAO, gperf was selected to provide that function. It was extended and enhanced to meet the more demanding needs of TAO and a derived version was placed in the ACE application libraries, `ace_gperf`. When using TAO under certain circumstances you may elect to use that version of `ace_gperf`, which is part of the ACE distribution of examples and optional programs. Both the current FSF gperf and `ace_gperf` are based on the original implementation. Since `ace_gperf` is derived from the original GPL'ed version, it too is licensed under the GPL Version 2.

The following terms are found in the source files for gperf:

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, or visit their web site www.gnu.ai.mit.edu.

IDL Compiler Front End

This product is protected by copyright and distributed under the following license restricting its use.

The Interface Definition Language Compiler Front End (CFE) is made available for your use provided that you include this license and copyright notice on all media and documentation and the software program in which this product is incorporated in whole or part. You may copy and extend functionality (but may not remove functionality) of the Interface Definition

Language CFE without charge, but you are not authorized to license or distribute it to anyone else except as part of a product or program developed by you or with the express written consent of Sun Microsystems, Inc. ("Sun").

The names of Sun Microsystems, Inc. and any of its subsidiaries or affiliates may not be used in advertising or publicity pertaining to distribution of Interface Definition Language CFE as permitted herein.

This license is effective until terminated by Sun for failure to comply with this license. Upon termination, you shall destroy or return all code and documentation for the Interface Definition Language CFE.

INTERFACE DEFINITION LANGUAGE CFE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.

INTERFACE DEFINITION LANGUAGE CFE IS PROVIDED WITH NO SUPPORT AND WITHOUT ANY OBLIGATION ON THE PART OF Sun OR ANY OF ITS SUBSIDIARIES OR AFFILIATES TO ASSIST IN ITS USE, CORRECTION, MODIFICATION OR ENHANCEMENT.

SUN OR ANY OF ITS SUBSIDIARIES OR AFFILIATES SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY INTERFACE DEFINITION LANGUAGE CFE OR ANY PART THEREOF.

IN NO EVENT WILL SUN OR ANY OF ITS SUBSIDIARIES OR AFFILIATES BE LIABLE FOR ANY LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL DAMAGES, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

Sun, Sun Microsystems and the Sun logo are trademarks or registered trademarks of Oracle Corporation.

Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065

NOTE:

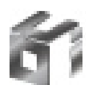
SunOS, SunSoft, Sun, Solaris, Sun Microsystems or the Sun logo are trademarks or registered trademarks of Oracle Corporation.

IIOP Engine

This notice applies to all files in this software distribution that were originally derived from SunSoft IIOP code (these files contain Sun Microsystems copyright notices).

COPYRIGHT AND LICENSING

Copyright 1995 Sun Microsystems, Inc. Printed in the United States of America. All Rights Reserved.



This software product (LICENSED PRODUCT), implementing the Object Management Group's "Internet Inter-ORB Protocol", is protected by copyright and is distributed under the following license restricting its use. Portions of LICENSED PRODUCT may be protected by one or more U.S. or foreign patents, or pending applications.

LICENSED PRODUCT is made available for your use provided that you include this license and copyright notice on all media and documentation and the software program in which this product is incorporated in whole or part.

You may copy, modify, distribute, or sublicense the LICENSED PRODUCT without charge as part of a product or software program developed by you, so long as you preserve the functionality of interoperating with the Object Management Group's "Internet Inter-ORB Protocol" version one. However, any uses other than the foregoing uses shall require the express written consent of Sun Microsystems, Inc.

The names of Sun Microsystems, Inc. and any of its subsidiaries or affiliates may not be used in advertising or publicity pertaining to distribution of the LICENSED PRODUCT as permitted herein.

This license is effective until terminated by Sun for failure to comply with this license. Upon termination, you shall destroy or return all code and documentation for the LICENSED PRODUCT.

LICENSED PRODUCT IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.

LICENSED PRODUCT IS PROVIDED WITH NO SUPPORT AND WITHOUT ANY OBLIGATION ON THE PART OF SUN OR ANY OF ITS SUBSIDIARIES OR AFFILIATES TO ASSIST IN ITS USE, CORRECTION, MODIFICATION OR ENHANCEMENT.

SUN OR ANY OF ITS SUBSIDIARIES OR AFFILIATES SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY LICENSED PRODUCT OR ANY PART THEREOF.

IN NO EVENT WILL SUN OR ANY OF ITS SUBSIDIARIES OR AFFILIATES BE LIABLE FOR ANY LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL DAMAGES, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

SunOS, SunSoft, Sun, Solaris, Sun Microsystems and the Sun logo are trademarks or registered trademarks of Oracle Corporation.

Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065

Make Project Creator (MPC)

ACE and TAO are delivered with an easy to use, open source, freely available, build environment called MPC. The following are the terms for its usage.

Copyright and Licensing Information for MPC

MPC (Licensed Product) is protected by copyright, and is distributed under the following terms.

MPC (Make, Project, and Workspace Creator) is an open-source tool developed by OCI and written in Perl. It is designed to generate a variety of tool-specific project files from a common baseline. Through the powerful combination of inheritance and defaults, MPC is able to reduce the maintenance burden normally associated with keeping multiple target platforms, their unique build tools, and inconsistent feature sets current. It is also easily extensible to support new build environments. The objective is to solve the prevalent problem of fragile build environments usually experienced by developer groups by replacing it with a singular, robust build environment and an active community of users committed to its evolution.

Since MPC is open source and free of licensing fees, you are free to use, modify, and distribute the source code, as long as you include this copyright statement.

In particular, you can use MPC to build proprietary software and are under no obligation to redistribute any of your source code that is built using MPC. Note, however, that you may not do anything to the MPC code, such as copyrighting it yourself or claiming authorship of the MPC code, that will prevent MPC from being distributed freely using an open-source development model.

Warranty

LICENSED PRODUCT IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY, AND FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

Support

LICENSED PRODUCT IS PROVIDED WITH NO SUPPORT AND WITHOUT ANY OBLIGATION ON THE PART OF OCI OR ANY OF ITS SUBSIDIARIES OR AFFILIATES TO ASSIST IN ITS USE, CORRECTION, MODIFICATION, OR ENHANCEMENT.

Support may be available from OCI to users who have agreed to a support contract.

Liability

OCI OR ANY OF ITS SUBSIDIARIES OR AFFILIATES SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS, OR ANY PATENTS BY LICENSED PRODUCT OR ANY PART THEREOF.

IN NO EVENT WILL OCI OR ANY OF ITS SUBSIDIARIES OR AFFILIATES BE LIABLE FOR ANY LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT

AND CONSEQUENTIAL DAMAGES, EVEN IF OCI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

MPC copyright OCI. St. Louis MO USA, 2003-2013.

Part 1

Introduction to TAO Programming

CHAPTER 1

Introduction

TAO (The ACE ORB) is an open source, advanced, CORBA-compliant, real-time Object Request Broker (ORB) that has been developed under the direction of Dr. Douglas C. Schmidt by members of the Distributed Object Computing (DOC) Group. The DOC Group is a distributed research consortium lead by Dr. Schmidt and consisting of the DOC group in the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, Nashville, the Center for Distributed Object Computing in the Computer Science department at Washington University, and the Laboratory for Distributed Object Computing in the Electrical Engineering and Computer Science department at the University of California, Irvine. The DOC Group also includes members of several companies and organizations all over the world, including Object Computing, Inc., Siemens ZT, University of Maryland, Remedy IT, Riverace Corporation, PrismTech, LMCO-ATL, Qualcomm, Hewlett-Packard, and Automated Trading Desk.

The purpose of the DOC group is “to support advanced research and development (R&D) on middleware and modeling tools using an open source software development model, which allows academics, developers, and end-users to participate in leading-edge R&D projects driven by the free market of ideas, requirements, and resources.”

In addition to supporting advanced R&D projects, TAO is being used in many commercial and government distributed applications in the areas of defense, telecommunications, multimedia, finance, manufacturing, information and systems management, and many others. Such users benefit from (and contribute to) ACE and TAO's open source model, yet demand stable, well documented, and commercially supported releases of TAO.

OCI meets the needs of this demanding TAO user community through the publication of this guide, a controlled release cycle, free source code plus inexpensive binary distributions, commercial support (including 24x7 support for organizations using TAO in their deployed applications), training for beginning through advanced users, and participation in the OMG.

To learn more, visit OCI's web site at `<http://www.ociweb.com>` or `<http://www.theaceorb.com>`, or contact `<sales@ociweb.com>`.

1.1 Design Goals

The design goals of TAO include the following:

- *Well-suited for real-time environments.*

  TAO's design includes real-time requirements considerations, including avoiding end-to-end priority inversion, maintaining upper bounds on latency and jitter, and providing bandwidth guarantees. In particular, TAO improves the predictability of many intra-ORB functions, such as endpoint and request demultiplexing, concurrency control, and operation dispatching. Moreover, TAO enables applications to specify their quality of service (QoS) requirements to ORB endsystems.

- *Well-suited for conventional environments.*

  TAO is well-suited for conventional (i.e., non real-time) environments, and may be used as a general-purpose ORB without leveraging any of its real-time features. This allows developers already familiar with CORBA to get up to speed quickly using TAO and later take advantage of its real-time QoS support as needed.

- *Exhibits high-performance characteristics.*

  TAO meets the stringent throughput requirements that are necessary to support performance-sensitive industrial-strength applications. TAO

optimizes many of the key determinants of ORB performance, including request demultiplexing and operation dispatching, presentation layer conversions, synchronization, context switching, memory management, and data copying. As a result, TAO's performance is very competitive with lower-level networking APIs, such as sockets.

- *Complies with open standards.*

  TAO was designed to be compliant with OMG CORBA specifications. In general, the current version of TAO is compliant with the CORBA 3.0 specification. Because of its compliance with OMG specifications, TAO is interoperable with other ORBs, and TAO application code written in accordance with the OMG's IDL to C++ language mapping is portable to other compliant ORBs. See Appendix I for complete information on TAO's level of compliance with various specifications.

- *Configurable.*

  When used in special purpose (e.g., embedded) environments, software must often be tailored to meet the special demands imposed by that environment. TAO's modular architecture and run-time configurability allow developers to tailor it to meet the specific needs of their application's operating environment (e.g., to reduce the run-time memory footprint or to strategize request demultiplexing for more deterministic performance). In addition, TAO fully supports the standard CORBA policy framework that allows developers to control ORB behavior programmatically. TAO is designed to give the developer a great deal of control over the run-time environment.

- *Extensible.*

  TAO comes with default resource and strategy factories for configuring clients and servers, and default pluggable protocol factories for choosing among certain transport protocols. These default factories are designed to provide enough flexibility to meet the needs of the vast majority of applications, even for very demanding environments. However, in cases where the default factories cannot satisfy a project's special needs, you can develop custom resource, strategy, and protocol factories that can be "plugged in" to TAO's core framework-based architecture with no impact on application code.

  TAO also supports the OMG Portable Interceptors specification. Portable Interceptors provide hooks that are invoked at predefined points in the

request and reply paths of an operation invocation or during the generation of an IOR. Interceptors are registered with the ORB via ORB initializers. Developers can define their own code to be executed at each interception point to perform application-specific tasks such as logging, debugging, or security and authentication. See Chapter 9 for more information on Portable Interceptors.

TAO also supports Smart Proxies, which allow developers to replace the normal proxy implementations, generated by the IDL compiler, with application-specific proxies for customizing the behavior of client request invocations. See Chapter 11 for more information on Smart Proxies.

In addition, because TAO is open source, you are free to modify it in almost any way to meet your project's unique requirements.

- *Portable.*

  For an ORB to be an effective tool, it must be implemented on all the platforms on which it is needed. TAO is built on top of the ADAPTIVE Communication Environment (ACE) framework, which is a C++ framework that provides object-oriented abstractions to operating system, networking, and interprocess communications facilities, as well as higher level patterns that encapsulate common communications mechanisms. By utilizing ACE, the TAO source code is more easily ported to diverse target platforms than would be the case if specific APIs were utilized. Moreover, the use of ACE mitigates the least common denominator approach of typical porting solutions. Thus, TAO can be optimized for any platform.

1.2 Development History

Since the early 1990s, Dr. Schmidt has led teams conducting advanced research and development on distributed-object computing middleware using an open source software development model. The open source model is a very pragmatic way of evolving software in a rapidly changing environment. It harnesses the collective wisdom, experiences, expertise, and requirements of its most demanding users to ensure that their needs are rapidly met.

Traditionally, ORBs have implemented only *best-effort* service models. Many corporate and government organizations have sponsored the development of TAO because they need both standards-based middleware and the ability to meet and enforce QoS for their applications and distributed systems. These

organizations require not just classic hard real-time characteristics, but soft real-time and best-effort support, as well. A partial list of these industries and government organizations, and their applications, includes:

- *Telecommunications*—switching, network management, software-defined radio, and mobile/hand-held systems.
- *Medical*—imaging, integrated patient monitoring, and tele-medicine.
- *Aerospace and Defense*—avionics, signal processing, simulation, command and control, and training.
- *Financial Services*—trading services, portfolio analysis, real-time risk analysis, and simulation.
- *Manufacturing*—machine tools, robotics, and process control.
- *Information and Systems Management*—storage management, systems and data recovery, customer information management, asset management, capacity management, and infrastructure and application control.

In addition to requirements for real-time and high-performance systems, TAO's sponsors require close conformance to the OMG specifications to ensure that their developers can design compliance into their baseline systems architecture. Moreover, there are often specific customer-application-generated requirements that ensure a pragmatic set of extensions to the OMG specifications. These extensions must meet the needs of real-time developers who typically demand complete control of all the system resources to guarantee success.

Original contributors to TAO's technical architecture, strategies and techniques include senior developers from sponsoring organizations who have extensive experience with first generation ORBs, understand real-time issues, and come from diverse industry backgrounds. The result of such wide-ranging inputs is an ORB with a highly adaptable architecture, well-suited for a diverse and demanding customer base. When combined with the thousands of contributors from the ACE and TAO open source community, it is fair to say that no other ORB has had such a signficant degree of participation from its users and sponsors.

Note *You can see a full list of contributors to ACE/TAO (over 2000 individuals) at* `<http://www.cs.wustl.edu/~schmidt/ACE-members.html>`. *This list*

indicates the size and diversity of the open source community that has grown around these products.

1.3 Architecture of TAO

In this section, we describe the architecture of TAO. If you are new to CORBA, you should read *Advanced CORBA Programming with C++*, Chapter 2 (especially sections 2.4 and 2.5) before reading this section.

Figure 1-1 shows the relationships among TAO's ORB endsystem components. An *ORB endsystem* is an endsystem (e.g., PC, workstation,

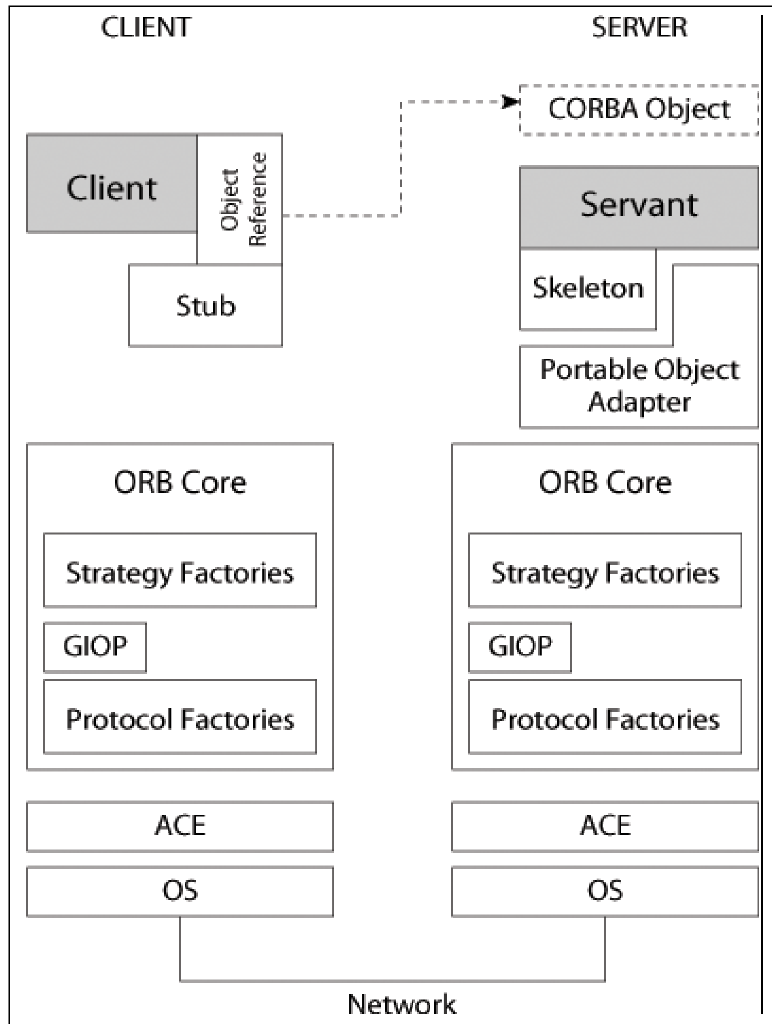


Figure 1-1: TAO Architecture

embedded processor board) that contains one or more network interfaces, an I/O subsystem (e.g., containing the operating system's protocol stacks like TCP/IP), an ORB, and possibly various standard services (e.g., Naming and Event). As a developer, you will typically write the client and servant (shaded

components in Figure 1-1). Components of the TAO architecture are described below.

1.3.1 Client

A CORBA client has two responsibilities: (1) obtain object references to CORBA objects and (2) invoke operations on them. The client is unaware of where or how the CORBA object is implemented. The only operations the client is able to invoke are those defined in the object's interface, expressed in OMG Interface Definition Language (IDL).

1.3.2 CORBA Object

CORBA objects are abstract entities. Each CORBA object has a unique identity and an interface, defined in IDL. A CORBA object is associated with a concrete implementation of the interface at run time by an Object Adapter.

1.3.3 Servant

A servant provides a concrete implementation for a CORBA object. In object-oriented programming languages such as C++, servants are implemented as objects and live within a server process or task. Normally, you will create an implementation class for each IDL interface and your servant objects will be instances of this class. The client is completely unaware of how an interface is implemented and has no knowledge of servants. A servant is associated with a CORBA object at run time via an Object Adapter.

1.3.4 IDL Stubs and Skeletons

The TAO IDL compiler generates C++ stubs and skeletons from IDL interface definitions. Stubs are used on the client side to provide a strongly typed, static invocation interface (SII) that converts C++ function calls into CORBA requests, including marshaling operation parameters into a common binary representation. The generated stubs can also optimize operation invocations when the target object is collocated with (i.e., in the same address space as) the client. Skeletons provide a static skeleton interface (SSI) that demarshals the binary data back into C++ types that are meaningful to servant implementations. You will normally compile and link the generated stubs and skeletons into your application code. See Chapter 4 for more information on TAO's IDL compiler.

In addition to the SII and SSI model described above, TAO also supports the Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI), defined by the CORBA specification.

1.3.5 Portable Object Adapter

The Portable Object Adapter (POA) specification, introduced in CORBA 2.2, replaces the Basic Object Adapter (BOA) defined in earlier versions of the CORBA specification. An Object Adapter associates servants with CORBA object references, demultiplexes incoming requests, and dispatches these requests to servants.

TAO fully implements the POA specification, including support for multiple nested POAs per ORB, applying policies to POAs at creation time, and portability of server implementation code. In addition, TAO's POA, by default, optimizes request demultiplexing and operation dispatching, using active demultiplexing and perfect hashing. These optimizations improve the predictability of CORBA applications by ensuring constant O(1) time operation dispatches, regardless of the number of active client connections, the number of activated servants, and the number of operations defined in an IDL interface. Alternative lookup strategies are configurable, such as linear search, binary search, and dynamic hashing.

The `RTPortableServer` extension to the POA interface, which is part of the real-time CORBA specification, adds operations that permit the application to associate priorities with object activations and to define thread pools for operation dispatching. The RT CORBA specification also defines a system of portable priorities that can be mapped to native operating system priorities. Thus, RT CORBA provides a single, “global” priority model that simplifies system design and improves code portability and extensibility. RT CORBA provides a standard mechanism for servers to allocate, partition, and manage thread resources and control dispatching of requests onto threads according to priority, thereby helping to ensure end-to-end predictability. In addition, RT CORBA gives the developer control over the allocation and selection of communication resources via explicit binding, protocol configuration, and protocol selection. See Chapter 8 for more information on TAO’s implementation of real-time CORBA.

1.3.6 ORB Core

A client ORB communicates with a server ORB to deliver client request messages and return responses, if any, to the client. On the server side, the ORB Core delivers the request to the appropriate Object Adapter and returns a reply message to the client-side ORB. ORBs also actively manage the transport-level connections that are used to transmit these request and reply messages.

The OMG defines the General Inter-ORB Protocol (GIOP) for enabling interoperable communications among disparate ORB implementations. TAO's ORB Core supports GIOP version 1.2 (and its realization atop the TCP transport protocol, known as the Internet Inter-ORB Protocol or IIOP). In addition, TAO's pluggable protocols framework allows GIOP messaging to operate over a wide range of transport protocols, including user-defined transports. In addition to IIOP, TAO provides alternate pluggable transport protocols, such as: UIOP, for inter-ORB communications over local IPC (or UNIX domain sockets); SHMIOP, for inter-ORB communications via shared memory; DIOP, for limited but highly-efficient inter-ORB communications using UDP; SSLIOP, for secure inter-ORB communications using Secure Sockets Layer (SSL); MIOP, for inter-ORB communications over unreliable multicast protocol; SCIOP, for inter-ORB communications over the Stream Control Transmission Protocol (SCTP); and HTIOP, which tunnels inter-ORB communications over Hypertext Transfer Protocol (HTTP). Each pluggable transport protocol must provide a *protocol factory* that is loaded and configured at run time. See Chapter 14 for more information on using TAO's pluggable protocols.

In addition to the pluggable protocols framework, TAO's ORB Core employs various strategies to configure certain aspects of the ORB's behavior for both the client and server sides. For example, on the client side, strategies are used to affect concurrency, to determine if multiple requests are allowed to share a communication channel, to control connection management by the ORB core, and various other behaviors. On the server side, strategies are used to control how the object adapter demultiplexes requests and to control concurrency. To obtain these strategies, the ORB core uses *strategy factories* that are loaded and configured at run time via the ACE Service Configurator framework. See Part 3, "Run-time Configuration of TAO," for more information on configuring the ORB's strategy factories.

1.3.7 ACE

TAO is implemented atop ACE, which is infrastructure middleware that implements the core concurrency and distribution patterns for communication software. ACE provides reusable C++ wrapper façades and framework components that support the QoS requirements of high-performance, real-time applications. ACE is a highly portable, multiplatform framework that spans both real-time and general-purpose operating systems.

1.4 CORBA Compliance

It is not necessary to use TAO as a real-time ORB. In fact, TAO provides out-of-the-box standard CORBA conformance. TAO was designed to be compliant with OMG CORBA specifications, as summarized below and as detailed in Appendix I.

- TAO is mainly compliant with the OMG CORBA 3.1 specifications (OMG Documents formal/08-01-04, formal/08-01-06, and formal/08-01-08).
- TAO implements the *CORBA for embedded* specification (OMG Document formal/08-11-06).
- TAO implements the real-time CORBA 1.2 specification (OMG Document formal/05-01-04).
- TAO is compliant with the CORBA C++ Language Mapping specification, version 1.2 (OMG Document formal/08-01-09).
- TAO complies with the Internet Inter-ORB Protocol (IIOP) specification, version 1.2, including support for bi-directional communications over a single connection. Therefore, TAO can interoperate seamlessly with other ORBs that use the standard IIOP (including ORBs that use IIOP versions 1.0 and 1.1). TAO does not technically support IIOP 1.3 or 1.4, but does support many of the component-related and IPv6 features of these versions through its IIOP 1.2 implementation.
- TAO supports the static invocation interface (SII) and static skeleton interface (SSI), as well as the dynamic invocation interface (DII) and dynamic skeleton interface (DSI) models.

- TAO fully implements the Portable Object Adapter (POA) specification, including advanced POA features, such as servant managers and adapter activators.
- TAO provides many of the standard CORBA services, as follows:
    - **Audio/Video Streaming Service**—implements the Control and Management of Audio/Video Streams specification.
    - **Concurrency Control Service**—allows objects in a distributed system to acquire and release locks.
    - **Event Service**—decouples communication between objects by providing an asynchronous *supplier/consumer* style of event propagation among objects.
    - **Interface Repository**—maintains a repository of information about IDL interfaces and types and provides lookup capabilities to clients.
    - **Life Cycle Service**—provides a standard means to locate, move, copy, and remove objects.
    - **Load Balancing Service**—provides random, round-robin, and least-loaded load balancing strategies to forward requests to registered replica services.
    - **Logging Service**—provides event-based logging and log-record query capabilities.
    - **Naming Service**—maps names to object references, organized in a hierarchy.
    - **Notification Service**—extends the CORBA Event Service with the addition of features such as event filtering and structured events.
    - **Property Service**—allows applications to associate properties with objects dynamically.
    - **Security Service**—provides a comprehensive treatment of security as it relates to distributed object systems and applications.
    - **Time Service**—provides globally-synchronized time to distributed objects.
    - **Trading Service**—maps properties to object references and provides constraint-based object lookup capabilities to clients.

In addition, TAO provides the following additional service that demonstrates TAO's capabilities in various real-time environments:

- **TAO Real-Time Event Service**—augments the standard CORBA Event Service model by providing source- and type-based event filtering, event correlation, priority-based dispatching, and event channel federation.

See Part 4, “TAO Services,” for more information on the various services implemented by TAO.

1.5 High Performance and Real-Time Support

Historically, CORBA has supported only “best-effort” quality of service to applications. Developers with stringent QoS or performance requirements could not rely on CORBA to provide the level of performance or predictability they needed.

TAO was designed from the beginning with support for real-time and other demanding applications in mind. Because this kind of support was lacking from the CORBA specifications, TAO supplied extensions to the CORBA specifications to support applications that required higher performance, real-time determinism, and end-to-end priority propagation.

Because the CORBA specification now supports more demanding applications, ORB implementations can now provide much greater QoS and performance guarantees without sacrificing CORBA compliance. TAO is in the forefront of support for these latest aspects of the CORBA specification as follows:

- TAO implements the CORBA policy framework as defined by the CORBA Messaging specification and supports the creation of policies for controlling request/reply timeouts, synchronization scope for oneway requests, support for bi-directional GIOP communications, and other aspects of inter-ORB communications.
- TAO implements the real-time CORBA specification, including the real-time ORB and real-time PortableServer features, such as portable priorities, client-propagated and server-declared priority models, RT CORBA threadpools, and priority-banded connections. In addition, TAO provides an implementation of RT CORBA dynamic scheduling.

TAO also provides the following extensions to the CORBA specifications to support specific application needs:

- TAO’s ORB Core provides an efficient and predictable communication infrastructure for high-performance and real-time applications. It provides a range of client and server concurrency models.
- TAO’s ORB Core supports nested upcalls with several of its concurrency models.
- TAO’s implementation of RT CORBA thread pools with lanes provides a *reactor-per-lane* configuration that requires no context switches throughout the life of an upcall, thereby greatly decreasing the likelihood of priority inversions.
- TAO’s ORB Core allows custom transport protocols to be plugged into the ORB without affecting standard CORBA application programming interfaces.
- Some custom transport protocols supported by TAO improve request transmission performance relative to the standard IIOP protocol under certain conditions.
- TAO’s implementation of the POA and generated skeletons are designed using patterns that provide an extensible and highly optimized set of request demultiplexing and operation dispatching strategies, such as perfect hashing and active demultiplexing. These strategies allow for constant-time lookup of nested POAs and servants, based on object keys, and operation names contained in CORBA requests.

1.6 Relationship Between ACE and TAO

Many components in TAO, such as its ORB Core, POA, and generated stubs and skeletons, are based on patterns and components provided by the ACE framework. Key patterns used in TAO include the Acceptor, Connector, Reactor, Active Object, Half-Sync/Half-Async, Service Configurator, Thread-Specific Storage, Strategy, Proxy, Adapter, Bridge, and Abstract Factory.

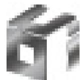
To improve portability, TAO uses ACE’s high-performance, small-footprint operating system adaptation layer for all operating system access, rather than

invoking non-portable system calls directly. Because ACE supports numerous operating systems, porting TAO to a new platform is simplified considerably.

For more on these patterns and concepts, see the following references, found in the References section near the end of this guide.

- *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (GoF).
- *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* (POSA2), by Doug Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann.
- *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns* (C++NPv1), by Doug Schmidt and Steve Huston.
- *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks* (C++NPv2), by Doug Schmidt and Steve Huston.
- *The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming*, by Steve Huston, James CE Johnson, and Umar Syyid.

Also, for more information on ACE, visit the ACE home page via `<http://www.theaceorb.com/references/>`.



CHAPTER 2

Building ACE and TAO

2.1 Introduction

This chapter walks you through the steps to build ACE and TAO for two common system/compiler combinations: Linux with GNU C++ and Windows Visual C++. Although TAO supports a variety of operating systems and platforms, in this chapter we will walk through a build using these common system/compilers using the default build options. This will allow you to quickly build the necessary libraries and executables so that you try the examples provided in this book.

Note *Appendix A discusses the full details on configuring ACE/TAO builds and some of the options available via each mechanism.The detailed instructions for building ACE and TAO for various operating systems and compilers are provided in other appendices.*

2.2 Where to Get ACE and TAO

The source code for ACE and TAO that this book is based on can be downloaded from <http://download.ociweb.com/TAO-2.2a/>. There you will find both the zip file ACE+TAO-2.2a.zip and compressed tar file ACE+TAO-2.2a.tar.gz. These archives contain the GNU makefiles and Visual Studio projects files that we will use to build ACE and TAO in this chapter.

Also in these archives you will find the source to the examples in this book. The examples using core TAO are in directory $TAO_ROOT/DevGuideExamples and examples using TAO's ORB services are in directory $TAO_ROOT/orbsvcs/DevGuideExamples.

Note *For other OCI releases of ACE and TAO, and additional information about OCI's support of ACE and TAO, visit OCI's TAO web site:* <http://www.theaceorb.com>.

Note *The link to the DOC group ACE and TAO source code repository, which OCI's version of ACE and TAO is based on, can be found at* <http://www.theaceorb.com/references/>.

2.3 System Requirements

You should have at least 512 MB of memory, but more memory will improve build times. You'll want to have several GB of free space on your drive to hold the build results. For both Windows and Linux, TAO can be built on 32 and 64 bit Intel and AMD processors.

For building under Windows, you will need Visual C++ 7.1 or later (TAO 2.2a has been tested using up to Visual C++ 10).

For building using Linux, you will need GNU C++ 3.3.x or later, although version 4.2 is the recommended version. You will also need to have the GNU Make program installed.

For either compiler, you will want to have perl installed to run example and test scripts.

Further details about system requirements can be found in A.1.

2.4 Steps to Build ACE and TAO

2.4.1 Building on Windows using Visual Studio

Extract the ACE+TAO archive to a path that does not contain spaces. For the discussion that follows we will assume the zip file was extracted to C:\, which places the files under `C:\ACE_wrappers`.

In `C:\ACE_wrappers\ace`, create the file `config.h` and add the following line to it:

```
#include "ace/config-win32.h"
```

Now open the solution file corresponding to the version of Visual Studio you are using as given in Table 2-1.

Table 2-1 Visual Studio Solution Files

Visual Studio Version	Solution File
7.1	TAO_ACE.sln
8	TAO_ACE_vc8.sln
9	TAO_ACE_vc9.sln
10	TAO_ACE_vc10.sln

In Visual Studio build the `Naming_Service` project. This will build all dependent projects as needed.

Note *Building the* `Naming_Service` *project ensures that the essential libraries and executables that we need will be built. However, to save compile time and storage space, the tests and examples are not built.*

In Chapter 3 you will learn how to set up your environment to run the examples in this book.

For information on generating custom solution files and building with various compiler options, see Appendix D.

2.4.2 Building on Linux using GNU C++

For the discussion that follows we will assume the distribution archive was extracted to /opt/ACE_wrappers.

In /opt/ACE_wrappers/ace, create the file config.h and add the following line to it:

```
#include "ace/config-linux.h"
```

The ACE and TAO GNU makefiles requires the presence of certain environment variables to work properly. These are shown in the following table.

Table 2-2 TAO Environment Variables

Environment Variable	Definition
ACE_ROOT	The root directory of the ACE files
TAO_ROOT	The root directory of the TAO files.
PATH	List of directories to search for executables that should include $ACE_ROOT/bin.
LD_LIBRARY_PATH	List of directories for the executable to find libraries that should include $ACE_ROOT/lib.

For our case, if using the bash shell we can set them as follows:

```
export ACE_ROOT=/opt/ACE_wrappers
export TAO_ROOT=$ACE_ROOT/TAO
export PATH=$PATH:$ACE_ROOT/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ACE_ROOT/lib
```

Now we need to specify certain GNU Make variables needed for building on Linux. This is done by creating the file $ACE_ROOT/include/makeinclude/platform_macros.GNU and adding the following:

```
debug=1
optimize=0
include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU
```

This will give us a build that we can use for debugging code under Linux that uses TAO.

You should now be able to start a build as follows:

```
cd $TAO_ROOT
make
```

For information on generating custom makefiles and building with various compiler options, see Appendix C.

CHAPTER 3

Getting Started

3.1 Introduction

This chapter guides you through the process of building and running a simple client/server application using TAO. You should already have TAO installed (from binaries supplied by OCI or another vendor) or built (from source code) on your system. If not, see Chapter 2. If you are new to CORBA, you may find it helpful to read Chapter 3 of *Advanced CORBA Programming with C++* before proceeding.

TAO uses a tool that makes using TAO essentially identical on all platforms. MakeProjectCreator (MPC) is capable of generating build files for each platform from simple text data files. So, whether you are getting started with TAO on Linux, Windows, Solaris, or one of the other many platforms supported by TAO, the steps are essentially the same.

3.1.1 Road Map

In this chapter, you will learn how to:

- Set up your environment for using TAO (see 3.2).

- Develop a simple server and client using TAO (see 3.3).

Full source code for the example presented in this chapter can be found in the TAO 2.2a source code distribution in the directory `$TAO_ROOT/DevGuideExamples/GettingStarted`.

3.2 Setting Up Your Environment

Certain environment variables are required during the compilation and run-time phases of TAO applications. These environment variables are presented here. If you built TAO yourself, these variables are probably already set and you may skip this section. The environment variables are shown first using UNIX syntax, with Windows syntax shown in parentheses.

- `ACE_ROOT`

  The base directory where you installed ACE and TAO, such as `/usr/local/ACE_wrappers` (`C:\ACE_wrappers`).

- `TAO_ROOT`

  The base path for all TAO-related code, normally `$ACE_ROOT/TAO` (`%ACE_ROOT%\TAO`).

- `PATH`

  Scripts and executables for TAO will be installed in `$ACE_ROOT/bin` (`%ACE_ROOT%\bin`). You should add this location to your `PATH` environment variable.

- Library path

  All required libraries will be installed in `$ACE_ROOT/lib` (`%ACE_ROOT%\lib`). You should add this location to your `LD_LIBRARY_PATH` environment variable or its equivalent. (On Windows, add this directory to your `PATH` so DLLs can be located at run time.)

3.3 A Simple Example

In this section, we guide you step-by-step through the creation of a simple TAO example. We create our IDL files, implement our servants, create client and server applications, generate build files, build, and run the application.

Our example consists of a server called `MessengerServer` that implements a simple `Messenger` interface, plus a client called `MessengerClient` that accesses and uses a `Messenger` CORBA object that the `MessengerServer` provides. Imagine that a full implementation of the `MessengerServer` might send e-mail, access a pager, or even make a phone call using voice synthesizer technology. To keep our example simple, we just write the client's message to standard output. In later chapters, we will expand on this example to illustrate various TAO and CORBA features.

Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/DevGuideExamples/GettingStarted`.

3.3.1 Create a Workspace

First, create a working directory for our example. We will place all of our code in a single directory for this example, but in larger projects you may use a different directory structure. For example, you may wish to separate code for libraries, servers, and clients into separate subdirectories.

```
mkdir Messenger
cd Messenger
```

3.3.2 Messenger Interface Definition Language (IDL) File

Create a new file called `Messenger.idl` to contain the interface definition for our simple Messenger. This interface simply defines an operation that we will use to send text messages between a client and server. A reply may be returned in the last parameter, and the return value indicates whether the message was accepted.

```
interface Messenger
{
  boolean send_message(in string user_name,
                       in string subject,
                       inout string message);
};
```

3.3.2.1 Run the IDL Compiler

The IDL compiler (`tao_idl`) generates stub and skeleton code from the IDL interface definitions contained in `Messenger.idl`. Details about using the IDL compiler are found in Chapter 4. We use the `-GI` option to cause `tao_idl` to generate *starter* implementation (servant) files. We then modify the generated starter code for our actual implementation. Using the `-GI` option to automatically generate starter code is a convenient way to make sure our implementation class function signatures are correct.

```
tao_idl -Sa -St -GI Messenger.idl
```

After running the IDL compiler as shown, our starter implementation class for the Messenger interface will be in files named `MessengerI.*`. Client-side stubs will be in files named `MessengerC.*` and server-side skeletons will be in files named `MessengerS.*`. Other files may also be generated, but they do not concern us for this simple example.

3.3.3 Create the Messenger_i Implementation Class

Normally, you will want to rename the generated starter implementation files `MessengerI.h` and `MessengerI.cpp` to `Messenger_i.h` and `Messenger_i.cpp`. That way, you will not inadvertently overwrite existing files if you run the IDL compiler with the `-GI` option again.

UNIX

```
mv MessengerI.h Messenger_i.h
mv MessengerI.cpp Messenger_i.cpp
```

Windows

```
ren MessengerI.h Messenger_i.h
ren MessengerI.cpp Messenger_i.cpp
```

3.3.3.1 C++ Header for the Messenger_i Class

Our `Messenger_i` implementation class inherits from the `POA_Messenger` skeleton class found in `MessengerS.h`. We have removed some comments and an unneeded constructor and destructor from the generated starter implementation files.

```
#include "MessengerS.h"
```

```
class Messenger_i : public virtual POA_Messenger
{
public:
  virtual CORBA::Boolean send_message (
    const char* user_name,
    const char* subject,
    char*& message);
};
```

3.3.3.2 C++ Implementation of the Messenger_i Class

The file `Messenger_i.cpp` already contains much of the code we need for implementing the `Messenger_i` class. Here is the file with our additions and changes shown in **bold** text. Once again, we have removed the unneeded constructor, destructor, and some generated comments.

```
#include "Messenger_i.h" // renamed from MessengerI.h
#include <iostream>

CORBA::Boolean Messenger_i::send_message (
    const char* user_name,
    const char* subject,
    char*& message)
{
    std::cout << "Message from: " << user_name << std::endl;
    std::cout << "Subject:      " << subject << std::endl;
    std::cout << "Message:      " << message << std::endl;
    CORBA::string_free(message);
    message = CORBA::string_dup("Thanks for the message.");
    return true;
}
```

3.3.4 C++ Implementation of the MessengerServer

We next create a `MessengerServer` to give our Messenger object a place to live. In `main()`, we create an instance of our `Messenger_i` implementation class, activate it in the `RootPOA`, and wait for requests from clients.

Create `MessengerServer.cpp` with the following contents:

```
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main(int argc, char* argv[])
{
```

```
    try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

      //Get a reference to the RootPOA.
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

      // Activate the POAManager.
      PortableServer::POAManager_var mgr = poa->the_POAManager();
      mgr->activate();

      // Create a servant.
      PortableServer::Servant_var<Messenger_i> servant = new Messenger_i();

      // Register the servant with the RootPOA, obtain its object
      // reference, stringify it, and write it to a file.
      PortableServer::ObjectId_var oid = poa->activate_object(servant.in());
      obj = poa->id_to_reference(oid.in());
      CORBA::String_var str = orb->object_to_string(obj.in());
      ofstream iorFile("Messenger.ior");
      iorFile << str.in() << std::endl;
      iorFile.close();
      std::cout << "IOR written to file Messenger.ior" << std::endl;

      // Accept requests from clients.
      orb->run();
      orb->destroy();

      return 0;
    }
    catch (CORBA::Exception& ex) {
      std::cerr << "MessengerServer CORBA exception: " << ex << std::endl;
    }
    return 1;
  }
```

3.3.5 C++ Implementation of the MessengerClient

We complete our example by creating a `MessengerClient`, which obtains an object reference to the Messenger object and sends it a message via its `send_message()` operation.

Create `MessengerClient.cpp` with the following contents:

```
  #include "MessengerC.h"
  #include <iostream>
```

```
int main(int argc, char* argv[])
{
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Read and destringify the Messenger object's IOR.
    CORBA::Object_var obj = orb->string_to_object("file://Messenger.ior");
    if( CORBA::is_nil(obj.in())) {
      std::cerr << "Could not get Messenger IOR." << std::endl;
      return 1;
    }

    // Narrow the IOR to a Messenger object reference.
    Messenger_var messenger = Messenger::_narrow(obj.in());
    if( CORBA::is_nil(messenger.in())) {
      std::cerr << "IOR was not a Messenger object reference." << std::endl;
      return 1;
    }

    // Send a message the the Messenger object.
    CORBA::String_var msg = CORBA::string_dup("Hello!");
    messenger->send_message("TAO User", "Test", msg.inout());

    // Print the Messenger's reply.
    std::cout << "Reply: " << msg.in() << std::endl;

    return 0;
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "MessengerClient CORBA exception: " << ex << std::endl;
  }
  return 1;
}
```

3.3.6 Create Build Files for the Example

Originally, creating the necessary files for building TAO projects involved manually creating separate build tool files for each platform. For example, to build the above example on UNIX using GNU Make and on Windows using Visual C++ required building and maintaining both Makefiles and Visual C++ project/solution files. In such cross-platform environments, creating and maintaining different build files for different build tools was tedious and error-prone. This process has been greatly simplified with the introduction of a tool called *MakeProjectCreator* (MPC). With MPC, multiple build environments can now be supported very simply. All we have to do is create a

simple mpc file with the information that is unique to our project. We then run MPC to generate build files for use with GNU Make (`gmake`), Microsoft Visual Studio (V7.1, V8, V9, V10), Microsoft `nmake`, Borland make and others. For more information on MPC see `<http://www.ociweb.com/products/MPC>`.

To support builds of our Messenger example, we create a file called `GettingStarted.mpc` with the following contents:

```
project(*idl): taoidldefaults {
  IDL_Files {
    Messenger.idl
  }
  custom_only = 1
}

project(*Server): taoserver {
  exename = MessengerServer
  after += *idl
  Source_Files {
    Messenger_i.cpp
    MessengerServer.cpp
  }
  Source_Files {
    MessengerC.cpp
    MessengerS.cpp
  }
  IDL_Files {
  }
}

project(*Client): taoclient {
  exename = MessengerClient
  after += *idl
  Source_Files {
    MessengerClient.cpp
  }
  Source_Files {
    MessengerC.cpp
  }
  IDL_Files {
  }
}
```

The `GettingStarted.mpc` defines three projects: one for the IDL processing, one for the server, and one for the client. This mpc file relies on various settings (such as include paths and link libraries) inherited from *base*

projects. The IDL processing project inherits from `taoidldefaults` which supplies the necessary defaults to process the IDL file for this example. This is broken into a separate project as it is required by both the client and server. Our server project inherits from `taoserver` which provides all the necessary project attributes to build a TAO server executable. In a similar manner, the client project inherits from `taoclient` which enables it to be a pure TAO client (with no server-side CORBA functionality). The projects will be named `GettingStarted_Idl`, `GettingStarted_Client` and `GettingStarted_Server`, because we used the '`*`' wild card character in our project name declarations. The output files will be named `MessengerClient` and `MessengerServer`, because these are the names of the source files in each project that contain `main()`. To prevent MPC from automatically detects the existence of our IDL files and implicitly adding these to our source files, we keep the `IDL_Files` group empty in the client and server projects. In our client project we explicitly add `MessengerC.cpp` to the list of source files to prevent MPC from implicitly adding `MessengerS.cpp`, which we do not want to build into our client.

Note *To use MPC, you must have Perl version 5.6.1 or greater.*

The next step depends upon your development environment:

- **UNIX with GNU Make**

  On UNIX or UNIX-like systems, run `mwc.pl` in the project directory to generate GNU makefiles for use with the ACE+TAO make system:

  ```
  mwc.pl -type gnuace
  ```

  The above command will generate the following files for use with GNU Make:

  - `GNUmakefile`
  - `GNUmakefile.GettingStarted_Idl`
  - `GNUmakefile.GettingStarted_Client`
  - `GNUmakefile.GettingStarted_Server`

- **Windows with Visual Studio (2003 and later)**

On Windows, using Visual Studio, run `mwc.pl` in the project directory to generate solution and project files:

```
mwc.pl -type vc10
```

Replace `vc10` with `vc71, vc8, or vc9 for` targeting VC++ 7.1, VC++ 8, or VC++ 9 respectively.

The above command will generate the following files: `GettingStarted.sln`, `GettingStarted_Idl.vcproj`, `GettingStarted_Client.vcproj`, and `GettingStarted_Server.vcproj` for use with Visual Studio.

Visual C++ 7.0 and older are not recommended for use with ACE and TAO.

3.3.7 Build the MessengerServer and MessengerClient

Once the build files are generated, you can build the test applications.

Using GNU Make:

```
gmake (or make)
```

Using Visual Studio:

```
devenv GettingStarted.sln /build debug
```

Using Visual C++ Express Editions:

```
msbuild /p:Configuration=Debug GettingStarted.sln
```

3.3.8 Running the Application

You are now ready to run the `MessengerServer` and `MessengerClient`. The server must be running before the client is started.

Run the `MessengerServer` in one terminal window with the following command:

```
./MessengerServer
```

Wait for the message "`IOR written to file Messenger.ior`", then run the `MessengerClient` from a different terminal window in the same directory with the following command:

```
./MessengerClient
```

You should see the following messages from the `MessengerServer`:

```
Message from: TAO User
Subject:      Test
Message:      Hello!
```

In the `MessengerClient`'s terminal, you should see:

```
Reply: Thanks for the message.
```

indicating that the client has received a reply from the server. The client then exits and your normal command prompt reappears.

Note that the `MessengerServer` will still be running, waiting for more client requests. You can run the client again if you like. To kill the `MessengerServer`, just type `Ctrl-C` in its terminal window or use the `kill(1)` command to terminate it.

3.4 Summary

In this chapter, you have seen how to develop a simple server and client using TAO. Topics covered included: how to set up your environment for building applications that use TAO; how to set up a working directory for a simple example and the files to create therein; creating and using a simple mpc file for building the example; and running the example.

You are now ready to explore other chapters of this guide that expand on this simple example to illustrate various features of TAO and various services that can be used by TAO applications. *Have fun!*

Part 2

Features of TAO

CHAPTER 4

TAO IDL Compiler

4.1 Introduction

To use IDL interfaces with the static invocation approach, you must generate skeleton and stub C++ code so requests can traverse from a client to a servant. TAO includes an IDL compiler, `tao_idl`, that generates C++ skeletons and stubs from your IDL file.

Note *The generated code is only usable by TAO. The output from IDL-to-C++ compilers cannot be interchanged among CORBA implementations. However, the code generated by TAO's IDL compiler is platform-independent, making it possible to use TAO in cross-compilation environments*

TAO's IDL compiler maps IDL files to C++ according to the CORBA C++ mapping specification (OMG Document formal/08-01-09). The basic C++ mapping uses C++ exceptions to report system and user exceptions. An alternate mapping for environments that do not use native C++ exceptions is defined in the specification but is no longer supported by the TAO IDL compiler.

The IDL compiler is modularized into a top-level executable, plus libraries for the front- and back-ends. This modular design allows the front-end lexing and parsing engine to be reused and different back-ends to be “plugged in” to produce different outputs (e.g., to populate the Interface Repository).

4.2 Executables

UNIX and UNIX-like Systems

The IDL compiler executable is `$TAO_ROOT/TAO_IDL/tao_idl`, with a symbolic link in `$ACE_ROOT/bin`.

Windows Systems

The IDL compiler executable is `%ACE_ROOT%\bin\tao_idl.exe`.

General Usage

The general usage of the TAO IDL compiler is as follows:

```
tao_idl <options> IDL-file(s)
```

IDL file names must be listed after the options. The options will apply to all of the IDL files. For example:

```
tao_idl -GI hello.idl Messenger.idl
```

4.3 Output Files Generated

By default, six files are generated for every IDL file the `tao_idl` compiler processes. Three of these files provide the stub code used by the client, and three files provide the skeleton code used by the server. The generation of these files ensures that the generated code is portable and optimized for a wide variety of C++ compilers. However, your client and server applications only need to include two header files directly.

For an IDL file named `Messenger.idl`, running the command

```
tao_idl Messenger.idl
```

generates the following files (we show how to customize these names later):

Table 4-1 C++ Files Generated

File Name	Contents
MessengerC.h	Stub class definitions.
MessengerC.inl	Inline stub member function definitions.
MessengerC.cpp	Stub member function definitions.
MessengerS.h	Skeleton class definitions.
MessengerS.inl	Inline skeleton member function definitions.
MessengerS.cpp	Skeleton member function definitions.

4.3.1 Tips for Working with the Output Files

- The client includes `MessengerC.h` and links to `MessengerC.o`. The server includes `MessengerS.h` and links to both `MessengerC.o` and `MessengerS.o`.
- The stubs and skeletons are decoupled. The client implementation need not include the `*S.h` file or link with `*S.o`. However, the server requires knowledge of the stub classes and IDL types found in the `*C.h` files. Therefore, the generated `*S.h` files that you include in your server code include the corresponding `*C.h` files. In addition, your server implementation needs to link with the object code produced from both the `*C.cpp` and `*S.cpp` files.
- The compiler interprets the functions in the `*.inl` files as inline code only if the build defines the preprocessor macro `__ACE_INLINE__`. This makes it possible to build applications with inlining disabled (e.g., to facilitate debugging) or with inlining enabled (e.g., to improve performance). You can find the definition of this preprocessor option by looking for the `__ACE_INLINE__` macro in `$ACE_ROOT/ace/config.h`.

4.4 Using TAO IDL Compiler Options

We discuss command line options available with the IDL compiler in 4.5 through 4.14. To see a complete list of the IDL compiler's options, enter the following:

```
tao_idl -u
```

The file `$TAO_ROOT/docs/compiler.html` also contains information on IDL compiler options.

You can specify IDL compiler options directly on the command line, or you can specify them in an MPC file by adding your IDL compiler options to the `idlflags` keyword. For example:

```
idlflags += -I$(TAO_ROOT)/orbsvcs -GI
```

4.5 Preprocessing Options

As required by the CORBA specification, IDL files can contain directives defined by the C++ preprocessor. This means, for example, that one IDL file can include another IDL file by using the `#include` directive. Likewise, conditional compilation can be done by using the `#ifdef` or `#if defined` directives. TAO's IDL compiler does not include a preprocessor, rather it invokes an external preprocessor. By default, the preprocessor is specified by the `TAO_IDL_PREPROCESSOR` environment variable. This variable is defined when `tao_idl` is built. The C++ compiler's preprocessor is used if this variable is not defined. See 4.5.1 for information on how to override the default and specify the preprocessor during IDL compilation.

Some common preprocessor options can be passed to the IDL compiler, which will then pass them through to the preprocessor. The IDL compiler also

supports the passing of any option to the preprocessor via its `-Wp` option. Table 4-2 provides details of the options related to preprocessing.

Table 4-2 Preprocessing Options

Option	Definition	Example
`-E`	Run the preprocessor on the IDL file, but do not generate any C++ code.	`tao_idl -E Messenger.idl`
`-D` *`macro-def`*	Defines a macro.	`tao_idl -D CORBA_IMPL=tao Messenger.idl`
`-U` *`macro-def`*	Undefines a macro.	`tao_idl -U unix Messenger.idl`
`-I` *`include-path`*	Add `include-path` to the list of paths searched for include files.	`tao_idl -I /idl/exceptions -I $TAO_ROOT/orbsvcs/orbsvcs Messenger.idl`
`-A` *`assertion`*	Make an assertion.	`-A system(gnu)`
`-Yp,`*`preproc-loc`*	Tells the TAO IDL compiler to use a specific preprocessor.	`tao_idl -Yp,/usr/bin/cpp Messenger.idl`
`-Wp,`*`arg1`*`,`*`arg2`*`,...`	Passes arguments to the preprocessor.	`-Wp,-undef`

The `-D`, `-U`, `-I`, and `-A` options are all passed directly through to the preprocessor. If the preprocessor you are using is the same as your C++ preprocessor (the likely case), then you should see your C++ compiler documentation for details about these options. The `-Wp` option will only pass the text between the commas to the preprocessor (stripping off the leading “`-Wp`” and all commas).

In addition to accepting preprocessor directives such as `#define`, `#include`, and `#if`, TAO’s IDL compiler recognizes and handles the following preprocessor directives:

Table 4-3 Additional Preprocessor Directives

Directive	Definition	Example
`#file "`*`file name`*`"`	Identifies the name of the file being preprocessed. These directives are inserted by some preprocessors. The IDL compiler simply ignores them.	`#file "Messenger.idl"`

Table 4-3 Additional Preprocessor Directives

Directive	Definition	Example
`#pragma ident "`*`id string`*`"`	Provides an identification string for source code control or other purposes. The IDL compiler simply copies the entire line, unmodified, to the top of each output file.	`#pragma ident "$Id$"`
`#pragma prefix "`*`prefix string`*`"`	Provides a prefix string used in generating RepositoryIds. The `typeprefix` keyword can also be used to set the prefix (as described in the CORBA specification).	`#pragma prefix "omg.org"`
`#pragma version` *`name major.minor`*	Provides major and minor version numbers used in generating RepositoryIds.	`#pragma version Messenger 1.1`
`#pragma ID` *`name id`*	Assigns a user-specified repository ID to the IDL type with the given name. The `typeid` keyword can also be used to set the prefix (as described in the CORBA specification).	`#pragma ID Messenger "IDL:Messenger:1.1"`

4.5.1 Environment Variables Affecting Preprocessing

Environment variables that impact the preprocessing stage of the IDL compiler are described in Table 4-4.

Table 4-4 Preprocessing Environment Variables

Environment Variable	Description	Default Behavior
`TAO_IDL_PREPROCESSOR`	Specifies the command to access the C++ preprocessor.	Uses the preprocessor used to build the IDL compiler itself.
`TAO_IDL_PREPROCESSOR_ARGS`	Provides additional options for the IDL compiler to pass to the preprocessor.	Always passes `-DIDL` and `-I`, in addition to any specified options.
`INCLUDE`	If set, append its contents to the include path.	Not passed.
`TAO_ROOT`	If set, passes `-I$(TAO_ROOT)/tao`.	Not passed.

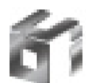


Table 4-4 Preprocessing Environment Variables

Environment Variable	Description	Default Behavior
`ACE_ROOT`	If set, passes `-I$(ACE_ROOT)/TAO/tao`.	Not passed.

Either `ACE_ROOT` or `TAO_ROOT` must be defined for `tao_idl` to find the file `orb.idl` when it is included by another IDL file (e.g., `#include <orb.idl>`). If neither `ACE_ROOT` nor `TAO_ROOT` is defined, the IDL compiler will display a warning message.

4.6 Output File Options

The TAO IDL compiler has an option that allows you to specify a target directory for the output files. In addition, there are options to control the file names of the C++ source code generated. These file names always start with the base name of the IDL file being processed. You can change the suffix of the file names that are generated by using the options listed in Table 4-5.

Table 4-5 Output File Options

Option	Description	Default
`-o` *`output-directory`*	Directory in which to place the generated stub and skeleton files.	Current directory
`-oS` *`output-directory`*	Directory in which to place the generated *S.* files.	Current directory or value of -o option
`-oA` *`output-directory`*	Directory in which to place the generated *A.* files.	Current directory or value of -o option
`-iC` *`path`*	Overrides the default include path for *C.h files included in generated *A.h files	`$TAO_ROOT/tao`
`-hc` *`filename-ending`*	Stub header suffix.	`C.h`
`-ci` *`filename-ending`*	Stub inline functions suffix.	`C.inl`
`-cs` *`filename-ending`*	Stub non-inline functions suffix.	`C.cpp`
`-hs` *`filename-ending`*	Skeleton header file name suffix.	`S.h`
`-si` *`filename-ending`*	Skeleton inline functions file name suffix.	`S.inl`
`-ss` *`filename-ending`*	Skeleton non-inline functions file name suffix.	`S.cpp`

Table 4-5 Output File Options

Option	Description	Default
`-hT` *`filename-ending`*	Skeleton template header file name suffix.	`S_T.h`
`-sT` *`filename-ending`*	Skeleton template non-inline functions file name suffix.	`S_T.cpp`
`-hI` *`filename-ending`*	Starter implementation header file name suffix (use with `-GI`).	`I.h`
`-sI` *`filename-ending`*	Starter implementation file name suffix (use with `-GI`).	`I.cpp`

As an example of these options, suppose you are migrating from a different CORBA implementation to TAO. When processing an IDL file named `Messenger.idl`, this implementation generates files named `Messenger.hh` and `Messenger.cc` for the stub code, and files named `MessengerS.hh` and `MessengerS.cc` for the skeleton code.

For TAO to emulate these naming conventions, invoke the IDL compiler as:

```
tao_idl -hc .hh -ci .i -cs .cc -hs S.hh -si S.i -ss S.cc Messenger.idl
```

This will produce header and source file names consistent with the other CORBA implementation.

4.7 Starter Implementation Files

To help you start writing implementation code, `tao_idl` optionally generates starter servant implementation files. The starter files contain empty C++ member function definitions that you must fill in with your implementation code. If you are new to CORBA or have a lot of operations to implement, this can be a great time saver. Table 4-6 lists the options related to this feature.

Note *Running the IDL compiler with the starter implementation options overwrites any existing implementation files of the same names. Any modifications will be*

lost unless you rename the starter implementation files after they are generated (recommended)!

Table 4-6 Starter Implementation Generation Options

Option	Description	Default
`-GI`	Generate starter implementation code.	Do not generate starter implementation code.
`-GIh` *`file-ending`*	Sets the starter implementation header file suffix to `file-ending`.	`I.h`
`-GIs` *`file-ending`*	Sets the starter implementation source file suffix to `file-ending`.	`I.cpp`
`-GIb` *`class-prefix`*	Prefix with which to begin the starter implementation class names.	No default prefix.
`-GIe` *`class-prefix`*	Suffix with which to end the starter implementation class names.	`_i`
`-GIc`	Create an empty copy constructor.	No copy constructor defined.
`-GIa`	Create an empty assignment operator.	No assignment operator defined.
`-GId`	Generate debug information (source file name and line number) in the starter implementation.	No debug information generated.

Note *The* `-GIh` *and* `-GIs` *options have the same effect as the* `-hI` *and* `-sI` *options presented in Table 4-5.*

For example, consider the following:

```
// Messenger.idl
interface Messenger {
  boolean send_message(in    string user_name,
                       in    string subject,
                       inout string message );
};
```

Suppose the convention you use includes implementation files that end with `_i.h` and `_i.cpp`. Then invoking the IDL compiler with:

```
tao_idl -GIh _i.h -GIs _i.cpp Messenger.idl
```

creates the `Messenger_i.h` and `Messenger_i.cpp` starter files.

Note *The* `-GI` *option was not needed in this case since using* `-GIh` *and* `-GIs` *implies* `-GI`.

The generated starter implementations of operations and attributes contain no code, only comments of the form "`//Add your implementation here.`" For example:

```
CORBA::Boolean Messenger_i::send_message (
    const char* user_name,
    const char* subject,
    char*& message
  )
  {
    //Add your implementation here
  }
```

To implement the interface, search for these comments and replace them with your own code. However, the IDL compiler does not generate comments for constructors and destructors; do not forget to fill in these functions as well.

4.8 Additional Code Generation Options

In addition to generating starter implementation classes, the TAO IDL compiler provides options for generating reply handler classes for use with the Asynchronous Method Invocation (AMI) callback model, servant and response handler classes for use with Asynchronous Method Handling (AMH), smart proxy factory and default smart proxy classes, optimized TypeCodes, servant tie classes, and explicit template instantiations. Options for using the above features are shown in Table 4-7. For more information on

AMI callbacks, see 6.2. For more information on AMH, see Chapter 7. For more information on smart proxies, see Chapter 11.

Table 4-7 Additional Code Generation Options

Option	Description	Default
`-GC`	Generate AMI callback reply-handler classes.	AMI callback classes not generated.
`-GH`	Generate AMH servant and response-handler classes.	AMH classes not generated.
`-Gsp`	Generate smart proxy factory and default smart proxy classes.	Smart-proxy-related classes not generated.
`-Gt`	Generate optimized TypeCodes.	TypeCodes not optimized.
`-GA`	Generate `any` operators and TypeCodes into a separate `*A.cpp` stub file.	Generate `any` operators and TypeCodes into the`*C.cpp` stub file
`-GT`	Generate the TIE classes, and the *S_T.h and *S_T.cpp files that contain them. Use this option only if your application uses the “tie” class to implement servants.	TIE classes are not generated.
`-Guc`	If an IDL constant is declared at `module` scope, assign its value in the stub’s `.cpp` file rather than inline in the stub’s header file.	A constant with `module` scope is assigned its value inline in the stub’s header file.
`-Gos`	Generates iostream-style insertion operators (`operator<<`) for IDL-defined types. If your IDL-defined types depend on types included via `orb.idl`, TAO must also be built with the `gen_ostream` MPC feature enabled.	No operators are generated.
`-Gse`	Generates an explicit export of any sequence's template base class. This is sometimes required as a workaround for bugs in certain versions of Visual Studio (.Net 2002, .Net 2003, 2005).	Do not generate explicit exports for sequence base classes.
`-Gce`	Limit the code generated to that specified by the CORBA/e (CORBA for embedded) configuration.	Generate the normal CORBA-specified code.
`-Gmc`	Limit the code generated to that specified for the minimum CORBA configuration.	Generate the normal CORBA-specified code.
`-in`	Generate `#includes` within “<>"	Uses `""` by default.
`-ic`	Generate `#includes` within “”	Uses `""` by default.

Table 4-7 Additional Code Generation Options

Option	Description	Default
`-b`	Generates code to allow cloning of arguments on oneway calls. This greatly speeds up some scenarios when using Custom Servant Dispatching (CSD). There is a small cost of increased footprint when using this option.	Arguments and return values are marshalled and demarshalled when using CSD.

4.9 OpenDDS-related Options

OpenDDS uses ACE, TAO, and the TAO IDL compiler as part of its underlying architecture. The TAO IDL compiler has two options related to OpenDDS which are listed in Table 4-8. For normal TAO users, these options can be ignored. For further information related to OpenDDS see `<http://www.opendds.org/>`.

Table 4-8 Additional Code Generation Options

Option	Description
`-Gdcps`	Generate appropriate marshaling and instance key support code for OpenDDS DCPS-enabled types.
`-Gdcpsonly`	Only generate DCPS-related code for OpenDDS (for internal OpenDDS use).

4.10 Operation Lookup Strategy Options

When a server receives a request from a client, the POA needs to find the skeleton function associated with the operation name in the request. This step involves looking up an operation, based on a string contained in the client request. There are many ways to do this; each has its strengths and weaknesses. Table 4-9 shows the operation lookup strategies for which the IDL compiler can generate code. For most cases we recommend that you use

the default strategy, perfect hashing, which is usually optimal in both time and space.

Table 4-9 Operation Lookup Strategies

Option	Type	Lookup time
`-H perfect_hash` (default)	Perfect hashing	Constant. Suitable for hard real-time systems.
`-H linear_search`	Linear search	Proportional to the number of operations. Represents a worst-case strategy for comparison purposes.
`-H binary_search`	Binary search	Proportional to the log of the number of operations. Adding more operations has minimal impact on lookup time.
`-H dynamic_hash`	Dynamic hash	Constant for the average case. Worst case similar to linear search. Inappropriate for hard real-time systems.

To support the perfect hashing operation lookup strategy, the TAO IDL compiler relies on `ace_gperf`, a general purpose perfect hashing function generator that is a separate application program distributed in source code form with ACE and TAO. It is invoked by the TAO IDL compiler when the perfect hashing operation lookup strategy is selected. The default path for ACE's `ace_gperf` is `$ACE_ROOT/bin/ace_gperf`. To override this value, use the `-g` option and specify the *full path* to the location of the `ace_gperf` executable. For example, if `ace_gperf` is installed in `/usr/local/bin` instead of `$ACE_ROOT/bin`, you should invoke the IDL compiler as follows:

```
$TAO_ROOT/TAO_IDL/tao_idl -g /usr/local/bin/ace_gperf Messenger.idl
```

4.11 Collocation Strategy Options

The use of collocated stubs allows requests on collocated servants to be dispatched more directly by permitting requests to bypass several layers of marshaling, networking, demultiplexing, demarshaling, and dispatching logic.

TAO provides two strategies for generating and using collocated stubs.

- The `thru_poa` collocation strategy delivers the request through the servant's POA and is considered the standard collocated stub.

- The `direct` collocation strategy delivers the request directly from the stub to the servant as a normal C++ virtual function call, thereby bypassing the POA completely.

Collocation strategy options are shown in Table 4-10. For more details on using these collocation strategies at run time, see 17.13.5.

Table 4-10 Collocation Strategy Options

Option	Description
`-Gp` (default)	Generate collocated stubs that use `thru_poa` collocation strategy.
`-Gd`	Generate collocated stubs that use `direct` collocation strategy.

Note *The run-time ORB initialization options that affect collocation must be compatible with the types of generated collocated stubs. For example, using the* `-Gd` *option at compile time, then using the* `-ORBCollocationStrategy thru_poa` *option at run time, is inconsistent and results in a run-time exception.*

4.12 Back End Options

The `-Wb` option can be used to pass options to the TAO-IDL-compiler back end that generates the C++ code. The general format for these options is as follows:

```
-Wb,option_list
```

The option list is a comma-separated list that may contain any of the options shown in Table 4-11. These options mainly control platform-specific behavior of the back end.

Currently, the majority of the supported back end options are related to export macros. Export macros help control what symbols are visible external to libraries. These macros are required on Windows platforms and are also used by default in GCC versions 4.0 and later.

When defining your own libraries, header files that define export macros can be generated using the `%ACE_ROOT\bin\generate_export_file.pl` script. IDL code that will be included in your library needs to use the export macros that you generate for that library. The back-end options that end in

export_include are used to identify the export macro header file to use for a given generated file type. The actual macro to use is identified by the corresponding back-end option that ends in export_macro. For example:

```
generate_export_file.pl Messenger > MessengerExport.h
```

The header file is used by export_include as follows:

```
tao_idl -Wb,export_macro=Messenger_Export
-Wb,export_include=MessengerExport.h Messenger.idl
```

The export_macro triggers the inclusion of these macros in the class definition, which is necessary on platforms that control visibility. These macros allow the export of symbols from the library.

Table 4-11 Back End Options for -Wb

Option	Description
export_macro=*macro*	IDL compiler will emit the macro after each class or extern keyword (both stub and skeleton).
export_include=*path*	IDL compiler will include the file specified by *path* at the top of the client header.
skel_export_macro=*macro*	IDL compiler will emit the macro after each class or extern keyword in the skeleton code.
skel_export_include=*path*	IDL compiler will include the file specified by *path* at the top of the server headers.
stub_export_macro=*macro*	IDL compiler will emit the macro after each class or extern keyword in the stub code.
stub_export_include=*path*	IDL compiler will include the file specified by *path* at the top of the client header.
pch_include=*path*	IDL compiler will include the file specified by *path* in all generated files (used to support pre-compiled header mechanisms).
pre_include=*file*	IDL compiler will include the file specified by *file* at the beginning of the generated header file.
post_include=*file*	IDL compiler will include the file specified by *file* at the end of the generated header file.
obv_opt_accessor	IDL compiler will generate code to optimize access to base class data for valuetypes.
anyop_export_macro=*macro*	IDL compiler will emit the macro before each Any operator or extern typecode declaration in the generated stub code.

Table 4-11 Back End Options for -Wb

Option	Description
`anyop_export_include=path`	When using the -GA option to generate separate files for any operators, the use of this option causes the IDL compiler to include the file specified by *path* at the top of the any such headers.
`include_guard=path`	Only for internal TAO use. This option causes the IDL compiler to add code to the headers that prevent users from including the generated files.
`safe_include=file`	Only for internal TAO use. This option causes the IDL compiler to use the specified file in place of the normal generated file (*C.h).
`unique_include=file`	Only for internal TAO use. Include this file in the *C.h file instead of the normal TAO includes.

Here is an example showing how to use the back end options `export_macro` and `export_include`. This example shows how the IDL compiler is invoked when building the `TAO_CosNaming` shared library:

```
tao_idl -Wb,pch_include=CosNaming_pch.h -Wb,export_macro=TAO_Naming_Export
-Wb,export_include=Naming/naming_export.h -I../.. -I../../orbsvcs
-Wb,pre_include=ace/pre.h -Wb,post_include=ace/post.h CosNaming.idl
```

4.13 Suppression Options

Table 4-12 shows options you can use to suppress the generation of code that corresponds to certain CORBA features particular applications may not need. Suppressing some of the code normally generated for these features may produce smaller skeletons and stubs, which is important for memory-constrained systems.

Table 4-12 Suppression Options

Option	Description	Restrictions
`-Sa`	Suppress generation of the `any` operators.	The application cannot use the `any` data type in operation parameter lists.
`-Sal`	Suppress generation of the `any` operators for local interfaces only.	The application cannot store local objects in a `CORBA::Any`

Table 4-12 Suppression Options

Option	Description	Restrictions
-St	Suppress generation of TypeCodes for IDL-defined types. Automatically implies -Sa.	The application can neither use the any data type in operation parameter lists nor use a TypeCode for any type declared in the IDL file.
-Sp	Suppress generation of thru_poa collocated stubs.	Collocation must be disabled or the direct collocation strategy must be used (see 17.5).
-Sd	Suppress generation of direct collocation stubs.	Collocation must be disabled or the thru_poa collocation strategy must be used (see 17.5).
-Sm	Disable processing of IDL3 constructs.	IDL interfaces cannot use IDL3 constructs (See Chapter 32, the CORBA Component Model).
-Se	Suppresses use of custom header extension for TAO's IDL files. This allows applications to specify their custom header extensions via -hc and -hs while existing TAO IDL files can be included and keep their existing extension.	

Note *The run-time ORB initialization options that affect collocation must be compatible with the types of generated collocated stubs. For example, using the* -Sp *option at compile time, then using the* -ORBCollocationStrategy thru_poa *option at run time, is inconsistent and results in a run-time exception.*

4.14 Options Used Internally by TAO

Table 4-13 lists options used internally by TAO. These are documented and described for the sake of completeness but should not generally be used by applications.

Table 4-13 Output Used Internally by TAO

Option	Description
-Sorb	Suppress generation of the ORB.h include in generated files.

Table 4-13 Output Used Internally by TAO

Option	Description
`-Sci`	Suppress generation of the client inline file (`*C.inl`).
`-Scc`	Suppress generation of the client stub file (`*C.cpp`).
`-Ssi`	Suppress generation of the server inline file (`*S.inl`).
`-Ssc`	Suppress generation of the server stub file (`*S.cpp`).
`-SS`	Suppresses generation of skeleton (server) files. Only an empty `*S.h` file is generated.
`-GX`	Causes generation of empty *A.h files.

4.15 Output and Reporting Options

Table 4-14 lists options you can use to control the output of various warning, error, and informational messages, as well as the location of temporary files generated by the IDL compiler.

Table 4-14 Output and Reporting Options

Option	Description	Default
`-t` *`dir`*	Directory used by the IDL compiler for temporary files.	In UNIX, uses the value of the `TMPDIR` environment variable, if set, or `/tmp` by default. In Windows, uses the value of the `TMP` environment variable, if set, or the `TEMP` environment variable, if set, or the `WINNT` directory (on NT).
`-v`	Verbose flag. IDL compiler will print progress messages after completing major phases.	No progress messages displayed.
`-d`	Print the Abstract Syntax Tree (AST) to `stdout`.	AST is not displayed.
`-w`	Suppress warnings.	All warnings displayed.
`-V`	Print version information for front end and back end.	No version information displayed.
`-Cw`	Output a warning if two identifiers in the same scope differ in spelling only by case.	Error output is default.

Table 4-14 Output and Reporting Options

Option	Description	Default
`-Ce`	Output an error if two indentifiers in the same scope differ in spelling only by case.	Error output is default.

CHAPTER 5

Error Handling

5.1 Introduction

The inherent complexity of distributed applications increases the opportunity for errors to occur. To handle errors, distributed computing middleware needs a mechanism to communicate errors between components. Likewise, clients must be able to handle the error conditions communicated to them by servers.

Distributing an application across several processes and/or several hosts creates more opportunities for errors to occur. Figure 5-1 illustrates a distributed application with several objects distributed across three processes on two hosts. Possible errors include a hardware failure on one of the hosts, the loss of a network connection between the hosts, and a software failure in one of the server processes.

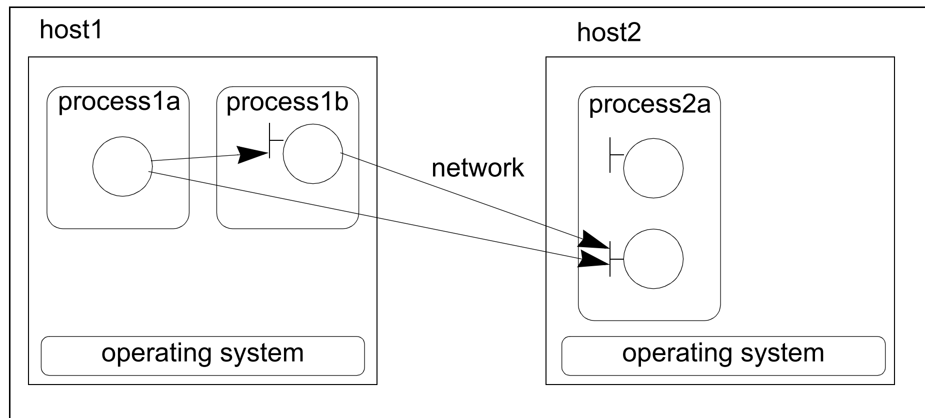


Figure 5-1 Sample Distributed Application

Before getting into TAO specifics, we first summarize the error-handling mechanisms common to all CORBA ORBs. There are two types of errors that may occur in a distributed system:

- System-level errors: These error conditions can happen in any distributed system. For example, a server can exit unexpectedly, thereby causing a loss of communication. Likewise, a client may send a request to an object that does not exist.
- User-level errors: These are application- and domain-specific error conditions defined by the architects and designers of a distributed system during application design and development. For example, a bank customer may attempt to withdraw more money from her bank account than is in the account.

By default, C++ exceptions are used by the CORBA IDL-to-C++ mapping to communicate error conditions between the client and server components. For more details on error handling in CORBA, see *Advanced CORBA Programming with C++*, 7.15.

Sometimes C++ exceptions are not available or desired. For instance, some platforms and compilers do not support C++ exceptions. Moreover, some applications cannot tolerate the performance impact or increase in code size that using C++ exceptions causes. The OMG defines an alternate mapping for such systems that passes error information through a `CORBA::Environment` parameter with each invocation. Older versions of TAO supported the alternate mapping, but TAO 2.2a no longer does. If your application requires

the alternate mapping you will need to use an older release of TAO, such as TAO 1.4a.

5.2 CORBA System Exceptions

By default, CORBA uses C++ exceptions to communicate error conditions between clients and servers. System-level errors automatically communicate with clients through a set of standard CORBA system exceptions. A system exception can be raised during any remote invocation. Table 5-1 lists some common system exceptions. For a complete list of system exceptions, please see *Advanced CORBA Programming with C++*, 7.15.

Table 5-1 Common System Exceptions

Name	Description
CORBA::COMM_FAILURE	The client's request was accepted, but a failure occurred (e.g., unexpected server termination) while processing the request.
CORBA::INV_OBJREF	The client attempted to invoke an operation on an invalid object reference.
CORBA::OBJECT_NOT_EXIST	The client attempted to invoke an operation on a non-existent object (e.g., not activated in the POA).
CORBA::TRANSIENT	The request was not able to reach its destination because a critical resource needed to carry out the request (e.g., the POA, the server, a connection) was not available.
CORBA::TIMEOUT	A request could not be completed within the specified time-to-live period as defined by the effective messaging quality of service (QoS) policies.

In C++, all CORBA exceptions derive from the class `CORBA::Exception`. All system exceptions derive from `CORBA::SystemException`, as shown in Figure 5-2.

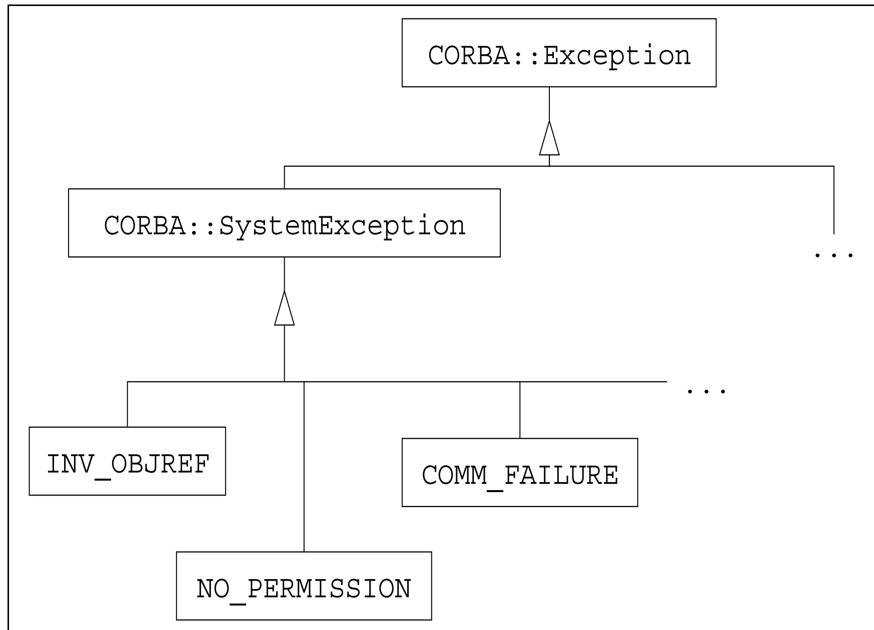


Figure 5-2 System Exception Hierarchy

A client must catch system exceptions raised by remote operations. CORBA system exceptions contain information that can aid in debugging.

Table 5-2 lists several operations you can use to get information from a CORBA exception.

Table 5-2 CORBA::Exception Operations

Operation	Description
`const char* _rep_id()`	Returns the Interface Repository ID of the exception.
`const char* _name()`	Returns the name of the exception.
`void _raise()`	Throws the exception.
`static CORBA::Exception* _downcast(CORBA::Exception*)`	"Downcast" an exception to a more-derived type. (Similar to `_narrow()` for object references.)
TAO Extensions	
`CORBA::TypeCode_ptr _type()`	Returns the typecode of the exception.
`int _is_a (const char* rep_id)`	Returns non-zero if Repository ID of exception matches `rep_id`.

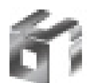


Table 5-2 CORBA::Exception Operations

Operation	Description
`void _tao_print_exception (const char *info, FILE *f = stdout)`	Print helpful debugging information about the exception to `f`, prepended by `info`.
`ACE_CString _info()`	Returns information printed by `_tao_print_exception()` as a string.
CORBA::SystemException Operations	
`CORBA::ULong minor()`	Returns an ORB-specific error code that conveys additional information about the error. See 5.4 for the TAO minor codes.
`CORBA::CompletionStatus completed()`	Returns an enumerated value that indicates whether the operation completed or not before the exception was raised. Valid values are `YES`, `NO`, and `MAYBE`.

The following examples illustrate the use of system exceptions in client code. Both of these examples are based on a simple interface of a hotel with guest rooms. In these examples, the client makes the following remote invocations:

- Getting the object reference from the Naming Service.
- Obtaining the hotel name with `hotel->name()`.
- Acquiring a reference to a guest room via `hotel->checkIn()`.
- Obtaining the room number using `room->roomNumber()`.

Any one of these invocations has the potential to raise a system exception.

Here is the IDL for our simple hotel:

```
interface GuestRoom
{
  readonly attribute short roomNumber;
  readonly attribute float balance;

  void checkOut();
};

interface Hotel
{
  readonly attribute string name;

  GuestRoom checkIn(in short numNights);
};
```

Example Here is an example of a client that catches system exceptions generically. Note that the catch code is highlighted.

```
#include <corba.h>
#include <iostream>

int main(int argv, char* argc[])
{
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    Hotel_var hotel = // get a Hotel proxy, possibly through the Naming Service

    CORBA::String_var name = hotel->name();
    std::cout << "The name of the hotel is " << name << std::endl;

    CORBA::Short numNights = 5;
    GuestRoom_var room = hotel->checkIn(numNights);

    std::cout << "The room number is " << room->roomNumber() << std::endl;
    orb->destroy();
  }
  catch (CORBA::SystemException& ex) {
    std::cerr << "A CORBA System Exception was caught: ";
    std::cerr << "ID = " << ex._rep_id()
              << "; minor code = " << ex.minor() << std::endl;
  }
  // catch all CORBA non-system exceptions
  catch (CORBA::Exception& ex) {
    std::cerr << "A CORBA Exception was caught: ";
    std::cerr << "ID = " << ex._rep_id() << std::endl;
  }
}
```

In addition to printing specific fields of the exception with the `_rep_id()` and `minor()` member functions, it is possible to simply insert the entire exception into an output stream, such as `std::cerr`. For instance, the catch clauses of the above example can be rewritten as follows:

```
  catch (CORBA::SystemException& ex) {
    std::cerr << "A CORBA System Exception was caught: " << ex << std::endl;
  }
  // catch all CORBA non-system exceptions
  catch (CORBA::Exception& ex) {
    std::cerr << "A CORBA Exception was caught: " << ex << std::endl;
  }
```

In TAO, the output stream insertion `operator <<()` has been overloaded for CORBA exceptions to print some details of the exception, such as its unqualified type name and its interface repository id. Here is what is printed by the first catch clause above if a `CORBA:TRANSIENT` exception is raised:

```
A CORBA System Exception was caught: TRANSIENT (IDL:omg.org/CORBA/TRANSIENT:1.0)
```

Example Here is an example of a client performing the same remote invocations but catching specific system exceptions.

```
#include <corba.h>
#include <iostream>

int main(int argv, char* argc[])
{
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    Hotel_var hotel = // get a Hotel proxy, possibly through the Naming Service

    CORBA::String_var name = hotel->name();
    std::cout << "The name of the hotel is " << name << std::endl;

    CORBA::Short numNights = 5;
    GuestRoom_var room = hotel->checkIn(numNights);

    std::cout << "The room number is " << room->roomNumber() << std::endl;
    orb->destroy();
  }
  catch (CORBA::COMM_FAILURE& ex) {
    std::cerr << "A communication failure occurred: "
              << "; minor code = " << ex.minor() << std::endl;
  }
  catch (CORBA::TRANSIENT& ex) {
    std::cerr << "A transient failure occurred: "
              << "; minor code = " << ex.minor() << std::endl;
  }
  // catch all other system exceptions
  catch (CORBA::SystemException& ex) {
    std::cerr << "A CORBA System Exception was caught: ";
    std::cerr << "ID = " << ex._rep_id()
              << "; minor code = " << ex.minor() << std::endl;
  }
  // catch all CORBA non-system exceptions
  catch (CORBA::Exception& ex) {
    std::cerr << "A CORBA Exception was caught: ";
    std::cerr << "ID = " << ex._rep_id() << std::endl;
```

```
  }
}
```

5.3 CORBA User Exceptions

Distributed system designers can use exceptions to communicate application-defined error conditions by specifying IDL modules and interfaces that contain CORBA User Exceptions. As shown in Figure 5-3, in C++, all user exceptions derive from `CORBA::UserException`, which itself derives from `CORBA::Exception`.

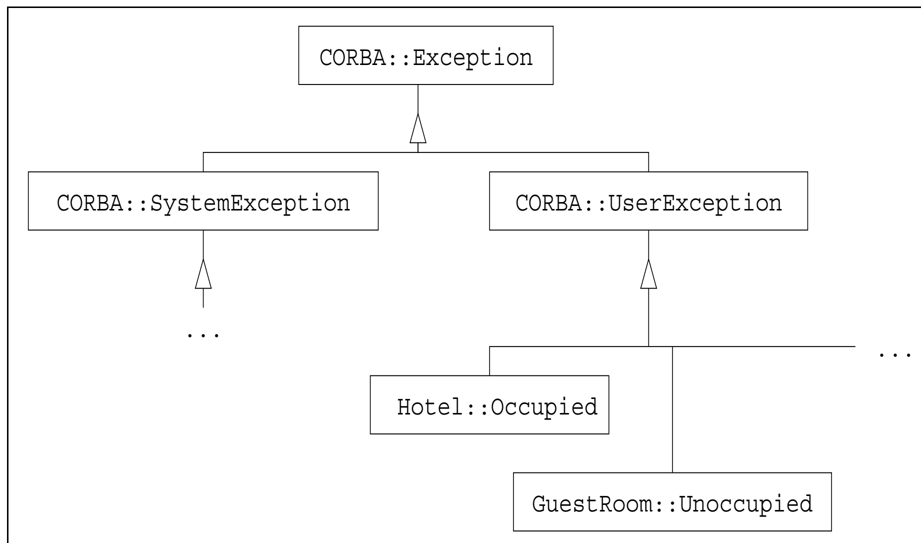


Figure 5-3 User Exceptions

A user exception is similar to an IDL `struct` in that it can contain data members. For example, you can extend the `GuestRoom` interface as follows:

```
interface GuestRoom
{
  exception Unoccupied
  {
    short  daysEmpty;
    string lastOccupant;
  };

  readonly attribute short roomNumber;
  readonly attribute float balance;
```

```
  void checkOut() raises (Unoccupied);
};
```

The client can catch the `Unoccupied` exception whenever `GuestRoom::checkOut()` is called:

```
  GuestRoom_var room = // get a proxy to a GuestRoom;
  try {
    room->checkOut();
  }
  catch (GuestRoom::Unoccupied& ex) {
    std::cerr << "Cannot check out. Room "
              << room->roomNumber() << " has been empty for "
              << ex.daysEmpty << " days. The last occupant was "
              << ex.lastOccupant << std::endl;
  }
  // catch all other user-defined CORBA exceptions
  catch (CORBA::UserException& ex) {
    std::cerr << "A CORBA User Exception was caught: ";
    std::cerr << "ID = " << ex._rep_id() << std::endl;
  }
  catch (CORBA::SystemException& ex) {
    std::cerr << "A CORBA System Exception was caught: ";
    std::cerr << "ID = " << ex._rep_id()
              << "; minor code = " << ex.minor() << std::endl;
  }
  // should never be reached, because all CORBA exceptions are
  // either system exceptions or user exceptions.
  catch (CORBA::Exception& ex) {
    std::cerr << "A CORBA Exception was caught: ";
    std::cerr << "ID = " << ex._rep_id() << std::endl;
  }
```

On the server side, the server's implementation of the `checkOut()` operation throws an `Unoccupied` exception if a `checkOut()` is attempted on an empty room. For example, suppose `GuestRoom_i` is the name of the server-side class that implements the `GuestRoom` interface:

```
void GuestRoom_i::checkOut()
{
  if ( /* the room is occupied */ ) {
    // check the guest out of the room
  }
  else { // the room is unoccupied; ERROR
    CORBA::Short daysEmpty =         // # of days room is empty
    CORBA::String_var lastOccupant = // name of last occupant

    throw GuestRoom::Unoccupied (daysEmpty,lastOccupant);
  }
}
```

Conceptually, the exception is thrown “across the wire” to the client. Actually, what happens is a little more complicated. First, the exception thrown by the implementation is caught by the CORBA infrastructure (skeleton or ORB) on the server side, marshalled, and returned to the client in the reply message. The client-side CORBA infrastructure (ORB and stub) unmarshals the reply body into a C++ exception, then throws it. Finally, the client code can catch the exception and handle it in an application-specific manner. Conceptually, therefore, the exception appears to be thrown directly from the server to the client.

5.4 TAO Minor Codes

A CORBA system exception includes information about the problem that occurred. You can obtain this information by calling `CORBA::Exception::minor()`. The value returned from `minor()` is a 32-bit value known as the *minor code value*. The minor code value contains several pieces of information about the exception in the following groups of bits:

- The high order 20 bits of the minor code value define the particular ORB implementation that is transmitting the exception. The OMG assigns this value to the ORB vendor or other responsible party to ensure it is unique for all ORB implementations. TAO’s unique identifier is the hexadecimal value `0x54410` (represented by ASCII `"TA"` followed by `0x0`).
- The low order 12 bits are assigned no special significance by the OMG. These bits comprise the implementation-specific minor code. For TAO, the low order 12 bits are further divided as follows:
    - The first 5 bits comprise the *location code* that identifies the location in TAO where the exception was raised.
    - The remaining 7 bits encode an error number (`errno`) associated with the exception.

TAO’s location codes are described in 5.4.1 and the meanings of various error numbers are described in 5.4.2.

5.4.1 Location Codes

Each location code is assigned a preprocessor macro definition for improved source code readability. The file `$TAO_ROOT/tao/ORB_Constants.h` contains the definitions of these macros. To determine why an exception is being thrown, you can step through the TAO source code in a debugger. However, that can be time consuming. Instead, you can search the TAO source code for these macro definitions and examine the code that detected the failure. Doing so often leads to insights as to the exact cause of the error.

`0x01U TAO_INVOCATION_LOCATION_FORWARD_MINOR_CODE`
If a client attempts to connect to a server and receives a `LOCATION_FORWARD` reply, it retries the connection at the address contained in the reply. If the retry fails, the client ORB raises a `CORBA::TRANSIENT` exception with the minor code set to `TAO_INVOCATION_LOCATION_FORWARD_MINOR_CODE`.

`0x02U TAO_INVOCATION_SEND_REQUEST_MINOR_CODE`
If a client attempts to send a request to a server for the current profile and fails, the client ORB raises a `CORBA::TRANSIENT` exception with the minor code set to `TAO_INVOCATION_SEND_REQUEST_MINOR_CODE`.

`0x03U TAO_POA_DISCARDING`
A POAManager in the `PortableServer::POAManager::DISCARDING` state causes the associated POAs to discard all incoming requests. Requests for which processing has already begun are allowed to continue. When a request is discarded, the POA will raise a `CORBA::TRANSIENT` exception to indicate that the client should retry the request. The POA will set the minor code to `TAO_POA_DISCARDING`.

`0x04U TAO_POA_HOLDING`
A POAManager in the `PortableServer::POAManager::HOLDING` state may cause the POAs associated with it to queue incoming requests, up to an implementation-defined limit. If this limit is exceeded, the POA may discard requests and raise the `CORBA::TRANSIENT` exception with minor code of `TAO_POA_HOLDING` to indicate to the client that it should retry the request.

Note *TAO's implementation of the POA does not support queuing of requests, so requests are immediately rejected with a* `CORBA::TRANSIENT` *exception if the POAManager is in the* `HOLDING` *state.*

0x05U TAO_UNHANDLED_SERVER_CXX_EXCEPTION
If a servant implementation throws a native C++ exception that is not handled within the application code, the server ORB cannot propagate the exception to the client. Because the CORBA specification provides no standards for marshaling or demarshaling native C++ exceptions, the server ORB raises a `CORBA::UNKNOWN` exception with the minor code set to `TAO_UNHANDLED_SERVER_CXX_EXCEPTION`.

0x06U TAO_INVOCATION_RECV_REQUEST_MINOR_CODE
If a client sends a request to a server and detects an error (other than a timeout) while waiting for a reply, the client ORB raises a `CORBA::COMM_FAILURE` exception with a `TAO_INVOCATION_RECV_REQUEST_MINOR_CODE` minor code.

0x07U TAO_CONNECTOR_REGISTRY_NO_USABLE_PROTOCOL
If none of the connector objects in the ORB's `Connector_Registry` are able to parse a particular URL-style stringified IOR (e.g., it may not be formatted properly or it may specify an unrecognized protocol), the ORB raises a `CORBA::INV_OBJREF` exception with the minor code set to `TAO_CONNECTOR_REGISTRY_NO_USABLE_PROTOCOL`. See Chapter 14 for more information on using TAO's pluggable protocols and 17.13.43 for more information on specifying URL-style object references.

0x08U TAO_MPROFILE_CREATION_ERROR
If the ORB, in attempting to parse a string (e.g., a stringified IOR), encounters an error creating the `MProfile` (the list of profiles contained within the object reference), it raises a `CORBA::INV_OBJREF` exception with the minor code set to `TAO_MPROFILE_CREATION_ERROR`.

0x09U TAO_TIMEOUT_CONNECT_MINOR_CODE
If a client fails to connect to a server within a specified timeout period, the client ORB raises a `CORBA::TIMEOUT` exception with the minor code set to `TAO_TIMEOUT_CONNECT_MINOR_CODE.`

0x0AU TAO_TIMEOUT_SEND_MINOR_CODE
If a client attempts to invoke an operation on a CORBA object, but the invocation can not be completed within a specified timeout period, the client ORB raises a `CORBA::TIMEOUT` exception with the minor code set to `TAO_TIMEOUT_SEND_MINOR_CODE.`

0x0BU TAO_TIMEOUT_RECV_MINOR_CODE
If a client sends a request to a server but fails to receive a reply within a specified timeout period, the client ORB raises a `CORBA::TIMEOUT` exception with the minor code set to `TAO_TIMEOUT_RECV_MINOR_CODE.`

`0x0CU TAO_IMPLREPO_MINOR_CODE`
General-purpose code indicating an error related to the TAO Implementation Repository (IMR). For example, a server could not notify the IMR of its start up, the IMR could not forward a client request to a server, or a server is not running and the server's activation mode does not allow the IMR to automatically start it. In all cases, a `CORBA::TRANSIENT` exception is raised with the minor code set to `TAO_IMPLREPO_MINOR_CODE`.

`0x0DU TAO_ACCEPTOR_REGISTRY_OPEN_LOCATION_CODE`
A server ORB opens an acceptor to allow it to accept client connections on one or more endpoints. Endpoints can be specified when the ORB is initialized (e.g., via the `-ORBListenEndpoints` option). If no endpoints are specified, the ORB will attempt to open acceptors on default endpoints for the loaded transports protocols. Opening an acceptor can fail for a variety of reasons, including: The ORB is unable to create or open an acceptor, the ORB is unable to add an acceptor to its acceptor registry, an invalid endpoint was specified, no usable transport protocol has been loaded, or a specified endpoint is already in use by another service. Such errors usually result in a `CORBA::BAD_PARAM` exception being raised with the minor code set to `TAO_ACCEPTOR_REGISTRY_OPEN_LOCATION_CODE`, usually during POA activation.

`0x0EU TAO_ORB_CORE_INIT_LOCATION_CODE`
An error can occur during ORB core initialization for a variety of reasons, including: An invalid argument was supplied to an `-ORBInitRef` option, an unrecognized argument starting with "`-ORB`" was passed to `CORBA::ORB_init()`, or an invalid endpoint was specified. In these cases, the ORB will raise `CORBA::BAD_PARAM`. Other reasons ORB initialization can fail include internal errors in TAO's configuration causing the ORB to fail to load a resource factory or server strategy factory. In these cases, the ORB will raise `CORBA::INTERNAL`. Other errors during ORB initialization, such as failure to initialize a codeset manager, reactor, pluggable protocol factories, or default policies, will result in a `CORBA::INITIALIZE` exception. In all these cases, the exception's minor code will be set to `TAO_ORB_CORE_INIT_LOCATION_CODE.`

`0x0FU TAO_POLICY_NARROW_CODE`
Not applicable.

`0x10U TAO_GUARD_FAILURE`
TAO's POA and Real-Time POA maintain internal locks using the scoped locking idiom, commonly known as a *guard* or lock monitor. If the POA or

the RT CORBA thread pool mechanism fails to acquire its internal lock via the guard, a `CORBA::INTERNAL` exception will be raised with the minor code set to `TAO_GUARD_FAILURE`.

`0x11U TAO_POA_BEING_DESTROYED`
An attempt to use the POA after it has been destroyed or as it is being destroyed can result in a `CORBA::BAD_INV_ORDER` exception with the minor code set to `TAO_POA_BEING_DESTROYED`.

`0x12U TAO_POA_INACTIVE`
An attempt to use the POA after it as been deactivated via its POA manager will result in a `CORBA::OBJ_ADAPTER` exception with the minor code set to `TAO_POA_INACTIVE`.

`0x13U TAO_CONNECTOR_REGISTRY_INIT_LOCATION_CODE`
If an invocation (such as the first invocation on an object reference) requires a new connection, but initialization of the ORB core's connector registry fails, a `CORBA::INITIALIZE` exception will be raised with the minor code set to `TAO_CONNECTOR_REGISTRY_INIT_LOCATION_CODE`.

`0x14U TAO_AMH_REPLY_LOCATION_CODE`
In an application using TAO's Asynchronous Method Handling (AMH) feature, if the response handler is deleted before a reply as been sent to the client, a `CORBA::NO_RESPONSE` exception will be generated and sent to the client. In other cases, if the application attempts to use a response handler incorrectly, a `CORBA::BAD_INV_ORDER` exception will be raised. In both cases, the minor code will be set to `TAO_AMH_REPLY_LOCATION_CODE`.

`0x15U TAO_RTCORBA_THREAD_CREATION_LOCATION_CODE`
If an error occurs during thread creation in an RT CORBA thread pool, a `CORBA::INTERNAL` exception will be raised with the minor code set to `TAO_RTCORBA_THREAD_CREATION_LOCATION_CODE`.

5.4.2 Error Number Codes

Many system exceptions are the result of a failure when accessing a system function. Most system functions set a global error number, known as `errno`, that identifies the reason for the failure. Table 5-3 gives the TAO preprocessor

macro definition and a short description associated with each possible error number value included in the minor code.

Table 5-3 Minor Code Error Numbers

Hex Value	Macro Definition	Description
0x0U	TAO_UNSPECIFIED_MINOR_CODE	No error number associated with the exception.
0x1U	TAO_ETIMEDOUT_MINOR_CODE	Connection timed out.
0x2U	TAO_ENFILE_MINOR_CODE	System file table is full.
0x3U	TAO_EMFILE_MINOR_CODE	Process has too many open files.
0x4U	TAO_EPIPE_MINOR_CODE	No process to read the data from a pipe.
0x5U	TAO_ECONNREFUSED_MINOR_CODE	Target machine refused a connection.
0x6U	TAO_ENOENT_MINOR_CODE	A file or directory does not exist.
0x7U	TAO_EBADF_MINOR_CODE	A file descriptor refers to a file that is not open, or trying to read from a file opened only for writing.
0x8U	TAO_ENOSYS_MINOR_CODE	Operation not applicable.
0x9U	TAO_EPERM_MINOR_CODE	Not the super user.
0xAU	TAO_EAFNOSUPPORT_MINOR_CODE	Address not compatible with requested protocol.
0xBU	TAO_EAGAIN_MINOR_CODE	System process table is full.
0xCU	TAO_ENOMEM_MINOR_CODE	Not enough memory.
0xDU	TAO_EACCES_MINOR_CODE	File access denied.
0xEU	TAO_EFAULT_MINOR_CODE	Attempting to access a bad address.
0xFU	TAO_EBUSY_MINOR_CODE	Device busy or lock is held.
0x10U	TAO_EEXIST_MINOR_CODE	File not expected to exist.
0x11U	TAO_EINVAL_MINOR_CODE	An invalid argument was used.
0x12U	TAO_ECOMM_MINOR_CODE	Communication error on send.
0x13U	TAO_ECONNRESET_MINOR_CODE	Connection reset by peer.
0x14U	TAO_ENOTSUP_MINOR_CODE	Function not implemented.

5.5 Summary

- Exceptions provide a mechanism to communicate error information between CORBA clients and servers. Exceptions are mapped to native C++ exceptions.
- A set of `CORBA::SystemException` exceptions is defined for system-level errors.
- Domain- or application-specific exceptions are defined in IDL. The generated C++ exception classes inherit from `CORBA::UserException`.
- When a CORBA system exception is thrown, a minor code is provided to help identify the reason for the failure. Decoding this minor code can help in identifying the cause of the failure.

CHAPTER 6

CORBA Messaging

6.1 Introduction

The OMG introduced a CORBA Messaging specification to facilitate the development of portable CORBA code that efficiently supports the following:

- Making requests that do not require a client to block while waiting for a reply from a server. This is referred to as Asynchronous Method Invocation (AMI).
- Handling replies that are returned after the client process that submitted the associated request has terminated. This is referred to as Time Independent Invocation (TII).
- Allowing Quality of Service (QoS) to be specified for method invocations at the application level.

This functionality is now part of the CORBA specification (OMG Document formal/08-01-04, Chapter 17).

TAO supports the callback model of AMI and a subset of the Messaging specification's QoS policies. TAO does not support Time-Independent Invocation of requests. The AMI callback model is described in 6.2. The QoS

policies are described in 6.3. A Bi-Directional GIOP policy is described in 6.4. Bi-Directional GIOP is defined in Part 2 of the CORBA Core specification (OMG Document formal/08-01-06) sections 9.8 and 9.9.

6.2 AMI Callback Model

The Problem

When a synchronous CORBA operation is invoked, the invoking client is blocked until it receives the reply to the operation request. A client cannot, however, always afford to spend its time exclusively waiting for the reply. Consider the following situations:

- *During time-consuming invocations to distributed objects, the user is not confident that the client is running properly.* While waiting for a reply, it may be desirable for the client application to provide periodic feedback to the user, verifying that the program is still waiting for a reply from the server.
- *A client wishes to make concurrent, rather than consecutive, requests to numerous servers.* Suppose that to obtain the total inventory count for a particular auto part, a number of warehouses must be queried. Whereas a single query may not be very time consuming, the cumulative waiting time for consecutive requests could be excessive. If all the queries are made concurrently, the total inventory count will be less time consuming.
- *The reply represents an event*. A client may want to be informed when some external event occurs, but does not want to block all other activities while waiting for such an event.

Prior to the communication models defined in the Messaging specification, the only models provided by CORBA for asynchronous communication were the *deferred synchronous* and *oneway* models.

Because the deferred synchronous model requires the use of the Dynamic Invocation Interface (DII), much more code is required for its implementation than for the synchronous model. In addition, the DII is tedious to use, not type-safe, and inefficient.

The oneway model requires the creation of a *reply-handler* object on the client side for handling replies to `oneway` operations that the client invokes on the server. An object reference to the reply handler is passed to the server in a

oneway operation. To reply to a oneway request, the server must invoke an operation on this reply handler. There are several disadvantages to this approach:

- The callback interface adds to the complexity of the IDL code.
- The interface for the server object must be altered to include the oneway definition and the callback object reference parameter.
- The server code needs to be written to also play the role of a client, so that it can invoke an operation on the callback object.
- Only the oneway operation is implemented. If the client application also needs a synchronous version of this operation, it must be defined separately.
- Traditional oneways guarantee neither non-blocking semantics nor reliable delivery.

The AMI Callback Solution

The Asynchronous Method Invocation (AMI) callback[1] model, defined by the Messaging specification and fully implemented in TAO 1.6a, addresses the above concerns by providing asynchronous operations that are not oneway operations and do not use the DII. These operations are referred to as sendc_ operations throughout this chapter. A sendc_ operation is provided to the client in addition to, rather than instead of, the corresponding synchronous operation, so the client may invoke either one at any time.

A sendc_ operation has two purposes:

1. To cause the client ORB to send a request message to the server.
2. To provide the client ORB with an object reference to a reply handler.

AMI is a client-side language mapping issue, so enabling AMI does not alter the CORBA interface, and changes to server implementations are normally not required[2]. Thus, from the server's perspective, a request message initiated by a sendc_ operation invocation is identical to a request message initiated by the corresponding synchronous operation invocation.

The client ORB is the workhorse of the AMI callback model. It transforms the sendc_ operation invocations into request messages and internally maps the

1. An AMI polling model is also defined by the specification, but is not supported by TAO.
2. Servers may require changes to handle transactional asynchronous requests.

request ID to the reply-handler object reference. When it receives a reply to a request message, it uses this internal mapping to invoke an operation on the designated reply handler. A reply-handler skeleton class is implicitly generated as part of the AMI callback model, so no explicit reply-handler interface need be defined. The application developer simply writes a reply handler that derives from this skeleton.

The AMI callback model is enabled by invoking the TAO IDL complier with the `-GC` option. This produces the following additions to the resultant code for each IDL interface:

1. A set of `sendc_` member functions.
2. A reply-handler skeleton class.

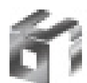
These components are described in 6.2.1 through 6.2.3. The process of writing a reply handler is covered in 6.2.4. The chain of events that is set in motion by invoking a `sendc_` operation is illustrated in Figure 6-1 and described in detail in 6.2.5.

The Messaging specification makes use of a concept called *implied-IDL* to define the implementation of these components in a language-independent manner. To distinguish it from actual IDL code, all implied-IDL code in this chapter is displayed in *italic* type.

Drawbacks to using AMI

There are some drawbacks to using AMI that are worth noting:

- Additional stub code is generated for *every* operation and attribute in an interface. If only a small percentage of the operations and attributes are accessed using asynchronous invocations, and executable size is an issue, then it may make more sense to use alternative asynchronous techniques.
- Support for AMI callback is just beginning to appear in ORBs, so if client application portability among different ORB implementations is desired, it may be premature to deploy AMI-based application code. Since the use of AMI callback does not alter the CORBA interface, this is a client-side issue only. AMI callback clients are fully interoperable with non-AMI-callback servers.
- Client programmers have to write more code to support the AMI callback model than the normal synchronous invocation model. In particular, you must implement the reply-handler class, and you must supply an event loop on the client side to handle asynchronous replies. Also, exception



handling is considerably more complicated with the AMI callback model than with normal synchronous invocations.

6.2.1 Asynchronous sendc_ Operations

When invoked with the `-GC` option, the TAO IDL complier generates `sendc_` member functions for the proxy[3] (stub) class in addition to the synchronous member functions. These additional member functions can be thought of as being generated from implied-IDL operations[4] that are added to the IDL interface. All `sendc_` operations have a return type of `void` and are defined as follows:

For each synchronous IDL operation `opName`, an implied-IDL operation named *sendc_opName* is defined according to the following rules:

- The first parameter is an *in* parameter named *ami_handler*, a reference to the designated reply handler.
- Each `in` and `inout` parameter in `opName` becomes an *in* parameter in *sendc_opName*.
- If `opName` has a `context` expression (specifying which elements of the client's context may affect the performance of a request by the object), then *sendc_opName* will have an identical *context* expression.

The return value and `out` parameters of `opName` are ignored because they are handled by the reply handler.

For each IDL attribute `attrName`, an implied-IDL operation named *sendc_get_attrName* is defined. Its only parameter is an *in* parameter named *ami_handler*, a reference to its reply handler.

For each *non-readonly* IDL attribute `attrName`, an *additional* implied-IDL operation named *sendc_set_attrName* is defined according to the following rules:

- The first parameter is an *in* parameter named *ami_handler*, a reference to its reply handler.
- There is a second `in` parameter named *attrName* that has the same type as `attrName` and is used to set the attribute value.

3. See *Advanced CORBA Programming with C++*, 7.3 for more information about proxy classes.
4. These new operations are not considered to be real IDL because they do not correspond to entries in the Interface Repository.

If a `sendc_` operation is invoked with a nil *ami_handler* value, no response will be returned for that invocation.

Suppose we have the following IDL definition for `MyInterface`:

```
exception UserExcep {string usr_exc;};

interface MyInterface {
  boolean opName(in short a_short,
                 inout long a_long,
                 out float a_float)
          raises(UserExcep);
  attribute short attrib1;
  readonly attribute short attrib2;
};
```

The implied-IDL *sendc_* operations for `MyInterface` are:

```
  void sendc_opName(in AMI_MyInterfaceHandler ami_handler
                    in short a_short,
                    in long a_long);

  void sendc_get_attrib1(in AMI_MyInterfaceHandler ami_handler);
  void sendc_set_attrib1(in AMI_MyInterfaceHandler ami_handler,
                         in short attrib1);

  void sendc_get_attrib2(in AMI_MyInterfaceHandler ami_handler);
```

The `sendc_` member functions of the C++ proxy (stub) class are:

```
  virtual void sendc_opName (AMI_MyInterfaceHandler_ptr ami_handler,
                             CORBA::Short a_short,
                             CORBA::Long a_long);

  virtual void sendc_get_attrib1 (AMI_MyInterfaceHandler_ptr ami_handler);

  virtual void sendc_set_attrib1 (AMI_MyInterfaceHandler_ptr ami_handler,
                                  CORBA::Short attrib1);

  virtual void sendc_get_attrib2 (AMI_MyInterfaceHandler_ptr ami_handler);
```

6.2.2 The ExceptionHolder

When a `sendc_` operation is invoked, the client ORB attempts to send a request message to the server. If this attempt fails, the `sendc_` operation raises

a system exception with a completion status of COMPLETED_NO. Otherwise, the sendc_ operation returns normally and the client application continues.

If an exception occurs during the processing of a sendc_ request, the server returns this exception to the client ORB in the reply message, just as in the case of a synchronous operation. Unlike the synchronous case, however, the sendc_ operation cannot raise an exception because it returns before the reply is received.

When a reply to a sendc_ operation contains an exception, the client ORB receiving the reply must deliver this exception to the designated reply handler. Because CORBA exceptions cannot be passed as arguments in an IDL interface, the exception is inserted into an ExceptionHolder for delivery to the designated reply handler.

The CORBA specification defines the Messaging::ExceptionHolder valuetype:

```
module Messaging
{
  typedef CORBA::OctetSeq MarshaledException;
  native UserExceptionBase;
  valuetype ExceptionHolder {
    void raise_exception() raises (UserExceptionBase);
    void raise_exception_with_list(in CORBA::ExceptionList exc_list)
              raises (UserExceptionBase);
    private boolean is_system_exception;
    private boolean byte_order;
    private MarshaledException marshaled_exception;
  };
};
```

TAO implements this valuetype with the Messaging::ExceptionHolder C++ class. When called back with an exception, AMI applications will typically call the raise_exception() operation, which throws the corresponding C++ exception. Applications then catch this exception like any C++ exception. The following section shows this behavior in the Reply Handler.

Note *Earlier versions of TAO generate type-specific Exception Holder classes for each interface that supports AMI as prescribed by the CORBA 2.6 specification (OMG Document formal/01-12-35).*

6.2.3 Reply Handler Operations

When the TAO IDL compiler is invoked with the `-GC` option, it generates a C++ reply-handler skeleton class for each interface on which it is invoked. This class can be thought of as having been compiled from an implied-IDL reply-handler interface. For an interface named `MyInterface`, the name of the implied-IDL interface is *`AMI_MyInterfaceHandler`* and the name of the generated skeleton class is `POA_AMI_MyInterfaceHandler`. The application developer writes a reply-handler class that inherits from the skeleton class and that is usually instantiated as a servant within the client.

However, the reply-handler servant does not have to be located within the client application. It can be located in another process. For example, if multiple instances of a client are instantiated, it may be desirable to handle all replies in only one of the instances. It is important to understand, however, that the client ORB that sends a request message will always receive the reply to this message. It is the client ORB that then invokes the reply-handler operation.

Reply-handler operations are invoked only by an ORB. They do not raise exceptions because they are never invoked by a client and thus have no client to respond to the exceptions. All reply-handler operations have a return type of `void` because their only purpose is to pass information to the reply handler.

An implied-IDL reply-handler interface contains two reply-handler operations for each `sendc_` operation, one to handle normal (non-exception) replies and another to handle exception replies. Thus there are two types of reply-handler operations:

- *Non-exception*: Delivers `sendc_` operation results.
- *Exception*: Delivers exceptions that occur during a `sendc_` operation.

Non-Exception Replies

Non-exception reply-handler operations are defined as follows:

For each IDL operation `opName`, an implied-IDL reply-handler operation named *opName* is defined according to the following rules:

- If the operation has a return value, then the first parameter is an `in` parameter named *ami_return_val* which is the return value of the IDL operation.

- Each `inout` and `out` parameter in `opName` (the IDL operation) becomes an *in* parameter in *opName* (the reply-handler operation).

The in parameters of the IDL operation are ignored in the reply-handler operation because they are not part of the reply.

For an IDL attribute `attrName`, the implied-IDL operation *get_attrName* is defined. It has a single *in* parameter named *ami_return_val* of the same type as the attribute.

For a *non-readonly* IDL attribute `attrName`, an *additional* implied-IDL operation named *set_attrName* with no parameters is defined.

There are two cases where the above rules will result in a reply-handler operation with no parameters:

- An IDL operation that has a return type of `void` and no `inout` or `out` parameters.
- A non-readonly attribute (the set operation does not return a value).

In these cases, the reply-handler operation simply acknowledges a successful completion of the IDL operation.

Exception Replies

When an exception occurs during the processing of a `sendc_` operation, the exception is returned to the client ORB in the reply message, just as it is in the case of a synchronous operation. In the case of the `sendc_` operation, however, the client ORB inserts the exception into an `ExceptionHolder` value and then invokes the designated reply-handler operation with this `ExceptionHolder` as its parameter.

Reply-handler operations that deliver exceptions have a single *in* parameter named *excep_holder* and are defined as follows:

For each IDL operation `opName`, an implied-IDL reply-handler operation named *opName_excep* is defined.

For an IDL attribute `attrName`, an implied-IDL reply-handler operation named *get_attrName_excep* is defined.

For a *non-readonly* IDL attribute `attrName`, an *additional* reply-handler implied-IDL operation named *set_attrName_excep* is defined.

Applying the above rules to `MyInterface` from 6.2.1 yields

```
// Reply-handler implied-IDL
interface AMI_MyInterfaceHandler {
  void opName(in boolean ami_return_val,
              in long a_long,
              in float a_float);

  void opName_excep(in Messaging::ExceptionHolder excep_holder);

  void get_attrib1(in short ami_return_val);
  void get_attrib1_excep(in Messaging::ExceptionHolder excep_holder);

  void set_attrib1();
  void set_attrib1_excep(in Messaging::ExceptionHolder excep_holder);

  void get_attrib2(in short ami_return_val);
  void get_attrib2_excep(in Messaging::ExceptionHolder excep_holder);
};
```

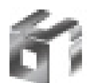
The generated C++ reply-handler skeleton class for `MyInterface` is:

```
class POA_AMI_MyInterfaceHandler: public virtual POA_Messaging::ReplyHandler
{
public:
// AMI callback exception support and TAO implementation code not shown.
   virtual void op(CORBA::Boolean ami_return_val,
                   CORBA::Long a_long,
                   CORBA::Float a_float) = 0;
   virtual void opName_excep(Messaging::ExceptionHolder* excep_holder) = 0;

   virtual void get_attrib1(CORBA::Short ami_return_val) = 0;
   virtual void get_attrib1_excep(Messaging::ExceptionHolder* excep_holder) = 0;

   virtual void set_attrib1() = 0;
   virtual void set_attrib1_excep(Messaging::ExceptionHolder* excep_holder) = 0;

   virtual void get_attrib2(CORBA::Short ami_return_val) = 0;
   virtual void get_attrib2_excep(Messaging::ExceptionHolder* excep_holder) = 0;
};
```

6.2.4 Creating a Reply-Handler Class

6.2.4.1 Generate Starter Code

When the options `-GC` (generate stub code for AMI callback support) and `-GI` (see 4.7) are simultaneously passed to the TAO IDL compiler, reply-handler class starter code is automatically generated for each interface in an IDL file. The two files generated are suffixed with `I.h` and `I.cpp`. For `MyInterface`,



the generated files are `MyInterfaceI.h` and `MyInterfaceI.cpp`. Since these files also contain servant starter code that is only relevant to the server side, the reply-handler code should be extracted and placed in separate files. The following reply-handler code was extracted from `MyInterfaceI.cpp` and inserted into the file `MyReplyHandler.cpp`.

```
// ACE exception code is not shown
void AMI_MyInterfaceHandler_i::opName (
    CORBA::Boolean ami_return_val,
    CORBA::Long a_long,
    CORBA::Float a_float)
  {
    //Add your implementation here
  }

void AMI_MyInterfaceHandler_i::opName_excep (
    Messaging::ExceptionHolder* excep_holder)
  {
    //Add your implementation here
  }

void AMI_MyInterfaceHandler_i::get_attrib1 (
    CORBA::Short ami_return_val)
  {
    //Add your implementation here
  }

void AMI_MyInterfaceHandler_i::get_attrib1_excep (
    Messaging::ExceptionHolder* excep_holder)
  {
    //Add your implementation here
  }

void AMI_MyInterfaceHandler_i::set_attrib1 ()
  {
    //Add your implementation here
  }

void AMI_MyInterfaceHandler_i::set_attrib1_excep (
    Messaging::ExceptionHolder* excep_holder)
  {
    //Add your implementation here
  }

void AMI_MyInterfaceHandler_i::get_attrib2 (
    CORBA::Short ami_return_val)
  {
    //Add your implementation here
```

```
    }

  void AMI_MyInterfaceHandler_i::get_attrib2_excep (
      Messaging::ExceptionHolder* excep_holder
    )
    {
      //Add your implementation here
    }
```

The AMI_MyInterfaceHandler_i reply-handler class inherits from the POA_AMI_MyInterfaceHandler class shown in 6.2.3. To complete the reply-handler class, the application developer needs to replace “Add your implementation here” with the desired functionality in each member function.

The type of functionality that is added to the starter code depends, first of all, on where the reply handler is to reside. Remember that the reply handler is not restricted to residing in the client process from which the sendc_ operation is invoked. It may reside in any process. The only restrictions are that the client must be able to obtain an object reference to the reply-handler object (to pass it in the sendc_ invocation) and the client process from which the operation was invoked must still be running when the reply is returned. (Remember that even if the reply handler is not part of the client application, the client ORB that invoked the request must receive the reply and invoke the reply-handler operation.)

In 6.2.4.2 and 6.2.4.3, we describe how to add the needed functionality to the reply-handler starter code.

6.2.4.2 Non-Exception Reply-Handler Functions

In general, if the reply handler resides within the client that invoked the sendc_ function, the reply handler copies the return value and other out parameters into appropriate variables and/or outputs a message. A common way to store the return values and out parameters is to declare these variables as private members of the reply-handler class.

First, we show an example using non-AMI invocations. Using the MyInterface example again, the following client application makes use of the synchronous form of opName (assume that the IOR of the MyInterface object is stored in my_interface.ior):

```
#include "MyInterfaceC.h"
```

```
int main(int argc, char* argv[])
{
  try {
    CORBA::Boolean my_return_value;
    CORBA::Long my_long;
    CORBA::Float my_float;

    // Get an object reference to MyInterface object.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var obj = orb->string_to_object("file://my_interface.ior");
    MyInterface_var myInterface = MyInterface::_narrow(obj.in());

    my_return_value = myInterface->opName(10, my_long, my_float);

    // do other stuff...

    orb->destroy();
  }
  catch (CORBA::Exception&) {
    // Handle CORBA exceptions...
  }
}
```

In the above code segment:

- The return value of `opName` is stored in `my_return_value`.
- `opName`'s second variable, an `inout` variable, is stored in `my_long`.
- `opName`'s third variable, an `out` variable, is stored in `my_float`.

To achieve the same result using `sendc_opName`, first add the above three variables to `MyReplyHandler.h` as private data members of the `AMI_MyInterfaceHandler_i` class and add an accessor method for each. You may also need to add a mutual exclusion (mutex) lock to ensure thread-safe access to these private data members.

```
class AMI_MyInterfaceHandler_i : public virtual POA_AMI_MyInterfaceHandler
{
  public:
    AMI_MyInterfaceHandler_i (void);
    virtual ~AMI_MyInterfaceHandler_i (void);
    virtual void opName(CORBA::Boolean ami_return_val,
                        CORBA::Long a_long,
                        CORBA::Float a_float);
    virtual void opName_excep(Messaging::ExceptionHolder* excep_holder);
    virtual void get_attrib1(CORBA::Short ami_return_val);
    virtual void get_attrib1_excep(Messaging::ExceptionHolder* excep_holder);
```

```
    virtual void set_attrib1 ();
    virtual void set_attrib1_excep(Messaging::ExceptionHolder* excep_holder);
    virtual void get_attrib2(CORBA::Short ami_return_val);
    virtual void get_attrib2_excep(Messaging::ExceptionHolder* excep_holder);

    CORBA::Boolean get_my_return_value (void);
    CORBA::Long get_my_long (void);
    CORBA::Float get_my_float (void);

  private:
    CORBA::Boolean my_return_value_;
    CORBA::Long my_long_;
    CORBA::Float my_float_;

    ACE_Thread_Mutex lock_;
};
```

Now define the reply handler for `opName()` so that the reply variables are loaded into these private data members:

```
ACE_Thread_Mutex AMI_MyInterfaceHandler_i::lock_;

void AMI_MyInterfaceHandler_i::opName(CORBA::Boolean ami_return_val,
                                      CORBA::Long a_long,
                                      CORBA::Float a_float)
{
  ACE_Guard<ACE_Thread_Mutex> guard(lock_);
  my_return_value_ = ami_return_val;
  my_long_ = a_long;
  my_float_ = a_float;
}
```

These data members can be accessed within the client application as follows:

```
#include "MyReplyHandler.h"
#include "MyInterfaceC.h"

// Assume that -1 is not an allowable return value for my_long
// Initialize my_long to -1 so we can check later to see if it has changed
CORBA::Long AMI_MyInterfaceHandler_i::my_long  = -1;

int main (int argc, char* argv[])
{
  try {
    // Get an object reference to MyInterface object.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var obj = orb->string_to_object("file://my_interface.ior");
    MyInterface_var myInterface = MyInterface::_narrow(obj.in());
```

```
    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Create a reply-handler servant.
    PortableServer::Servant_var<AMI_MyInterfaceHandler_i>
      replyHandler_servant = new AMI_MyInterfaceHandler_i();
    PortableServer::ObjectId_var oid =
      poa->activate_object(replyHandler_servant.in());
    CORBA::Object_var handler_obj = poa->id_to_reference(oid.in());
    AMI_MyInterfaceHandler_var replyHandler =
      AMI_MyInterfaceHandler::_narrow(handler_obj.in());

    // Invoke the operation asynchronously.
    myInterface_obj->sendc_opName(replyHandler.in(), 10);

    // do other stuff...

    while(1) {
      // Check to see if reply has been returned.
      if(orb->work_pending()) {
        orb->perform_work();  // Client ORB will invoke reply handler here
        // If the value of my_long has been changed, break out of while loop.
        if (replyHandler_servant->get_my_long() != -1) {
          break;
        }
      }
    }

    orb->destroy();
  }
  catch (CORBA::Exception&) {
    // Handle CORBA exceptions...
  }
}
```

In the above client application, the ORB invokes the reply-handler function `opName()` on the client after the `orb->perform_work()` call is made from the client application. The result of this asynchronous client application differs from the synchronous one only in that the variables are private members of the reply-handler class. The variables will receive the same values in both cases.

6.2.4.3 Exception Reply-Handler Functions

We handle exceptions by adding `try` and `catch` blocks to each reply-handler exception function. Using `MyInterface` again in the example below, we add a `try` block and two `catch` blocks to the `opName` exception reply-handler:

```
void AMI_MyInterfaceHandler_i::opName_excep (
  Messaging::ExceptionHolder* excep_holder)
{
  try {
    excep_holder->raise_exception();
  }
  catch (CORBA::SystemException& e) {
    std::cout << "opName System Exception " << e << std::endl;
  }
  catch (CORBA::UserException& e) {
    std::cout << "opName user exception " << e.usr_exc << std::endl;
  }
}
```

The client ORB calls the `opName_excep()` reply handler when an exception is thrown during the processing of `opName()`. This exception is inserted into the `Messaging::ExceptionHolder` object and passed to the `opName_excep()` reply-handler operation. The only way to gain access to this exception is to call the `Messaging::ExceptionHolder` member function `raise_exception()`. This function demarshals the exception and throws it just as synchronous `opName()` does.

Calling the synchronous `opName()` from within a `try` block would have the same effect, as shown below:

```
#include "MyInterfaceC.h"
#include <iostream>

int main (int argc, char* argv[])
{
  CORBA::Boolean return_value;
  CORBA::short a_short;
  CORBA::long a_long;
  // Get object reference to MyInterface object.
  try {
    return_value = myInterface_obj->opName(10, a_short, a_long);
  }
  catch (CORBA::SystemException& e) {
    std::cout << "opName System Exception exception " << e << std::endl;
  }
  catch (CORBA::UserException& e) {
```

```
      std::cout << "opName user exception " << e.usr_exc << std::endl;
    }
    // do other stuff
  }
```

In the above code, the same exception is thrown and the same exception message is printed as in the asynchronous example.

6.2.4.4 Associating Replies with Requests

Before invoking a `sendc_` operation, a client must generate an object reference for the reply handler. In most cases, this object reference is generated once, then used repeatedly by the client application. However, there are situations where a client application needs to associate a unique identifier with each invocation of a `sendc_` operation so that it can distinguish between these requests at a later time. Also, there may be situations in which the client side needs to instantiate more than one instance of a reply handler.

Using one reply-handler instance to handle all replies coming from multiple server objects of the same type is technically correct, but not necessarily useful, since there is no way to distinguish callbacks resulting from AMI calls to different server objects. Here are some common strategies for addressing this problem:

- **Servant-per-AMI-call strategy:** This strategy involves the instantiation and activation of a separate reply-handler instance for each AMI call. The drawback, of course, is that if there are many simultaneous asynchronous calls, the memory footprint of the client will increase. This strategy is simpler to implement programmatically than the activation-per-AMI strategy and results in less data being marshaled/demarshaled and sent over the wire than the server-differentiated-reply strategy.
- **Activation-per-AMI-call strategy:** One way to distinguish separate AMI calls without using a separate reply-handler instance for each invocation is to explicitly activate the same servant multiple times in the client's POA. Using the `PortableServer::IdAssignmentPolicy` of `MULTIPLE_ID` with a non-root POA, you can activate a servant multiple times, each time with a different user-chosen object id. The reply-handler callback methods examine this object id to determine which request caused the reply.

  Before making AMI calls, the application creates a POA with the `MULTIPLE_ID` and `USER_ID` policies. For each AMI call, the application

creates a special object id and maps the object id to the reply-handler servant using the `PortableServer::activate_object_with_id()` operation. After the client makes the AMI call, the reply arrives at the reply handler. The reply-handler uses the `PortableServer::Current` interface to obtain the object id and associate the reply with the correct request. See *Advanced CORBA Programming with C++* 11.4.2 and 11.4.3 for more information on the `USER_ID` and `MULTIPLE_ID` policies and *Advanced CORBA Programming with C++* 11.7.4 for more information on the `PortableServer::Current`.

Although this approach is more complex to implement, it is more scalable than the Servant-per-AMI-call strategy because it uses a single servant for all asynchronous calls. However, both strategies require an entry-per-AMI-call in the client POA's active object map. One way to reduce this overhead is to use a Servant Locator that activates the client's reply handler on demand, thereby minimizing memory utilization. The activation-per-AMI-call has the advantage over the Server-differentiated-reply strategy of reduced marshaling/demarshaling and less data traveling over the wire.

- **Server-differentiated-reply strategy:** This strategy provides an alternative for differentiating multiple AMI calls, but requires a small modification to the IDL interface. An `out` parameter is added to the function signature for use by the server side to add information that will assist the client-side reply handler in distinguishing which reply goes with which request. Thus, just one servant is needed for distinguishing between all AMI callbacks, and it only needs to be activated once in the client's POA.

  However, compared to allocating a different servant for each AMI call, the use of an `out` parameter is obtrusive and incurs more network overhead to pass the added parameter back to the client. The network overhead can be limited by using the Asynchronous Completion Token (ACT) pattern of adding a small, fixed-size `inout` parameter to the function call. The ACT is first initialized by the client to indicate a particular AMI call and then passed to the server. The server subsequently returns the ACT unchanged as a parameter to the reply handler callback. The reply handler maps the ACT to the associated actions and states necessary to complete the reply processing. If the size of the ACT is

smaller than the `out` parameter described in the earlier part of this strategy, the network bandwidth consumption is reduced somewhat.

The above strategies are further described, with examples, in <`http://www.cs.wustl.edu/~schmidt/PDF/ami1.pdf`>.

6.2.5 The Processing of an AMI sendc_ Operation

When an AMI `sendc_` operation is invoked, the following sequence of steps is initiated (See Figure 6-1):

1. The client invokes an AMI `sendc_` operation on its server object reference.
2. The object reference passes the request to the client ORB.
3. The client ORB:
   - Assigns a unique ID number to the request.
   - Creates a mapping between the ID and the reply-handler object reference.
   - Packages the request message and hands it off to the OS.
4. The client OS sends the request message to the server.
5. The server OS stores the request in the server ORB's message buffer.
6. The server ORB:
   - Gets the request from the message buffer.
   - Unpacks the message.
   - Invokes the synchronous operation on the servant.
7. The servant processes the operation and returns the reply to the ORB.
8. The server ORB packages the reply message and hands it off to the OS.
9. The server OS sends the reply message to the client.
10. The client OS stores the reply in the client ORB's message buffer.

    Because the `sendc_` operation is asynchronous, the client is able to process other tasks while steps 2 through 10 are taking place. At some point after invoking the `sendc_` operation (unless the client is multithreaded and the ORB is running in its own thread, or the reply handler is not located in the client application), the client must invoke either `perform_work()` or `run()` on the ORB to retrieve the reply. The following sequence of steps is then initiated:

11. The client invokes either `run()` or `perform_work()` on the client ORB.
12. The client ORB:
    - Gets the reply from the message buffer.
    - Unpacks the reply message and extracts the ID number.
    - Uses the ID number to locate the designated reply-handler object reference.
    - Invokes the appropriate reply-handler operation on the reply-handler object reference.
13. The client processes the reply-handler operation.

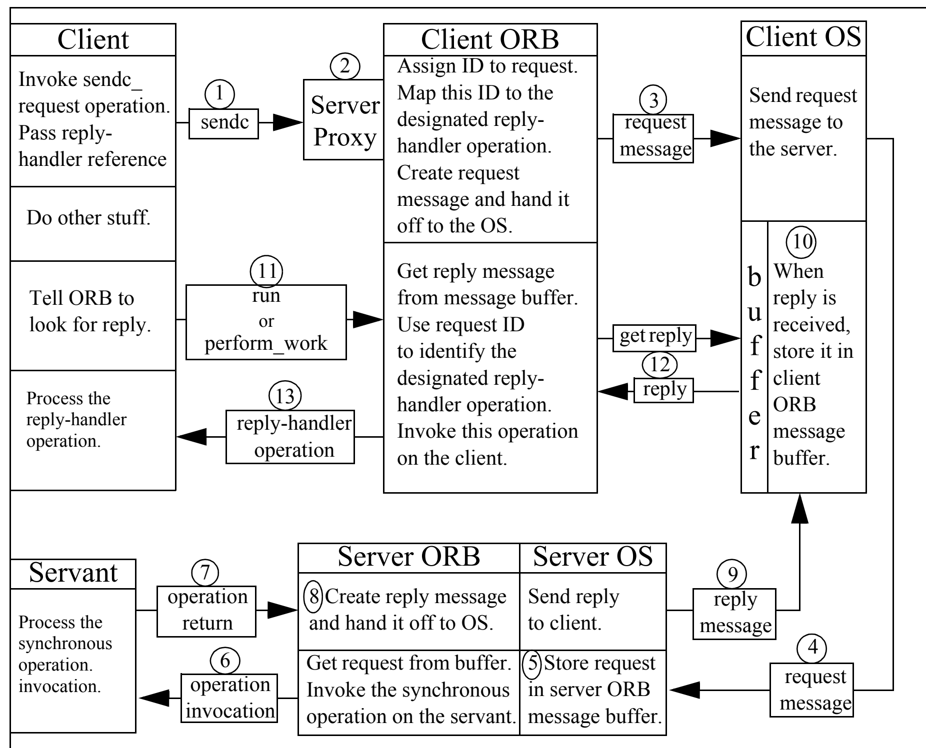


Figure 6-1 AMI Sequence of Steps

6.2.6 AMI Callback Example

Now that you know how to use a `sendc_` operation and write a reply handler, we show a complete example that uses the AMI callback feature of TAO.

The example shown here is based on the Messenger example, first introduced in Chapter 3. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/DevGuideExamples/Messaging/AMIcallback`.

6.2.6.1 IDL Definitions

The IDL file used for the Messenger is shown below:

```
// Messenger.idl

exception MessengerUnableToSendMessage
{
};

interface Messenger
{

  boolean send_message(in string user_name,
                       in string subject,
                       inout string message,
                       out long time_sent)
    raises (MessengerUnableToSendMessage);
};
```

In the above code, the string message parameter is an `inout` parameter since only part of the message may get displayed (on a pager, for example). The returned value in this case is the partial message sent to the user.

The implied-IDL for the reply handler for the Messenger interface is:

```
interface AMI_MessengerHandler
{
  void send_message(in boolean ami_return_val,
                    in string message,
                    in long time_sent);

  void send_message_excep(in Messaging::ExceptionHolder excep_holder);
};
```

The Messenger interface with the implied-IDL `sendc_` operations included is as follows:

```
interface Messenger
{

  boolean send_message(in string user_name,
                       in string subject,
                       inout string message,
                       out long time_sent)
    raises (MessengerUnableToSendMessage);

  void sendc_send_message(in AMI_MessengerHandler ami_handler,
                          in string user_name,
                          in string subject,
                          in string message);
};
```

6.2.6.2 Generating Starter Implementation Code

To minimize the code generated by the IDL compiler, AMI callback stub code is not generated by default. Therefore, we need to inform the IDL compiler to generate this code by passing the `-GC` option. To minimize the amount of code we need to write, we tell the compiler to generate starter implementation code by using the `-GIh` and `-GIs` options. The resulting command line appears as follows:

```
tao_idl -GC -GIh _i.h -GIs _i.cpp Messenger.idl
```

After this command is run, the starter code for the Messenger servant and AMI reply-handler implementations can be found in the files `Messenger_i.h` and `Messenger_i.cpp`. Invoke the IDL compiler manually (instead of through a Makefile) to avoid overwriting implementation code that you have added to the generated starter code.

Since the AMI reply-handler code is for the client side only, we remove the reply-handler starter code from `Messenger_i.h` and `Messenger_i.cpp`, then place it into files named `MessengerHandler.h` and `MessengerHandler.cpp`, respectively.

6.2.6.3 The Messenger Servant Code

To help illustrate the usage of AMI, we add a private data member called `seconds_to_wait_` to the `Messenger_i` class defined in `Messenger_i.h`. By having the server artificially wait `seconds_to_wait_` seconds before it sends the reply, we can mimic the effects of an actual server that may take a while to send a reply.

In addition, `Messenger_i` has a `CORBA::Boolean` data member called `throw_exception_` that the `send_message()` implementation uses to force an exception to be thrown, thus allowing the client-side exception handling code to be exercised.

The constructor for `Messenger_i` accepts arguments to initialize `seconds_to_wait_` and `throw_exception_`. These arguments are set based on command-line arguments passed to the server executable (see `$TAO_ROOT/DevGuideExamples/Messaging/AMIcallback/MessengerServer.cpp` in the TAO source code distribution to see how this is done).

The `send_message()` member function is shown below (the code in bold has been added to the IDL-compiler-generated starter code):

```
CORBA::Boolean Messenger_i::send_message (
  const char * user_name,
  const char * subject,
  char *& message,
  CORBA::Long_out time_sent
  )
{
  if (throw_exception_)
  {
   std::cout << "Throwing MessengerUnableToSendMessage exception." << std::endl;
    throw MessengerUnableToSendMessage();
  }

  std::cout << "Write a letter to " << user_name << " as follows:" << std::endl;
  std::cout << "Subject: " << subject << std::endl;
  std::cout << "Dear " << user_name << ',' << std::endl;
  std::cout << message << std::endl;

  if (seconds_to_wait_ > 0)
  {
   std::cout << "Waiting for " << seconds_to_wait_ << " seconds..." << std::flush;
    ACE_OS::sleep(seconds_to_wait_);
    std::cout << " Done waiting" << std::endl;
  }
```

```
  // Record the time the message was sent
  time_sent = ACE_OS::gettimeofday().sec();

  // We will assume the message has been sent, so return true
  return true;
}
```

6.2.6.4 The Reply-Handler Class Definition

For this example, the `MessengerHandler` reply-handler echoes the server's response to standard output, including the time the message was sent. It shuts down the ORB after one message.

The reply-handler functions related to the `send_message()` operation are shown below:

```
void MessengerHandler::send_message (
   CORBA::Boolean ami_return_val,
   const char * message,
   CORBA::Long time
   )
{
  if (ami_return_val)
  {
    time_ = time;
    time_t t = time_;
    const char * time_str = ACE_OS::ctime(&t);
    if (time_str != 0) {
      std::cout << std::endl << "Message sent at " << time_str << std::endl;
    }
    std::cout << "Content of message: " << message << std::endl;
  }
  else
  {
    std::cerr << "Error: Message was not sent." << std::endl;
  }
  // Our simple test just shuts down after sending one message.
  orb_->shutdown(0);
}

void MessengerHandler::send_message_excep (
  Messaging::ExceptionHolder* excep_holder
  )
{
  // We'll print an error message and shut down the orb
  try
  {
    excep_holder->raise_exception();
  }
```

```
  catch (CORBA::Exception& ex)
  {
    std::cerr << "A CORBA Exception was thrown: " << ex << std::endl;
  }
  orb_->shutdown(0);
}
```

6.2.6.5 The Client Code

Since the reply handler will be called back by the ORB when the reply arrives from the server, it needs to be registered with the POA as a servant just like servants are registered in server code. The client code is:

```
#include "MessengerC.h"
#include "MessengerHandler.h"

int
main(int argc, char * argv[])
{
  try {

    // assume any command line parameter means we want an automated test.
    bool automated = argc > 1;

    // Initialize orb
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    CORBA::Object_var obj = orb->string_to_object("file://MessengerServer.ior");
    if (CORBA::is_nil(obj.in())) {
      std::cerr << "Nil Messenger reference" << std::endl;
      return 1;
    }

    // Narrow
    Messenger_var messenger = Messenger::_narrow(obj.in());
    if (CORBA::is_nil(messenger.in())) {
      std::cerr << "Argument is not a Messenger reference" << std::endl;
      return 1;
    }

    // Get reference to Root POA.
    obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

    // Activate POA manager
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Register an AMI handler for the Messenger interface
```

```
PortableServer::Servant_var<MessengerHandler> servant =
  new MessengerHandler(orb.in());
PortableServer::ObjectId_var oid = poa->activate_object(servant.in());
obj = poa->id_to_reference(oid.in());
AMI_MessengerHandler_var handler = AMI_MessengerHandler::_narrow(obj.in());
```

For our example, we will get the necessary information needed to send a message to a particular user from standard input.

```
CORBA::String_var user = CORBA::string_alloc(81);
CORBA::String_var subject = CORBA::string_alloc(81);
CORBA::String_var message = CORBA::string_alloc(81);

if (! automated) {
  std::cout << "Enter user name -->";
  std::cin.getline(user, 81);

  std::cout << "Enter subject -->";
  std::cin.getline(subject, 81);

  std::cout << "Enter message -->";
  std::cin.getline(message, 81);
} else {
  user = CORBA::string_dup("TestUser");
  subject = CORBA::string_dup("TestSubject");
  message = CORBA::string_dup("Have a nice day.");
}

// Record the time the request was made.
ACE_Time_Value time_sent = ACE_OS::gettimeofday();

messenger->sendc_send_message(handler.in(),
                              user.in(),
                              subject.in(),
                              message.in());
```

Now we will run an event loop that runs the ORB in a non-blocking fashion. Doing so allows us to provide feedback to the user when control is handed back to the main thread.

Note *See Advanced CORBA Programming with C++, 11.11.2, for details on performing non-blocking event handling with CORBA.*

```
// Do some work to prove that we can send the message asynchronously, then come
// back later and retrieve the results.
```

```
    for (int i = 0; i < 10; ++i) {
      ACE_OS::printf(".");
      ACE_OS::sleep(ACE_Time_Value(0, 10 * 1000));
    }

    // Our simple servant will exit as soon as it receives the results.
    orb->run();

    if (servant->message_was_sent())
    {
      // Note : We cannot use the time sent by the server to compare with
      // the time value here in the client because the server machine's
      // clock may not be synchronized with the client's clock.

      ACE_Time_Value delay = ACE_OS::gettimeofday() - time_sent;
      std::cout << std::endl << "Reply Delay = "
                << delay.msec() << "ms" << std::endl;
    }

    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA::Exception: " << ex << std::endl;
    return 1;
  }

  return 0;
}
```

6.2.6.6 Building Applications that use AMI

TAO's support of AMI is implemented in the `TAO_Messaging` library. Thus, applications that use AMI must link with this library. MPC projects for applications that use AMI can simply inherit from the `messaging` and `ami` base projects. For example, below is the MPC file for the AMI callback example in `$TAO_ROOT/DevGuideExamples/Messaging/AMIcallback`:

```
project(*Server): messaging, taoexe, portableserver, ami {
  Source_Files {
    Messenger_i.cpp
    MessengerServer.cpp
  }
}

project(*Client): messaging, taoexe, portableserver, ami {
  Source_Files {
    MessengerHandler.cpp
```

```
    MessengerClient.cpp
  }
}
```

For more information on MPC, see
`<http://www.ociweb.com/products/MPC>`.

6.2.7 Controlling the Delivery of AMI-based Requests

TAO's implementation of AMI permits the Messaging `SyncScope` policy to be applied to the delivery of requests that use AMI. This feature allows fine-grained control over the ORB's return of control back to the client application code. See 6.3.4 for more information on the Messaging `SyncScope` policy. In addition, the TAO-specific buffering constraint policy can be applied to specify the conditions under which a queue of requests should be buffered and transmitted. See 6.3.5 for more information on TAO's buffering constraint policy.

6.3 Quality of Service Policies

Quality of Service (QoS) is a general concept that is used to specify the behavior of a service. Programming service behavior by means of QoS settings offers the advantage that application developers only need to specify *what* they want rather than *how* it should be achieved.

Generally speaking, quality of service comprises several QoS *policies*. Each policy is an independent description that associates a name with a value. Describing QoS by means of a list of independent QoS policies gives rise to greater flexibility in application design.

The CORBA Messaging specification defines mechanisms by which clients and servers can set required and supported qualities of service with respect to requests. It describes a standard QoS framework within which CORBA services can define their service-specific qualities. In this framework, all QoS settings are local interfaces derived from `CORBA::Policy`. Many of these QoS policies are defined in the `Messaging` module in IDL. TAO defines some additional QoS policies, also derived from `CORBA::Policy`, that fit within the CORBA Messaging QoS framework.

The following sections describe the messaging QoS policies supported by TAO and how to use them.

6.3.1 Policy Management

6.3.1.1 Creating Policies

As stated above, all CORBA Messaging QoS policies inherit from `CORBA::Policy` (the same base interface used to specify POA policies). `CORBA::ORB` has a generic factory operation, `create_policy()`, that can be used to create new policy objects. This operation is defined in the following IDL:

```
module CORBA {
  typedef unsigned long PolicyType;
  interface Policy {};

  typedef short PolicyErrorCode;
  const PolicyErrorCode BAD_POLICY = 0;
  const PolicyErrorCode UNSUPPORTED_POLICY = 1;
  const PolicyErrorCode BAD_POLICY_TYPE = 2;
  const PolicyErrorCode BAD_POLICY_VALUE = 3;
  const PolicyErrorCode UNSUPPORTED_POLICY_VALUE = 4;
  exception PolicyError {PolicyErrorCode reason;};

  interface ORB {
    Policy create_policy(in PolicyType type, in any val) raises(PolicyError);
  };
};
```

Each messaging QoS policy is assigned a unique `PolicyType`. For example, the policy type for the relative round-trip timeout policy, described in 6.3.2, is defined as:

```
module Messaging {
  const CORBA::PolicyType RELATIVE_RT_TIMEOUT_POLICY_TYPE = 32;
};
```

The `val` parameter passed to `create_policy()` is a `CORBA::Any` that contains the desired value for the policy. If the `CORBA::Any` does not contain an acceptable value or if the `CORBA::PolicyType` is not supported by the ORB, a `CORBA::PolicyError` exception is raised.

Before policies can be applied, they must be added to a `CORBA::PolicyList`. For example:

```
  // Create a policy and add it to a CORBA::PolicyList.
```

```
CORBA::Any policy_value_as_any;
// initialize "policy_value_as_any" with a value
CORBA::PolicyList policy_list;
policy_list.length(1);
policy_list[0] =
  orb->create_policy (SOME_POLICY_TYPE, policy_value_as_any);
```

On the client side, policies are applied to various objects, such as the ORB, the current thread of execution, or a specific object reference. On the server side, policies are applied to the POA.

6.3.1.2 Client Side Policy Management

Messaging QoS policies can be applied on the client side at three different scoping levels. This permits you to work with a level of granularity that is appropriate for your application. These levels are as follows:

1. **The ORB level.** Policies applied at the ORB level will apply to all requests delivered by the specified ORB. The `CORBA::PolicyManager` is used to set policies at this level. For example:

```
CORBA::Object_var obj = orb->resolve_initial_references("ORBPolicyManager");
CORBA::PolicyManager_var policy_manager =
  CORBA::PolicyManager::_narrow(obj.in());
policy_manager->set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);
```

2. **The thread level.** Using thread-level policies allows quality-of-service values to be applied to operations invoked from a certain thread. The `CORBA::PolicyCurrent` object is used to set policies at this level. For example:

```
CORBA::Object_var obj = orb->resolve_initial_references("PolicyCurrent");
CORBA::PolicyCurrent_var policy_current =
  CORBA::PolicyCurrent::_narrow (obj.in());
policy_current->set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);
```

3. **The object reference level.** For the most fine-grained level of control, quality-of-service policies can be set on a per-object-reference basis. Assuming we have some object reference `obj`, we can apply policies to the object reference as follows:

```
CORBA::Object_var new_obj =
  obj->_set_policy_overrides (policy_list, CORBA::SET_OVERRIDE);
```

Note that `CORBA::Object::_set_policy_overrides()` returns an object reference that you must narrow to a specific interface type before invoking operations on it. It does *not* modify the object reference upon which it is called. In the above example, the `new_obj` object reference contains the new `policy_list` policies.

Policy overrides applied at the object-reference level take precedence over those applied at the thread or ORB level. Likewise, policy overrides applied at the thread level take precedence over those applied at the ORB level.

Whether new policy settings are *added to* or *replace* existing policy settings is controlled by the second parameter in `set_policy_overrides()`. If the second parameter is `CORBA::SET_OVERRIDE`, the policies in the policy list completely replace the existing policies set at the relevant level of granularity. If the second parameter is `CORBA::ADD_OVERRIDE`, the new policies are added to the existing policies, unless a given policy in effect has the same `PolicyType` as one of the policies in the list, in which case the new policy replaces the existing policy.

6.3.1.3 Server Side Policy Management

On the server side, messaging policies are associated with a POA. Policies that are applicable to server-side behavior can be passed via a `CORBA::PolicyList` to the `POA::create_POA()` operation. Request processing through a POA is subject to the policies applied to that POA at its creation. Some policies applied to a POA are exported to clients via object references created through that POA.

For more information on POA creation and POA policies, see *Advanced CORBA Programming with C++*, Chapter 11.

6.3.1.4 Destroying Policies

Once the policies have been applied at either the client side or the server side, the policy objects themselves should be destroyed using the `CORBA::Policy::destroy()` operation. For example:

```
// Destroy the Policy objects.
for (CORBA::ULong i = 0; i < policy_list.length(); ++i) {
  policy_list[i]->destroy ();
}
policy_list.length(0);
```

Alternately, you can also use the `TAO::Utils::PolicyList_Destroyer`. This TAO-specific class is used in place of `CORBA::PolicyList` to hold your policies. Upon destruction, it automatically calls `destroy()` on each policy in it. To use it, include `tao/Utils/PolicyList_Destroyer.h` in the files where you are using policies, change your policy list types, and inherit your MPC projects from the `utils` base project.

6.3.2 Request and Reply Timeouts

The CORBA Messaging specification defines a relative round-trip timeout policy. Relative round-trip timeouts are used to limit the total amount of time spent completing the following steps:

1. The client attempts to make a connection with a server.
2. The client passes a request to the server.
3. The client waits for a reply from the server.

At each step, the time spent since the start of the request is checked against a user-specified timeout value. If the time exceeds this timeout value a `CORBA::TIMEOUT` exception is raised. Specifying a relative round-trip timeout value is useful in real-time and fault-tolerant systems, since the client can take appropriate action if the server becomes unresponsive or cannot complete a request within a specified time interval.

Only clients are impacted by the use of the relative round-trip timeout policy; no timing requirements are passed to the server. If a `CORBA::TIMEOUT` exception is raised and a server reply arrives sometime afterward, the reply is simply ignored.

The `PolicyType` for the relative round-trip timeout policy is `Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE`. Its value is a `CORBA::Any` containing a `TimeBase::TimeT` as defined in the CORBA Time Service specification. The timeout value is a 64-bit value interpreted as hundreds of nanoseconds. If either the `CORBA::Any` reference does not contain a `TimeBase::TimeT` or the `CORBA::PolicyType` is not supported, then a `CORBA::PolicyError` exception is raised.

The following code shows how to create a relative round-trip timeout policy with a timeout value of one millisecond, and apply it at the ORB level. A complete example showing how to use this policy is included in the TAO source code distribution in the directory `$TAO_ROOT/DevGuideExamples/Messaging/RelativeRoundtripTimeout`.

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Set the policy value to 1 millisecond (10 * 1000 msec/usec).
TimeBase::TimeT relative_rt_timeout = 10000;  // 1 millisecond
CORBA::Any relative_rt_timeout_as_any;
relative_rt_timeout_as_any <<= relative_rt_timeout;

// Create the policy and add it to a CORBA::PolicyList.
CORBA::PolicyList policy_list;
policy_list.length(1);
policy_list[0] =
  orb->create_policy (Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE,
  relative_rt_timeout_as_any);

// Apply the policy at the ORB level.
CORBA::Object_var obj = orb->resolve_initial_references("ORBPolicyManager");
CORBA::PolicyManager_var policy_manager =
  CORBA::PolicyManager::_narrow(obj.in());
policy_manager->set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);

// Destroy the Policy objects.
for (CORBA::ULong i = 0; i < policy_list.length(); ++i) {
  policy_list[i]->destroy ();
}
policy_list.length(0);
```

A client that invokes a request on a server while a relative timeout policy is in effect may receive an exception of type `CORBA::TIMEOUT`. This exception is generated by the underlying invocation implementation upon expiration of the specified time limit. Prior to the exception being thrown, the request is cancelled if a response has not yet been received from the server.

Note *In addition to relative round-trip timeouts, five additional timeout policies are defined in the CORBA Messaging specification. These additional timeout policies apply to request start time, request end time, reply start time, reply end time, and relative request delivery time. None of these is available in TAO 1.6a. However, TAO 1.6a does provide a TAO-specific connection timeout policy, described in 6.3.3.*

6.3.3 Connection Timeouts

In addition to the relative round-trip timeout policy described in 6.3.2, TAO provides a policy to control connection timeouts. The connection timeout

policy is used to limit the total amount of time a client spends establishing a connection with a server. If the connection time exceeds the value specified in the policy, a `CORBA::TIMEOUT` exception is raised. Specifying a connection timeout value is useful in real-time and fault-tolerant systems, since the client can take appropriate action if the server becomes unresponsive or if a network interruption occurs.

The TAO connection-timeout-policy local interface is defined in `$TAO_ROOT/tao/Messaging/TAO_Ext.pidl` as follows:

```
#pragma prefix "tao"

module TAO
{
  const CORBA::PolicyType CONNECTION_TIMEOUT_POLICY_TYPE = 0x54410008;

  local interface ConnectionTimeoutPolicy : CORBA::Policy {
    readonly attribute TimeBase::TimeT relative_expiry;
  };
};
```

The following example shows how to set the `ConnectionTimeoutPolicy` to 200 milliseconds on an object reference. The example uses the CORBA Messaging `_validate_connection()` operation to explicitly open the connection and verify that the connection can be made within the specified timeout.

```
try {

  // Initialize the ORB.
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

  // Set the policy value (1.0e-3 * 1.0e7 is 1 millisecond).
  TimeBase::TimeT connection_timeout = 1.0e-3 * 1.0e7 * 200;
  CORBA::Any connection_timeout_as_any;
  connection_timeout_as_any <<= connection_timeout;

  // Create the policy and add it to a CORBA::PolicyList.
  CORBA::PolicyList policy_list;
  policy_list.length(1);
  policy_list[0] =
    orb->create_policy (TAO::CONNECTION_TIMEOUT_POLICY_TYPE,
                        connection_timeout_as_any);

  // Obtain an object reference.
  CORBA::Object_var obj = orb->string_to_object("file://MessengerServer.ior");
```

```
  // Apply the policy to the object reference; returns a new object reference.
  CORBA::Object_var new_obj =
    obj->_set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);

  // Destroy the Policy objects.
  for (CORBA::ULong i = 0; i < policy_list.length(); ++i) {
    policy_list[i]->destroy ();
  }
  policy_list.length(0);

  // Explicitly bind a connection to the server (may timeout).
  CORBA::PolicyList_var inconsistent_policies;
  CORBA::Boolean status =
    new_obj->_validate_connection (inconsistent_policies.out());

  // _narrow() and use the new_obj object reference as usual...

}
catch (CORBA::TIMEOUT&) {
  // The connection attempt timed out.
}
catch (CORBA::Exception&) {
  // Some other CORBA exception was raised.
}
```

You can see another example of using the connection timeout policy in `$TAO_ROOT/tests/Connection_Timeout`. See the `README` file in that directory for more information.

Note *The connection timeout policy is specific to TAO. It is not part of the CORBA Messaging specification.*

6.3.4 Reliable Oneway calls using the SyncScope Policy

The CORBA Messaging specification defines a policy called `SyncScope`. This policy permits clients to specify at what stage during a oneway message invocation control is returned back to the client application code. The specification defines four possible values that can be used for the `SyncScope` policy.

- **SYNC_NONE** Using this policy value causes the client ORB to return control to the client application before the request is passed to the transport protocol. For this case the client is guaranteed not to block

during a request invocation. This policy value provides the lowest guarantee of delivery.

- **SYNC_WITH_TRANSPORT** Setting the `SyncScope` policy to this value causes control to return to the client application code after the transport has accepted the request. Use of this policy value does not guarantee that the request has been delivered to the server. For example, if IIOP is being used, then limited TCP buffer space may cause unbounded delays in transmission. `SYNC_WITH_TRANSPORT` is the default `SyncScope` policy value in TAO.
- **SYNC_WITH_SERVER** When the `SyncScope` policy is set to this value, the server sends its reply before invoking the target servant. This setting is useful if the reliability of the network is of concern and the time spent executing the servant code dominates the time involved in waiting for a reply. The stage at which the server sends back an acknowledgement is right after the use of any servant manager, but before the target servant is invoked.
- **SYNC_WITH_TARGET** This policy value has the same effect as turning a oneway call into a synchronous call by removing the oneway qualifier in the operation signature. Control is returned to a client application only after the reply has been received from the target servant. Use this policy value if you need complete confidence that a reply has been received from the server and only if it is appropriate for the client application to block while the target servant is preparing a reply.

In addition, TAO defines a TAO-specific value for this policy:

- **SYNC_DELAYED_BUFFERING** This policy value is a variant of SYNC_NONE. See 6.3.5, for discussion of the `BufferingConstraint` policy and its interaction with this value.

Both the **SYNC_NONE** and **SYNC_WITH_TRANSPORT** policy values are valid interpretations of the original oneway semantics defined by the CORBA specification.

The following example shows how to set the `SyncScope` policy such that oneway invocations do not return control to the client until the client has received an acknowledgement from the server that the message has been delivered to the servant. In this example, we apply the policy at the ORB level:

```
// Initialize the ORB.
```

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Set the policy value.
Messaging::SyncScope sync_with_target = Messaging::SYNC_WITH_TARGET;
CORBA::Any sync_with_target_any;
sync_with_target_any <<= sync_with_target;

// Create the policy and add it to a CORBA::PolicyList.
CORBA::PolicyList policy_list;
policy_list.length(1);
policy_list[0] =
 orb->create_policy (Messaging::SYNC_SCOPE_POLICY_TYPE, sync_with_target_any);

// Apply the policy at the ORB level.
CORBA::Object_var obj = orb->resolve_initial_references ("ORBPolicyManager");
CORBA::PolicyManager_var policy_manager =
  CORBA::PolicyManager::_narrow(obj.in());
policy_manager->set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);

// Destroy the Policy objects.
for (CORBA::ULong i = 0; i < policy_list.length(); ++i) {
  policy_list[i]->destroy ();
}
policy_list.length(0);

// ... rest of application ...
```

Figure 6-2 shows the effects of the various settings of the `SyncScope` policy on `oneway` invocations.

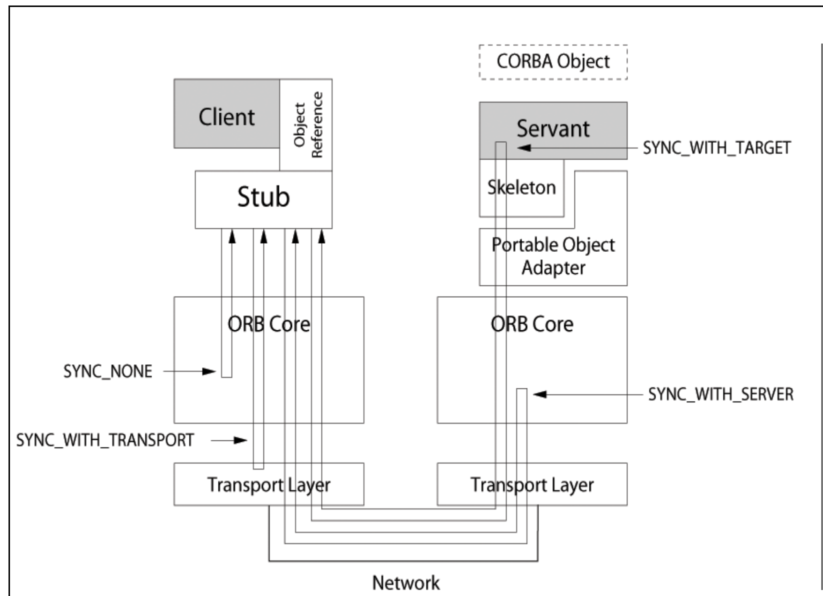
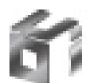


Figure 6-2 Effect of SyncScope Policy on Oneway Invocations

6.3.5 Buffered Oneway and Asynchronous Requests

TAO provides a `BufferingConstraint` policy to control the dispatching of oneway and asynchronous requests from a client. This policy is not part of the CORBA Messaging specification, but uses the same QoS policy framework as the other policies described in this chapter. Using the `BufferingConstraint` policy, it is possible to specify that oneway and asynchronous requests should be buffered in the client's ORB and dispatched only when one or more of the following conditions, controllable via the policy value, has been met:

- A specified timeout value has expired.
- A specified maximum message byte count has been reached.
- A specified maximum message count has been reached.
- An explicit buffer flush has been issued.
- The ORB has been shut down.



By default, oneway and asynchronous requests are not buffered. To set the BufferingConstraint policy, you create and initialize a structure of type TAO::BufferingConstraint to describe how request buffering is to be performed. When specifying this policy, applications are also required to set the SyncScope policy, described in 6.3.4, to either Messaging::SYNC_NONE or TAO::SYNC_DELAYED_BUFFERING. The only difference between these two values is that when the buffer is empty, the SYNC_DELAYED_BUFFERING value will attempt a send before queueing. The SYNC_NONE value will always queue the message in the buffer.

The BufferingConstraintPolicy interface, BufferingConstraint structure, and related constants are defined in $TAO_ROOT/tao/TAO.pidl as follows:

```
#pragma prefix "tao"

module TAO {

  typedef unsigned short BufferingConstraintMode;
  const BufferingConstraintMode BUFFER_FLUSH         = 0x00;

  // Note that timeout, message_count, and message_bytes can be or’d.
  const BufferingConstraintMode BUFFER_TIMEOUT       = 0x01;
  const BufferingConstraintMode BUFFER_MESSAGE_COUNT = 0x02;
  const BufferingConstraintMode BUFFER_MESSAGE_BYTES = 0x04;

  struct BufferingConstraint
  {
    BufferingConstraintMode mode;
    TimeBase::TimeT timeout;
    unsigned long message_count;
    unsigned long message_bytes;
  };

  const CORBA::PolicyType BUFFERING_CONSTRAINT_POLICY_TYPE = 0x54410001;
  local interface BufferingConstraintPolicy : CORBA::Policy
  {
    readonly attribute BufferingConstraint buffering_constraint;
  };
};
```

To initialize the mode data member of the BufferingConstraint structure, compute the bitwise OR of one or more TAO::BufferingConstraintMode constants (e.g., TAO::BUFFER_TIMEOUT|TAO::BUFFER_MESSAGE_COUNT).

Depending upon the value of `mode`, one or more of the `timeout`, `message_count`, or `message_bytes` data members should also be set.

The following example shows how to set the `BufferingConstraint` policy such that oneway and asynchronous invocations are buffered in the client ORB until a particular message count or total buffer size has been reached. In this example, we apply the policy at the ORB level:

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Set the SyncScope policy for oneways to SYNC_NONE.
Messaging::SyncScope sync_none = Messaging::SYNC_NONE;
CORBA::Any sync_none_any;
sync_none_any <<= sync_none;

// Set the BufferingConstraint policy to buffer up to 5 requests
// or until a total of 4K bytes have been buffered.
TAO::BufferingConstraint buffering_constraint;
buffering_constraint.mode =
  TAO::BUFFER_MESSAGE_COUNT | TAO::BUFFER_MESSAGE_BYTES;
buffering_constraint.message_count = 5;
buffering_constraint.message_bytes = 4096;
buffering_constraint.timeout = 0;
CORBA::Any buffering_constraint_any;
buffering_constraint_any <<= buffering_constraint;

// Create the policies and add them to a CORBA::PolicyList.
CORBA::PolicyList policy_list;
policy_list.length(2);
policy_list[0] =
  orb->create_policy (Messaging::SYNC_SCOPE_POLICY_TYPE, sync_none_any);
policy_list[1] = orb->create_policy (TAO::BUFFERING_CONSTRAINT_POLICY_TYPE,
                                     buffering_constraint_any);

// Apply the policies at the ORB level.
CORBA::Object_var obj = orb->resolve_initial_references ("ORBPolicyManager");
CORBA::PolicyManager_var policy_manager =
  CORBA::PolicyManager::_narrow(obj.in());
policy_manager->set_policy_overrides (policy_list, CORBA::ADD_OVERRIDE);
```

You can see another example using the `BufferingConstraint` policy with `oneway` requests in `$TAO_ROOT/tests/Oneway_Buffering`. You can see an example of using the `BufferingConstraint` policy with asynchronous (AMI) requests in `$TAO_ROOT/tests/AMI_Buffering`. See the `README` file in each of those directories for more information.

Note *The buffering constraint policy is specific to TAO. It is not part of the CORBA Messaging specification.*

6.3.5.1 Building Applications that use Messaging QoS

TAO's support of Messaging QoS is implemented in the `TAO_Messaging` library. Thus, applications that use these features must link with this library. MPC projects for applications that use Messaging QoS can simply inherit from the `messaging` base project. For example, below is the MPC file for the Timeout test in `$TAO_ROOT/tests/Timeout` that uses the relative round-trip timeout policy:

```
project(*Server): taoexe, portableserver {
  Source_Files {
    test_i.cpp
    server.cpp
  }
}

project(*Client): messaging, taoexe, portableserver {
  requires += corba_messaging

  Source_Files {
    testC.cpp
    client.cpp
  }
}
```

For more information on MPC, see Chapter 4.

6.4 Bi-Directional GIOP

Bi-directional GIOP provides a solution to the problem of invoking callback operations on clients behind a firewall. Imagine you have a client application that resides on hosts that are inside firewalls. That client contacts a server outside of the firewall and provides a callback object for the server. When the server invokes upon the client callback object, the server attempts to open a new connection back to the client application. Unless the firewall at every installation of the client is configured to allow access by the server, the callback invocation fails. Configuring the firewall to allow the server to open

a callback connection may impose a significant installation cost and/or violate site security policies.

Bi-directional GIOP solves this problem by allowing the callback invocation to use the connection that already exists between the client and the server, which is the connection that was used to transmit the client's initial request to the server. The server's callback invocation doesn't need to open a new connection to the client; thus, a firewall does not block the callback.

Connection management restrictions imposed in GIOP versions 1.0 and 1.1 state that only clients can send requests, and only servers can respond to requests. (By definition, an application or process that initiates a connection is a client, and an application or process that accepts connections is a server. Connection management, however, is orthogonal to the sending of requests and replies.) This restriction can be overcome in GIOP v1.2 by specifying a bi-directional policy of `BOTH`. This policy allows the server to invoke the client's callback operations, and the client to respond to these invocations, on the same connection that the client established initially. The client and server use only one connection instead of two.

The code in `$TAO_ROOT/DevGuideExamples/BiDirectionalGIOP` provides an example of a bi-directional connection between a client and server. The client creates a callback object and passes its object reference to the server so that the server may invoke an operation on the callback object. Bi-directional GIOP allows the server to invoke a callback operation on the client without creating another connection to the client. There is also a test case in `$TAO_ROOT/tests/BiDirectional` that behaves in a similar fashion.

To create a bi-directional connection, *both* the client and server must specify a BidirectionalPolicyValue of `BOTH` when creating their POAs. The default policy is `NORMAL`.

6.4.1 Bi-Directional GIOP Example

The following example shows how to set the bi-directional GIOP policy on a new POA:

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get the RootPOA and its POAManager.
CORBA::Object_var poa_obj = orb->resolve_initial_references ("RootPOA");
```

```
PortableServer::POA_var root_poa = PortableServer::POA::_narrow (poa_obj.in());
PortableServer::POAManager_var poa_manager = root_poa->the_POAManager ();

// Create policies for the child POA to be created.
CORBA::PolicyList policy_list;
policy_list.length(1);

CORBA::Any bi_dir_policy_as_any;
bi_dir_policy_as_any <<= BiDirPolicy::BOTH;
policy_list[0] =
  orb->create_policy (BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE,
  bi_dir_policy_as_any);

// Create a POA as a child of RootPOA with the above policies. This POA
// will receive requests on the same connection on which it sent the request.
PortableServer::POA_var child_poa =
  root_poa->create_POA ("biDirPOA", poa_manager.in(), policy_list);

// Destroy the Policy objects.
for (CORBA::ULong i = 0; i < policy_list.length(); ++i) {
  policy_list[i]->destroy ();
}
policy_list.length(0);

// Activate both POAs.
poa_manager->activate ();

// ... rest of application ...
```

6.4.2 Building Applications that use Bi-Directional GIOP

TAO's support of bi-directional GIOP is implemented in the `TAO_BiDirGIOP` library. Thus, applications that use bi-directional GIOP features must link with this library. MPC projects for applications that use bi-directional GIOP can simply inherit from the `bidir_giop` base project. For example, below is the MPC file for the bi-directional GIOP test in `$TAO_ROOT/tests/BiDirectional`:

```
project(*idl): taoidldefaults {
  IDL_Files {
    test.idl
  }
  custom_only = 1;
}

project(*Server): taoserver, anytypecode, avoids_minimum_corba, bidir_giop,
avoids_corba_e_micro {
```

```
  after += *idl
  Source_Files {
    testC.cpp
    testS.cpp
    test_i.cpp
    server.cpp
  }
  IDL_Files {
  }
}

project(*Client): taoserver, anytypecode, avoids_minimum_corba, bidir_giop,
avoids_corba_e_micro {
  exename = client
  after += *idl
  Source_Files {
    testC.cpp
    testS.cpp
    test_i.cpp
    client.cpp
  }
  IDL_Files {
  }
}
```

The `avoids_minimum_corba` *base project, from which the above projects inherit, indicates that these projects will* ***not*** *build if minimum_corba=1.*

For more information on MPC, see
`<http://www.ociweb.com/products/MPC>`.

Note *The current implementation of TAO sets the BiDirPolicy at the ORB level rather than in each POA. Thus, every connection in that ORB will have the* `BiDirPolicy::BOTH` *value.*

Note *Warning: There are security issues involved in using Bi-Directional GIOP. See the last paragraph of section 9.8, "Bi-Directional GIOP", in Part 2 of the CORBA specification (OMG Document formal/08-01-06) for a complete description. See section 15.8.1.1, "IIOP/SSL Considerations," for issues related to using Bi-Directional GIOP over IIOP/SSL.*

6.5 Endpoint Policy

The Endpoint Policy is a TAO-specific policy that enables applications running on multi-homed hosts to limit the endpoints specified for some objects. This allows an ORB to listen to both public and private interfaces, and then publish object references that are only public or private. For instance, a process may provide public access to some business logic object, but private access, perhaps only on localhost, to an administrative object.

The Endpoint policy makes use of a new policy scope, being applied to a `POA_Manager` rather than to an Object, POA, or ORB. The policy will affect all POAs associated with the constrained `POA_Manager`.

The Endpoint Policy is located in the `TAO_EndpointPolicy` library. The source code is located in `$TAO_ROOT/tao/EndpointPolicy`. MPC-based applications that use this policy should derive their server projects from the `endpointpolicy` base project.

6.5.1 Using the Endpoint Policy

The Endpoint Policy object is constructed with a sequence of endpoint objects that identify the different endpoints that should be used. The policy is used to create a POA Manager via the POA Manager Factory. When POAs are created with that POA manager, any objects activated within those POAs restrict their object references to the endpoints that match those found in the corresponding Endpoint Policy object.

The following server example code is adapted from the TAO test in `$TAO_ROOT/tests/POA/EndpointPolicy`. First, we need to include the header files for the Endpoint Policy and the IIOP Endpoint Value:

```
#include "tao/EndpointPolicy/EndpointPolicy.h"
#include "tao/EndpointPolicy/IIOPEndpointValue_i.h"
```

After initializing the ORB, we'll initialize an Endpoint List with an IIOP endpoint value containing a host name of "localhost" and a port of 1234:

```
  EndpointPolicy::EndpointList list;
  list.length (1);
  list[0] = new IIOPEndpointValue_i ("localhost", 1234);
```

Once we have finished populating the Endpoint List, we can use it to create a an Endpoint Policy:

```
CORBA::PolicyList policies;
policies.length (1);

CORBA::Any policy_value;
policy_value <<= list;
policies[0] = orb->create_policy (EndpointPolicy::ENDPOINT_POLICY_TYPE,
                                  policy_value);
```

Now we are ready to create a new POA Manager. In order to create a POA Manager with new policies, we'll need to get the POA Manager Factory from an existing POA (such as the Root POA).

```
PortableServer::POAManagerFactory_var poa_manager_factory;
poa_manager_factory = root_poa->the_POAManagerFactory ();

local_pm = poa_manager_factory->create_POAManager ("localPOAManager",
                                                  policies);
```

When creating new POAs that we want to apply this policy to, we need to pass out POA Manager to the `create_POA()` operation.

```
    PortableServer::POA_var local_poa = root_poa->create_POA ("localPOA",
                                                              local_pm.in (),
                                                              poa_policies);
```

We start the server with two endpoints, one on the external network and one with localhost:

```
server -ORBListenEndpoints iiop://localhost:1234 \
       -ORBListenEndpoints iiop://zippy:9999
```

Any CORBA objects activated with the Root POA includes both of these endpoints in their object references. Any CORBA objects activated with our local POA only includes the localhost endpoint and are only able to be used by clients located on the same host.

Endpoints are matched in their final form. This means that if an IIOP Endpoint makes use of the `hostname_in_ior` attribute, that is the name the policy must match.

6.5.2 Limitations

Currently, only IIOP endpoints are supported. This means that only IIOP endpoints can be added to the policy and selected for use. Any non-IIOP endpoints are not matched and are never selected when the Endpoint Policy is used.

6.6 Specifying Differentiated Services with TAO

Some environments provide support for differentiated classes of network service, and allow applications to specify their network quality of service needs. A common mechanism for providing differentiated classes of service on IP networks is the *Differentiated Services* (diffserv) architecture defined by the Internet Engineering Task Force (IETF) Diffserv Working Group. In the diffserv architecture, applications encode a particular six-bit pattern into a field, called the *DS field*, of the IP packet header, thereby marking a packet to receive a particular forwarding treatment, or *per-hop behavior* (PHB), at each network node. The Diffserv Working Group has standardized a small number of specific per-hop behaviors and a recommended bit pattern, or *codepoint*, for each one. These PHBs and their recommended codepoints are defined in various IETF Requests for Comments (RFCs). For more information on Differentiated Services and Diffserv Codepoints (DSCPs), see RFC 2474, RFC 2475, RFC 2597, RFC 2598, and RFC 3246, all of which are available from the IETF at `<http://www.ietf.org/rfc/>`.

TAO allows applications to control the setting of DSCPs on requests and replies via client and server policies. Servers can apply the Network Priority Policy and clients can apply the Client Network Priority Policy. Either policy is used to set DSCP values for requests and replies. The network priority model is used to determine which DSCP to apply to a particular request or reply.

Support for the Network Priority Policies is located in the `TAO_DiffServPolicy` library. The source code is located in `$TAO_ROOT/tao/DiffservPolicy`. MPC-based applications that use this policy should derive their server projects from the `diffservpolicy` base project.

Note *TAO can also set DSCPs as part of its Real-Time CORBA support. See 8.5.2 for details.*

6.6.1 Using the Network Priority Policies

Both the Client Network Priority and Network Priority Policies use the `NetworkPriorityPolicy` policy type that is defined in `$TAO_ROOT/tao/DiffServPolicy/DiffServPolicy.pidl`:

```
module TAO
{
  typedef long DiffservCodepoint;

  enum NetworkPriorityModel
  {
    CLIENT_PROPAGATED_NETWORK_PRIORITY,
    SERVER_DECLARED_NETWORK_PRIORITY,
    NO_NETWORK_PRIORITY
  };

  const CORBA::PolicyType CLIENT_NETWORK_PRIORITY_TYPE = 0x54410003;
  const CORBA::PolicyType NETWORK_PRIORITY_TYPE = 0x54410004;

  local interface NetworkPriorityPolicy : CORBA::Policy
    {
      attribute NetworkPriorityModel network_priority_model;
      attribute DiffservCodepoint request_diffserv_codepoint;
      attribute DiffservCodepoint reply_diffserv_codepoint;
    };
};
```

Here is some sample client code for specifying the Client Network Priority Policy on the ORB:

```
CORBA::Policy_var client_network_policy =
  orb->_create_policy (TAO::CLIENT_NETWORK_PRIORITY_TYPE);

TAO::NetworkPriorityPolicy_var nw_priority =
  TAO::NetworkPriorityPolicy::_narrow (client_network_policy.in ());

nw_priority->request_diffserv_codepoint (20);  // AF22
nw_priority->reply_diffserv_codepoint (22);    // AF23
nw_priority->network_priority_model (
  TAO::CLIENT_PROPAGATED_NETWORK_PRIORITY);

CORBA::PolicyList policy_list;
```

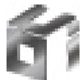
```
policy_list.length (1);
policy_list[0] = nw_priority;

policy_manager->set_policy_overrides (policy_list, CORBA::SET_OVERRIDE);

policy_list[0]->destroy ();
```

This sets the Client Network Priority Policy with the client-propagated model, meaning we want the request and reply to be taken from this policy object. The request DSCP is set to 20 and the reply DSCP is set to 22.

In order for this example to function properly, we should also set the server policy in the POA to a compatible value:

```
CORBA::Policy_var npp =
  orb->_create_policy (TAO::NETWORK_PRIORITY_TYPE);

TAO::NetworkPriorityPolicy_var nw_priority =
  TAO::NetworkPriorityPolicy::_narrow (npp.in ());

nw_priority->request_diffserv_codepoint (24);  // CS3
nw_priority->reply_diffserv_codepoint (16);    // CS2
nw_priority->network_priority_model(
  TAO::CLIENT_PROPAGATED_NETWORK_PRIORITY);

CORBA::PolicyList policy_list;
policy_list.length (1);
policy_list[0] = nw_priority;

PortableServer::POA_var child_poa =
  root_poa->create_POA ("Child_POA",
                        poa_manager.in (),
                        policy_list);

policy_list[0]->destroy ();
```

These settings ensure that the client-specified request and reply DSCPs are used.



CHAPTER 7

Asynchronous Method Handling

7.1 Introduction

Processing CORBA requests may require a long duration of activity resulting in the blocking of subsequent requests. This can reduce the server's responsiveness as threads that might otherwise be used to service incoming requests must block while waiting for a response. Concurrency strategies such as thread-per-connection and thread-pool can be used in such cases to increase the system responsiveness, but these approaches may not scale well as the number of threads increase due to increased number of client connections or client requests. Asynchronous Method Handling (AMH) is a TAO specific feature that addresses this situation without requiring you to implement complicated concurrency strategies.

Note *Although AMH is TAO specific, it has been submitted to the Object Management Group for possible inclusion into the CORBA specification.*

AMH provides server implementers the means to have a request for an operation be handled by one thread, while having the response to that request

delivered by another thread. For example, a server that needs to invoke a remote operation in the course of handling a request can combine the use of AMH with Asynchronous Method Invocation (AMI), discussed in 6.2, to have an AMI callback handler complete the processing of the original invocation, and return the result to the caller. In this way, AMH greatly reduces the risk of a runaway stack that may arise from the use of concurrency and wait strategies such as the Leader-Follower strategy.

Clients of AMH based services are unaware of this responsibility hand-off. However, using AMH does impose some limitations:

- *All requests must be passed as GIOP messages.* No direct or through-POA collocation is possible with AMH. The use of AMH in an application must be anticipated at design time.
- *AMH may be considered to violate certain aspects of the contract between servants, POAs, servant managers, and other server objects.* This violation arises from the possibility that a thread calling a servant during an invocation may return control back to the POA before the actual operation is complete without throwing an exception.

Note *Code using AMH can be found in TAO itself. In particular, TAO's Implementation Repository uses AMH to improve its performance when there are numerous clients trying to connect to it.*

7.1.1 When AMH is Useful

Situations where AMH can be useful include:

- *You have numerous client connections or client requests to a service that depends on systems or other services that may cause the serving thread to block.* Scenarios that cause such blockage include media I/O, database access, reliance on long-running services, or computation intensive activities such as numerical analysis. A scenario where AMH is particularly useful is when it is used in conjunction with AMI for the middle-tier server in a three-tier architecture where the middle-tier server offloads requests to servers that can be time consuming to fulfill. For this scenario, neither the client nor the target server processing the request are aware of the AMH/AMI usage by the middle-tier server. Details on using AMH in combination with AMI can be found in 7.6.

- *A very large number of clients are concurrently connected to your server.* Empirical data from *Design and Performance of Asynchronous Method Handling for CORBA* shows that when the number of concurrent clients connected to a server gets large enough, when using standard concurrency strategies, the request throughput becomes unacceptably low. In this case, AMH becomes the only practical solution available.
- *You wish to simplify concurrency support in your server code.* Using AMH, it is possible to avoid multithreaded programming, which can be difficult to write and maintain.
- *It is important that requests be processed in the order in which they are received.* The special skeleton code generated for using AMH ensures that client requests are processed in the order received.

7.1.2 When AMH is not Useful

Some situations where AMH may not be applicable include the following:

- *There will rarely be more than a medium load of concurrent clients.* Additional empirical data provided in *Design and Performance of Asynchronous Method Handling for CORBA* shows that AMH is slightly less efficient than other concurrency models. However, you should weigh this slight reduction in efficiency to the possibly simpler server code that can be written when using AMH to support concurrent requests.
- *You are using certain advanced CORBA features.* Some advanced CORBA features assume that the thread that starts an invocation is the same one that finishes it. However, AMH breaks this assumption. Further discussion about using advanced CORBA features along with AMH can be found in 7.5.
- *Your application can throw a number of exceptions.* You must be careful when using AMH in an application that raises exceptions. AMH response handlers, the code responsible for sending a reply back to the client, are typically not invoked by a POA or a skeleton. Therefore, care must be taken to catch all exceptions in order to communicate them to the client. There is no framework that will assist in automating this task.

7.2 Participants in an AMH Servant

Server applications using AMH rely on the interaction of the following classes:

- A special skeleton class generated by the IDL compiler for supporting AMH.
- A Response Handler class also generated by the IDL compiler that takes the reply information and passes it to the client. An instance of this class is generated by the AMH skeleton.
- The AMH servant code you write.

AMH servants are derived from the AMH skeletons. The implementation methods of these skeletons differ from the ordinary server-side IDL to C++ mapping. Only `in` and `inout` style arguments are passed to implementation methods, and the method has a void return, regardless of the return type of the IDL operation. The response handler's interface deals with the outputs for each operation.

The response handler is a local CORBA object. Its role is to gather any output (`out` and `inout` arguments, and return values) from an IDL operation and prepare a GIOP reply message for the client. A given response handler is only valid during a single operation. Once the reply is sent, references to the response handler should be discarded. For each IDL operation, the response handler has two methods, one for ordinary returns, and another for returning exceptions.

7.2.1 Simple Example

Consider the following IDL:

```
interface EchoTest
{
   string echo (in string message);
};
```

The following code fragments illustrate the participants in an AMH servant.

AMH Skeleton

```
class AMH_EchoTestResponseHandler_ptr;
class POA_AMH_EchoTest
```

```
{
public:
  virtual void echo (AMH_EchoTestResponseHandler_ptr rh,
  const char * message) = 0;
};
```

Response Handler

```
class AMH_EchoTestResponseHandler : public virtual CORBA::Object
{
public:
   virtual void echo (const char *return_value)
};
```

Servant

```
class AMH_EchoTest_i : public virtual POA_AMH_EchoTest
{
public:
  virtual void echo (EchoTestResponseHandler_ptr rh,
                     const char * message );
}
```

The following diagram illustrates how these participants interact to handle an invocation.

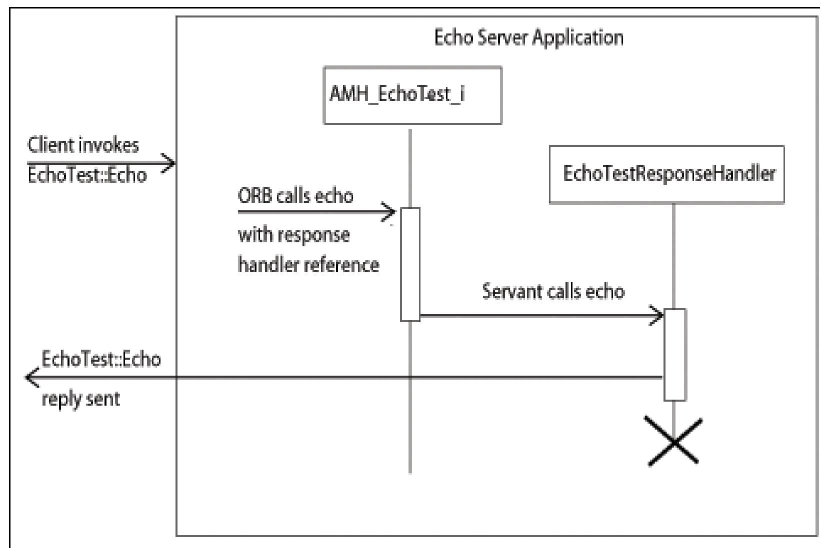


Figure 7-1AMH Servant and Response Handler

As shown in Figure 7-1, the ORB infrastructure processes the incoming request from the client and invokes the servant. The reply to the request is not sent until the application calls the appropriate response handler method. This can happen from the servant code, or the application may retain a reference to the response handler and send the reply at a later time.

7.3 Generating AMH Related Code

The TAO IDL compiler takes a single command line option, `-GH`, which triggers the generation of the AMH skeleton and response handler classes. The IDL compiler and its options are discussed at length in Chapter 4.

For each interface in the IDL file, the compiler generates an AMH skeleton and a response handler. The name for the AMH skeleton is similar to that of the ordinary skeleton, except that `AMH_` is prepended to the interface name. Recall that CORBA compliant skeletons are named by prepending `POA_` to the fully qualified interface name, including any modules. In general the form is `POA_`[<modules>`::`]`AMH_`<interface>

For an interface declared as:

```
module DevGuide
{
  interface Messenger
  {
  ...
 };
};
```

The IDL compiler will generate `POA_DevGuide::Messenger` and `POA_DevGuide::AMH_Messenger`. If the interface were declared outside of any module, the generated names would be `POA_Messenger` and `POA_AMH_Messenger`.

The response handler generated by the IDL compiler for each interface is given a name derived from the interface name in this way: [<module>`::`]`AMH_`<interface>`ResponseHandler`. For example, the response handler for the interface above is `DevGuide::AMH_MessengerResponseHandler`.

Response handlers are reference counted local CORBA objects. As such, the response handler has a stub class, a `var`, `ptr`, and an `out` type related to it.

These are all necessary to allow a response handler instance to be created during the receipt of an invocation, then held until the invocation is complete. Response handler references are kept by the skeleton infrastructure until a reply or exception operation is invoked. It is the servant's responsibility to either use the response handler or store its reference for future use. It is essential that either a reply or exception reply method be invoked before losing the reference to the response handler. Failure to do so will cause the client to hang.

7.4 An AMH Example Program

Now let us take a look at the Messenger example, extended to use an AMH based servant. The full source code for this example can be found at `$TAO_ROOT/DevGuideExamples/AMH`.

The IDL is nearly identical to that used in other code examples, except that here the Messenger interface is enclosed in a module so that the application of the naming convention can be seen.

```
module DevGuide
{
  interface Messenger
  {
    boolean send_message(in string user_name,
                         in string subject,
                         inout string message);
  };
};
```

The `send_message()` operation has data that flows in and out through its parameters, as well as returning a value. Also, even though no user exceptions are raised by the operation, it is still liable to raise system exceptions.

7.4.1 The Generated Stub Classes

When the IDL is compiled, a stub is created for the Messenger interface that is identical to the stub used for the non-AMH case. The same stub is used as a proxy for objects served by synchronous servants as well as asynchronous ones.

A stub for the response handler is also created. This stub is defined in the `MessengerC.h` and `MessengerC.cpp` files. Although they are of no interest to the clients, AMH-based servers will use these stubs. For the Messenger interface shown above, the response handler stub appears as follows:

```
namespace DevGuide
{
   class AMH_MessengerResponseHandler : public virtual CORBA::Object
   {
      virtual void send_message (
            ::CORBA::Boolean return_value,
            const char * message
          ) = 0;

        virtual void send_message_excep (
            ::DevGuide::AMH_MessengerExceptionHolder * holder
          ) = 0;
   };
};
```

For each operation in the interface, two corresponding response handler member functions are created. The first, which has the same name as the operation, takes arguments for the operation's return value, and any `out` (or `inout`) arguments. The second method generated for responding to an operation is used for sending exceptions back to the client. The name of this method is generated by appending `_excep` to the operation name. The exception reply method takes a single argument, a reference to an exception holder. The exception holder is a valuetype object that is capable of raising whatever exception it contains.

Note *The use of an exception holder valuetype and inclusion of AMH stubs make the generated client-side files dependent on the valuetype library. Any application linked to these generated files must also be linked to the valuetype library. A way to minimize the impact of this dependency is to generate two sets of stub definitions: one with AMH for use in server development, and one without for use on the client side.*

The exception holder is a specialized value type object that is unique to a particular interface. The exception holder has a method for each operation that is responsible for raising whatever exception needs to be propagated back to

the client. An example showing the use of the exception holder is shown in 7.6.1.

7.4.2 The AMH Servant

The TAO IDL compiler, when given the `-GH` option, generates an ordinary skeleton as well as an AMH skeleton for every interface. Using the example IDL shown above, the ordinary skeleton and AMH skeleton are as follows.

```
namespace POA_DevGuide
{
   class Messenger : public virtual PortableServer::ServantBase
   {
   public:
      virtual CORBA::Boolean send_message (
               const char * user_name,
               const char * subject,
               char *& message
             ) = 0;

   };

   class AMH_Messenger : public virtual PortableServer::ServantBase
   {
   public:
      virtual void send_message (
               DevGuide::AMH_MessengerResponseHandler_ptr _tao_rh,
               const char * user_name,
               const char * subject,
               const char * message
             ) = 0;

   };
};
```

Note that the AMH skeleton's definition of the `send_message()` method differs from that of the ordinary skeleton in two ways:

1. The leading argument is a reference to the response handler for this particular invocation. This response handler must be invoked by the servant for any operation to send a GIOP reply message to the client.
2. The remaining `send_message()` parameters map to the operation's `in` and `inout` arguments. The return from `send_message()` is void, even though the IDL operation returns a string. The return value is passed to the appropriate method on the response handler, as are any out and out-bound inout values, if any are defined.

Now lets take a look at a very simple implementation for the AMH version of `send_message()`. In this case, we are not really taking advantage of the benefit of AMH because we are directly invoking the response handler immediately from the servant.

```
void
AMH_Messenger_i::send_message (
    DevGuide::AMH_MessengerResponseHandler_ptr _tao_rh,
    const char * user_name,
    const char * subject,
    const char * message
  )
{
  std::cout << "Message from: " << user_name << std::endl;
  std::cout << "Subject:      " << subject << std::endl;
  std::cout << "Message:      " << message << std::endl;
  CORBA::String_var inout_message_arg =
    CORBA::string_dup("Thanks for the message.");
  CORBA::Boolean result = true;

  _tao_rh->send_message(result,inout_message_arg.inout());
}
```

The parameters passed to the response handler's `send_message()` method are supplied using `in` parameter semantics. This means that the caller is still responsible for releasing memory that was used by any intermediate values that may be returned. So, for example, the response handler must duplicate the `in` parameters it receives.

This behavior exists due to an inversion of control that results from the asynchronous request processing. In an ordinary request/reply invocation, the servant has control only when the thread is in the implementation method, thus control goes away when the method returns. Another way to look at this is that the lifespan of the invocation is only as long as the duration of the invocation method. This means that it is the responsibility of the caller of the method (the generated skeleton class) to clean up any allocated resources being passed back to the client.

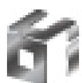
By contrast, when using AMH it is possible for the lifespan of the invocation to exceed the duration of the invocation method. The example shown above happened to use the response handler right away, but it is perfectly valid to store a reference to the response handler, then invoke the appropriate method on it at a later time. For instance, the example code shown above alternatively could have spawned a thread to handle the response.

```
#include <ace/Thread.h>
void
AMH_Messenger_i::send_message (
    DevGuide::AMH_MessengerResponseHandler_ptr _tao_rh,
    const char * user_name,
    const char * subject,
    const char * message
  )
{
  std::cout << "Message from: " << user_name << std::endl;
  std::cout << "Subject:      " << subject << std::endl;
  std::cout << "Message:      " << message << std::endl;
  DevGuide::AMH_MessengerResponseHandler_ptr  dup_rh =
   DevGuide::AMH_MessengerResponseHandler::_duplicate (_tao_rh)
  ACE_Thread::spawn (send_message_reply, dup_rh);
};
```

The response handler reference must be managed, and sending the response does not end the method. The response handler is a reference counted local CORBA object. An instance of the response handler is created in the skeleton just prior to calling the implementation method, and the skeleton releases its reference to the response handler when the implementation method returns. This is why, in the implementation of `send_message()` above, the servant duplicates the reference to the response handler before returning.

```
void send_message_reply (void * arg)
{
  DevGuide::AMH_MessengerResponseHandler_ptr rh =
    (DevGuide::AMH_MessengerResponseHandler_ptr)arg

  CORBA::String_var inout_message_arg =
    CORBA::string_dup("Thanks for the message.");
  CORBA::Boolean result = true;

  ACE_OS::sleep (5);

  rh->send_message(result,inout_message_arg.in());
  CORBA::release (rh);
}
```

In this case, the response handler is passed to a thread function that waits a few seconds before proceeding to send the reply. Here, the invocation spans the life of both the implementing method called by the skeleton, as well as the second thread function. The first method returns to the caller immediately after spawning the thread, but the client does not receive a reply until after the



thread function completes. The thread function must release a reference to the response handler to offset the reference duplication done in `AMH_Messenger_i::send_message()`.

Invoking the response handler CORBA object results in the sending a response back to the original client without having any effect on any of the values passed to it. Therefore, any allocation of storage, such as the string shown above, is still available for reuse (or leaking if not managed properly).

7.4.3 AMH and Oneway Invocations

Operations that have no out or inout arguments and a void return type will still have a response handler that must be called. This is also true for oneway operations. Oneways may be invoked by a client that has set the `SYNC_WITH_TARGET` synchronization scope policy, which requires the server to send a GIOP response when the operation completes. See 6.3.4 for more details on this policy.

7.4.4 Throwing Exceptions

Exceptions are sent back to clients using a special form of the GIOP reply message. The message header contains a flag indicating that the reply contains an exception and the message data contains the marshaled exception. Because of the inversion of control mentioned above, we cannot simply throw an exception and expect it to be propagated back to the original client. Exceptions are thrown by the use of specialized methods on the response handler. Along with each method for returning values (even if void) from an interface's attributes and operations, the response handler also has a method for each of these to raise exceptions. These methods are named by appending `_excep` to the name of the operation to which it is related. Thus, the `DevGuide::AMH_MessengerResponseHandler::send_message()` method shown above is accompanied by `send_message_excep()` for raising exceptions.

There is a distinct exception method for each operation. The data supplied to the method is a reference to a value type object that contains the exception to be thrown. This exception holder object is similar to the exception holder defined by the CORBA Messaging specification for AMI. The difference being that in AMI, the framework calls the `_excep` method you implement in your callback handler, whereas with AMH, you call the `_excep` method supplied to you by the framework.

The following example code shows how to throw an exception via the response handler. To make this example a little clearer, a new interface is defined with an operation that raises a user exception.

```
module DevGuide
{
  exception demo {};
  interface Asynch_Except_Demo
  {
    boolean trigger () raises (demo);
  };
};
```

This could be implemented by an AMH servant method such as this one:

```
void
Excep_Example_i::trigger (DevGuide::AMH_Asynch_Except_DemoResponseHandler_ptr
_tao_rh)
{
  DevGuide::demo *d = new DevGuide::demo;
  DevGuide::AMH_Asynch_Except_DemoExceptionHolder ex_holder(d);
  _tao_rh->trigger_excep (&ex_holder);
  std::cout << "Done sending exception." << std::endl;
}
```

You will see there is something peculiar about this code example. The exception instance, `d`, is a pointer to an exception allocated on the heap. This code does not free the exception object after invoking the exception sender. This is because the exception is given to the exception holder, which takes ownership of the exception.

The AMH exception holder is initialized by supplying a pointer to an exception instance allocated on the heap. This requirement is a consequence of the mechanism used to propagate the exception back to the caller. This is done by the response handler invoking the operation-specific raise method of the exception holder, which in turn guards the pointer to the exception with an auto-pointer, then calls a method on the exception that causes it to throw itself. Afterward, the auto-pointer releases the holder's reference to the exception, causing it to be deleted.

Remember that invoking the appropriate response method was semantically identical to making an invocation with `in` parameters. The exception methods violate this notion because these methods require the exception to be allocated on the heap, and consume it as a side effect of sending the exception.

This behavior results from the mechanism used to actually form and transmit the GIOP message containing the exception. This code relies on the exception holder to throw the exception, then uses existing skeleton helper methods to generate the GIOP method. By throwing the exception, the exception holder extracts the exception instance from the exception holder, assigns it to an auto pointer, then throws the exception.

7.4.5 AMH And The Server Main

A process hosting objects served by asynchronous servants is only different from one hosting synchronous objects in that it has a choice as to which type of servant it wishes to attach to an activated object. The server must still initialize an ORB, obtain a POA, and use that POA to associate the servant (asynchronous or synchronous) with an object.

7.4.6 AMH and the Client

As with the main function of the server, clients are unaware of the synchrony of the servant behind any objects it uses. Currently, TAO imposes a side-effect on clients of AMH enabled services. The AMH response handler and exception holder classes are defined in the generated stub header file. The exception holder is a value type, as is the exception holder in an AMI callback object. This creates a dependency between the client application and the TAO value type library.

A second side-effect is manifest in MPC files related to applications that directly build the IDL for an AMH based server. As is the case with the AMH example code provided in `$TAO_ROOT/DevGuideExamples/AMH`, the client subproject must be dependant on the AMH base project, `amh.mpb`. This ensures that if this project is used to compile the IDL, the proper AMH elements will be generated. For example, the definition for an MPC project that inherits from the AMH base project could look like:

```
project(*Client): …, amh
{
  ...
}
```

7.5 AMH and Advanced CORBA Features

CORBA servers frequently have many details to manage in addition to running implementation code. Examples include:

- An application may make use of servant managers to control the deployment of servant instances.
- Applications may make use of certain CORBA Current objects to gain access to information that is specific to a particular invocation context.
- Applications may be written in such a way that a servant is collocated with a client, and may wish to use strategies such as direct collocation for performance optimization.

Many of these advanced features of CORBA assume that the thread that starts an invocation is the same thread that finishes it. AMH makes it possible to invalidate this assumption. Therefore, care must be taken to ensure that any invocation-specific context data is separately managed so that any subsequent thread that participates in an invocation is able to access this information. The following subsections outline some of the obvious situations where such context dependent information may be needed, although it is not a complete list.

7.5.1 Portable Interceptors

Portable Interceptors are specialized client or server objects that are used to process inbound or outbound messages, very close to the transport layer. Interceptors are used to initialize the context within any CORBA Current objects used by the server, and they may be used to process service contexts, which contain meta-information attached to a message. For example, this meta-information might include security information such as authorization tokens or credentials, or may include transaction tracking information.

When using any service that relies on interceptors, you should be very careful when AMH is also used. For example, TAO’s RT Scheduling Service assumes that all responses will be sent by the same thread as that on which the original request was received.

If you are implementing your own interceptor to manage meta-information, such as request auditing or other activities, and you use AMH, be sure to write

your interceptor in such a way as to avoid relying on any information that is specific to the thread receiving a request or sending a reply.

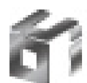
7.5.2 Servant Locators

Servant managers work with the POA to supply servant instances on demand. As discussed in *Advanced CORBA Programming with C++*, 11.7.3, there are two kinds of servant managers: activators and locators. Activators provide a servant to the POA that is retained in the POA's active object map, and remains associated with a given object until that object is deactivated or the POA itself shuts down. Locators are servant managers that provide servants only for the duration of a single invocation. Because of the limited lifespan of servants when using locators, care must be taken using AMH when servants are managed using locators.

Whether the servant is held by the POA for the duration of a single invocation or many, it is handed back to the servant manager for clean up. With AMH, this may occur before the response handler has sent a reply message. If you are using AMH and a servant locator then you must ensure that any reference to the original servant is not used, as the servant locator may have destroyed the servant. Similarly, you may have implemented the servant locator to manage a pool of servant objects. While the servant instance may still be valid, you must be aware that any state information that you update while handling an invocation may be modified if the invocation is handed off to another thread for completion.

7.5.3 Invocation Related CORBA::Current Objects

CORBA Current objects are locality constrained interfaces, derived from the empty `CORBA::Current` interface, that give application code access to information that is specific to the current thread of control. This access is provided through an object reference supplied by the ORB. Some current objects, such as `PortableServer::Current` and `PortableInterceptor::Current`, are specific to the current invocation. Because this invocation information is thread-specific, it is not sufficient to hand a current object to whatever thread will complete an AMH invocation. The initial servant thread must extract any information from the current and hand that data separately to the invocation completing thread. If a completion handler needs invocation-specific information from the current, such as the object ID from a `PortableServer::Current`, the information must be



accessed in the original servant method and handed off to the completing thread along with the response handler.

Since current objects are context aware, the completing thread cannot simply use a reference to the invocation related current. Any invocation on the current by that thread would result in a `NO_CONTEXT` exception being raised. From the point of view of the current, the second thread is outside of the context of an invocation.

7.5.4 Reference Counted Servants

Servants may be reference counted in order to avoid memory leaks when objects associated with the servants are destroyed. A servant's reference count is incremented during an invocation and decremented again when the invocation completes. This way, an invocation that deactivates an object, such as one calling `POA::deactivate_object()`, will not cause a crash when the POA removes its reference to the servant.

When an AMH servant method returns, regardless of the disposition of any pending reply, the skeleton code will treat this as a completion of the invocation and will decrement the reference count on the servant. This means that if your AMH servant uses a companion object to complete an invocation, the companion object should either have no association with the original servant, or must manage the reference count of that servant. Managing the reference count may be done explicitly by calling `_add_ref()` and `_remove_ref()` on the servant, or implicitly by holding the reference to the servant in a `PortableServer::Servant_var<>`.

7.5.5 Collocation

Collocation occurs when the servant for an object is in the same address space as the client making an invocation on that object. There are two forms of collocation, thru-POA and direct. Collocated invocations that go through the POA have an opportunity to also go through interceptors and are subject to the control imposed by the POA such as being rejected because the POA is in the discarding state. Direct invocations are forwarded straight from the stub method to the servant method. In both cases, the caller's thread is actually used to perform the invocation.

Since an AMH servant's implementation of an operation has a different signature than what is expected based on the IDL definition of the operation, it does not match what is expected by the collocated stub. Further, AMH makes

it possible or likely that the invocation is not complete when the servant method returns. There is currently no mechanism in TAO's collocated stubs to enable the calling thread to wait until some other thread invokes the response handler and provides results to the caller. Therefore, there is no support in TAO for combining collocation and AMH.

Given that the desire to use AMH is often the result of bottlenecks in the servant, we might find that the efficiency gained by using collocated calls that avoid marshaling would be minimal anyway. Therefore, it is reasonable to explicitly disable collocated calls in an application that is implemented using AMH based servants. Keep in mind that such control is imposed at IDL compilation time. Be sure to isolate the definitions of interfaces that will be implemented using AMH from those for which you wish to support collocated access.

Note *Future versions of TAO may support collocated invocation of AMH servants.*

7.6 Combining AMH with AMI

CORBA servers are often used in multi-tier applications, with middle layers serving as concentrators or gateways. The middle layer may process requests by turning around and sending invocations to other servers to complete. Consider the following IDL interfaces:

```
// file: middle.idl
interface Middle
{
   string get_the_answer (in string question);
};


// file: inner.idl
interface Inner
{
  string answer (in string question);
};
```

A client might invoke `Middle::get_the_answer()`, which in turn invokes `Inner::answer()`, waits for the response, then replies back to the client with the answer. Imagine that invoking `Inner::answer()` takes a long time,

and that the client load is variable. Sometimes there may be two or three client requests pending, while at other times there may be hundreds. The thread waiting for a reply is essentially a wasted resource. As the client load increases, if there are insufficient threads available to handle the load, then clients may not be able to have their requests processed within their time constraints.

Traditionally, to avoid making a thread a wasted resource, the server implementing the `Middle` interface would process a request invoking `get_the_answer()` in a separate thread using some threading strategy. Before this thread returns, it in turn invokes `Inner::answer()` and waits for the result. While waiting, this thread may be blocked so that it cannot handle other requests. If TAO's thread pool strategy is being used, and the thread is part of a thread pool, then it is at risk of being "borrowed" to process another incoming request. This is illustrated in Figure 7-2.

Note *Further discussion about TAO's threading models and wait strategies can be found in Chapter 15.*

.

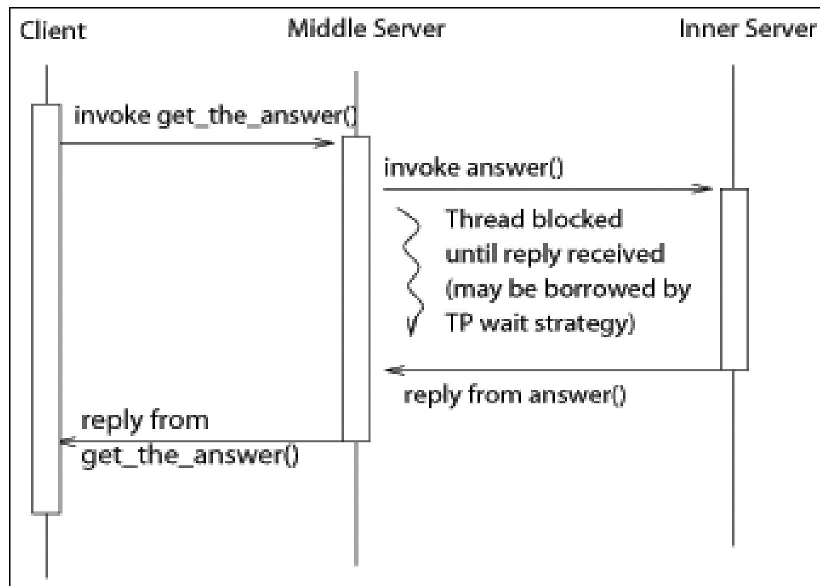


Figure 7-2 Middle-tier Server Without AMH and AMI

In addition, using TAO's thread pool strategy raises the possibility of recursive requests (also called "nested upcalls"), wherein a thread waiting for the `Inner::answer()` response may be required to handle another incoming client request. Of course, the problem could be resolved by adding more threads, but this does not scale well as the number of client requests increases. This is because context switching between many threads may overwhelm the system.

The best solution to this problem is to implement the Middle server using AMH to handle the incoming `get_the_answer()` requests and AMI to invoke `Inner::answer()`, as shown in Figure 7-3.

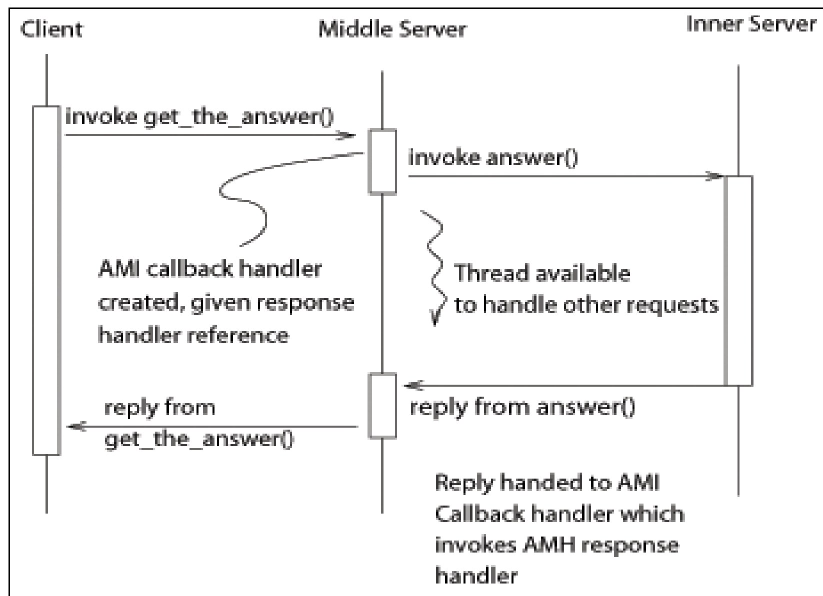


Figure 7-3 Middle-tier Server Using AMI and AMH together

7.6.1 AMH/AMI Example

An example implementation of the Middle server using AMH and AMI together is given here. The full code for this example is in the directory `$TAO_ROOT/DevGuideExamples/AMH_AMI.`

7.6.1.1 AMI Callback Handler

To realize the benefits of AMH/AMI, an AMI callback handler in the middle tier must be supplied to manage asynchronous replies from the Inner server.

Since AMI callback handlers are CORBA servants we must supply an implementation for the handler. `Inner_callback_i` is the AMI callback handler invoked when the reply to `Inner::answer()` is received. The role of this callback handler is to forward the answer back to the original caller. Therefore, the callback servant must be initialized with the correct response handler shown here:

```
class Inner_callback_i : public virtual POA_AMI_InnerHandler
{
public:
  Inner_callback_i (PortableServer::POA_ptr poa,
                    AMH_MiddleResponseHandler_ptr _tao_rh);
  virtual void answer (const char * ami_return_val);
  virtual void answer_excep (Messaging::ExceptionHolder * excep_holder);

private:
  PortableServer::POA_var poa_
  AMH_MiddleResponseHandler_var response_handler_;
};
```

As you will see below, we will make sure the AMI reply handler is initialized with the AMH response handler. The callback handler reference must be supplied for each `sendc_` call, in order to allow the ORB's dispatcher to deliver the reply to the appropriate handler.

As is the case for all AMI callback handlers, our callback handler has two methods that must be supplied for each operation in the interface. One for dealing with ordinary replies, another for dealing with exceptions. In this example, we have a callback handler class, `Inner_callback_i`, with methods `answer()` and `answer_excep()`.

Note *Because we are using AMI, the Inner server does not need to change to support AMH-based asynchronous replies.*

Consider now the handling of ordinary replies:

```
void
Inner_callback_i::answer (const char * ami_return_val)
{
  this->response_handler_->get_the_answer (ami_return_val);
  PortableServer::ObjectId_var oid = this->poa_->servant_to_id(this);
  this->poa_->deactivate_object (oid.in());
```

```
}
```

Handling return values is straightforward. The return value and any `out/inout` parameters are supplied as `in` parameters to the callback handler. It simply passes those values on to the AMH response handler, again as `in` parameters, which forwards the response to the original client. After that, this servant has done its job, so it deactivates itself.

Now consider the handling of exceptions. As shown in section 7.4.4, the AMH exception holder is initialized with a local copy of the exception extracted from the AMI exception holder. Combining this behavior with the AMI exception callback interface requires that we explicitly duplicate the exception in order to pass it on to the client. This may be done by having the AMI exception holder throw the exception, which we then catch, duplicate, and pass to the client via the AMH response handler. This technique is shown here.

```
void
Inner_callback_i::answer_excep (Messaging::ExceptionHolder* excep_holder)
{
  try {
    excep_holder->raise_exception();
  }
  catch (CORBA::Exception& ex) {
    CORBA::Exception* local_ex = ex._tao_duplicate();
    AMH_MiddleExceptionHolder amh_excep_holder (local_ex);
    this->response_handler_->get_the_answer_excep (&amh_excep_holder);
  }
  catch (...) {
    CORBA::Exception* unknown_ex = new CORBA::UNKNOWN;
    AMH_MiddleExceptionHolder amh_excep_holder (unknown_ex);
    this->response_handler_->get_the_answer_excep (&amh_excep_holder);
  }

  std::cout << "inner_callback_i deactivating self" << std::endl;
  PortableServer::ObjectId_var oid = this->poa_->servant_to_id(this);
  this->poa_->deactivate_object (oid.in());
}
```

7.6.1.2 AMH Servant

Consider now the interface to servant for the middle-tier servant:

```
#include "middleS.h"
#include "innerC.h"
```

```
class Asynch_Middle_i :
  public virtual AMH_POA_Middle
{
public:
    Asynch_Middle_i (PortableServer::POA_ptr poa, Inner_ptr inner);
    virtual void get_the_answer (AMH_MiddleResponseHandler_ptr _tao_rh,
                                 const char * question);
private:
  PortableServer::POA_var poa_
  Inner_var inner_;
};
```

The middle servant is the only piece that needs to be AMH aware, therefore it is the only class to derive from a `POA_AMH_*` base class.

Asynch_Middle_i::`get_the_answer()` is passed an AMH response handler, which is used to initialize an instance of an AMI reply handler, as shown below:

```
void
Asynch_Middle_i::get_the_answer (AMH_MiddleResponseHandler_ptr _tao_rh,
                                 const char * question)
{
  PortableServer::ServantBase_var servant =
       new Inner_callback_i (this->poa_.in(),_tao_rh.in());

  PortableServer::ObjectId_var objid =
       this->poa_->activate_object (servant.in());
  CORBA::Object_var obj = this->poa_->id_to_reference(objid.in());
  AMI_InnerHandler_var cb =  AMI_InnerHandler::_narrow(obj.in());
  this->inner_->sendc_answer (cb.in(),question);
}
```

The call to `Inner::sendc_answer()` sends the invocation request message to the Inner server and returns immediately.

In general, after this point the servant thread is free to handle any other incoming messages, whether they are new requests from clients, or AMI replies from the Inner server. The AMI reply handler will be invoked on some thread when the Inner server sends a reply message from the `answer()` invocation. It will then immediately invoke the `get_the_answer()` method of the AMH response handler, which sends a reply back to the originating client.

CHAPTER 8

Real-Time CORBA

8.1 Introduction

In 1999, the OMG introduced the Real-Time CORBA specification (ptc/99-06-02) (RT CORBA 1.0) to provide CORBA developers with policies and mechanisms for controlling allocation of system resources and improving the predictability of system execution. The RT CORBA 1.0 specification was originally defined as a set of extensions to the CORBA core and the CORBA Messaging specifications. In August 2002, the OMG published a minor revision of the RT CORBA specification, RT CORBA 1.1 (formal/02-08-02). RT-CORBA 2.0 (formal/03-11-01), released in November 2003, introduced the dynamic scheduling model. Finally, RT-CORBA 1.2 (formal/05-01-04) integrated the dynamic scheduling details with the existing static scheduling model. Despite the unusual version numbering sequencing, RT-CORBA 1.2, released in January of 2005, is the most current version of the RT CORBA specification.

8.1.1 Road Map

In this chapter, we explore the topic of real-time CORBA from the perspective of an application with real-time predictability requirements, as well as from the perspective of the features available in TAO's implementation. While the chapter is designed to be taken as a whole, you may find benefit to reading certain sections independently.

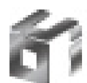
If you want to learn more about...

- *The motivation for and scope of RT CORBA*, see 8.2, "Real-Time CORBA Overview."
- *The new modules and interfaces introduced by RT CORBA*, see 8.3, "Real-Time CORBA Architecture."
- *The latest dynamic scheduling features of RT CORBA*, see 8.4, "Dynamic Scheduling."
- *Building and configuring* applications that use TAO's implementation of RT CORBA, see 8.5, "TAO's Implementation of Real-Time CORBA." This section also discusses TAO's *extensions* to RT CORBA.
- *Sample code that uses TAO's RT CORBA features*, see 8.6, "Client-Propagated Priority Model," 8.7, "Server-Declared Priority Model," and 8.8, "Using the RTScheduling::Current." These sections present examples of client and server application code using various RT CORBA features and priority models. In addition, 8.9, "Real-Time CORBA Examples," lists further examples and tests in TAO that use several of the RT CORBA features discussed in this chapter.

8.2 Real-Time CORBA Overview

The standard CORBA specification has historically done a very good job of supporting the requirements of distributed object-oriented systems, such as location transparency, programming language and operating system independence, separation of interface from implementation, and interoperability in heterogeneous environments.

However, in real-time systems, the timeliness of a system is as important as its functional requirements. That is, success is determined not only by logical correctness, but also by the time required to reach a correct solution or complete a task. A correct result that is reached outside the predetermined



time interval is still considered a failure. Such systems must be predictable and deterministic.

The applicability of CORBA to real-time systems has been limited, due to CORBA's lack of standard mechanisms for specifying and enforcing Quality of Service (QoS) across distributed objects and supporting real-time programming techniques.

The goal of the RT CORBA specification is to address the shortcomings of CORBA for distributed real-time systems without sacrificing the spirit of CORBA and without placing a burden on developers of non-real-time systems. RT CORBA adds QoS control to standard CORBA with the goal of improving application predictability. RT CORBA achieves this by bounding priority inversions and managing resources end-to-end.

Specifically, RT CORBA provides policies and mechanisms for resource configuration and control in the following areas:

- Processor Resources:
    - RT CORBA defines *portable priorities* and a mechanism for mapping them to native operating system priorities.
    - RT CORBA enables *end-to-end priority propagation* via standard priority models and mechanisms so that clients and servers can specify request-priority propagation semantics.
    - RT CORBA adds *thread pools* and mechanisms for servers to allocate, partition, and manage thread characteristics.
    - RT CORBA defines *standard synchronizers* for coordinating contention for system resources in a consistent fashion.
    - RT CORBA defines *distributable threads* and *schedulers* for managing static or dynamic scheduling.
- Communication Resources:
    - RT CORBA adds *protocol properties* to enable selection and configuration of protocols by clients and servers.
    - RT CORBA enables mechanisms for *explicit binding* to establish and manage connections between clients and servers.
- Memory Resources:

- RT CORBA enables *request buffering* by servers when all available threads are currently servicing requests.

Figure 8-1 shows how various RT CORBA policies and mechanisms (in *italics*) relate to the standard CORBA architecture.

CLIENT
SERVER
End-to-End Priority Propagation
CORBA Object
Client
Object Reference
Servant
Skeleton
Thread Pools
Stub
Portable Object Adapter
Request Buffering
Explicit Binding
Standard Synchronizers
ORB Core
ORB Core
ACE
ACE
OS
OS
Portable Priorities
Network
Protocol Properties

Figure 8-1 Real-Time CORBA Policies and Mechanisms

To achieve its goals with regard to the above QoS policies, the RT CORBA specification leverages the QoS policy framework defined in the CORBA Messaging specification. See Chapter 6 for more information on CORBA Messaging.

For example, using the QoS policy framework, a client can override default policy settings at the ORB, thread, or object reference level to affect qualities such as request priority, message delivery, and request/reply timeouts. Likewise, servers can use QoS policies with the Portable Object Adapter's `create_POA()` operation to affect server-side qualities such as request queuing and the creation and management of thread pools.

The remainder of this chapter describes the specific QoS policies addressed by the RT CORBA specification and how they are supported by TAO. We extend the `Messenger` example from previous chapters to show how to use TAO's implementation of the RT CORBA specification to address the QoS requirements of real-time applications.

For more information on RT CORBA, read "Object Interconnections: Real-time CORBA, Part 1: Motivation and Overview," by Douglas C. Schmidt and Steve Vinoski.

8.3 Real-Time CORBA Architecture

The RT CORBA specification extends the standard CORBA specification with the addition of several new modules and interfaces to achieve end-to-end predictability and control over the management of resources. Developers of non-real-time CORBA applications need not be burdened by these extensions. RT CORBA extensions to the standard CORBA architecture include:

- Real-time ORB (`RTCORBA::RTORB`)
- Real-time POA (`RTPortableServer::POA`)
- Real-time CORBA priority (`RTCORBA::Priority`)
- Real-time Current (`RTCORBA::Current`)
- Real-time mutex (`RTCORBA::Mutex`)
- Thread pools (`RTCORBA::ThreadpoolId`)
- Thread pool lanes (`RTCORBA::ThreadpoolLane`)

Figure 8-2 shows how key entities defined by the RT CORBA extensions relate to the standard CORBA architecture.

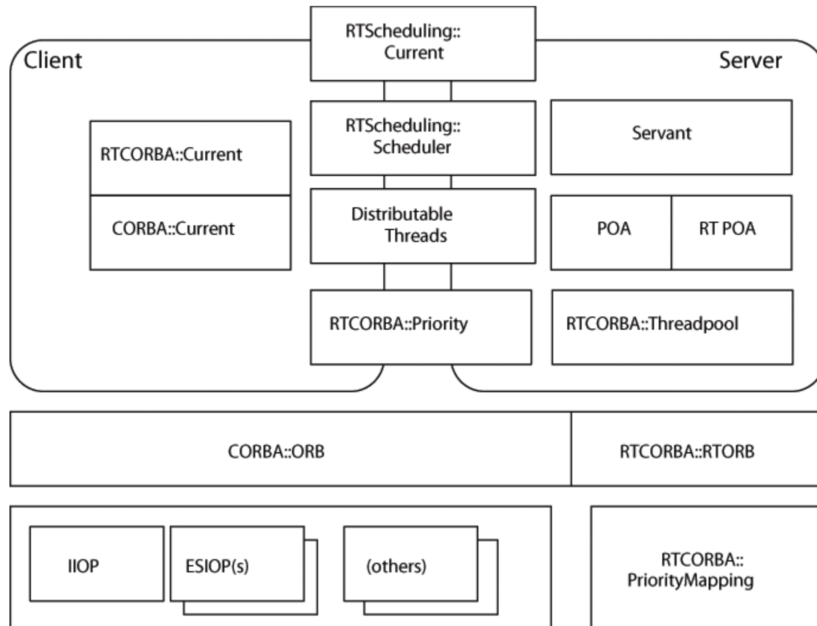


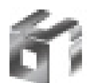
Figure 8-2 RT CORBA Extensions

Figure 8-2 also shows elements of Dynamic Scheduling, an extension to the real-time CORBA specification. Dynamic Scheduling, its components, features, and services are discussed at length in 8.4.

8.3.1 Real-Time CORBA Modules

The RT CORBA specification introduces additional IDL modules, `RTCORBA` `RTPortableServer`, and `RTScheduling`, which contain definitions of RT CORBA interfaces and types.

- The `RTCORBA` module contains definitions for portable priorities, priority mapping, thread pools, real-time policies, and protocol properties. It also contains definitions for real-time `Current`, `Mutex`, and `ORB` interfaces. Entities defined in the `RTCORBA` module are used by both clients and servers.



- The `RTPortableServer` module contains the definition of the real-time Portable Object Adapter (POA) interface for use by servers.
- The `RTScheduling` module defines the components used to support dynamic and static scheduling and distributable threads. The `RTScheduling::Current` may be used in both client and server processes.

Note *The interface definition for module* `RTCORBA` *is quite large. We will be examining portions of it throughout this chapter. The definition is included in the TAO source distribution in* `$TAO_ROOT/tao/RTCORBA/RTCORBA.pidl`. *Likewise, the definition of RTPortableServer is found in* `$TAO_ROOT/tao/RTPortableServer/RTPortableServer.pidl` *and the definition of RTScheduling is found in* `$TAO_ROOT/tao/RTScheduling/RTScheduler.pidl.`

8.3.2 The Real-Time ORB

The RT CORBA specification introduces an interface for a real-time ORB, `RTCORBA::RTORB`. The RT ORB is a local interface used to create resources necessary to manage real-time applications. The `RTORB` interface does not inherit from `CORBA::ORB`, rather it is simply a helper object that applications use to create and manage instances of various RT CORBA types, such as mutexes, thread pools, and policies.

TAO's RT CORBA library supplies the implementation of `RTORB`. The RT CORBA library uses specialized ORBInitializers, as defined in the Portable Intercepter specification, to initialize the RTORB. Each ORB has an `RTORB` instance. Applications obtain a reference to the RTORB by calling `resolve_initial_references("RTORB")` on the ORB.

Rather than showing the entire `RTORB` interface in one place, the various operations are introduced in smaller functional groupings as we describe how to use them in subsequent sections.

The `RTCORBA::RTORB` is initialized when its associated `CORBA::ORB` is initialized during `CORBA::ORB_init()`. The RT CORBA specification defines a new ORB initialization parameter, `ORBRTPriorityRange`, used to constrain the range of CORBA priorities the `RTORB` may use. However, this option is not currently supported by TAO, meaning that any CORBA priority

value may be used by any `RTORB`. CORBA Priority values are discussed in 8.3.4.

Here we show an example that obtains the RTORB from the ORB. For the sake of clarity, we have omitted error-handling code:

```
#include <tao/corba.h>
#include <tao/RTCORBA/RTCORBA.h>

int main (int argc, char* argv[])
{
  // Initialize the ORB.
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

  // Get the RTORB.
  CORBA::Object_var obj = orb->resolve_initial_references ("RTORB");
  RTCORBA::RTORB_var rt_orb = RTCORBA::RTORB::_narrow (obj.in());

  // Use the RTORB to access RT CORBA features (e.g., create_threadpool() )
};
```

8.3.3 The Real-Time POA

The RT CORBA specification introduces the real-time POA interface, `RTPortableServer::POA`, which specializes `PortableServer::POA`. As shown below, the real-time POA adds new reference creation operations that accept a priority as an additional parameter.

```
module RTPortableServer
{
  local interface POA : PortableServer::POA
  {
    Object create_reference_with_priority (in CORBA::RepositoryId intf,
                                           in RTCORBA::Priority priority)
    raises (WrongPolicy);

    Object create_reference_with_id_and_priority
                                          (in PortableServer::ObjectId oid,
                                           in CORBA::RepositoryId intf,
                                           in RTCORBA::Priority priority)
    raises (WrongPolicy);

    PortableServer::ObjectId activate_object_with_priority
                                          (in PortableServer::Servant p_servant,
                                           in RTCORBA::Priority priority)
    raises (ServantAlreadyActive, WrongPolicy);
```

```
    void activate_object_with_id_and_priority (in PortableServer::ObjectId oid,
                                               in PortableServer::Servant p_servant,
                                                   in RTCORBA::Priority priority)
    raises ( ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy );

  };
};
```

When an application links to the `RTPortableServer` library, all POA references are implemented by the real-time POA. Thus an application may create a POA using the ordinary method of invoking `PortableServer::POA::create_POA()`, then narrow the newly-created POA reference to `RTPortableServer::POA`. The following example shows this technique. Once again, error checking has been omitted for the sake of clarity:

```
#include <tao/corba.h>
#include <tao/RTPortableServer/RTPortableServer.h>

int main (int argc, char* argv[])
{
  // Initialize the ORB.
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

  // Get the RootPOA.
  CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());

  // Create a child POA.
  CORBA::PolicyList policies;
  policies.length(2);
  policies[0] = poa->create_lifespan_policy (PortableServer::PERSISTENT);
  policies[1] = poa->create_id_assignment_policy (PortableServer::USER_ID);
  PortableServer::POAManager_var mgr = poa->the_POAManager();
  PortableServer::POA_var child_poa =
    poa->create_POA ("Child POA", mgr.in(), policies);

  // Use the new POA as a RT POA.
  RTPortableServer::POA_var rt_poa =
    RTPortableServer::POA::_narrow (child_poa.in());

};
```

8.3.4 Real-Time Priority Mapping

A Real-Time Operating System (RTOS) must support the concept of discrete thread priority to adequately leverage benefits of RT CORBA. Thread priorities are usually represented as a range of integer values and a direction of precedence. A thread's priority is used to determine its execution eligibility, with threads having higher precedence being eligible for execution ahead of threads with lower precedence. RT CORBA refers to an operating system's representation of priority as *native* priority. Native priorities are not used directly by RT CORBA, but are represented in IDL to provide a mechanism for allowing native code to interact with RT CORBA. The native priority type is defined as follows:

```
module RTCORBA {
  typedef short NativePriority;
 // ...
}
```

A short integer has the range -32768 to 32767, however only a subset of this range will be valid in any particular operating system.

To allow all objects participating in a distributed real-time application to have a consistent notion of thread priority, RT CORBA supplies a second type to represent portable priority:

```
module RTCORBA {
  typedef short Priority;
  const Priority minPriority = 0;
  const Priority maxPriority = 32767;
  //...
};
```

The type `Priority` is used when referring to "portable" priority values. Although it is a signed short integer, an RT CORBA priority value is always positive, its range being constrained by the constants `minPriority` and `maxPriority`. With CORBA priorities, higher values take precedence over lower values.

Conversion between native priority and RT CORBA priority is achieved through mapping functions. The RT CORBA specification uses the following declaration to represent the mapping between priority types:

```
module RTCORBA {
```

```
  native PriorityMapping;
};
```

The specification recognizes that this mapping behavior is frequently invoked. To minimize performance impact, the declaration uses a *native* type rather than an *interface*. Use of an interface would necessitate the use of a CORBA object, and require obtaining a reference to the same. On the other hand, native types are specified directly in the target language binding. The C++ binding for `PriorityMapping` is a class in the `RTCORBA` namespace.

```
namespace RTCORBA {
  class PriorityMapping
  {
  public:
    virtual CORBA::Boolean to_native (RTCORBA::Priority corba_priority,
                                      RTCORBA::NativePriority& native_priority );
    virtual CORBA::Boolean to_CORBA (RTCORBA::NativePriority native_priority,
                                     RTCORBA::Priority& corba_priority );
  };
};
```

The methods `to_native()` and `to_CORBA()` may be called several times by an ORB during an invocation. To provide the greatest possible efficiency these methods do not throw exceptions, not even CORBA System exceptions. However these functions will return `FALSE` if the input value is outside the allowed range for that type. For both of these functions, the first argument has the semantics of an *in*, supplying input, the second argument being an *out* for receiving the converted results.

An RT ORB conforming to the specification will make use of these mapping functions throughout the course of an invocation. If the call to either mapping function returns `FALSE`, the ORB is required to stop processing the invocation and throw a `DATA_CONVERSION` system exception to the invoking application. Note that this exception may not be propagated if the error is the result of a `oneway` operation.

8.3.5 The Real-Time Current

The `CORBA::Current` interface serves as a common base for interfaces providing context-specific information in clients and servers. Applications use the `RTCORBA::Current, which derives from CORBA::Current`, to determine the priority of the current invocation.

```
module RTCORBA {
  local interface Current : CORBA::Current
  {
    attribute Priority the_priority;
  };
};
```

The real-time Current is obtained by narrowing the reference to the object returned by calling `resolve_initial_references("RTCurrent")`.

The `RTCORBA::Priority` value obtained from the Current may be mapped to a native priority by using the `PriorityMapping` object, as discussed in section 8.3.4.

8.3.6 The Real-Time Mutex

Real-time CORBA defines the local *Mutex* object to present a portable interface for controlling access by multiple threads. RT CORBA mutexes are created by the RTORB. The RT CORBA Mutexes are local objects, thus references to the Mutexes are not allowed to cross process boundaries.

The interface definition for Mutex is in the `RTCORBA` module. Here is the Mutex definition.

```
module RTCORBA
{
  // Mutex.
  local interface Mutex
  {
      void lock ();
      void unlock ();
      boolean try_lock (in TimeBase::TimeT max_wait);
    // if max_wait = 0 then return immediately
  };
};
```

A real-time CORBA Mutex is functionally similar to a common mutual exclusion lock, which is used to ensure that only one thread has access to critical sections of code at a time. A mutex has two states, locked and unlocked. A mutex starts out in the unlocked state. The mutex operations are:

- `lock()` sets the mutex state to locked, when called on an unlocked mutex. If the mutex is already locked, then `lock()` blocks until the owning thread calls `unlock()`. A mutex is not recursive, therefore if a thread attempts to call `lock()` twice, it will deadlock.

- `try_lock()` attempts to set the state of a mutex to locked. It returns `TRUE` if it successfully locks the mutex, or `FALSE` if the mutex cannot be locked within the `max_wait` time period. If zero is passed as the `max_wait` time, `try_lock()` immediately returns `FALSE` if the mutex cannot be locked. The type `TimeT`, used as the argument to `try_lock()`, is defined in the module `TimeBase`.
- `unlock()` resets the mutex state back to unlocked. If there is a single thread waiting to acquire the lock, it will do so at this time. If multiple threads are waiting, and either `SCHED_FIFO` or `SCHED_RR` scheduling policies (described in Table 8-3) are in effect, the mutex is acquired in priority order. If the threads implementation does not support the aforementioned scheduling policies, or a different scheduling policy is used, the order in which threads are awarded the lock is undefined.

RT CORBA Mutexes are supplied by the `RTORB`. The operations which affect the life cycle of Mutexes are shown below:

```
module RTCORBA
{
  local interface RTORB
  {
   // ...
    Mutex create_mutex ();
    void destroy_mutex (in Mutex the_mutex);

    // TAO specific ...
  };
};
```

- `RTORB::create_mutex()` creates a new instance of a mutex and returns a reference to it. An RT CORBA mutex is a reference-counted object.
- `RTORB::destroy_mutex (in Mutex the_mutex)` cleans up the resources held by the mutex object. In TAO, `destroy_mutex()` removes the mutex from the internal table of named mutexes.

In addition to these RT-CORBA-compliant operations, TAO provides extra functionality. See 8.5.3 for details on the TAO extension.

8.3.7 Thread Pools

A *thread pool* is a collection of threads that are all separately available to perform work on behalf of the ORB. A typical thread pool consists of two or

more threads, all waiting for incoming requests. The threads in a pool may be of the same default priority, or they may be grouped together in *lanes*. Each lane is designated by a certain priority, and each thread in the lane executes at that priority. Upon creation, a thread pool may have a number of pre-created *static* threads. A number of *dynamic* threads may be created later if needed. By default, dynamic threads live forever after they are created. The `-RTORBDynamicThreadRunTime` and `-RTORBDynamicThreadIdleTimeout` options can be used to specify when dynamic threads should be destroyed. See 8.5.7.1 for details on these options. The RT CORBA specification provides a mechanism that allows a lane of higher priority to *borrow* a thread from a lane of lesser priority in the same thread pool, if needed.

Note *TAO thread pools are not fully compliant with the RT CORBA specification. Specifically, request buffering and thread borrowing are not supported. Attempts to specify request buffering or thread borrowing result in a* `CORBA::NO_IMPLEMENT` *exception.*

8.3.7.1 Interface Specifications

The `RTCORBA` module defines several types that are used in conjunction with thread pools, as shown here:

```
module RTCORBA
{
  // Threadpool types.
  typedef unsigned long ThreadpoolId;

  struct ThreadpoolLane
  {
    Priority lane_priority;
    unsigned long static_threads;
    unsigned long dynamic_threads;
  };
  typedef sequence <ThreadpoolLane> ThreadpoolLanes;

  // Threadpool Policy.
  const CORBA::PolicyType THREADPOOL_POLICY_TYPE = 41;
  local interface ThreadpoolPolicy : CORBA::Policy
  {
      readonly attribute ThreadpoolId threadpool;
  };
```

Thread pools are identified by a value of type `RTCORBA::ThreadpoolId`. A POA may be associated with a single thread pool. This is done by supplying a `ThreadpoolPolicy` as part of the `PolicyList` supplied to `create_POA()`. Because a POA may only be associated with a single thread pool, there is only one `ThreadpoolId` in the `ThreadpoolPolicy`. The thread pool policy may also be applied to the ORB to set the default thread pool used by subsequently-created POAs.

Thread pools are created by the RTORB, using the IDL operations shown here:

```
module RTCORBA
{
  local interface RTORB
  {
    // Threadpool creation/destruction.
    exception InvalidThreadpool {};

    ThreadpoolId create_threadpool (in unsigned long stacksize,
                                    in unsigned long static_threads,
                                    in unsigned long dynamic_threads,
                                    in Priority default_priority,
                                    in boolean allow_request_buffering,
                                    in unsigned long max_buffered_requests,
                                    in unsigned long max_request_buffer_size);

    ThreadpoolId create_threadpool_with_lanes (
                                      in unsigned long stacksize,
                                      in ThreadpoolLanes lanes,
                                      in boolean allow_borrowing,
                                      in boolean allow_request_buffering,
                                      in unsigned long max_buffered_requests,
                                      in unsigned long max_request_buffer_size);

    void destroy_threadpool (in ThreadpoolId threadpool)
      raises (InvalidThreadpool);

  };
};
```

8.3.7.2 Creating Thread Pools

Thread pools are managed by the ORB. This is similar to using multiple threads with the thread-pool reactor (see Chapter 18). The main advantage to using RT CORBA thread pools is that multiple lanes can be created with differing thread priorities.

Here we show the steps necessary to create a thread pool. First, the thread pool lanes must be defined. To do this, an `RTCORBA::ThreadpoolLanes` sequence is instantiated and filled with information specifying the lane priorities and the number and types of threads to create:

```
// Set the thread pool lane size.
const CORBA::ULong TOTAL_LANES = // get a value from somewhere
RTCORBA::ThreadpoolLanes lanes(TOTAL_LANES);
lanes.length(TOTAL_LANES);

// Initialize the lane information.
for (CORBA::ULong i = 0; i < TOTAL_LANES; ++i) {
  lanes[i].static_threads  = 1;
  lanes[i].dynamic_threads = 0;

  // Initialize the lane_priority (a value between 0 - 32767).
  lanes[i].lane_priority = // some priority value
}
```

Next, use the `RTORB` to create the thread pool and child POA:

```
// Create the threadpool and get back its ThreadpoolId.
RTCORBA::ThreadpoolId threadpool_id =
  rt_orb->create_threadpool_with_lanes(0,      // Stack Size
                                       lanes,
                                       false,  // Allow borrowing
                                       false,  // Allow request buffering
                                       0,      // Max buffered requests
                                       0);     // Max request buffer size

// Create a policy list.
CORBA::PolicyList poa_policy_list(2);
poa_policy_list.length(2);

// Set the priority model (client propagated for this example).
poa_policy_list[0] =
  rt_orb->create_priority_model_policy(RTCORBA::CLIENT_PROPAGATED, 0);

// Set the thread pool id.
poa_policy_list[1] =
  rt_orb->create_threadpool_policy(threadpool_id);

// Create the child poa with the policy list.
PortableServer::POA_var child_poa = root_poa->create_POA("child_poa",
                                                          poa_manager.in(),
                                                          poa_policy_list);
```

Operations dispatched to servants activated in this new POA will run in one of the threads from the thread pool at a priority requested by the client application. See the example in 8.9 for more details on the use of thread pools.

8.3.7.3 Thread Pool Lane Listen Endpoints

Persistent object references require the same endpoint(s) to be used each time the server is run. When thread pools with lanes are required to support persistent object references, you must supply explicit endpoints for each lane. This is accomplished by specifying the `-ORBLaneListenEndpoints` ORB initialization option. This option takes two parameters—the lane identifier and an endpoint specification. The lane identifier is a composite value of the form *n:m* where *n* is the thread pool number, starting with 1, and *m* is the lane index within the thread pool, starting with 0. The endpoint specification parameter is of the form of an ordinary endpoint specification such as may be provided to the `-ORBListenEndpoints` ORB initialization option.

For example, an application creating a thread pool with three lanes might specify the following lane listen endpoints:

```
-ORBLaneListenEndpoints 1:0 iiop://:1234 \
-ORBLaneListenEndpoints 1:1 iiop://:1235 \
-ORBLaneListenEndpoints 1:2 iiop://:1236
```

to define three explicit endpoints for the pool's lanes. Both of the thread pool number and lane index can use the wildcard character of '*'. This value means that the specified endpoint should be used for all lanes that match. For example:

```
-ORBLaneListenEndpoints *:* iiop://myhost
```

would specify that all lanes in all thread pools should use the network interface named myhost. This would be useful with systems that have multiple network interfaces.

Another ORB initialization option, `-ORBLaneEndpoint`, is an alias for `-ORBLaneListenEndpoints`. See 17.13.40 and 17.13.41 for more information about these options.

8.3.8 End-to-End Priority Propagation

One of the biggest challenges in using CORBA for real-time applications is making sure that the priority of an activity is honored by all of the objects and operations involved in carrying out that activity. End-to-end predictability requires that both client and server respect the system-wide priorities during request processing. Furthermore, the system needs to bound the priority inversions and latencies during end-to-end processing. Of course, ultimately the RT ORB relies upon the real-time operating system to schedule threads appropriately. The RT CORBA specification does not attempt to define or dictate real-time OS capabilities.

The RT CORBA specification defines two common priority propagation models: *client-propagated* and *server-declared*. The priority model is selected through the `RTCORBA::PriorityModelPolicy` as follows:

```
module RTCORBA
{
  // Priority Model Policy.
  const CORBA::PolicyType PRIORITY_MODEL_POLICY_TYPE = 40;
  enum PriorityModel
  {
    CLIENT_PROPAGATED,
    SERVER_DECLARED
  };

  local interface PriorityModelPolicy : CORBA::Policy
  {
    readonly attribute PriorityModel priority_model;
    readonly attribute Priority server_priority;
  };
};
```

The `PriorityModelPolicy` is a *client-exposed* policy, meaning that a client ORB knows what policy is in force and can adjust itself accordingly. As defined in the CORBA messaging specification, the value of this property is communicated through `IOP::ServiceContexts`. In 8.6 and 8.7, we describe how to use the client-propagated and server-declared priority models to specify how priorities are propagated end-to-end across ORB endsystems.

8.3.9 Explicit Binding

Frequently, real-time CORBA systems need to explicitly bind object references prior to their first use. To fulfill this requirement, RT CORBA

makes use of the operation `CORBA::Object::_validate_connection()` with the appropriate policies set, which preestablishes a connection between the client and server. This operation forces the server to allocate basic resources necessary to service requests on the physical connection used to generate the call.

8.3.10 Priority-banded Connections

The server uses the `create_priority_banded_connection_policy()` operation on the RTORB to create priority bands. This operation takes as a parameter a sequence of `PriorityBand` structures called `RTCORBA::PriorityBands`. As shown in the IDL below, a `PriorityBand` structure contains two priorities, `low` and `high`. The low priority represents the minimum priority of the band and the high priority represents the maximum priority of the band.

```
module RTCORBA
{
  // Priorities.
  typedef short NativePriority;
  typedef short Priority;

  // PriorityBandedConnectionPolicy.
  struct PriorityBand
  {
    Priority low;
    Priority high;
  };
  typedef sequence <PriorityBand> PriorityBands;
};
```

Each band corresponds to one or more lanes within a thread pool on the server. The following example shows how to create priority-banded connection policies.

```
  // Create the sequence of priority bands.
  const CORBA::ULong NUM_BANDS = // some number of bands
  RTCORBA::PriorityBands bands(NUM_BANDS);
  bands.length(NUM_BANDS);

  // Populate the priority band sequence.
  bands[0].low = low_value1;
  bands[0].high = high_value1;
  bands[1].low = low_value2;
  bands[1].high = high_value2;
```

```
...

// Create the policy list and add the priority banded connection policy.
CORBA::PolicyList policy_list(1);
policy_list.length(1);
policy_list[0] = rt_orb->create_priority_banded_connection_policy(bands);

// Create a child poa with the priority banded connection policy.
PortableServer::POA_var child_poa = root_poa->create_POA("child_poa",
                                                         poa_manager.in(),
                                                         policy_list);
```

The priority band chosen depends on the priority model specified by the server. See 8.6 and 8.7 for more information on priority models.

8.3.11 Private Connections

Ordinarily, the client ORB is allowed to reuse a connection to support many object references. However, multiplexing requests for different object references on a single connection carries a risk of blocking a thread if the connection is busy during an invocation triggered by another thread. The RT CORBA `PrivateConnectionPolicy` allows the application to specify that dedicated, non-multiplexed connections will be used for certain object references. When this policy is applied to an ORB or a thread, each object reference will have a private connection associated with it. Be aware that the connection is associated with the object reference, not the ORB or thread. If multiple threads use the same object reference, they may still share a connection.

An application should use the `RTORB` to create instances of the `RTCORBA::PrivateConnectionPolicy` policy. This policy may be applied to an `RTORB` via the `ORBPolicyManager`, or to a specific thread via the `PolicyCurrent`. In the following example, we apply the private connection policy to the `PolicyCurrent` to ensure that requests on object references invoked from within this thread will be carried over private connections to the servers hosting the referenced objects.

```
CORBA_Object_var obj = orb->resolve_initial_references("PolicyCurrent");
CORBA::PolicyCurrent_var policy_current =
  CORBA::PolicyCurrent::_narrow(obj.in());

// Create a policy list to supply to the PolicyCurrent.
CORBA::PolicyList policy_list;
policy_list.length(1);
```

```
  policy_list[0] = rt_orb->create_private_connection_policy();

  policy_current->set_policy_overrides (policy_list,
                                        CORBA::SET_OVERRIDE);
```

8.3.12 Protocol Properties

Protocol Properties were introduced into the RT CORBA specification to allow users to specify a preferred protocol to use for connections between clients and servers, and to fine tune the parameters of the physical transport over which GIOP requests are made. As shown below, the `ProtocolProperties` interface does not contain any operations or attributes:

```
module RTCORBA
{
  // Protocol Properties.
  local interface ProtocolProperties
  {
  };
};
```

The `TCPProtocolProperties` interface corresponds to GIOP over TCP/IP (IIOP) and allows the application to specify the sizes of the TCP send and receive buffers, as well as the TCP keep-alive, routing, and delay attributes. The `enable_network_priority` attribute is a TAO extension and is described in 8.5.2. The `TCPProtocolProperties` interface is shown here:

```
module RTCORBA
{
  local interface TCPProtocolProperties : ProtocolProperties
  {
    attribute long send_buffer_size;
    attribute long recv_buffer_size;
    attribute boolean keep_alive;
    attribute boolean dont_route;
    attribute boolean no_delay;
    attribute boolean enable_network_priority;
  };
};
```

TAO also provides protocol property interfaces for the TAO-specific transport protocols UIOP, DIOP, SCIOP, and SHMIOP as shown below:

```
module RTCORBA
{
  // Communication over Unix Domain Sockets (Local IPC).
  local interface UnixDomainProtocolProperties : ProtocolProperties
  {
    attribute long send_buffer_size;
    attribute long recv_buffer_size;
  };

  // Communication over Shared Memory.
  local interface SharedMemoryProtocolProperties : ProtocolProperties
  {
    attribute long send_buffer_size;
    attribute long recv_buffer_size;
    attribute boolean keep_alive;
    attribute boolean dont_route;
    attribute boolean no_delay;
    attribute long preallocate_buffer_size;
    attribute string mmap_filename;
    attribute string mmap_lockname;
  };

  // Communication over UDP (DIOP)
  local interface UserDatagramProtocolProperties : ProtocolProperties
  {
    attribute long send_buffer_size;
    attribute long recv_buffer_size;
    attribute boolean enable_network_priority;
  };

  // Communication over SCTP (SCIOP)
  local interface StreamControlProtocolProperties : ProtocolProperties
  {
    attribute long send_buffer_size;
    attribute long recv_buffer_size;
    attribute boolean keep_alive;
    attribute boolean dont_route;
    attribute boolean no_delay;
    attribute boolean enable_network_priority;
  };

};
```

RT CORBA specifies two new policies for configuring protocols:

- `ServerProtocolPolicy`
- `ClientProtocolPolicy`

When the server or client protocol policy is created, several protocols can be configured at the same time by specifying the protocols and their properties in a sequence called a `ProtocolList`. The order in which protocols are specified in the list indicates the order of preference. The server-side ORB lists protocol information in this same order in IORs created through that ORB; the client-side ORB considers the protocols in this same order when binding to the server. Server and client protocol policies are created via factory operations on the RT ORB interface. These type and interface definitions are shown below:

```
module RTCORBA
{
  local interface ProtocolProperties { };

  struct Protocol
  {
    IOP::ProfileId     protocol_type;
    ProtocolProperties orb_protocol_properties;
    ProtocolProperties transport_protocol_properties;
  };
  typedef sequence<Protocol> ProtocolList;

  // Server Protocol Policy
  local interface ServerProtocolPolicy : CORBA::Policy
  {
    readonly attribute ProtocolList protocols;
  };

  // Client Protocol Policy
  local interface ClientProtocolPolicy : CORBA::Policy
  {
    readonly attribute ProtocolList protocols;
  };

  // RT ORB factory operations for protocol policies
  local interface RTORB
  {
    // ...
    ServerProtocolPolicy create_server_protocol_policy (
      in ProtocolList protocols);

    ClientProtocolPolicy create_client_protocol_policy (
      in ProtocolList protocols);
  };
};
```

The following example shows how to specify protocol properties on the server side. In this example, we configure two protocols—IIOP and the TAO-specific UIOP—with IIOP specified as the preferred protocol (because it is first in the protocol list).

```
#include <tao/RTCORBA/RTCORBA.h>

int main (int argc, char* argv[])
{
  try
  {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get a reference to Root POA and activate it.
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Get the RTORB.
    obj = orb->resolve_initial_references ("RTORB");
    RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow (obj.in());

    // Create a protocol list with 2 elements
    RTCORBA::ProtocolList protocols(2);
    protocols.length(2);

    // Specify the TCP properties.
    CORBA::Long send_buffer_size = 16384;
    CORBA::Long recv_buffer_size = 16384;
    CORBA::Boolean keep_alive = true;
    CORBA::Boolean dont_route = false;
    CORBA::Boolean no_delay = true;

    // Create TCP protocol properties.
    RTCORBA::TCPProtocolProperties_var tcp_properties =
      rtorb->create_tcp_protocol_properties(
        send_buffer_size,
        recv_buffer_size,
        keep_alive,
        dont_route,
        no_delay );

    // Specify the TCP (IIOP) protocol as the primary protocol type.
    protocols[0].protocol_type = TAG_INTERNET_IOP;
    protocols[0].transport_protocol_properties =
      RTCORBA::ProtocolProperties::_duplicate (tcp_properties.in ());
```

```
// Next, use UIOP with the default values if IIOP fails.
protocols[1].protocol_type = TAO_TAG_UIOP_PROFILE;
protocols[1].transport_protocol_properties =
  RTCORBA::UnixDomainProtocolProperties::_nil ();

// Create server protocol policy and insert it into a policy list.
CORBA::PolicyList policy_list;
policy_list.length(1);
policy_list[0] = rtorb->create_server_protocol_policy (protocols);

// Set the policy on a new child POA.
PortableServer::POA_var child_poa = poa->create_POA (
  "childPOA", mgr.in (), policy_list);

//...
```

8.3.13 Other Real-Time CORBA Features

Other aspects of the RT CORBA specification are covered in other chapters of this guide.

- Timeouts

  See 6.3.3 for information on request and reply timeouts.

- Reliable oneways

  See 6.3.4 for information on specifying the reliability of `oneway` requests.

- Asynchronous Invocations

  Asynchronous invocations of operations on CORBA objects are covered in 6.2.

8.4 Dynamic Scheduling

CORBA Real-Time Scheduling is defined in section 3 of the Real-Time CORBA specification, version 1.2 (OMG document formal/05-01-04). This specification supersedes the static real-time scheduling service defined in version 1.0.

Note *This specification is also known as the “static” specification. The OMG’s document access page refers to a separate document, formal/03-11-01, as the “dynamic” specification. In fact, version 1.2 supersedes version 2.0, and*

represents an integration of the dynamic scheduling features first specified in version 2.0 back into the 1.x specification branch.

Static scheduling depends upon knowing the run time characteristics of a system *a priori* in order to determine the scheduling needs. Dynamic scheduling, on the other hand, is much more flexible, allowing for run time selection of scheduling behavior. This is achieved through the implementation of scheduling disciplines. The RT CORBA specification lists some well known disciplines, such as *Fixed Priority Scheduling* (i.e., *Static*) and *Earliest Deadline First*. These disciplines are used to evaluate thread execution order when using a scheduler to dispatch threads. See section 8.4.5 for more information on the scheduler.

A distributable thread is a schedulable entity that maps to an operating system thread while in the context of a particular process, but may pass between processing nodes carrying schedule requirements as service contexts. Distributable threads may be newly spawned, or may be created from an existing thread. Distributable threads carry context information with them as they span nodes. This context information is accessed by the scheduler in each node through the use of portable interceptors.

8.4.1 Distributable Threads

The RT CORBA specification defines *distributable threads* as schedulable threads that can run across many nodes in a distributed system. The interface `RTScheduling::DistributableThread` is used by both the `Current` and `Scheduler` interfaces, but may also be used by the application. Distributable thread creation occurs in two ways:

- Creating a new operating system thread by calling `RTScheduling::Current::spawn()`, described in 8.4.3.
- Invoking `begin_scheduling_segment()`, also described in 8.4.3. Calling `begin_scheduling_segment()` when not in a distributable thread makes the current thread distributable.

The distributable thread interfaces shown below provide only the means of canceling the thread while it is running. Distributable threads work in conjunction with *scheduling segments*, which in turn are managed by the `RTScheduling::Current`.

```
module RTScheduling
{
  local interface DistributableThread
  {
      enum DT_State
      {
        ACTIVE,
        CANCELLED
      };

      void cancel();
      readonly attribute DT_State state;
  };
};
```

The `RTScheduling::DistributableThread::cancel()` operation may be used to cancel a running thread. The thread's current state may be referenced via the `state` attribute. Any thread may cancel a distributable thread, however it is dangerous to try to cancel a thread other than the current thread. While a distributable thread may span many nodes of a distributed system, the interface shown above is local, meaning it only has effect in the current process. It is possible to obtain a reference to a `DistributableThread` object in one process while the actual head of the thread is in another process. Calling `cancel()` on such a `DistributableThread` will then not have the desired effect as the cancellation is indicated as a change of state in the span. While a distributable thread's `cancel()` operation may be invoked at any time, in any process, doing so does not necessarily alter the thread's processing.

8.4.2 Real-Time Scheduling Thread Action

Thread Action objects, which implement the `RTScheduling::ThreadAction` interface, are the RT CORBA equivalent of thread functions.

```
module CORBA
{
  native VoidData;
};

module RTScheduling
{
  local interface ThreadAction
  {
      void do (in CORBA::VoidData data);
```

```
    };
  };
```

Objects that implement the `ThreadAction` interface are required to provide a method to be invoked when using `RTScheduling::Current::spawn()` to start a new distributable thread. The `spawn()` operation takes a reference to a `ThreadAction` and invokes `do()` on it in the native thread creation method. The `data` argument is a native type, `CORBA::VoidData`, which is defined as a `void*` by the C++ language mapping and is used the same way that a `void*` argument is supplied to a C/C++ thread function.

8.4.3 Real-Time Scheduling Current

The real-time scheduling current, `RTScheduling::Current`, is a specialization of the `RTCORBA::Current` interface providing additional operations related to creation and management of distributable threads. Through the `RTScheduling::Current`, distributable threads may be spawned or the current thread may be converted into a distributable thread. A distributable thread may span several scheduling segments. The `Current` provides the means to identify all the segments of the currently operating thread.

TAO's definition of `RTScheduling::Current` deviates slightly from the RT CORBA specification in one important regard. In TAO, the signature of the `spawn()` operation contains additional parameters that are not present in the RT CORBA specification. The RT CORBA specification defines the `spawn()` operation as follows:

```
module RTScheduling
{
  local interface Current : RTCORBA::Current
  {
    // Standard RT CORBA spawn() operation definition.
    DistributableThread spawn (in ThreadAction    start,
                               in unsigned long     stack_size,
                               in RTCORBA::Priority base_priority);
  };
};
```

TAO extends this interface to include parameters that would normally be supplied to the `begin_scheduling_segment()` operation.

```
module RTScheduling
```

```
{
  local interface Current : RTCORBA::Current
  {
    // TAO's extended spawn() operation definition.
    DistributableThread spawn (in ThreadAction      start,
                               in CORBA::VoidData   data,
                               in string            name,
                               in CORBA::Policy     sched_param,
                               in CORBA::Policy     implicit_sched_param,
                               in unsigned long     stack_size,
                               in RTCORBA::Priority base_priority);
  };
};
```

TAO's extended `spawn()` operation combines two steps into one. The RT CORBA specification is very open in terms of how distributable threads may interact with schedules. It states that schedules, or schedulers, are not required, but does not completely describe what should happen when they are not present. By allowing distributable threads to exist outside of schedules, the specification imposes an explicit burden for any thread to also invoke `begin_scheduling_segment()` when starting. This requires that the schedule segment information either be known implicitly by the thread function, or it must be communicated to the thread function via the `data` parameter, which usually carries application data for the thread. TAO's implementation of `spawn()` carries the schedule segment information separately from any application-specific thread data. It also uses a wrapper function to invoke `begin_scheduling_segment()` before, and `end_scheduling_segment()` after, invoking the thread function.

The `RTScheduling::Current` is responsible for managing scheduling segments. Threads invoke `begin_scheduling_segment()` to start a new, and possibly nested, scheduling segment. Threads invoke `end_scheduling_segment()` at the completion of a scheduling segment. Threads may also invoke `update_scheduling_segment()` to modify the attributes of the current scheduling segment, if necessary.

```
module RTScheduling
{
  local interface Current : RTCORBA::Current
  {
    exception UNSUPPORTED_SCHEDULING_DISCIPLINE {};

    void begin_scheduling_segment (in string name,
                                   in CORBA::Policy sched_param,
```

```
                                    in CORBA::Policy implicit_sched_param)
        raises (UNSUPPORTED_SCHEDULING_DISCIPLINE);

    void update_scheduling_segment (in string name,
                                    in CORBA::Policy sched_param,
                                    in CORBA::Policy implicit_sched_param)
        raises (UNSUPPORTED_SCHEDULING_DISCIPLINE);

    void end_scheduling_segment (in string name);
  };
};
```

The operations above allow RT CORBA applications to manage scheduling segments. Scheduling segments may be named, and the name can be used when ending a scheduling segment to ensure the correct segment is ended; otherwise, the name has no purpose. The `sched_param` and `implicit_sched_param` parameters are used to describe how a scheduling segment relates to other scheduling segments within the scheduler. The `sched_param` parameter is used to explicitly define the new or updated scheduling segment. The `implicit_sched_param` parameter is used whenever a nested scheduling segment is started with a nil `sched_param` value. There are no constraints placed on the definition of the scheduling parameters by the RT CORBA specification other than the fact that they must derive from the `CORBA::Policy` interface. Scheduling parameters are created by schedulers that implement a particular scheduling discipline, by way of a factory operation that returns scheduling parameters specific to that discipline. The RT CORBA specification describes a number of well-known scheduling disciplines, and includes example IDL specifications for each. These scheduling disciplines are briefly described in section 8.4.5. See also section 3.7 of the RT CORBA 1.2 specification for a more complete description of these scheduling disciplines.

8.4.4 Real-Time Scheduling Resource Manager

The RT CORBA specification defines a scheduler-aware specialization of the `RTCORBA::Mutex` called `RTScheduling::ResourceManager`. This interface defines no new operations or attributes, it simply provides a means to distinguish a resource manager from a base mutex. A `ResourceManager` is created by calling the `create_resource_manager()` operation on the `RTScheduling::Scheduler` interface. See 8.4.5 for more details on the `RTScheduling::Scheduler` interface. Operations that acquire (`RTScheduling::ResourceManager::lock()` or

`RTScheduling::ResourceManager::try_lock()`) or release (`RTScheduling::ResourceManager::unlock()`) a real-time CORBA resource manager are defined as *scheduling points*, meaning the scheduler will have a chance to run and reassess the scheduling parameters, thereby ensuring that the scheduling discipline is maintained. Distributable threads may share local resources by using `RTScheduling::ResourceManager` operations.

8.4.5 Real-Time Scheduling Scheduler

A scheduler is responsible for allocating system resources and determining timeliness. The real-time CORBA scheduler is intended to be pluggable. ORB implementations may provide different scheduler implementations as long as they implement the `RTScheduling::Scheduler` interface, defined in `RTScheduler.pidl`. This pluggable architecture affords implementors the flexibility to provide specific schedulers, each with particular scheduling characteristics. The scheduler works in conjunction with portable interceptors to track scheduling segments of a distributable thread. The real-time scheduler interface is described in 8.4.5.3, where it is presented in context with the other elements of real-time scheduling.

8.4.5.1 TAO's RTScheduleManager

While the real-time CORBA specification describes the `RTScheduling::Scheduler` interface and its role in distributed real-time systems, it does not mandate an implementation or even a scheme for managing schedulers. The specification states that using a scheduler is optional, and for ORBs that have a scheduler, it may be accessed by calling `resolve_initial_references("RTScheduler")`. TAO, on the other hand, provides a TAO-specific initial reference called the `"RTScheduleManager"`. A call to `resolve_initial_references("RTSchedulerManager")` returns an object through which an application may supply its own scheduler.

8.4.5.2 Scheduling Disciplines

A scheduler is used to implement a particular scheduling discipline. It may implement a well-known scheduling discipline such as *Earliest Deadline First* or something esoteric such as *Most Important First*. The real-time CORBA specification describes four well-known scheduling disciplines:

- Fixed Priority

- Earliest Deadline First (EDF)
- Least Laxity First (LLF)
- Maximize Accrued Utility (MAU)

For each of these disciplines, the specification provides a recommended interface, including the definition of the particular scheduling parameter type and how it might be created.

In addition, TAO provides an example scheduler that implements a scheduling discipline that is not one of the above well-known disciplines, but that may be useful in some situations:

- Most Important First (MIF)

The following sections briefly describe each of the well-known scheduling disciplines, the associated scheduling parameter types, and the scheduler interface.

8.4.5.3 Common RTScheduling::Scheduler Interface

Each of the discipline-specific schedulers listed above extends the common base interface `RTScheduling::Scheduler`. The RT CORBA specification provides an interface for this base type. TAO extends the `Scheduler` interface, as shown in 8.5.4. The RT CORBA specified interface focuses on resource management and is shown here:

```
module RTScheduling
{
  local interface Scheduler
  {
      exception INCOMPATIBLE_SCHEDULING_DISCIPLINES {};

      attribute CORBA::PolicyList scheduling_policies;
      readonly attribute CORBA::PolicyList poa_policies;
      readonly attribute string scheduling_discipline_name;

      ResourceManager create_resource_manager (
        in string name,
        in CORBA::Policy scheduling_parameter);

      void set_scheduling_parameter (
        inout PortableServer::Servant resource,
        in string name,
        in CORBA::Policy scheduling_parameter);
  };
```

```
};
```

Note that the exception `INCOMPATIBLE_SCHEDULING_DISCIPLINES` is not explicitly referenced by any of the operations defined in the specification.

The `scheduling_policies` and `poa_policies` attributes provide the scheduler with a way of listing any POA policies that might be required for the scheduler to work. It is reasonable and typical for these attributes to return nil values.

The `scheduling_discipline_name` attribute is simply a name that may be queried by the application to identify the particular discipline implemented by the scheduler.

The `create_resource_manager()` operation is used to create instances of `RTScheduling::ResourceManager`, providing it with a name and optionally associating it with a scheduling parameter.

The `set_scheduling_parameter()` operation is intended to be a hook giving schedulers a way to associate certain schedule parameters, such as a fixed priority ceiling, to a particular resource. The resource in this case is a servant.

It is not necessary for a scheduler to implement all or any of these operations. They are defined merely as hooks to provide scheduler implementors a means to express any sort of scheduling needs in code.

8.4.5.4 Fixed Priority Scheduling

The *fixed priority* scheduling discipline, also known as *rate-monotonic* scheduling, is characterized by static schedules where the schedule is completely determined prior to deployment. An example of fixed priority scheduling is available in `$TAO_ROOT/examples/RTScheduling/Fixed_Priority_Scheduler`. The fixed priority scheduler is a suitable replacement for the older static scheduling model.

Fixed priority scheduling is defined in the `FP_Scheduling` module. The fixed priority scheduling parameter is defined by the `FP_Scheduling::SegmentSchedulingParameterPolicy` local interface. It has an attribute representing a single RT CORBA priority value. The fixed priority scheduler is defined by the `FP_Scheduling::FP_Scheduler` interface.

```
module FP_Scheduling
{
  local interface SegmentSchedulingParameterPolicy : CORBA::Policy
  {
    attribute RTCORBA::Priority value;
  };

  local interface FP_Scheduler : RTScheduling::Scheduler
  {
    SegmentSchedulingParameterPolicy create_segment_scheduling_parameter (
      in RTCORBA::Priority segment_priority);
  };
};
```

8.4.5.5 Earliest Deadline First Scheduling

The *earliest deadline first* (EDF) scheduling discipline places emphasis on those threads that must complete the soonest. EDF scheduling segments are based on a required completion time, which may be modified further by an indication of a thread's importance depending upon a particular scheduler's implementation. TAO does not currently provide a reference implementation of the EDF scheduling discipline. The RT CORBA specification, however, provides a suggested EDF scheduling interface in the `EDF_Scheduling` module, shown here:

```
module EDF_Scheduling
{
  struct SchedulingParameter
  {
    TimeBase::TimeT deadline;
    long importance;
  };

  local interface SchedulingParameterPolicy : CORBA::Policy
  {
    attribute SchedulingParameter value;
  };

  local interface Scheduler : RTScheduling::Scheduler
  {
    SchedulingParameterPolicy create_scheduling_parameter (
      in SchedulingParameter value);
  };
};
```

Note that the `deadline` member of the scheduling parameter is of type `TimeBase::TimeT`. This type is defined as part of the CORBA Time Service specification.

8.4.5.6 Least Laxity First Scheduling

The *least laxity first* (LLF) scheduling discipline favors threads that have the least amount of time they can wait before missing a deadline (i.e., being late). The real-time CORBA specification defines laxity as:

```
laxity = deadline - current time - estimated remaining time to completion
```

Where the *estimated remaining time to completion* is based on an estimated initial execution time and the current time executed. In the LLF scheduling discipline, threads that have a lower laxity value are scheduled first.

Least laxity first scheduling is defined in the `LLF_Scheduling` module:

```
module LLF_Scheduling
{
  struct SchedulingParameter
  {
    TimeBase::TimeT deadline;
    TimeBase::TimeT estimated_initial_execution_time;
    long importance;
  };

  local interface SchedulingParameterPolicy : CORBA::Policy
  {
    attribute SchedulingParameter value;
  };

  local interface Scheduler : RTScheduling::Scheduler
  {
    SchedulingParameterPolicy create_scheduling_parameter (
      in SchedulingParameter value);
  };
};
```

The LLF scheduling parameter is similar to the EDF scheduling parameter in that it includes a deadline and an importance qualifier. An estimate of the amount of time this schedule segment is anticipated to take to complete is also included, so that the scheduler can compute the laxity for this segment. TAO does not currently provide a reference implementation of the LLF scheduling discipline.

8.4.5.7 Maximize Accrued Utility Scheduling

The *maximize accrued utility* (MAU) scheduling discipline is a special case of the Earliest Deadline First scheduling discipline. In MAU scheduling, a special function, called the *utility function*, is used to compute a *utility* value for a thread. The utility of a thread is a measure of the likelihood that a thread will complete close to, but prior than, its deadline. A thread that can complete close to its deadline has a greater utility value than a thread completing much earlier. A thread that will complete after its deadline may have a zero or negative utility value. The MAU discipline seeks a schedule that results in maximal accrued (i.e., summed) utility.

Maximize accrued utility scheduling is defined in the `Max_Utility_Scheduling` module:

```
module Max_Utility_Scheduling
{
  struct SchedulingParameter
  {
    TimeBase::TimeT deadline;
    long importance;
  };

  local interface SchedulingParameterPolicy : CORBA::Policy
  {
    attribute SchedulingParameter value;
  };

  local interface Scheduler : RTScheduling::Scheduler
  {
    SchedulingParameterPolicy create_scheduling_parameter (
      in SchedulingParameter value);
  };
};
```

The MAU scheduling parameter is similar to the EDF scheduling parameter in that it includes a deadline and an importance qualifier. TAO does not currently provide a reference implementation of the MAU scheduling discipline.

8.4.5.8 Most Important First Scheduling

TAO provides an example of a scheduler that implements a somewhat different scheduling discipline from the well-known disciplines described above. The *most important first* (MIF) scheduler makes use of a thread's *importance* to determine which thread should execute. The MIF scheduler

interface is defined in the `MIF_Scheduling` module in `$TAO_ROOT/examples/RTScheduling/MIF_Scheduling.idl`.

```
module MIF_Scheduling
{
  local interface SegmentSchedulingParameterPolicy : CORBA::Policy
  {
    attribute short importance;
  };

  local interface MIF_Scheduler : RTScheduling::Scheduler
  {
    SegmentSchedulingParameterPolicy create_segment_scheduling_parameter (
      in short segment_importance);
  };
};
```

An implementation of MIF scheduling is provided in `$TAO_ROOT/examples/RTScheduling/MIF_Scheduler`.

8.5 TAO's Implementation of Real-Time CORBA

This section provides details on using TAO's implementation of RT CORBA and RT CORBA extensions provided in TAO. Topics in this section include:

- Priority Mapping in TAO
- Enabling Network Priority in TAO
- Using TAO's Named Mutexes
- Building RT CORBA Support into TAO
- Configuring the RT ORB Component

8.5.1 Priority Mapping In TAO

TAO offers three priority mappings: continuous, direct and linear. These priority mappings are explained in the following subsections.

8.5.1.1 Continuous Priority Mapping

The continuous priority mapping, as shown in Figure 8-3, uses only the first *n* priorities of CORBA's priority range, providing a one-to-one mapping of native-to-CORBA priorities, where *n* is the number of discrete native priority

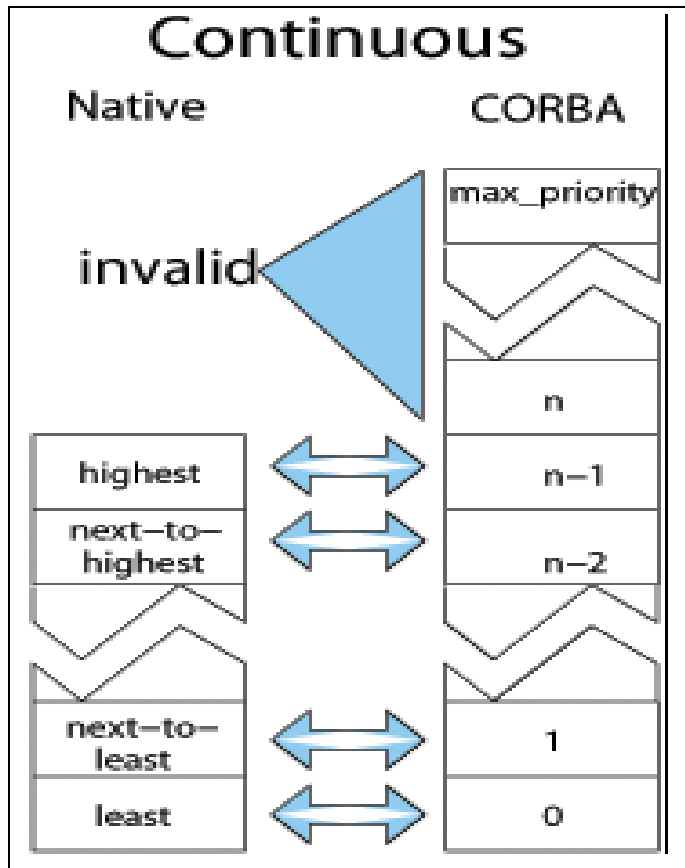


Figure 8-3 TAO Continuous Mapping Behavior

values permitted by the operating system. The lowest native priority maps to a CORBA priority value of 0, the next higher native priority maps to 1, etc.

Because the mapping functions, `PortableMapping::to_native()` and `PortableMapping::to_CORBA()` are idempotent, you can start with a priority value, convert it twice and end up with the same value. This advantage of continuous mapping is countered by the disadvantage that part of the CORBA priority range is invalid. If you are using more than one RTOS in a distributed environment, where each RTOS defines a different number of discrete priority values, some CORBA priorities will map to valid native priorities in one RTOS, but be invalid in another RTOS.

8.5.1.2 Direct Priority Mapping

The direct priority mapping is similar to the continuous priority mapping in that some of the CORBA priority values are invalid. However, native priority values are passed directly through as CORBA priority values. As shown in Figure 8-4, no priority transform is applied to the native value. This mapping would only be desirable where native priority values fall within the minimum and maximum values of the CORBA priority range.

It is possible to have native priority values that do not directly map onto the CORBA priority range. The direct mapping would be useless in this situation. An attempt to set the priority from a native value that does not map into the

CORBA priority range would cause a `CORBA::DATA_CONVERSION` exception to be raised.

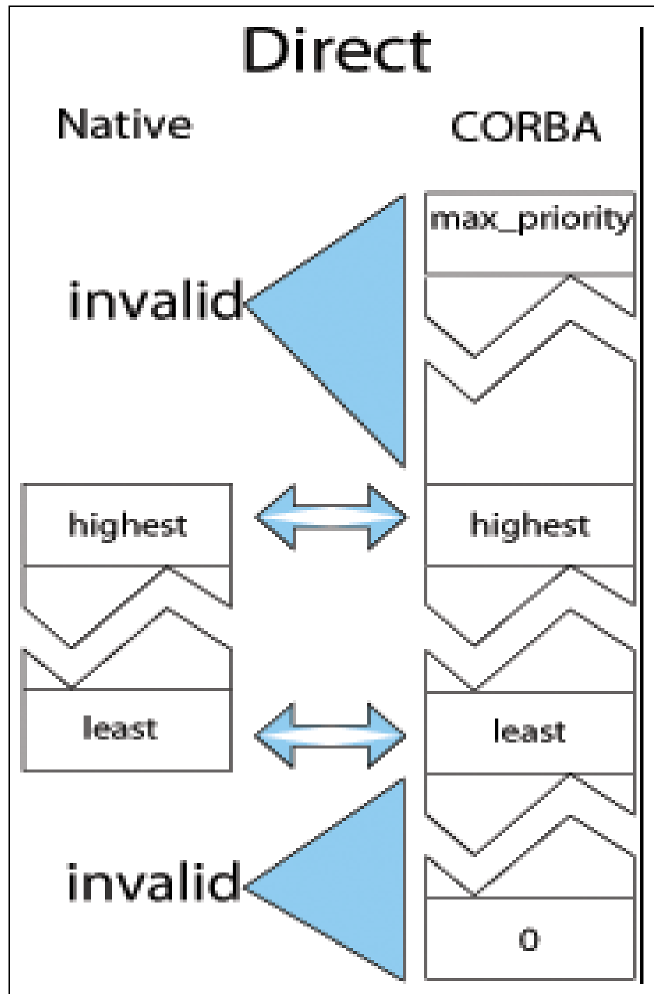


Figure 8-4 TAO Direct Mapping Behavior

8.5.1.3 Linear Priority Mapping

The linear priority mapping is one-to-many, providing a range of CORBA priority values for each native priority value. The size of the range is (`RTCORBA::maxPriority - RTCORBA::minPriority`)/*n*, where *n* is the number of discrete native priorities. For instance, a native system offering 16

priority values would result in a map with 2048 CORBA priority values for each native priority value.

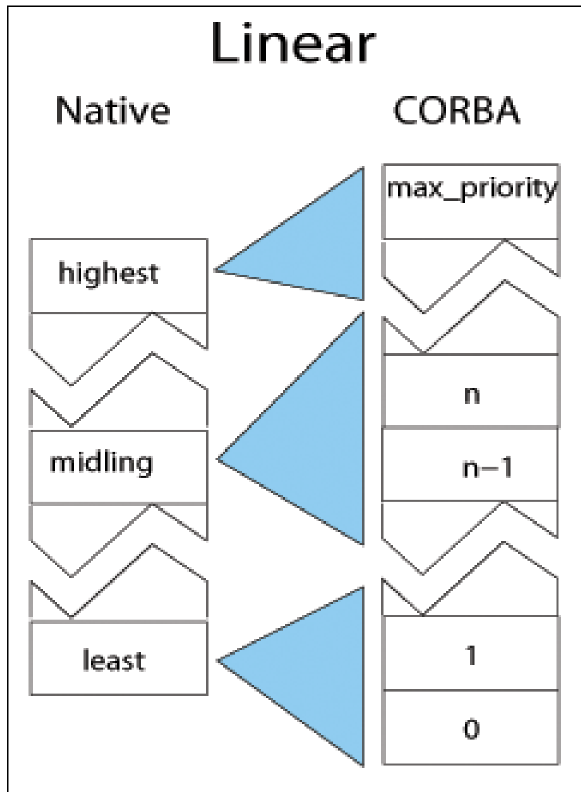


Figure 8-5 TAO Linear Mapping Behavior

The risk with the linear priority mapping is that rounding can occur. In other words, a CORBA priority converted to a native priority may be converted back to a different CORBA priority value than the original. Furthermore, if objects are hosted on processes using different real-time operating systems with sufficiently different priority ranges, the linear mapping may result in a native priority being communicated, then communicated back as a different native priority.

8.5.1.4 Using TAO's Priority Mappings

The RT CORBA specification only defines *what* a priority-mapping object is to do, not *how* it is to do it. Furthermore, the specification does not define a

means of accessing the priority-mapping object. What *is* defined is RTScheduling, which hides the details of priority mapping from the application.

To ensure that priority mappings are used consistently in an environment, TAO is configured at run time through the `RT_ORB_Loader` service configuration option `ORBPriorityMapping` (see 8.5.7). Within your application, the ORB's priority-mapping object may be obtained through a helper object, an instance of `TAO_Priority_Mapping_Manager` (or its alias `RTCORBA::PriorityMappingManager`) as shown below:

```
class TAO_RTCORBA_Export TAO_Priority_Mapping_Manager :
  public virtual CORBA::LocalObject
{
public:
//... implementation details not shown

  // Get the current priority mapping.
  RTCORBA::PriorityMapping* mapping (void);

  // Set a new priority mapping.
  void mapping (RTCORBA::PriorityMapping* new_mapping);

};
```

Since the priority-mapping manager, `TAO_Priority_Mapping_Manager`, inherits from `CORBA::LocalObject`, it ultimately inherits from `CORBA::Object`, which means that a reference to it may be obtained from the ORB via a call to `resolve_initial_references()`. The name used for this object is `PriorityMappingManager`. The object reference returned from `resolve_initial_references()` must be narrowed using a call to `RTCORBA::PriorityMappingManager::_narrow()`. The `mapping()` operation is then called on the resulting object reference to obtain a pointer to the `RTCORBA::PriorityMapping` object as shown below:

```
  CORBA::Object_var obj =
   orb->resolve_initial_references ("PriorityMappingManagaer");
  RTCORBA::PriorityMappingManager_var mapping_manager =
    RTCORBA::PriorityMappingManager::_narrow (obj.in());
  RTCORBA::PriorityMapping* priority_mapping = mapping_manager->mapping();
```

Note that the `PriorityMappingManager` retains ownership of the `PriorityMapping`, so it should not be deleted.

8.5.2 Enabling Network Priority in TAO

Some environments provide support for differentiated classes of network service, and allow applications to specify their network quality of service needs. A common mechanism for providing differentiated classes of service on IP networks is the *Differentiated Services* (diffserv) architecture defined by the Internet Engineering Task Force (IETF) Diffserv Working Group. In the diffserv architecture, applications encode a particular six-bit pattern into a field, called the *DS field*, of the IP packet header, thereby marking a packet to receive a particular forwarding treatment, or *per-hop behavior* (PHB), at each network node. The Diffserv Working Group has standardized a small number of specific per-hop behaviors and a recommended bit pattern, or *codepoint*, for each one. These PHBs and their recommended codepoints are defined in various IETF Requests for Comments (RFCs). For more information on Differentiated Services and Diffserv Codepoints (DSCPs), see RFC 2474, RFC 2475, RFC 2597, RFC 2598, and RFC 3246, all of which are available from the IETF at `<http://www.ietf.org/rfc/>`.

The RT CORBA specification does not provide a way to map RT CORBA priorities to network priorities. As an extension to RT CORBA, TAO provides a mechanism to map RT CORBA priorities to network priorities via diffserv codepoints. Applications enable this mapping via a TAO-specific extension to the `TCPProtocolProperties` described in 8.3.12. The interface is:

```
module RTCORBA
{
  local interface TCPProtocolProperties : ProtocolProperties
  {
    attribute long send_buffer_size;
    attribute long recv_buffer_size;
    attribute boolean keep_alive;
    attribute boolean dont_route;
    attribute boolean no_delay;
    attribute boolean enable_network_priority;
  };
};
```

When the `enable_network_priority` attribute is set to `TRUE`, mapping between RT CORBA priority and a corresponding network priority diffserv codepoint is enabled. The diffserv codepoint resulting from this mapping is encoded into the DS field in the IP packet header for GIOP requests and replies.

8.5.2.1 Enabling Network Priority in a Client

If you enable network priorities on the client side, the RT CORBA priority of the invoking thread is mapped to a corresponding diffserv codepoint and set in the IP packet header for GIOP requests. To enable network priorities on the client side, you must create a `TCPProtocolProperties` object, set the `enable_network_priority` attribute to `TRUE`, create a protocol properties policy, then set the protocol properties policy at the ORB, thread, or object level. An example of how to do this is shown below:

```
#include <tao/RTCORBA/RTCORBA.h>

int main (int argc, char* argv[])
{
  try
  {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get the RTORB.
    obj = orb->resolve_initial_references ("RTORB");
    RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow (obj.in());

    // Specify the TCP properties.
    CORBA::Long send_buffer_size = 16384;
    CORBA::Long recv_buffer_size = 16384;
    CORBA::Boolean keep_alive = true;
    CORBA::Boolean dont_route = false;
    CORBA::Boolean no_delay = true;

    // Create TCP protocol properties.
    RTCORBA::TCPProtocolProperties_var tcp_properties =
      rtorb->create_tcp_protocol_properties(
        send_buffer_size,
        recv_buffer_size,
        keep_alive,
        dont_route,
        no_delay );

    // Enable network priority.
    tcp_properties->enable_network_priority (1);

    // Create a protocol list and set the ProtocolProperties.
    RTCORBA::ProtocolList protocols;
    protocols.length (1);
    protocols[0].protocol_type = TAO_TAG_IIOP_PROFILE;
    protocols[0].transport_protocol_properties =
      RTCORBA::ProtocolProperties::_duplicate (tcp_properties.in ());
```

```
    // Create client protocol policy and insert it into a policy list.
    CORBA::PolicyList policy_list;
    policy_list.length (1);
    policy_list[0] = rtorb->create_client_protocol_policy (protocols);

    // Set the policy at the ORB level.
    CORBA::Object_var obj = orb->resolve_initial_references("ORBPolicyManager");
    CORBA::PolicyManager_var policy_manager =
      CORBA::PolicyManager::_narrow(obj.in());
    policy_manager->set_policy_overrides(policy_list, CORBA::SET_OVERRIDE);

    //...
```

In the example above, since the client protocol policy is set at the ORB level (by setting the policy overrides on the `ORBPolicyManager`), network priority mapping will be enabled for all requests invoked through that ORB.

8.5.2.2 Enabling Network Priority in a Server

If you enable network priorities on the server side, the RT CORBA priority of the request-dispatching thread is mapped to a corresponding diffserv codepoint and set in the IP packet header for the GIOP reply. To enable network priorities on the server side, create `TCPProtocolProperties` and set the `enable_network_priority` attribute to `TRUE`. Then, create a protocol-properties policy and set the policy on a new POA upon creation. An example of how to do this is shown below:

```
#include <tao/RTCORBA/RTCORBA.h>

int main (int argc, char* argv[])
{
  try
  {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get a reference to Root POA and activate it.
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate ();

    // Get the RTORB.
    obj = orb->resolve_initial_references ("RTORB");
    RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow (obj.in());
```

```
// Specify the TCP properties.
CORBA::Long send_buffer_size = 16384;
CORBA::Long recv_buffer_size = 16384;
CORBA::Boolean keep_alive = true;
CORBA::Boolean dont_route = false;
CORBA::Boolean no_delay = true;

// Create TCP protocol properties.
RTCORBA::TCPProtocolProperties_var tcp_properties =
  rtorb->create_tcp_protocol_properties(
    send_buffer_size,
    recv_buffer_size,
    keep_alive,
    dont_route,
    no_delay );

// Enable network priority.
tcp_properties->enable_network_priority (1);

// Create a protocol list and set the ProtocolProperties.
RTCORBA::ProtocolList protocols;
protocols.length (1);
protocols[0].protocol_type = TAO_TAG_IIOP_PROFILE;
protocols[0].transport_protocol_properties =
  RTCORBA::ProtocolProperties::_duplicate (tcp_properties.in ());

// Create server protocol policy and insert it into a policy list.
CORBA::PolicyList policy_list;
policy_list.length (1);
policy_list[0] = rtorb->create_server_protocol_policy (protocols);

// Set the policy on a new child POA.
PortableServer::POA_var child_poa = poa->create_POA (
  "childPOA", PortableServer::POAManager::_nil(), policy_list);

//...
```

In the example above, network priority mapping will be enabled in all replies generated from servants activated in the child POA. Replies generated from servants activated in the Root POA will not have network priority mapping enabled, so take care when using `_this` to get object references from servants.

8.5.2.3 Network Priority Mappings

Different mappings of RT CORBA priorities to network priorities are possible. A specific network priority mapping is provided via a TAO-specific

`NetworkPriorityMapping` interface. This mapping is similar to the RT CORBA priority to native priority mapping described in 8.3.4. The interface is shown below:

```
module RTCORBA
{
  typedef long NetworkPriority;
  native NetworkPriorityMapping;
};
```

The C++ binding for `NetworkPriorityMapping` is a class in the `RTCORBA` namespace as shown below:

```
namespace RTCORBA {
  class NetworkPriorityMapping
  {
  public:
    virtual CORBA::Boolean to_network (RTCORBA::Priority corba_priority,
                                       RTCORBA::NetworkPriority& network_priority );
    virtual CORBA::Boolean to_CORBA (RTCORBA::NetworkPriority network_priority,
                                     RTCORBA::Priority& corba_priority );
  };
};
```

The functions `to_network()` and `to_CORBA()` may be called several times by an ORB during an invocation. To provide the greatest possible efficiency these functions do not throw exceptions, not even CORBA System exceptions. However these functions will return `FALSE` if the input value is outside the allowed range for that type. For both of these functions, the first argument has the semantics of an *in*, supplying input, with the second argument being an *out* for receiving the converted results.

During an invocation, the RT ORB uses these mapping functions to map between RT CORBA priority and network priority. If the call to either mapping function returns `FALSE`, the ORB stops processing the invocation and throws a `DATA_CONVERSION` system exception to the invoking application.

By default, TAO uses a linear network-priority mapping that maps RT CORBA priority values to discrete diffserv codepoints recommended by various IETF RFCs. For example, RFC 2474 recommends specific codepoints for "default" per-hop behaviors and various Class Selector (CS) codepoints; RFC 2597 recommends specific codepoints for various Assured Forwarding

(AF) PHBs; RFC 3246 recommends a specific codepoint for Expedited Forwarding (EF) PHB. See the relevant RFCs for more information on the values and semantics of these PHBs and their recommended diffserv codepoints.

8.5.2.4 Using TAO's Network Priority Mappings

Similar to the run-time configuration of priority mappings described in 8.5.1.4, the network priority mapping used by TAO can be configured at run time through the `RT_ORB_Loader` option `RTORBNetworkPriorityMapping`, described in 8.5.7. Within your application, the ORB's network priority-mapping object may be obtained through either a helper object, an instance of `TAO_Network_Priority_Mapping_Manager` or its typedef `RTCORBA::NetworkPriorityMappingManager`.

```
class TAO_RTCORBA_Export TAO_Network_Priority_Mapping_Manager :
  public virtual CORBA::LocalObject
{
public:
//... implementation details not shown

  /// Get the current network priority mapping.
  RTCORBA::NetworkPriorityMapping* mapping (void);

  /// Set a new network priority mapping.
  void mapping (RTCORBA::NetworkPriorityMapping* new_mapping);

}
```

Because `TAO_Network_Priority_Mapping_Manager` is derived from `CORBA::LocalObject`, it ultimately is derived from `CORBA::Object`. As a descendant of `CORBA::Object`, a reference to the network priority mapping manager may be obtained from the ORB via a call to `resolve_initial_references()`. The name used for this object is `NetworkPriorityMappingManager`. The object reference returned from `resolve_initial_references()` must be narrowed using a call to `RTCORBA::NetworkPriorityMappingManager::_narrow()`. Then, a call must be made to the `mapping()` function on the resulting object reference to obtain a pointer to the `RTCORBA::NetworkPriorityMapping` object. An example of how to do this is shown below:

```
  CORBA::Object_var obj =
   orb->resolve_initial_references ("NetworkPriorityMappingManagaer");
  RTCORBA::NetworkPriorityMappingManager_var network_mapping_manager =
    RTCORBA::NetworkPriorityMappingManager::_narrow (obj.in());
  RTCORBA::NetworkPriorityMapping* network_priority_mapping =
    network_mapping_manager->mapping();
```

Note that the `NetworkPriorityMappingManager` retains ownership of the `NetworkPriorityMapping`, so it should not be deleted.

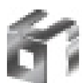
8.5.2.5 Implementing a Custom Network Priority Mapping

An application can implement a custom `NetworkPriorityMapping` by deriving a new class from `RTCORBA::NetworkPriorityMapping` and overriding the `to_network()` and `to_CORBA()` functions as shown below:

```
class CustomNetworkPriorityMapping :
  public virtual RTCORBA::NetworkPriorityMapping
{
  public:
    virtual CORBA::Boolean to_network (RTCORBA::Priority corba_priority,
                               RTCORBA::NetworkPriority& network_priority );
    virtual CORBA::Boolean to_CORBA (RTCORBA::NetworkPriority network_priority,
                                 RTCORBA::Priority& corba_priority );
};

CORBA::Boolean CustomNetworkPriorityMapping::to_network (
  RTCORBA::Priority corba_priority,
  RTCORBA::NetworkPriority& network_priority )
{
  network_priority = // map corba_priority to network_priority
  return true;
}

CORBA::Boolean CustomNetworkPriorityMapping::to_CORBA (
  RTCORBA::NetworkPriority network_priority,
  RTCORBA::Priority& corba_priority )
{
  corba_priority = // map network_priority to corba_priority
  return true;
}
```

To create an instance of our custom network priority mapping and instruct TAO to use it instead of the default network priority mapping, we do the following:

```
  CustomNetworkPriorityMapping* new_network_priority_mapping =
```



```
    new CustomNetworkPriorityMapping();
  network_mapping_manager->mapping(new_network_priority_mapping);
```

Note that the `NetworkPriorityMappingManager` takes ownership of the new network priority-mapping object.

8.5.3 Using TAO's Named Mutexes

The basic RT CORBA Mutex specification only requires that an RTORB create and destroy mutexes. TAO adds the option of maintaining a table of mutexes keyed by a name, freeing the application developer from managing references to the mutex as shown below:

```
module RTCORBA
{
  local interface RTORB
  {
    // ...
    // TAO specific
    // Named Mutex creation/opening
    exception MutexNotFound {};
    Mutex create_named_mutex (in string name,
                              out boolean created_flag);
    Mutex open_named_mutex (in string name)
      raises (MutexNotFound);

  };
};
```

In addition to supplying the RT-CORBA-compliant implementation of mutexes, TAO provides the following functions for accessing mutexes by name:

```
RTORB::create_named_mutex(in string name, out boolean created_flag)
```

Creates a new instance of a mutex and returns a reference to it, or returns a reference to an existing mutex of that name. The value of `created_flag` will be `TRUE` only if the mutex was created as a result of this call.

```
RTORB::open_named_mutex(in string name)
```

Returns a reference to an existing mutex only if it exists. Raises the TAO-specific exception `MutexNotFound` if the supplied name does not match any mutex.

8.5.4 Dynamic Scheduling and TAO

TAO fully supports the scheduling feature of RT CORBA. TAO deviates from the specification in a few important respects. In particular, the definitions of the `RTScheduling::Current` (see 8.4.3) and the base interface for the `RTScheduling::Scheduler` (see 8.4.5) are slightly different in TAO than in the specification.

TAO does not provide an implementation of `RTScheduling::Scheduler`. As discussed in 8.4.5, the scheduler is effectively an abstract interface, the real work of scheduling being handled by specialized schedulers. TAO does provide the framework for building your own scheduler and using it to manage the dispatching of distributable threads. TAO's framework includes a set of portable interceptors to provide message notification to the scheduler. These interceptors provide additional scheduling points in addition to the usual begin, update, and end of a scheduling segment.

TAO extends the base scheduler as defined in the RT CORBA specification to define several operations used by the portable interceptors or `RTScheduling::Current` as schedule evaluation points. TAO's scheduler interface definition is shown here:

```
module RTScheduling
{
  local interface Scheduler
  {
    void begin_new_scheduling_segment (in Current::IdType guid,
                                       in string name,
                                       in CORBA::Policy sched_param,
                                       in CORBA::Policy implicit_sched_param)
        raises (Current::UNSUPPORTED_SCHEDULING_DISCIPLINE);

    void begin_nested_scheduling_segment (in Current::IdType guid,
                                       in string name,
                                       in CORBA::Policy sched_param,
                                       in CORBA::Policy implicit_sched_param)
        raises (Current::UNSUPPORTED_SCHEDULING_DISCIPLINE);


    void update_scheduling_segment (in Current::IdType guid,
                                    in string name,
                                    in CORBA::Policy sched_param,
                                    in CORBA::Policy implicit_sched_param)
        raises (Current::UNSUPPORTED_SCHEDULING_DISCIPLINE);

    void end_scheduling_segment (in Current::IdType guid,
```

```
                                    in string name);

    void end_nested_scheduling_segment (in Current::IdType guid,
                                        in string name,
                                        in CORBA::Policy outer_sched_param);
  };
};
```

The scheduler's operations are invoked by the `RTScheduling::Current` whenever a new base or nested schedule segment is begun or ended, or when a segment is updated. These invocations provide the scheduler an opportunity to raise an exception (e.g., if an inappropriate exception is used) or to possibly dispatch or otherwise order existing distributable threads.

The TAO scheduler provides implementations for all of the portable interceptor interception points. With the exception of `receive_request()`, the signatures of all the interception operations is the same as that of the equivalent `PortableInterceptor::ServerInterceptor` or `PortableInterceptor::ClientInterceptor` operations. The difference in the signature of the `receive_request()` operation results from the need to supply additional information to the scheduler, as shown here:

```
module RTScheduling
{
  local interface Scheduler
  {
    void receive_request (in PortableInterceptor::ServerRequestInfo ri,
                          in Current::IdType guid,
                          in string name,
                          in CORBA::Policy sched_param,
                          in CORBA::Policy implicit_sched_param)
        raises (PortableInterceptor::ForwardRequest);
  };
};
```

As with the common scheduler operations shown in 8.4.5.3, TAO's additional operations provide additional hooks to allow the expression of any sort of scheduler that may be required by the application. The TAO-specific scheduler interface is intended to describe all possible schedule evaluation points, making up for apparent deficiencies in the RT CORBA specification's scheduler interface.

8.5.5 Enabling RT CORBA Support in TAO

RT CORBA features are controlled when building TAO by the rt_corba, corba_messaging, and minimum_corba build flags. The RT CORBA features are enabled in TAO by default because rt_corba is set to 1, corba_messaging is set to 1, and minimum_corba is set to 0 by default in the TAO make files. If any of these three flags is set to the opposite value, RT CORBA support will not be available in the TAO libraries. The preprocessor macros TAO_HAS_RT_CORBA, TAO_HAS_CORBA_MESSAGING, and TAO_HAS_MINIMUM_CORBA can also be used to set these flags (for example, in build environments where you are not using GNU Make). See A.4 for more information concerning these flags and macros.

8.5.6 Building Applications that use RT CORBA

TAO's support of basic RT CORBA features is implemented in the TAO_RTCORBA library, and the RTPortableServer module's features are implemented in the TAO_RTPortableServer library. Thus, applications that use TAO's RT CORBA features must link with one or both of these libraries. MPC projects for clients that use RT CORBA can simply inherit from the rt_client base project. MPC projects for servers that use RT CORBA can simply inherit from the rt_server base project. For example, here is the MPC file for the RT CORBA example in $TAO_ROOT/DevGuideExamples/RTCORBA:

```
project(*Server): rt_server {
  Source_Files {
    Messenger_i.cpp
    MessengerServer.cpp
    common.cpp
  }
}


project(*Client): rt_client {
  Source_Files {
    MessengerC.cpp
    MessengerClient.cpp
    common.cpp
  }
}
```

For more information on MPC, see <http://www.ociweb.com/products/MPC>.

8.5.7 Configuring RT CORBA at Run Time

Certain behavioral aspects of TAO's implementation of RT CORBA can be configured at run time.

8.5.7.1 RT ORB Loader

The RT ORB can be configured via the `RT_ORB_Loader` service object. The TAO `RT_ORB_Loader` takes initialization options that control the priority mapping type, the network priority mapping type, the scheduling policy, the thread scoping policy, and the lifetime of dynamic threads in a thread pool.

The `RT_ORB_Loader` object is initialized by supplying a service configuration directive, typically as a line in a `svc.conf` file. Service configuration directives are explained in further detail in Chapter 16. For example, an application that statically links the `RTCORBA` library may use a `static` directive as shown here. (The entire directive should appear on one line in the file.)

```
static RT_ORB_Loader "-ORBPriorityMapping linear -ORBSchedPolicy SCHED_FIFO
-ORBScopePolicy SYSTEM"
```

The possible configuration options for the `RT_ORB_Loader` service object are listed in the following tables.

Table 8-1 Priority Mapping Selection Option

Option	Description
`-ORBPriorityMapping` *`mapping_type`*	Selects the algorithm to use when mapping RT CORBA priorities to native operating system priorities.

Valid values for *mapping_type* are:

- `direct` (default)—RT CORBA priorities are passed directly through as native priorities. The entire range of RT CORBA priorities may not be usable.
- `linear`—The entire RT CORBA priority range is mapped to the entire native priority range.

- `continuous`—RT CORBA priorities are mapped onto native priorities based on the minimum native value. For example, if the minimum native value is 8 and the CORBA priority is 22 then the mapped priority would be 30.

See 8.5.1 for more information on TAO's priority mappings.

Table 8-2 Network Priority Mapping Selection Option

Option	Description
`-RTORBNetworkPriorityMapping` *mapping*	Selects the algorithm to use when mapping RT CORBA priorities to network priorities.

Currently, the only valid value for *mapping* is:

- `linear` (default)—RT CORBA priorities are mapped to a series of network priorities, represented as *diffserv codepoints*. The `linear` mapping is the only network priority mapping provided with TAO.

See 8.5.2 for more information on enabling network priorities in TAO.

Table 8-3 Scheduling Policy Selection Option

Option	Description
`-ORBSchedPolicy` *sched_policy*	Specifies the scheduling policy used when mapping RT CORBA priorities to native priorities.

On some operating systems, the choice of scheduling policy affects how the priority mapping computations are performed. Each scheduling policy may have different low and high priority values, and therefore would affect the priority at which threads may run. Valid values for *sched_policy* are:

- `SCHED_OTHER` (default)—System-dependent default scheduling policy.
- `SCHED_FIFO`—FIFO scheduling policy, wherein the highest priority thread that can run is scheduled first.
- `SCHED_RR`—Round-robin scheduling policy, wherein a fixed time-slice is provided for each thread.

You may need "super-user" or "Administrator" privileges to affect the scheduling policy.

Table 8-4 Scope Policy Selection Option

Option	Description
`-ORBScopePolicy` *`scope_policy`*	Specifies the thread-scheduling contention scope.

On some operating systems, the choice of scheduling contention scope affects the preemption and execution of the threads allocated to RT CORBA thread pools. Valid values for *scope_policy* are:

- `PROCESS` (default)—Threads compete for resources only with other threads in the same process.
- `SYSTEM`—Threads with system scheduling contention scope compete for resources against all threads in the system. This may not be available for all threading implementations.

Table 8-5 Dynamic Thread Options

Option	Description
`-RTORBDynamicThreadIdleTimeout` *`time`*	Specifies that dynamic threads should exit after being idle for *time* microseconds.
`-RTORBDynamicThreadRunTime` *`time`*	Specifies that dynamic threads should exit after *time* microseconds. Any work in progress will be completed before termination.

These options control when thread pools' dynamic threads should exit and end. If neither option is specified, dynamic threads will essentially run forever (until the ORB shuts down). The thread pool automatically creates new dynamic threads as required. See 8.3.7 for more details about thread pools.

8.5.7.2 RT Collocation Resolver

TAO normally optimizes collocated invocations (where the client and the target object are in the same address space). The effect of the ORB's default collocation optimization is such that the client thread is used to carry out the request. As described in 15.4.5, this effect may be undesirable in real-time applications, because the client thread may not be running at the priority at which the request should be processed, possibly leading to priority inversions.

Therefore, TAO's implementation of RT CORBA employs a special "real-time collocation resolver" (`RT_Collocation_Resolver`) to determine whether an invocation should be subject to collocation optimization. As described in 17.13.11, the RT collocation resolver considers factors other than just whether the request target object is in the same address space as the client when deciding if the collocation optimization should be applied.

The default behavior of TAO's RT collocation resolver is appropriate for most real-time CORBA applications. However, its behavior can be disabled if the default (non-RT) collocation optimization resolution mechanism is desired. The RT collocation resolver can be disabled using the `-ORBDisableRTCollocation` ORB initialization option.

Table 8-6 RT Collocation Resolver Option

Option	Description
`-ORBDisableRTCollocation {0	1}`	Controls how collocation optimization decisions are made in RT CORBA applications.

See 17.13.11 for more information on this option.

8.6 Client-Propagated Priority Model

In the client-propagated priority model, a CORBA request is executed at the priority specified by the client when the request is invoked. The CORBA priority of the request is carried with the invocation. In this model, the server is obligated to execute the servant code that handles the request in a thread running at the appropriate native priority, according to the selected priority mapping. The client's requested priority is carried to the server in a CORBA priority service context, and is passed back to the client from the server, along with the reply, through the service context.

8.6.1 Specifying the Client-Propagated Priority Model in the Server

A server specifies that it supports the client-propagated priority model by setting the `RTCORBA::CLIENT_PROPAGATED` policy during POA creation. The policy is then exported to clients via object references generated through that POA.

Here, we extend the Messenger example from Chapter 3 to set the `RTCORBA::CLIENT_PROPAGATED` priority model policy in the `Messenger` server:

```
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main (int argc, char* argv[])
{
  try
  {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get a reference to Root POA and activate it.
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Get the RTORB.
    obj = orb->resolve_initial_references ("RTORB");
    RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow (obj.in());

    // Use the RTORB to create the CLIENT_PROPAGATED priority model policy.
    CORBA::PolicyList policies;
    policies.length(1);
    RTCORBA::Priority default_server_priority = 30;
    policies[0] = rtorb->create_priority_model_policy(
      RTCORBA::CLIENT_PROPAGATED,
     default_server_priority);  // priority to use if not propagated from client

    // Create a child POA with CLIENT_PROPAGATED priority model in effect.
    PortableServer::POA_var child_poa =
      poa->create_POA ("MessengerPOA", mgr.in(), policies);

    // Create a Messenger_i servant.
    PortableServer::Servant_var<Messenger_i> messenger_servant
      = new Messenger_i(orb.in());

    // Register the servant with the new POA, obtain its object reference,
    // stringify it, and write it to a file
    PortableServer::ObjectId_var oid =
      child_poa->activate_object (messenger_servant.in());
    CORBA::Object_var messenger_obj = child_poa->id_to_reference (oid.in());
    CORBA::String_var str = orb->object_to_string (messenger_obj.in());
    std::ofstream iorFile ("Messenger.ior");
    iorFile << str.in() << std::endl;
```

```
    iorFile.close();
    std::cout << "IOR written to file Messenger.ior" << std::endl;

    // Accept requests from clients.
    orb->run();

    // Release resources.
    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
  }
  return 0;
}
```

8.6.2 Using the Client-Propagated Priority Model in the Client

To use the client-propagated priority model in the client, we first check to see if the object reference we obtain from the server is configured with the `RTCORBA::CLIENT_PROPAGATED` policy. We do this by calling the `_get_policy()` operation on the object reference, narrowing the resulting policy object to `RTCORBA::PriorityModelPolicy`, then testing it to see if its value is `RTCORBA::CLIENT_PROPAGATED`:

```
  // Get the Messenger object reference.
  CORBA::Object_var obj = orb->string_to_object ("file://Messenger.ior");
  Messenger_var messenger = Messenger::_narrow (obj.in());

  // Get the policy from the object reference.
  CORBA::Policy_var policy =
    messenger->_get_policy (RTCORBA::PRIORITY_MODEL_POLICY_TYPE);

  // Check to see if it is of type RTCORBA::PriorityModelPolicy.
  RTCORBA::PriorityModelPolicy_var priority_policy =
    RTCORBA::PriorityModelPolicy::_narrow (policy.in ());
  if (CORBA::is_nil (priority_policy.in ())) {
   std::cerr << "Messenger object does not support RTCORBA::PriorityModelPolicy"
              << std::endl;
    return 1;
  }

  // Check to see if the priority model is RTCORBA::CLIENT_PROPAGATED.
  RTCORBA::PriorityModel priority_model = priority_policy->priority_model();
  if (priority_model != RTCORBA::CLIENT_PROPAGATED) {
    std::cerr << "Messenger object does not support RTCORBA::CLIENT_PROPAGATED"
```

```
            << std::endl;
    return 1;
  }
```

Next, we use the `RTCORBA::Current` interface to set the priority of the calling thread to the priority we want propagated to the server with the request. We call `resolve_initial_references("RTCurrent")` on the ORB to obtain the `RTCORBA::Current`, then use the `RTCORBA::Current` object to set the priority of the current thread to the CORBA priority at which we want the request to be processed by the server:

```
CORBA::Object_var current_obj = orb->resolve_initial_references ("RTCurrent");
RTCORBA::Current_var current = RTCORBA::Current::_narrow (current_obj.in());
RTCORBA::Priority desired_priority = 10;
current->the_priority (desired_priority);
```

Now, when we invoke an operation on the `Messenger` object reference, the priority will be carried with the request to the server's ORB, where it will be used to set the priority of the thread that processes the request:

```
CORBA::String_var message = CORBA::string_dup ("Howdy!");
messenger->send_message ("TAO User", "Test", message.inout());
```

8.7 Server-Declared Priority Model

In the server-declared priority model, a CORBA request is executed at the priority specified by the server as the default for CORBA objects managed by the POA to which the policy applies, or at the priority specified on a per-object-reference basis. The server-declared model is appropriate when some operations must always be invoked at the same priority, regardless of the client thread making the request. In this model, the priority at which invocations on an object reference will be executed by the server is published in the object reference, where it is available to the client-side ORB. The client's ORB can use this priority information to, for example, set the priority of the calling thread or choose the appropriate connection to the server over which to send the request. The priority is not carried with the request through the service context list as it is with the client-propagated priority model.

8.7.1 Specifying the Server-Declared Priority Model in the Server

The server specifies that it supports the server-declared priority model by setting the RTCORBA::SERVER_DECLARED policy, and the default priority value to use for executing requests, during POA creation. By default, operation invocations on CORBA objects managed by a particular POA will be executed at the default priority value specified at that POA's creation. The server-declared policy and the priority value are exported to clients via object references generated through that POA. The priority can also be set on a per-object-reference basis, as explained later in this section.

Here, we extend the Messenger example from Chapter 3 to set the RTCORBA::SERVER_DECLARED priority model policy in the Messenger server:

```
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main (int argc, char* argv[])
{
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get a reference to Root POA and activate it.
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Get the RTORB.
    obj = orb->resolve_initial_references ("RTORB");
    RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow (obj.in());

    // Use the RTORB to create the SERVER_DECLARED priority model policy.
    CORBA::PolicyList policies;
    policies.length(1);
    RTCORBA::Priority default_server_priority = 30;
    policies[0] = rtorb->create_priority_model_policy(
      RTCORBA::SERVER_DECLARED,
      default_server_priority);  // default priority to use

    // Create a child POA with SERVER_DECLARED priority model in effect.
    PortableServer::POA_var child_poa =
      poa->create_POA ("MessengerPOA", mgr.in(), policies);
```

```
    // Create a Messenger_i servant.
    PortableServer::Servant_var<Messenger_i> messenger_servant
      = new Messenger_i(orb.in());

    // Activate the Messenger in the new POA.
    PortableServer::ObjectId_var oid =
      child_poa->activate_object (messenger_servant.in());

   // Obtain the Messenger object reference, stringify it, and write it to a file
    CORBA::Object_var messenger_obj = child_poa->id_to_reference (oid.in());
    CORBA::String_var str = orb->object_to_string (messenger_obj.in());
    std::ofstream iorFile ("Messenger.ior");
    iorFile << str.in() << std::endl;
    iorFile.close();
    std::cout << "IOR written to file Messenger.ior" << std::endl;

    // Accept requests from clients.
    orb->run();

    // Release resources.
    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
  }
  return 0;
}
```

You can use operations defined on the `RTPortableServer::POA` interface to override the POA's default priority on a per-object-reference basis. For example, we can use the `activate_object_with_priority()` function rather than `activate_object()` to activate the `Messenger` object with a CORBA priority other than the default, specified when the POA was created:

```
    // Use the POA as a RT POA.
    RTPortableServer::POA_var rt_poa =
      RTPortableServer::POA::_narrow (child_poa.in());

    // Activate the Messenger in the new POA with a specific priority.
    RTCORBA::Priority messenger_specific_priority = 50;
    PortableServer::ObjectId_var oid =
      rt_poa->activate_object_with_priority (
        messenger_servant.in(),
        messenger_specific_priority);
```

Other operations on the `RTPortableServer::POA` interface are used similarly, including

- `create_reference_with_priority()`,
- `create_reference_with_id_and_priority()`, and
- `activate_object_with_id_and_priority()`,

depending upon the POA policies (e.g., `PortableServer::USER_ID` or `PortableServer::SYSTEM_ID`) in effect. If the POA does not support the `RTPortableServer::SERVER_DECLARED` priority policy, and you try to use one of the above operations to set the priority of the object reference, the `PortableServer::WrongPolicy` exception is raised.

8.7.2 Using the Server-Declared Priority Model in the Client

You do not need to do anything special on the client-side to use an object reference with the server-declared priority model. The server automatically executes operations invoked on that object reference at the priority declared in the object reference. However, clients can use the priority information in the object reference to make decisions about how (or whether) to invoke the request. In the following example, our `MessengerClient` will invoke the `send_message()` operation only if the `Messenger` object reference supports the server-declared priority model, and the priority value is sufficiently high:

```
// Get the Messenger object reference.
CORBA::Object_var obj = orb->string_to_object ("file://Messenger.ior");
Messenger_var messenger = Messenger::_narrow (obj.in());

// Get the policy from the object reference.
CORBA::Policy_var policy =
  messenger->_get_policy (RTCORBA::PRIORITY_MODEL_POLICY_TYPE);

// Check to see if it is of type RTCORBA::PriorityModelPolicy.
RTCORBA::PriorityModelPolicy_var priority_policy =
  RTCORBA::PriorityModelPolicy::_narrow (policy.in ());
if (CORBA::is_nil (priority_policy.in ())) {
  std::cerr << "Messenger object does not support RTCORBA::PriorityModelPolicy"
            << std::endl;
  return 1;
}

// Check to see if the priority model is RTCORBA::SERVER_DECLARED.
RTCORBA::PriorityModel priority_model = priority_policy->priority_model();
```

```
if (priority_model != RTCORBA::SERVER_DECLARED) {
  std::cerr << "Messenger object does not support RTCORBA::SERVER_DECLARED"
            << std::endl;
  return 1;
}

// Check to see if the server's priority value is high enough for us.
RTCORBA::Priority desired_priority = 30;
RTCORBA::Priority server_priority = priority_policy->server_priority();
if (server_priority < desired_priority) {
  std::cerr << "Messenger object's priority is too low." << std::endl;
  return 1;
}

// Send the message.
CORBA::String_var message = CORBA::string_dup ("Howdy!");
messenger->send_message ("TAO User", "Test", message.inout());
```

8.8 Using the RTScheduling::Current

The `RTScheduling::Current` exists to manage distributable threads. Through its interface, an application is able to spawn new distributable threads, and begin, update, and end scheduling segments. The Current object provides a thread with an identity as well as a way to obtain references to other distributable threads. The interface is described in detail in 8.4.3.

8.8.1 Spawning New Distributable Threads

The Current's `spawn()` operation will create a new distributable thread, to invoke the `do()` operation of the supplied `RTScheduling::ThreadAction` reference. Spawning a new distributable thread has the side effect of creating a new operating system thread bound to the current process. The `spawn()` operation allows you to set the priority of the new thread and supply a scheduling parameter appropriate for whatever scheduling discipline was chosen for the application. TAO's implementation of `spawn()` implicitly associates the newly created distributable thread with a scheduling segment by invoking `begin_scheduling_segment()` right before calling the thread function, then invoking `end_scheduling_segment()` immediately after.

The following code fragment, found in `$TAO_ROOT/tests/RTScheduling/DT_Spawn/test.cpp`, shows how to use the `RTScheduling::Current::spawn()` operation.

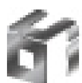
The application begins as any other CORBA application, setting up some pointers that will refer to various objects, then calling `CORBA::ORB_init()`. Note that, in this case, the `sched_param` and `implicit_sched_param` values are initialized to nil policy references. In this example, the particular scheduler we are using does not support any particular scheduling parameters.

```
#include "../Scheduler.h"   // for class TAO_Scheduler
#include "Thread_Action.h"  // for class Test_Thread_Action
#include "tao/RTScheduling/RTScheduler_Manager.h"

int main (int argc, char* argv [])
{
  CORBA::ORB_var orb;
  RTScheduling::Current_var current;

  const char * name = 0;
  CORBA::Policy_var sched_param = CORBA::Policy::_nil();
  CORBA::Policy_var implicit_sched_param = CORBA::Policy::_nil();

  Test_Thread_Action thread_action;

  try {
    orb = CORBA::ORB_init (argc, argv);
```

A distributable thread cannot exist without a scheduler, so the next portion of the code shows the initialization of a scheduler. TAO deviates a bit from the RT CORBA specification by supplying a *scheduler manager*, through which an externally created scheduler may be associated with the ORB. The scheduler used for this example is trivial; its implementation is available in the parent directory for this example, in `$TAO_ROOT/tests/RTScheduling/Scheduler.h.`

```
    CORBA::Object_var manager_obj =
      orb->resolve_initial_references ("RTSchedulerManager");

    TAO_RTScheduler_Manager_var manager =
      TAO_RTScheduler_Manager::_narrow (manager_obj.in());

    TAO_Scheduler* scheduler;
    ACE_NEW_RETURN (scheduler, TAO_Scheduler (orb.in ()), -1);

    manager->rtscheduler (scheduler);
```

Now that we have created a scheduler, we are ready to obtain a scheduling current and use it to spawn a distributable thread.



```
CORBA::Object_var current_obj =
  orb->resolve_initial_references ("RTScheduler_Current");
current = RTScheduling::Current::_narrow (current_obj.in());

try {
  ACE_DEBUG ((LM_DEBUG,
    ACE_TEXT("Invoking DT spawn without calling "
             "begin_scheduling_segment...\n")));

  ACE_CString data ("Harry Potter");
  char* thread_data = const_cast<char*>(data.c_str());
  CORBA::ULong stack_size = 0;
  RTCORBA::Priority base_priority = 0;
  current->spawn (&thread_action,
    thread_data,
    name,
    sched_param.in(),
    implicit_sched_param.in(),
    stack_size,
    base_priority);
```

At this point, the new thread is off and running. Notice that the `spawn()` operation takes several arguments

- `thread_action`—a reference to an object that derives from `RTScheduler::ThreadAction`. The `do()` operation of this object is invoked as the interesting part of the thread function. See 8.4.2 for more information on the `ThreadAction` interface. In this example, the class `Test_Thread_Action` is defined in `$TAO_ROOT/tests/RTScheduling/DT_spawn/Thread_Action.h`.
- `thread_data`—the argument that is passed to the thread function. In this case, the thread function takes a string as an argument.
- `name`—the identity supplied to the new distributable thread. As a side-effect, this identity is supplied to `begin_scheduling_segment()` as part of the thread function. In this example, the name is empty since no other distributable threads exist when `spawn()` is called.
- `sched_param`—depending on the scheduling discipline applied to the application, this argument may point to information required to schedule this thread. It is forwarded to `begin_scheduling_segment()`. In this example, the trivial scheduler does not require any parameters, thus `sched_param` is nil.

- `implicit_sched_param`—also passed to `begin_scheduling_segment()`, this argument is used to update the scheduling parameter that would be used if one were needed and `sched_param` was nil. In this example, scheduling parameters are not needed, so this value is also nil.
- `stack_size`—the next argument supplied is the size of the stack for the newly spawned thread. A non-zero value would indicate the desired stack size; passing zero, as in this example, indicates the operating system defined default stack size should be used.
- `base_priority`—the last argument supplied is the base priority value for the newly spawned thread. In this example, multiple thread priorities are not being used, so the base priority value is passed as zero.

The rest of the example in the distribution shows how to manage distributable threads using the current.

8.8.2 Managing Scheduling Segments

Scheduling segments are logical entities that may cross application boundaries. They are used to define the life span of a distributable thread. Depending upon the scheduling discipline used, a scheduling segment may have information associated with it, such as a completion deadline, that may cause the priority of the associated thread to be dynamically changed. Scheduling segments are used by the scheduler to associate requests or activities with application threads of appropriate priority. Scheduling segments have a beginning and an end; the end need not be in the same process as the beginning. Segments may be nested or consecutive. A distributable thread may span many scheduling segments. Scheduling segments may also be updated, changing the associated schedule parameter, and possibly raising or lowering the priority.

The following example demonstrates the management of scheduling segments. This example is found in `$TAO_ROOT/tests/RTScheduling/Current/Thread_Task.cpp`. Here, we look at `Thread_Task::svc()`, which provides an implementation of a thread function just like any class derived from `ACE_Task_Base`. The code below begins three nested scheduling segments, uses the current to obtain a list of the currently scheduled segments, then finally ends each segment.

```
int Thread_Task::svc (void)
```

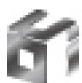
```
{
  try {
    const char * name = 0;
    CORBA::Policy_var sched_param = CORBA::Policy::_nil();
    CORBA::Policy_var implicit_sched_param = CORBA::Policy::_nil();

    this->current_->begin_scheduling_segment (
      "Fellowship of the Ring",
      sched_param.in(),
      implicit_sched_param.in());

    size_t count = 0;
    ACE_OS::memcpy (
      &count,
      this->current_->id()->get_buffer(),
      this->current_->id()->length());

    ACE_DEBUG ((LM_DEBUG,
      ACE_TEXT ("Starting Distributable Thread %d with 3 nested scheduling "
                "segments...\n"),
      count));

    // Start - Nested Scheduling Segment
    this->current_->begin_scheduling_segment (
      "Two Towers",
      sched_param.in(),
      implicit_sched_param.in());

    // Start - Nested Scheduling Segment
    this->current_->begin_scheduling_segment (
      "The Return of the King",
      sched_param.in(),
      implicit_sched_param.in());

    RTScheduling::Current::NameList* segment_name_list =
      this->current_->current_scheduling_segment_names();

    {
      ACE_GUARD_RETURN (TAO_SYNCH_MUTEX, ace_mon, *lock_, -1);
      ACE_DEBUG ((LM_DEBUG,
        ACE_TEXT ("Segment Names for DT %d :\n"),
        count));

      for (unsigned int i = 0; i < segment_name_list->length(); ++i)
      {
        ACE_DEBUG ((LM_DEBUG,
          ACE_TEXT ("%s\n"),
          (*segment_name_list)[i].in()));
      }
    }
```

```
      // End - Nested Scheduling Segment
      this->current_->end_scheduling_segment (name);

      // End - Nested Scheduling Segment
      this->current_->end_scheduling_segment (name);

      // End - Nested Scheduling Segment
      this->current_->end_scheduling_segment (name);

      ACE_DEBUG ((LM_DEBUG,
        ACE_TEXT ("DT %d terminated ...\n"),
        count));

      {
        ACE_GUARD_RETURN (TAO_SYNCH_MUTEX, ace_mon, *shutdown_lock_, -1);
        --active_thread_count_;
        if (active_thread_count_ == 0)
        {
          // Without this sleep, we will occasionally get BAD_INV_ORDER
          // exceptions on fast dual processor machines.
          ACE_OS::sleep (1);
          orb_->shutdown (0);
        }
      }
    }
    catch (CORBA::Exception& ex)
    {
      ACE_PRINT_EXCEPTION (ex, "Caught exception:");
      return -1;
    }

    return 0;
  }
```

8.9 Real-Time CORBA Examples

Throughout this chapter we have shown several code fragments illustrating the use of real-time CORBA features. Many of the examples we have shown are available as executable code distributed with TAO. This code is found in the following directories:

- `$TAO_ROOT/DevGuideExamples/RTCORBA`—This directory contains a variation of the Messenger application introduced in Chapter 3. This example uses the RTORB in the client to create a private connection



policy and uses the `RTCORBA::Current` for setting the priority to be consistent with the server. On the server side, the example shows the use of a thread pool with some number of lanes (the number depending on the native platform).

- `$TAO_ROOT/examples/RTCORBA/Activity`—The example in this directory highlights configuration of RTORB and RTPOA characteristics including the priority mapping policy.
- `$TAO_ROOT/examples/RTScheduling`—This directory contains several applications that demonstrate the use of distributable threads and various scheduling disciplines.
- `$TAO_ROOT/tests/RTCORBA`—This directory is the top level directory for a suite of tests that are run regularly to ensure the integrity of TAO's real-time CORBA implementation. These tests are focused on isolating particular features, but do provide insight into use of RT CORBA mutexes, priority-banded connections, client-propagated and server-declared priority models, collocation, and more.
- `$TAO_ROOT/tests/RTScheduling`—This directory includes a suite of tests focused on the use of the real-time scheduler. Specific tests highlight using distributable threads, canceling threads, and the scheduling current.

CHAPTER 9

Portable Interceptors

9.1 Introduction

Portable Interceptors in CORBA are objects that the ORB invokes at predefined points in the request and reply paths of an operation invocation (request interceptors) or during the generation of an IOR (IOR interceptors). As an application developer, you define the code executed in an interceptor. Since interceptors exist orthogonally to the operation invocations they monitor/modify, interceptor code can be added without affecting existing client and server code. Portable Interceptors can perform a variety of information collection and authentication tasks, including the following:

- Gathering debugging information about messages sent between clients and servers.
- Logging usage or access information about distributed objects.
- Performing security and authenticity checks in a distributed system.

TAO supports both request interceptors and IOR interceptors. Except for minor differences, TAO's implementation conforms to the CORBA 3.1 Portable Interceptor specification (Chapter 16 of OMG Document

formal/08-01-04). Portable interceptor support in TAO is controlled by the `interceptors` build flag. By default, interceptors are enabled unless `minimum_corba=1` or `interceptors=0`. For more information on build flags, refer to .

The Portable Interceptors functionality is mostly defined by local interfaces. In order to create your own portable interceptors you will need to be familiar with how to implement local objects. If you need more information how to implement local objects, see Chapter 12.

For more information on Portable Interceptors, read "*Object Interconnections: CORBA Metaprogramming Mechanisms, Part 1: Portable Interceptors Concepts and Components,*" by Douglas C. Schmidt and Steve Vinoski.

9.2 Using TAO Request Interceptors

TAO request interceptors can be attached at four points along the request/reply path of client and server communications. On the client, they can be activated when a request is sent or when a reply is received. On the server, they can be activated when a target operation is called, or when the reply is sent. There are ten interception operations, discussed in the next four sections. Figure 9-1 shows the relationship of clients and servers to the ten interception operations.

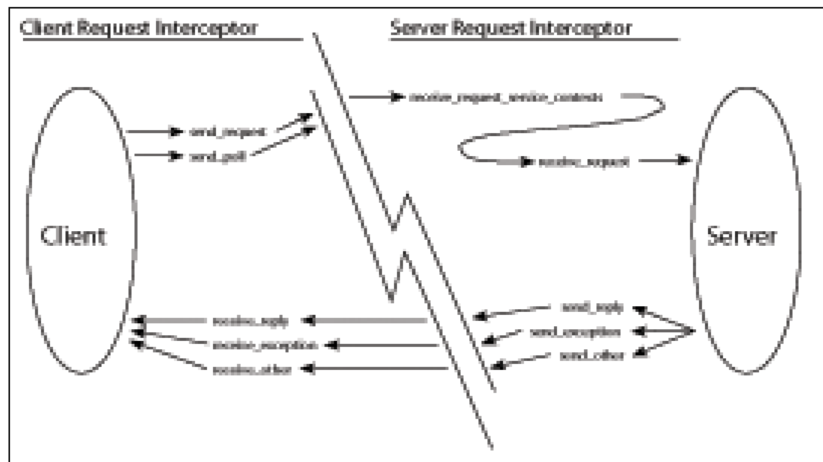


Figure 9-1 Client and Server Interception Operations

9.2.1 The Interceptor Interface

All interceptor interfaces defined in the CORBA specification are derived from the Interceptor interface:

```
module PortableInterceptor
{
  local interface Interceptor
  {
    readonly attribute string name;
    void destroy ();
  };
```

This interface is defined in `$TAO_ROOT/tao/PI/Interceptor.pidl`. All interceptors we define need to implement the `name` attribute and `destroy()` operation.

9.2.2 Client Request Interceptors

Client request interceptors implement the following interface, which ia defined in `$TAO_ROOT/tao/PI/ClientRequestInterceptor.pidl`:

```
module PortableInterceptor
{
  local interface ClientRequestInterceptor : Interceptor
  {
    void send_request  (in ClientRequestInfo ri) raises (ForwardRequest);
    void send_poll (in ClientRequestInfo ri);
    void receive_reply (in ClientRequestInfo ri);
    void receive_exception (in ClientRequestInfo ri)
      raises (ForwardRequest);
    void receive_other (in ClientRequestInfo ri) raises (ForwardRequest);
  };
};
```

To use interceptors on the client-side, developers define a new class that inherits from `PortableInterceptor::ClientRequestInterceptor` and `CORBA::LocalObject`, and implement the five operations that correspond to the client side interception points, plus operations that provide the name of the interceptor and destroy the interceptor.

TAO does not support the `send_poll()` *client interception point. The* `send_poll()` *operation is specific to time-independent invocations and TAO does not currently support time-independent invocations.*

9.2.2.1 Client Interception Points

The four client interception points available in TAO are `send_request()`, `receive_reply()`, `receive_exception()`, and `receive_other()`. The `send_request()` interception point allows an interceptor to monitor or change the service context before a request is sent to the server. The `receive_reply()` point intercepts a reply after it has returned from the server but before it has been passed to the client. The `receive_exception()` point is invoked when an exception occurs, before the exception is raised to the client. The `receive_other()` interception point allows the interceptor to monitor responses that are neither normal replies nor exceptions. An example of this would be a `LOCATION_FORWARD` response.

9.2.3 Server Request Interceptors

Interceptors on the server side implement the following interface, which is defined in `$TAO_ROOT/tao/PI_Server/ServerRequestInterceptor.pidl`:

```
module PortableInterceptor {

  local interface ServerRequestInterceptor : Interceptor
  {
    void receive_request_service_contexts (in ServerRequestInfo ri)
      raises (ForwardRequest);
    void receive_request (in ServerRequestInfo ri) raises (ForwardRequest);
    void send_reply (in ServerRequestInfo ri);
    void send_exception (in ServerRequestInfo ri) raises (ForwardRequest);
    void send_other (in ServerRequestInfo ri) raises (ForwardRequest);
  };

  // additional interfaces omitted for brevity.
};
```

To use interceptors on the server side, developers define a new class that inherits from `PortableInterceptor::ServerRequestInterceptor and CORBA::LocalObject`, and implement the five operations that correspond to the five server side interception points, plus operations that provide the name of the interceptor and destroy the interceptor.

9.2.3.1 Server Interception Points

The `receive_request_service_context()` interception point is called before the servant manager is called. The operation parameters that are passed as part the request’s service context are not available to the interceptor at this

point. Any service context information the interceptor needs must be obtained from the request scope `PICurrent`. The `PICurrent` is explained in 9.5. The `receive_request()` point allows the interceptor to monitor request information once all operation parameters are available. An interceptor that implements the `send_reply()` operation can monitor and modify the reply service context after the server operation has been invoked but before the reply is sent to the client. The `send_exception()` point allows the interceptor to inspect exception information and modify the reply service context before the exception is sent to the client. Interceptors implementing the `send_other()` operation can inspect the information available when the request results in something other than a normal reply or an exception.

9.2.4 Request Parameters

Request interceptors access request information through `ClientRequestInfo` and `ServerRequestInfo` objects, which are given as parameters to their respective interception points. Client and server `RequestInfo` objects inherit from a common interface defined in `$TAO_ROOT/tao/PI/RequestInfo.pidl`:

```
local interface RequestInfo
{
  readonly attribute unsigned long request_id;
  readonly attribute string operation;

  readonly attribute Dynamic::ParameterList arguments;
  readonly attribute Dynamic::ExceptionList exceptions;
  readonly attribute Dynamic::ContextList contexts;
  readonly attribute Dynamic::RequestContext operation_context;

  readonly attribute any result;
  readonly attribute boolean response_expected;
  readonly attribute Messaging::SyncScope sync_scope;

  readonly attribute ReplyStatus reply_status;
  readonly attribute Object forward_reference;

  any get_slot (in SlotId id) raises (InvalidSlot);
  IOP::ServiceContext get_request_service_context (in IOP::ServiceId id);
  IOP::ServiceContext get_reply_service_context (in IOP::ServiceId id);
};
```

The `ClientRequestInfo` interface extends the `RequestInfo` interface with attributes and operations of interest to client-side interceptors.

```
local interface ClientRequestInfo : RequestInfo
{
  readonly attribute Object target;
  readonly attribute Object effective_target;
  readonly attribute IOP::TaggedProfile effective_profile;

  readonly attribute any received_exception;
  readonly attribute CORBA::RepositoryId received_exception_id;

  IOP::TaggedComponent get_effective_component (in IOP::ComponentId id);
  IOP::TaggedComponentSeq get_effective_components (in IOP::ComponentId id);
  CORBA::Policy get_request_policy (in CORBA::PolicyType type);
  void add_request_service_context (
    in IOP::ServiceContext service_context,
    in boolean replace);
};
```

The `ServerRequestInfo` interface extends the `RequestInfo` interface with attributes and operations of interest to the server-side interceptors.

```
local interface ServerRequestInfo : RequestInfo
{
  readonly attribute any sending_exception;
  readonly attribute ServerId server_id;
  readonly attribute ORBId orb_id;
  readonly attribute AdapterName adapter_name;
  readonly attribute CORBA::OctetSeq object_id;
  readonly attribute CORBA::OctetSeq adapter_id;
  readonly attribute CORBA::RepositoryId target_most_derived_interface;

  CORBA::Policy get_server_policy (in CORBA::PolicyType type);
  void set_slot (in SlotId id, in any data) raises (InvalidSlot);
  boolean target_is_a (in CORBA::RepositoryId id);
  void add_reply_service_context (
    in IOP::ServiceContext service_context,
    in boolean replace);
};
```

For an explanation of the attributes and operations of a `RequestInfo` object, as well as their applicability to each interception point, see Chapter 16 of the CORBA 3.1 specification.

9.2.5 Registering Interceptors

As a developer, you provide the code to register your application's interceptors with the ORB. Interceptors are installed in the ORB with an

`ORBInitializer` object and registered by implementing its `pre_init()` or `post_init()` method and calling `PortableInterceptor::register_orb_initializer()` prior to calling `CORBA::ORB_init()`. The specifics of interceptor initialization are shown in the example in the next section. Developers need to be aware that certain operations that need a pointer to the ORB can not be invoked during interceptor registration, because the registration occurs within the call to `ORB_init()`, and no ORB pointer exists yet.

To allow for this, the `ORBInitInfo` interface contains functions and attributes that hold the arguments passed to `ORB_init()`, a reference to the `CodecFactory`, and additional information that would otherwise be unavailable.

9.2.6 Example: A Simple Authentication Interceptor

Our first example uses interceptors to add a user name to each request that a client makes. The client's interceptor provides the name that is sent with the request. The server's interceptor authenticates the user before the request is dispatched to the servant. Our example extends the Messenger example from Chapter 3. The complete source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/DevGuideExamples/PortableInterceptors/Auth`.

Note *This example is not meant as a secure authentication solution. Please refer to the Chapter 27 for a more thorough treatment of the subject of security.*

9.2.6.1 Messenger Interface

The definition of the Messenger interface has not changed:

```
interface Messenger
{
   boolean send_message (in string user_name,
                         in string subject,
                         inout string message);
};
```

9.2.6.2 Messenger Implementation Class

We now define the `Messenger_i` implementation class as follows:

```
#include "MessengerS.h"

class  Messenger_i : public virtual POA_Messenger
{
public:
  //Constructor
  Messenger_i (void);

  //Destructor
  virtual ~Messenger_i (void);

virtual CORBA::Boolean send_message (
  const char* user_name,
  const char* subject,
  char*& message
  );
};
```

The implementation of the `Messenger_i` class is as follows:

```
#include "Messenger_i.h"
#include <iostream>

// Implementation skeleton constructor
Messenger_i::Messenger_i (void)
{
}

// Implementation skeleton destructor
Messenger_i::~Messenger_i (void)
{
}

CORBA::Boolean Messenger_i::send_message (
  const char* user_name,
  const char* subject,
  char*& message
)
{
  std::cout << "Message from: " << user_name << std::endl;
  std::cout << "Subject:      " << subject << std::endl;
  std::cout << "Message:      " << message << std::endl;
  CORBA::string_free (message);
  message = CORBA::string_dup ("Thanks for the message.");
  return true;
}
```

9.2.6.3 Defining the Client Request Interceptor

The client-side interceptor is defined in the `ClientInterceptor` class. This class inherits from `CORBA::LocalObject` and `PortableInterceptor::ClientRequestInterceptor`.

```
#include <tao/PortableInterceptorC.h>
#include <tao/LocalObject.h>
#include <tao/PI/PI.h>

class ClientInterceptor :
              public virtual PortableInterceptor::ClientRequestInterceptor,
              public virtual CORBA::LocalObject
{
public:
  ClientInterceptor (void);
  virtual ~ClientInterceptor ();

  virtual char* name ();

  virtual void destroy ();

  virtual void send_poll (PortableInterceptor::ClientRequestInfo_ptr ri);

  virtual void send_request (PortableInterceptor::ClientRequestInfo_ptr ri);

  virtual void receive_reply (PortableInterceptor::ClientRequestInfo_ptr ri);

  virtual void receive_other (PortableInterceptor::ClientRequestInfo_ptr ri);

  virtual void receive_exception(PortableInterceptor::ClientRequestInfo_ptr ri);

private:
  const char *myname_;
};
```

In this example, we have overridden the operations that correspond to the five interception points as well as the `name()` and `destroy()` operations from `PortableInterceptor::Interceptor`.

9.2.6.4 Implementing the Client Request Interceptor

Next, we implement the client request interceptor. The constructor initializes the name of our interceptor. The `name()` operation's implementation is simple.

```
ClientInterceptor::ClientInterceptor (void)
```

```
  : myname_ ("Client_Authentication_Interceptor")
{
  std::cout << "Calling ClientInterceptor constructor." << std::endl;
}

char* ClientInterceptor::name ()
{
  std::cout << "Calling ClientInterceptor name() method" << std::endl;
  return CORBA::string_dup (this->myname_);
}
```

To illustrate how information can be passed using the service context list, we insert some user name information into the service context list at the client's `send_request()` interception point. Later, we will show how we can extract and use this as validation information on the server side at the `server's receive_request()` interception point.

In our interceptor's `send_request()` operation, we first log information about the request to standard output. Then, we insert information into the service context list. Recall that the service context list is a sequence of structures, each containing a context identifier of type `unsigned long` and a sequence of octets. You can use these fields to pass any information you want with the request. Here, we set the `context_id` field to an arbitrary value and populate the `context_data` field with the user's name after we convert the name from a string to a sequence of octets.

```
const CORBA::ULong service_ctx_id = 0xdeed;

void ClientInterceptor::send_request (
  PortableInterceptor::ClientRequestInfo_ptr ri)
{
  std::cout << "Calling send_request()." << std::endl;

  IOP::ServiceContext sc;
  sc.context_id = service_ctx_id;

  const char user_name[] = "Ron Klein";
  std::cout << "User's Name: " << user_name << std::endl;
  CORBA::ULong string_len = sizeof (user_name) + 1;
  CORBA::Octet *buf = new CORBA::Octet [string_len];

  ACE_OS::strcpy (reinterpret_cast<char *>(buf), user_name);

  sc.context_data.replace (string_len, string_len, buf, true);

  // Add this context to the service context list.
```

```
  ri->add_request_service_context (sc, false);

}
```

Our `receive_reply()`, `receive_other()`, and `receive_exception()` operations do nothing of any importance. They simply log information about the request to standard output.

```
void ClientInterceptor::receive_reply (
  PortableInterceptor::ClientRequestInfo_ptr)
{
  std::cout << "Calling receive_reply()." << std::endl;
}

void ClientInterceptor::receive_other (
  PortableInterceptor::ClientRequestInfo_ptr)
{
  std::cout << "Calling receive_other()." << std::endl;
}

void ClientInterceptor::receive_exception (
  PortableInterceptor::ClientRequestInfo_ptr)
{
  std::cout << "Calling receive_exception()." << std::endl;
}
```

9.2.6.5 Developing the Client and Installing the Interceptor

To use our client request interceptor, we implement the `post_init()` function of a `ClientInitializer` object, which inherits from `CORBA::LocalObject` and `PortableInterceptor::ORBInitializer`.

```
#include <tao/PortableInterceptorC.h>
#include <tao/LocalObject.h>
#include <tao/PI/PI.h>

class ClientInitializer : public virtual PortableInterceptor::ORBInitializer,
                          public virtual CORBA::LocalObject
{
  virtual void post_init (PortableInterceptor::ORBInitInfo_ptr info);
};
```

If we were registering another interceptor that needed access to this interceptor's initial services, we would choose to register this interceptor in `pre_init()`. Since no other interceptors need access to this interceptor's

services, we register this interceptor in `post_init()`. In contrast to `pre_init()`, `post_init()` is executed at the point in ORB initialization when all initial references are available. Our `post_init()` instantiates a `ClientRequestInterceptor` and registers it by calling `ORBInitInfo::add_client_request_interceptor()`.

```
void ClientInitializer::post_init (
  PortableInterceptor::ORBInitInfo_ptr info)
{
  // Create and register the request interceptors.
  PortableInterceptor::ClientRequestInterceptor_var ci = new ClientInterceptor();
  info->add_client_request_interceptor (ci.in());
}
```

With the initializer written, we develop a client and install our interceptor by creating and registering the `ClientInitializer` object before calling `CORBA::ORB_init()`. We include header files for both the Messenger interface and our Messenger initializer class definitions and instantiate an `ORBInitializer_var` which is passed as a parameter to `PortableInterceptor::register_orb_initializer()`.

```
#include "MessengerC.h"
#include "ClientInitializer.h"
#include <orbsvcs/CosNamingC.h>
#include <iostream>

int main (int argc, char* argv[])
{
  try {
    // Create and register our ORBInitializer.
    PortableInterceptor::ORBInitializer_var orb_initializer =
      new ClientInitializer;

    PortableInterceptor::register_orb_initializer (orb_initializer.in());

    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "Client ORB");
```

Next, we acquire a reference to the Messenger object and use it to send a message. The user name authentication is added transparently by the interceptor and passed through the service context with the rest of the request when `send_message()` is invoked.

```
    // Read and destringify the Messenger object's IOR.
```

```
    CORBA::Object_var obj = orb->string_to_object("file://Messenger.ior");
    if (CORBA::is_nil(obj.in())) {
      std::cerr << "Could not get Messenger IOR." << std::endl;
      return 1;
    }

    // Narrow the IOR to a Messenger object reference.
    Messenger_var messenger = Messenger::_narrow(obj.in());
    if (CORBA::is_nil(messenger.in())) {
      std::cerr << "IOR was not a Messenger object reference." << std::endl;
      return 1;
    }

    // Send a message the the Messenger object.
    CORBA::String_var message = CORBA::string_dup ("Hello!");
    messenger->send_message ("TAO User", "TAO Test", message.inout());

    // Print the Messenger's reply.
    std::cout << "Reply: " << message.in() << std::endl;
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "CORBA exception: " << ex << std::endl;
    return 1;
  }

  return 0;
}
```

9.2.6.6 Defining the Server Request Interceptor

The server-side interceptor is defined in the `ServerInterceptor` class. This class inherits from the `CORBA::LocalObject` and `PortableInterceptor::ServerRequestInterceptor` classes.

```
#include <tao/PortableInterceptorC.h>
#include <tao/LocalObject.h>
#include <tao/PI_Server/PI_Server.h>

class ServerInterceptor :
           public virtual PortableInterceptor::ServerRequestInterceptor,
           public virtual CORBA::LocalObject
{
 public:
  ServerInterceptor (void);
  ~ServerInterceptor ();

  virtual char* name ();
```

```
  virtual void destroy ();

  virtual void receive_request (PortableInterceptor::ServerRequestInfo_ptr ri);

  virtual void receive_request_service_contexts (
    PortableInterceptor::ServerRequestInfo_ptr ri);

  virtual void send_reply (PortableInterceptor::ServerRequestInfo_ptr ri);

  virtual void send_exception (PortableInterceptor::ServerRequestInfo_ptr ri);

  virtual void send_other (PortableInterceptor::ServerRequestInfo_ptr ri);

 private:
  const char *myname_;
};
```

Recall that the server request interceptor class should inherit from `PortableInterceptor::ServerRequestInterceptor`. In this example, we have overridden the operations that correspond to the five interception points as well as the `name()` and `destroy()` operations.

9.2.6.7 Implementing the Server Request Interceptor

Next, we implement the server request interceptor. The constructor initializes the name of our interceptor. The `name()` operation's implementation is simple.

```
ServerInterceptor::ServerInterceptor (void)
  : myname_ ("Server_Authentication_Interceptor")
{
  std::cout << "Calling ServerInterceptor constructor." << std::endl;
}

char* ServerInterceptor::name ()
{
  std::cout << "Calling ServerInterceptor name() method" << std::endl;
  return CORBA::string_dup (this->myname_);
}
```

Recall that the client request interceptor passes information along with the request in the service context list. We now want to extract this information on the server side at the server's `receive_request()` interception point.

In our interceptor's `receive_request()` operation, we call `target_is_a()` to verify that the remote invocation is requesting a

Messenger object. Then we obtain the service context data, which contains the user name as an octet sequence, and cast it to a char * so that it can be compared to allowed_users[]. If the user name matches and element in allowed_users[], authentication is successful.

```
const IOP::ServiceId service_id = 0xdeed;
const unsigned int num_allowed_users = 4;
const static char* allowed_users[num_allowed_users+1] =
  {"Ron Klein", "Scott Case", "Mark Hodge", "Greg Black", 0};
const char* restricted_interfaces[1] = {"IDL:Messenger:1.0"};

void ServerInterceptor::receive_request (
  PortableInterceptor::ServerRequestInfo_ptr ri)
{
  bool permission_granted = false;
  std::cout << "Calling receive_request()." << std::endl;

  if (ri->target_is_a(restricted_interfaces[0]))
  {
    IOP::ServiceId id = service_id;

    // Check that the request service context can be retrieved.
    IOP::ServiceContext_var sc = ri->get_request_service_context (id);
    CORBA::OctetSeq ocSeq = sc->context_data;
    const char * buf = reinterpret_cast<const char *>(ocSeq.get_buffer());

    for (unsigned int i=0; i<num_allowed_users; ++i) {
      if (ACE_OS::strcmp (buf, allowed_users[i]) == 0) {
        permission_granted = true;
        break;
      }
    }
  }

  if (permission_granted == true) {
    std::cout << "Permission Granted " << std::endl;
  }
  else {
    std::cout << "Permission Denied " << std::endl;
    throw CORBA::NO_PERMISSION();
  }
}
```

Our other interception point operations do nothing important. They log information about the request to standard output.

```
void ServerInterceptor::receive_request_service_contexts (
```

```
  PortableInterceptor::ServerRequestInfo_ptr)
{
  std::cout << "Calling receive_request_service_contexts()." << std::endl;
}

void ServerInterceptor::send_reply (
  PortableInterceptor::ServerRequestInfo_ptr)
{
  std::cout << "Calling send_reply()." << std::endl;
}

void ServerInterceptor::send_exception (
  PortableInterceptor::ServerRequestInfo_ptr)
{
  std::cout << "Calling send_exception()." << std::endl;
}

void ServerInterceptor::send_other (
  PortableInterceptor::ServerRequestInfo_ptr)
{
  std::cout << "Calling send_other()." << std::endl;
}
```

9.2.6.8 Developing the Server and Installing the Interceptor

We develop a server and install our interceptor by registering it with the ORB. The procedure is very similar to installation of the client interceptor. We are extending the `MessengerServer` example from Chapter 3. Few changes to the original code are required to use interceptors and we will explain them as we go along.

The code for the server interceptor initializer is similar to its client counterpart. The `ServerInitializer` class inherits from `CORBA::LocalObject` and `PortableInterceptor::ORBInitializer`, and we implement the `post_init()` operation by creating a `ServerRequestInterceptor` and using it as a parameter to `ORBInitInfo::add_server_request_interceptor()`.

```
void ServerInitializer::post_init (
  PortableInterceptor::ORBInitInfo_ptr info)
{

  // Create and register the request interceptors.
 PortableInterceptor::ServerRequestInterceptor_var si = new ServerInterceptor();
  info->add_server_request_interceptor (si.in());
}
```

In the server, we include header files for both the `Messenger` interface and our `Messenger` interceptor initializer class definitions. We then instantiate a new `ServerInitializer` object and register the server interceptor with a call to `PortableInterceptor::ORBInitializer()`. Then we initialize the ORB as usual.

```
#include "Messenger_i.h"
#include "MessengerS.h"
#include "ServerInitializer.h"
#include <iostream>
#include <fstream>

int main (int argc, char* argv[])
{
  try {
   // Create and register our ORBInitializer.
    PortableInterceptor::ORBInitializer_var orb_initializer =
      new ServerInitializer;
    PortableInterceptor::register_orb_initializer (orb_initializer.in ());

    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "Server ORB");
```

The next part of the server is unchanged from the `MessengerServer` in Chapter 3.

```
    //Get reference to the RootPOA.
    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

    // Activate the POAManager.
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Create a servant.
    PortableServer::Servant_var<Messenger_i> messenger_servant
      = new Messenger_i();

    // Register the servant with the RootPOA, obtain its object
    // reference, stringify it, and write it to a file.
    PortableServer::ObjectId_var oid =
      poa->activate_object(messenger_servant.in());
    CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
    CORBA::String_var str = orb->object_to_string(messenger_obj.in());
    std::ofstream iorFile("Messenger.ior");
    iorFile << str.in() << std::endl;
```

```
    iorFile.close();
    std::cout << "IOR written to file Messenger.ior" << std::endl;

    // Accept requests from clients.
    orb->run();
    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "CORBA exception: " << ex << std::endl;
    return 1;
  }

  return 0;
}
```

9.2.7 Running the Application

Run the `MessengerServer` and `MessengerClient` as before. The `MessengerServer` will write a stringified object reference to the file named `Messenger.ior`. The `MessengerClient` will read the stringified object reference from the file, convert it to an object reference, and use it to invoke operations upon the Messenger object.

9.2.8 Program Output

Each operation invoked by the client will go through the client and server request interceptors that have been registered with their respective ORBs. The client's output should look something like this:

```
> ./MessengerClient
Calling ClientInterceptor constructor.
Calling ClientInterceptor name() method
Calling send_request().
User's Name: Ron Klein
Calling receive_reply().
```

The server's output should look something like this:

```
> ./MessengerServer
Calling ServerInterceptor constructor.
Calling ServerInterceptor name() method

IOR written to file Messenger.ior
Calling receive_request_service_contexts().
Calling receive_request().
Permission Granted
```

```
Message from: TAO User
Subject:      TAO Test
Message:      Hello!
Calling send_reply().
```

9.3 Marshaling and the Service Context

Recall that data in the service context is a sequence of octets or raw bytes. In the previous example, where it was intended that the client and server run on the same machine, no special care was taken to ensure the integrity of the data in the service context. Due to different hardware having different endianess, the data in the service context can be corrupted if the client and server exist on different machines. To keep this from occurring, octet sequences are converted to a network byte order, or *marshaled*. This is accomplished by the `Codec`.

9.3.1 The Codec

The `Codec` interface encodes and decodes between a sequence of octets and a `CORBA::Any`. A `codec` is obtained from the `CodecFactory`. There are multiple ways to obtain a reference to it. If the ORB is available, a reference to the `CodecFactory` can be obtained from a call to `ORB::resolve_initial_references("CodecFactory")`. During interceptor initialization, when the ORB is not available, a reference to the `CodecFactory` can be obtained from `ORBInitInfo::codec_factory()` or `ORBInitInfo::resolve_initial_references("CodecFactory")`.

The `Codec` interface has four distinct functions that encode/decode either the value of the data or the value and typecode of the data. This interface is defined in `$TAO_ROOT/tao/CodecFactory/IOP_Codec.pidl`:

```
local interface Codec
{
  exception InvalidTypeForEncoding {};
  exception FormatMismatch {};
  exception TypeMismatch {};

  CORBA::OctetSeq encode (in any data) raises (InvalidTypeForEncoding);
  any decode (in CORBA::OctetSeq data) raises (FormatMismatch);
  CORBA::OctetSeq encode_value (in any data) raises (InvalidTypeForEncoding);
  any decode_value (in CORBA::OctetSeq data, in CORBA::TypeCode tc)
    raises (FormatMismatch, TypeMismatch);
```

```
};
```

9.3.2 Example: Using the Codec

Encoding data with a `codec` is straightforward. First, a reference to the `CodecFactory` is obtained. Then an encoding scheme is specified and used to create a `codec` instance. This `codec` object can encode and decode between a `CORBA::Any` and a `CORBA::OctetSeq`. Files that utilize the `CodecFactory` interface should add the following include:

```
#include <tao/CodecFactory/CodecFactory.h>
```

This example extends the first interceptor example from 9.2.6 by encoding all the request information. The simple authentication scheme has changed from user name to user and group ids (uid/gid), which are passed as `CORBA::Long`, and would be corrupted if client and server reside on different-endian hosts. This example only shows the `codec`-specific code, as the majority of the source has not changed. The complete source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/DevGuideExamples/PortableInterceptors/SimpleCodec`.

9.3.2.1 The Client

The user id is encoded before the `send_message()` function is invoked. First we obtain an initial reference to the CodecFactory, then we obtain a `codec` by passing an encoding scheme to the `CodecFactory` reference.

```
// Obtain a reference to the CodecFactory.
CORBA::Object_var obj2 = orb->resolve_initial_references ("CodecFactory");
if (CORBA::is_nil(obj2.in())) {
  std::cerr <<  "Error: codec_factory" << std::endl;
  return 1;
}

IOP::CodecFactory_var codec_factory = IOP::CodecFactory::_narrow (obj2.in());
std::cout << "got codec factory" << std::endl;

// Set up a structure that contains information necessary to
// create a GIOP 1.2 CDR encapsulation Codec.
IOP::Encoding encoding;
encoding.format = IOP::ENCODING_CDR_ENCAPS;
encoding.major_version = 1;
encoding.minor_version = 2;
```

```
// Obtain the CDR encapsulation Codec.
IOP::Codec_var codec = codec_factory->create_codec (encoding);
```

The user id is inserted into a `CORBA::Any` which is encoded and returns a `CORBA::OctetSeq` that is used in the call to `send_message()`.

```
// our user id
CORBA::Long uid = 64321;
CORBA::Any uid_as_any;
uid_as_any <<= uid;
CORBA::OctetSeq client_uid = *codec->encode (uid_as_any);
messenger->send_message (client_uid);
```

9.3.2.2 The Client Interceptor

In this example, the client interceptor passes group id information. Since the gid is also a `CORBA::Long`, it too is marshaled. Recall that interceptors are registered within the call to `ORB_init()`. Since an ORB reference is not yet available, interceptors that need access to a `codec` must obtain it from operations in the `ORBInitInfo` interface.

```
void
ClientInitializer::post_init (
  PortableInterceptor::ORBInitInfo_ptr info)
{
  // get Codec factory
  IOP::CodecFactory_var codec_factory = info->codec_factory();
```

The interceptor gets the CodecFactory reference through an argument passed to its constructor by the `ORBInitializer`.

```
   // Create and register the request interceptors.
  PortableInterceptor::ClientRequestInterceptor_var ci =
    new ClientInterceptor (codec_factory);
  info->add_client_request_interceptor (ci.in());
}
```

The client interceptor constructor has been modified to accept the reference to the `CodecFactory` and create a codec. The `send_request()` interception point will use the codec to encode the gid. It then adds the encoded octet sequence to the service context. The marshaling code is identical to that in the client.

```
ClientInterceptor::ClientInterceptor (IOP::CodecFactory_var cf)
  : myname_ ("Client_Authentication_Interceptor")
{
  std::cout << "Calling ClientInterceptor constructor." << std::endl;

  // Set up a structure that contains information necessary to
  // create a GIOP 1.2 CDR encapsulation Codec.
  IOP::Encoding encoding;
  encoding.format = IOP::ENCODING_CDR_ENCAPS;
  encoding.major_version = 1;
  encoding.minor_version = 2;

  // Obtain the CDR encapsulation Codec.
  this->codec = cf->create_codec (encoding);
}

void
ClientInterceptor::send_request (
  PortableInterceptor::ClientRequestInfo_ptr ri)
{
  std::cout << "Calling send_request()." << std::endl;

  IOP::ServiceContext sc;
  sc.context_id = service_ctx_id;

  const CORBA::Long gid = 9007;
  std::cout << "GID: " << gid << std::endl;

  CORBA::Any gid_as_any;
  gid_as_any <<= gid;

  sc.context_data = *codec->encode(gid_as_any));

  // Add this context to the service context list.
  ri->add_request_service_context (sc, false);
}
```

9.3.2.3 The Server

The octet sequence is decoded in the `send_message()` function of the server. The code is very similar to the code in the client. A reference to the `CodecFactory` is passed an encoding structure and returns a codec that is used to demarshal the uid. In practice, creating the codec could be moved to the constructor, but in this example it is left in the `send_message()` function for clarity.

```
// Obtain a reference to the CodecFactory.
```

```
CORBA::Object_var obj = orb->resolve_initial_references ("CodecFactory");
if (CORBA::is_nil(obj.in())) {
  std::cerr <<  "Error: codec_factory" << std::endl;
  return 1;
}

IOP::CodecFactory_var codec_factory = IOP::CodecFactory::_narrow (obj.in ());
std::cout << "Server got codec factory" << std::endl;

// Set up a structure that contains information necessary to
// create a GIOP 1.2 CDR encapsulation Codec.
IOP::Encoding encoding;
encoding.format = IOP::ENCODING_CDR_ENCAPS;
encoding.major_version = 1;
encoding.minor_version = 2;

// Obtain the CDR encapsulation Codec.
IOP::Codec_var codec = codec_factory->create_codec (encoding);

CORBA::Any uid_as_any;
uid_as_any = *(codec->decode(user_name));

CORBA::Long uid;
if (uid_as_any >>= uid) {
  std::cout << "UID: " << uid << std::endl;
} else {
  std::cerr << "Could not extract UID from any." << std::endl;
}
```

9.3.2.4 The Server Interceptor

As in the client, the server interceptor is not able to access the `CodecFactory` through the ORB because an ORB reference is not available. Instead, a reference to the `CodecFactory` is obtained in the `ORBInitializer` from `ORBInitInfo::codec_factory()`. This reference is passed as an argument to the interceptor's constructor. The code that does this is identical to the client code and is not shown here.

Decoding in the server interceptor is more complex than the decoding in the server's `send_message()` function. Previously we decoded an octet sequence that was passed as a parameter to `send_message()`. At the `receive_request()` interception point, we do not have direct access to an octet sequence and must extract it from the service context. There is no service context member function that returns the resident data as an octet sequence, but there is a way to construct an octet sequence from the data in the service context.

```
// need to construct an octet seq for decoding
CORBA::OctetSeq ocSeq = CORBA::OctetSeq(
  sc->context_data.length(),
  sc->context_data.length(),
  sc->context_data.get_buffer(),
  false);
```

Once the data has been obtained as a `CORBA::OctetSeq`, it can be decoded using the codec that was created in the interceptor's constructor.

```
CORBA::Any gid_as_any;
gid_as_any = *codec->decode(ocSeq);

CORBA::Long gid;
if (gid_as_any >>= gid) {
  for (int i=0; i<3; ++i) {
    if (gid == allowed_gid[i]) {
      permission_granted = true;
    }
  }
} else {
  permission_granted = false;
  std::cerr << "Could not extract GID from any." << std::endl;
}
```

9.3.3 Program Output

The output of this example is similar to the previous example. The client output should resemble:

```
> ./MessengerClient
Calling ClientInterceptor constructor.
Calling ClientInterceptor name() method
got codec factory
Calling send_request().
GID: 9007
Calling receive_reply().
message was sent
```

The server output should resemble:

```
> ./MessengerServer
Calling ServerInterceptor constructor.
Calling ServerInterceptor name() method
IOR written to file Messenger.ior
Calling receive_request_service_contexts().
```

```
Calling receive_request().
Permission Granted
Server got codec factory
UID: 64321
Calling send_reply().
```

9.4 IOR Interceptors

A second class of interceptors, IOR Interceptors, can add user-defined information (called tagged components) to an object's IOR at the time of the creation of the IOR. This is useful in cases where developers might want to add information about a server's or object's capabilities or requirements. IOR interceptors implement the following interfaces, which are defined in `$TAO_ROOT/tao/IORInterceptor/IORInterceptor.pidl`.

```
local interface IORInterceptor : Interceptor
{
  void establish_components (in IORInfo info);
};

local interface IORInterceptor_3_0 : IORInterceptor
{
  void components_established (in IORInfo info);

  void adapter_manager_state_changed (in AdapterManagerId id,
                                      in AdapterState state);
  void adapter_state_changed (in ObjectReferenceTemplateSeq templates,
                              in AdapterState state);
   };
```

The `IORInterceptor_3_0` interface was added in CORBA 3.0 and adds additional operations and capabilities to IOR Interceptors. In general, you should use this interface when implementing your own IOR Interceptors.

The server calls `establish_components()` for all registered IOR interceptors in the course of assembling the data that will be included in an IOR, typically when the adapter (POA) is created. Per the CORBA specification, the ORB ignores any exceptions thrown by `establish_components()`. By using the `IORInfo` object passed, IOR interceptors can access adapter information and add tagged components during this call. The adapter template is not available during this call.

Once `establish_components()` is called on each IOR interceptor, the ORB calls `components_established()` on each interceptor that implements the `IORInterceptor_3_0` interface. The adapter template is available during this call. If this operation throws an exception, then POA creation fails.

The orb calls the `adapter_manager_state_changed()` operation each time an adapter manager (POA Manager) undergoes a state change. For example a call to `activate()` on a `POAManager` results in a call to `adapter_manager_state_changed()` with a value for `state` of `PortableInterceptor::ACTIVE`. The id parameter identifies the `POAManager`.

State changes on the adapter itself result in a call to `adapter_state_changed()`.

The `IORInfo` interface is used to get profile information and to add components to an IOR. The `IORInfo` interface is defined in `$TAO_ROOT/tao/IORInterceptor/IORInfo.pidl`.

```
local interface IORInfo
{
  CORBA::Policy get_effective_policy (in CORBA::PolicyType type);
  void add_ior_component (in IOP::TaggedComponent component);
  void add_ior_component_to_profile (
    in IOP::TaggedComponent component,
    in IOP::ProfileId profile_id);

  readonly attribute AdapterManagerId manager_id;
  readonly attribute AdapterState state;
  readonly attribute ObjectReferenceTemplate adapter_template;
  attribute ObjectReferenceFactory current_factory;
};
```

For more information about the `IORInfo` interface, Object Reference Factories, and Object Reference Templates see the Portable Interceptors chapter of the CORBA specification.

9.4.1 Defining and Implementing the IOR Interceptor

An IOR interceptor multiply inherits from `PortableInterceptor::IORInterceptor_3_0` and `CORBA::LocalObject`. It must implement the `establish_components()`,

components_established(), adapter_manager_state_changed(), adapter_state_changed(), name(), and destroy() methods.

```
#include "tao/PortableInterceptorC.h"
#include "tao/LocalObject.h"
#include "tao/IORInterceptor/IORInterceptor.h"

class ServerIORInterceptor :
  public virtual PortableInterceptor::IORInterceptor_3_0,
  public virtual CORBA::LocalObject
{
public:
  virtual char* name ();

  virtual void destroy ();

  virtual void establish_components (PortableInterceptor::IORInfo_ptr info);
  virtual void components_established (PortableInterceptor::IORInfo_ptr info);
  virtual void adapter_manager_state_changed (
    const char* id,
    PortableInterceptor::AdapterState state);
  virtual void adapter_state_changed (
    const PortableInterceptor::ObjectReferenceTemplateSeq& templates,
    PortableInterceptor::AdapterState state);
};
```

9.4.2 Registering the IOR Interceptor

Similar to request interceptors, IOR Interceptors are registered in the pre_init() or post_init() methods of the ORBInitializer. A newly created instance of IORInterceptor is registered by passing it as an in parameter to ORBInitInfo::add_ior_interceptor().

9.4.3 Extracting Tagged Information

Tagged components in an IOR can be extracted on the client-side at any of the four implemented client interception points (send_poll() is not supported). The tagged component of an IOR is returned from ClientRequestInfo::get_effective_component().

9.4.4 Example: “ServerRequiresAuth” Tag in IOR

The previous examples in 9.2.6 and 9.3.2 show how interceptors can be used to provide simple authentication. In a more complex environment, different

levels of security would exist and clients would not necessarily know what types of authentication they might need to provide with remote requests.

This example demonstrates how IOR interceptors can allow an object to advertise the type of authentication it requires. The following code extends the `codec` example from 9.3.2 on the server side by publishing the string "`ServerRequiresAuth`" as a component tag in the IOR. The client extracts this new tag. The complete source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/DevGuideExamples/PortableInterceptors/IOR`.

9.4.4.1 Developing the IOR Interceptor

The `name()` method of the `IORInterceptor` is simple.

```
char* ServerIORInterceptor::name ()
{
  return CORBA::string_dup ("ServerIORInterceptor");
}
```

To implement `establish_components()`, we create an `IOP::TaggedComponent` and choose an arbitrary numeric value for its `tagID`.

Note *Were we developing an application that needed to interface with other CORBA applications, we would want to make sure our tagID was unique, and would ask the OMG to assign a Component ID. See ftp://ftp.omg.org/pub/docs/ptc/99-02-01.txt for additional information*

We then copy the string “ServerRequiresAuth” into the `TaggedComponent` buffer and call `add_ior_component()`.

```
void ServerIORInterceptor::establish_components (
  PortableInterceptor::IORInfo_ptr info )
{
  const char * permission = "ServerRequiresAuth";

  // arbitrary tag.
  CORBA::ULong tagID = 9654;

  // populate the tagged component
  IOP::TaggedComponent myTag;
```

```
  myTag.tag = tagID;
  myTag.component_data.length (ACE_OS::strlen(permission) + 1);

  CORBA::Octet *buf = myTag.component_data.get_buffer();
  ACE_OS::memcpy (buf, permission, ACE_OS::strlen(permission) + 1);

  // add the tagged component
  info->add_ior_component (myTag);
  std::cout << "Created Tagged IOR." << std::endl;
}
```

The remaining member functions (`components_established()`, `adapter_manager_state_changed()`, `adapter_state_changed()`, and `destroy()`) have empty definitions

9.4.4.2 Installing the IOR Interceptor

To install the IOR interceptor, we implement the `post_init()` method of the `ServerInitializer` class. This method is similar to the `post_init()` method previously shown for Request interceptors. The IOR interceptor is registered by creating a new instance of the interceptor and passing it to the `add_ior_interceptor()` method.

```
void ServerInitializer::post_init (
  PortableInterceptor::ORBInitInfo_ptr info)
{
  // get reference to the codec_factory
  IOP::CodecFactory_var codec_factory = info->codec_factory();

  // Create and register the request interceptors.
  PortableInterceptor::ServerRequestInterceptor_var si =
    new ServerInterceptor(codec_factory);
  info->add_server_request_interceptor (si.in());

  // add IOR Interceptor
  PortableInterceptor::IORInterceptor_var iori = new ServerIORInterceptor;
  info->add_ior_interceptor (ior.in());
}
```

9.4.4.3 Decoding the Tag in the Client

In this example, the client accesses the `IOP::TaggedComponent` information at the `send_request()` interception point. It retrieves an `IOP::TaggedComponent` based upon a known `tagID`. If the `TaggedComponent` does not exist, the operation

`get_effective_component()` raises a `CORBA::BAD_PARAM` exception, hence the `try/catch` block below.

```
const CORBA::ULong tagID = 9654;
try {
  IOP::TaggedComponent_var myTag = ri->get_effective_component(tagID);
  char *tag = reinterpret_cast<char *>(myTag->component_data.get_buffer());
  std::cout << "IOR Tag is : " << tag << std::endl;
}
catch(CORBA::BAD_PARAM&) {
  std::cerr << "Tagged Component not found" << std::endl;
}
```

9.4.5 Program Output

The server generates a tagged IOR which can be viewed in a more readable form using the utility `tao_catior` (in `$TAO_ROOT/utils/catior`):

```
> $TAO_ROOT/utils/catior/tao_catior -f Messenger.ior
reading the file Messenger.ior

here is the IOR
IOR:010000001200000049444c3a4d657373656e6765723a312e30000000010000000000000000870
000000101020010000000636869702e6f63697765622e636f6d00878200001b00000014010f0052
53540e02573c698d0c000000000001000000010000000003000000000000000008000000010000000
04f415401000000140000000135b340010001000000000009010100000000000b625000013000000
53657276657252657175697265734175746800

decoding an IOR:
The Byte Order: Little Endian
The Type Id:    "IDL:Messenger:1.0"
Profile Count:  1
IIOP Version:   1.2
    Host Name:  chip.ociweb.com
    Port Number:        33415
    Object Key len:     27
    Object Key as hex:
    14 01 0f 00 52 53 54 0e 02 57 3c 69 8d 0c 00 00
    00 00 00 01 00 00 00 01 00 00 00
    The Object Key as string:
    ....RST..W<i..............
    The component <0> has tag <0>
          Component Value len:  8
          Component Value as hex:
          01 00 00 00 00 4f 41 54
          The Component Value as string:
          .....OAT
    The component <1> has tag <1>
          Component Value len:  20
```

```
        Component Value as hex:
        01 35 b3 40 01 00 01 00 00 00 00 00 09 01 01 00
        00 00 00 00
        The Component Value as string:
        .5.@...............
    The component <2> has tag <9654>
        Component Value len:  19
        Component Value as hex:
        53 65 72 76 65 72 52 65 71 75 69 72 65 73 41 75
        74 68 00
        The Component Value as string:
        ServerRequiresAuth.
```

At the `send_request()` interception point, the client obtains the `TaggedComponent` and casts it to a `char*` for output. The output of the client should resemble:

```
./MessengerClient
Calling ClientInterceptor constructor.
Calling ClientInterceptor name() method
got codec factory
Calling send_request().
IOR Tag is : ServerRequiresAuth
GID: 9007
Calling receive_reply().
message was sent
```

9.5 The PortableInterceptor::Current

The `PortableInterceptor::Current` or `PICurrent` is a slot table that is used to transfer thread context information between the request and reply service contexts. The `PICurrent` is an ancillary object to Portable Interceptors. Its use is not required, but it is helpful in propagating data when the service context is not available or not yet available.

9.5.1 Using PICurrent

The `PortableInterceptor::Current` interface is defined in `$TAO_ROOT/tao/PI/PICurrent.pidl`:

```
module PortableInterceptor
{
  typedef unsigned long SlotId;
```

```
  exception InvalidSlot {};
  local interface Current : CORBA::Current
  {
    any get_slot (in SlotId id) raises (InvalidSlot);
    void set_slot (in SlotId id, in any data) raises (InvalidSlot);
  };
};
```

A reference to the PICurrent is obtained from a call to `ORB::resolve_initial_references("PICurrent")`. Data in the form of a `CORBA::Any` is inserted into a slot with `set_slot()` and is retrieved with `get_slot()`.

9.5.2 When to use PICurrent

There are special instances where `PICurrent` can be helpful on both the server- and client-side. Recall that the `receive_request_service_context()` interception point does not have access to the service context. Any data that is needed at this interception point can be copied into the `PICurrent` before the invocation and will be available at the interception point. The CORBA specification discusses a client-side case where the `PICurrent` is useful in stopping recursion. The following example demonstrates how interceptors can be called recursively and how the `PICurrent` can be used to pass a flag between the client and server service contexts to allow the client to recognize that it is making a recursive call.

9.5.3 Example: Stopping Client-side Recursion

Infinite recursion can happen if an interceptor makes an ORB-mediated invocation on a CORBA object. Suppose the client makes an invocation, which calls `send_message()`, and `send_message()` makes its own invocation, which will call `send_message()` indefinitely until somehow the interceptor realizes it is recursing or the application crashes. The `PICurrent` can be used to pass a flag that the `ClientRequestInterceptor` can use to keep from making recursive calls.

Note *Another way to solve this potential recursion problem may be to use the Processing Mode Policy to disable interceptors for collocated invocations. If the CORBA object that the interceptor is invoking an operation on is in the same process, then disabling interceptors for collocated calls would avoid*

recursively calling the interceptor. See 9.6.1 for details of the Processing Mode Policy.

In this example, we assume that the client needs to know the server's date and time with each invocation operation. Rather than returning the date and time from a call to `send_message()`, we choose to extend our IDL to add a `get_time()` function. The complete source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/DevGuideExamples/PortableInterceptors/PICurrent`.

Here is our Messenger interface with the addition of a `get_time()` operation:

```
interface Messenger
{
  boolean send_message ( in    string user_name,
                         in    string subject,
                         inout string message );

  string get_time ();
};
```

The `get_time()` operation is implemented on the server as:

```
char* Messenger_i::get_time ()
{
  time_t thetime;
  struct tm* timeinfo;

  ACE_OS::time(&thetime);
  timeinfo = ACE_OS::localtime(&thetime);
  char* timestring = CORBA::string_dup(ACE_OS::asctime(timeinfo));

  return timestring;
}
```

The functionality we would like to see in the client is that every time `Messenger::send_message()` is called, `Messenger::get_time()` is called. If we add a call to `get_time()` in the `send_request()` interception point, we will cause `send_request()` to be called recursively. Using the `PICurrent`, we can detect this recursion.

Before we can use the `PICurrent`, we must get a reference to it. This is done in the `ORBInitializer`. Because we want to call `get_time()`, we need a reference to the Messenger object. In the earlier examples in this chapter, we

read an IOR from a file and called `ORB::string_to_object()` to obtain the object reference, but interceptors do not have access to the ORB in the `ORBInitializer`, so `string_to_object()` is not available. For this example we made the `Messenger` object an initial reference by passing the server's IOR as part of the `-ORBInitRef` command line argument. Alternately, we could bind the `Messenger` object in the Naming Service and resolve the Naming Service as an initial reference in the `ORBInitializer`.

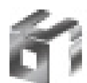
```
void ClientInitializer::post_init (
  PortableInterceptor::ORBInitInfo_ptr info)
{
  // resolve Messenger object
  CORBA::Object_var obj = info->resolve_initial_references ("Messenger");
  Messenger_var messenger = Messenger::_narrow (obj.in());
  if (CORBA::is_nil(messenger.in())) {
    std::cerr << "Not a Messenger reference" << std::endl;
    // We could throw an exception here, or just ignore the error and go on.
  }
```

We then allocate a slot in the current for our use. Since we cannot obtain a reference to the `PICurrent` through `ORB::resolve_initial_references()`, we obtain it through `ORBInitInfo::resolve_initial_references()`.

```
  // allocate slot
  PortableInterceptor::SlotId slot = info->allocate_slot_id();

  // get PICurrent
 CORBA::Object_var current_obj = info->resolve_initial_references ("PICurrent");
  PortableInterceptor::Current_var pic =
    PortableInterceptor::Current::_narrow (current_obj.in());
```

A `CORBA::Boolean` serves as our recursion flag. Initially there is no recursion, so we set a false value in the `PICurrent` slot and finish installing the interceptor.

```
  // set recurion flag
  CORBA::Any flag;
  CORBA::Boolean x = false;
  flag <<= CORBA::Any::from_boolean(x);

  pic->set_slot(slot, flag);

  // Create and register the request interceptors.
  PortableInterceptor::ClientRequestInterceptor_var ci =
```

```
    new ClientInterceptor (messenger, pic.in(), slot);
  info->add_client_request_interceptor (ci.in());
}
```

In the client interceptor, the code pertinent to `Messenger::send_message()` is unchanged.

```
void ClientInterceptor::send_request (
  PortableInterceptor::ClientRequestInfo_ptr ri)
{
  std::cout << "Calling send_request()." << std::endl;

  IOP::ServiceContext sc;
  sc.context_id = service_ctx_id;

  const char user_name[] = "Ron Klein";
  std::cout << "User's Name: " << user_name << std::endl;
  CORBA::ULong string_len = sizeof (user_name) + 1;
  CORBA::Octet *buf = 0;
  ACE_NEW (buf, CORBA::Octet[string_len]);
  ACE_OS::strcpy (reinterpret_cast<char *>(buf), user_name);

  sc.context_data.replace (string_len, string_len, buf, true);
```

Before calling `get_time()`, the interceptor retrieves the flag from the `PICurrent`. If it is false, the interceptor inserts a value of true, which will stop the recursion on the next invocation. Then it calls `get_time()`. Finally, before exiting `send_request()` the value in the `PICurrent` is reset to false, so that later invocations will call `get_time()`.

```
  // recursive call setup
  CORBA::Any* recurse = ri->get_slot(slot);
  CORBA::Boolean x;
  if (*recurse >>= CORBA::Any::to_boolean(x)) {

    CORBA::Any flag;
    if (x == false) {
      x = true;
      flag <<= CORBA::Any::from_boolean(x);

      pic->set_slot(slot, flag);

      // get server time
      std::cout << "Server Time = " << messenger->get_time() << std::endl;
    }
  } else {
```

```
      std::cerr << "Could not extract a boolean value from any" << std::endl;
    }

    // Add this context to the service context list.
    ri->add_request_service_context (sc, false);

    // reset recursion test
    x = false;
    flag <<= CORBA::Any::from_boolean(x);
    pic->set_slot(slot,flag);
  }
```

9.5.4 Program Output

The server output should resemble:

```
> ./MessengerServer
Calling ServerInterceptor constructor.
Calling ServerInterceptor name() method

IOR written to file Messenger.ior
Calling receive_request_service_contexts().
Calling receive_request().
Permission Granted
Calling send_reply().
Calling receive_request_service_contexts().
Calling receive_request().
Permission Granted
Message from: TAO User
Subject:      TAO Test
Message:      Hello!
Calling send_reply().
```

The client output should resemble:

```
> ./MessengerClient -ORBInitRef Messenger=file://Messenger.ior
Calling ClientInterceptor constructor.
Calling ClientInterceptor name() method
Calling send_request().
User’s Name: Ron Klein
Calling send_request().
User’s Name: Ron Klein
Calling receive_reply().
Server Time = Tue Jan 29 13:19:16 2002
```

Notice that `send_request()` is called twice. The first is from the `send_message()` operation invocation. The second is from the `get_time()` operation invocation.

9.6 Interceptor Policy

The CORBA 3.1 specification adds the ability to control interceptor behavior by applying policies during the registration of interceptors. The `ORBInitInfo` interface was extended, via inheritance to support this capability. The new interface is called `ORBInitInfo_3_1` and is defined in `$TAO_ROOT/tao/PI/ORBInitInfo.pidl`:

```
local interface ORBInitInfo_3_1 : ORBInitInfo
{
  void add_client_request_interceptor_with_policy(
       in ClientRequestInterceptor interceptor,
       in CORBA::PolicyList policies)
       raises (DuplicateName, CORBA::PolicyError);
  void add_server_request_interceptor_with_policy(
       in ServerRequestInterceptor interceptor,
       in CORBA::PolicyList policies)
       raises (DuplicateName, CORBA::PolicyError);
  void add_ior_interceptor_with_policy(
       in IORInterceptor interceptor,
       in CORBA::PolicyList policies)
       raises (DuplicateName, CORBA::PolicyError);
};
```

In addition to supporting all the existing `ORBInitInfo` functionality, the new interface adds the ability to register each of the three interceptor types with a list of policies to apply. Applying an invalid or incompatible policy results in a `CORBA::PolicyError` exception.

9.6.1 Processing Mode Policy

Currently, the only portable interceptor policy is the Processing Mode Policy. This policy can limit the conditions under which the interceptor is invoked. The supported values are:

- **LOCAL AND REMOTE**: Call the interceptor for both local and remote invocations. This is the default value for this policy (when it is not specified).

- **REMOTE ONLY**: Only call the interceptor for remote invocations. This will suppress calling the interceptor for collocated invocations.
- **LOCAL ONLY**: Only call the interceptor for local invocations. This will suppress calling the interceptor for remote invocations and only do so for collocated invocations.

Here is a short example that shows registering a client interceptor with the Processing Mode Policy set to `REMOTE_ONLY`:

```
#include "tao/PI/ProcessingModePolicyC.h"
#include "tao/PI/ORBInitInfo.h"

void
ClientInitializer::post_init (PortableInterceptor::ORBInitInfo_ptr info)
{
  PortableInterceptor::ORBInitInfo_3_1_var info_3_1 =
    PortableInterceptor::ORBInitInfo_3_1::_narrow(info);

  PortableInterceptor::ClientRequestInterceptor_var
    client_interceptor = new MyClientInterceptor;

  CORBA::Any client_proc_mode_as_any;
  client_proc_mode_as_any <<= PortableInterceptor::REMOTE_ONLY;

  CORBA::PolicyList policy_list (1);
  policy_list.length (1);
  policy_list[0] =
    orb->create_policy (PortableInterceptor::PROCESSING_MODE_POLICY_TYPE,
                        client_proc_mode_as_any);

  info_3_1->add_client_request_interceptor_with_policy (
                                  client_interceptor.in (),
                                  policy_list);

  policy_list[0]->destroy ();
}
```

9.7 Summary

This chapter showed how to develop applications using TAO and its CORBA 3.1 compliant Portable Interceptors implementation. Several aspects of Portable Interceptors were discussed, including:

- Portable Interceptors are used to monitor and modify transparently the requests and replies between clients and server. They can be implemented with few modifications to existing code.
- `Codecs` (coders/decoders) can be used to marshal the service context, so that byte order differences between systems do not corrupt request/reply data.
- IOR Interceptors are used to add tagged components to IORs. This allows servers and other objects to advertise their capabilities or requirements to clients.
- The `PICurrent` is a table that can be used to transfer data between the reply and request service contexts.
- The Processing Mode Policy can be applied to client request interceptors to limit the conditions under which the interceptor is invoked.

CHAPTER 10

Value Types

10.1 Introduction

The original CORBA specification focused on the challenges of remotely invoking operations on objects, regardless of location. A number of data types were introduced to allow rich interfaces to be defined, but a strict separation was maintained between the functionality of a system, specified via interfaces, and the data of a system, specified via data types. With *value types* (see Chapter 9 the CORBA Core specification, OMG Document formal/08-01-04), it is possible to define objects in IDL that contain both functionality and data (called *state members*) that can be passed remotely by their value. These value types are defined using the `valuetype` keyword. Value types can be helpful for the case where an object role is primarily to hold data, or it is common to copy an object with a large amount of data.

This chapter provides an introduction to the subset of value type features that are implemented in TAO. For additional information about value types, see Chapter 11 of *Pure CORBA*.

10.2 Uses for Value Types

A value type can be used in several situations that may otherwise be awkward in CORBA. You can use a value type as a replacement for an IDL `struct`, as another type of local interface, or as a way to combine data with operations, like a C++ class. They can also be passed through any of the TAO CORBA event services such as the Notification Service described in Chapter 25.

Using a value type in place of a `struct` allows you to use inheritance as a way to organize your structures. This is easier to work with than building complex structures using containment, because it can be tedious to access many levels of nested `struct`s.

In many cases, using value types can be more straightforward and more efficient than using CORBA local interfaces. An operation invocation on a value type does not go through an ORB. Public accessor and mutator operations on value types do not transfer ownership, so there is no need for `_var` types and reference counting, but operations and attributes still have typical interfaces. Unlike a local interface, a value type is not derived from `CORBA::Object`, and therefore avoids considerable compilation and memory overhead.

Note *It is important to keep in mind that for each language used in your distributed system an implementation of your value type must be made. Each language implementation must behave identically for your value type for work correctly.*

Perhaps the most common use of a value type is for passing objects by value. A normal CORBA interface leaves the data portion of an object undefined, allowing you to implement the interface operations with whatever servant best suits the needs of your application. By using value types, your IDL interface can constrain the data types, thus trading a little flexibility in how the servant is implemented for having the ORB pass your objects by value. With value types, you are still required to provide an implementation for the interface.

You can also pass value types through any of the included event services, or through any custom interface that passes values using Any. Your value type implementation need only be linked into the code that ultimately extracts the value type from the Any.

Note *The CORBA* `eventtype` *is a specialization of the* `valuetype`.

When a value type is passed from one process to another, only the state of the object is passed. It is the responsibility of the receiving process to provide an implementation of that value type, as well as a factory suitable for reconstructing the object from the data marshalled using CORBA. A base factory class will be generated and will contain virtual methods corresponding to each of the `factory` operations defined in the value type IDL. Processes operating on a value type are not required to use the same implementation. You must register a factory object with the ORB using `CORBA::ORB::register_value_factory()`.

Note *Value types with only state members will have a factory generated automatically, although you must still register an instance of this factory with the ORB.*

Value types break one of CORBA's original tenets, that the IDL is all that is required for different processes to communicate, regardless of implementation language, architecture, or location. When using value types, you are required to supply value type implementations, which may restrict the processes and implementations with which you can communicate. Although this restriction is often offset by the potential efficiency of using value types, it sometimes forces developers to avoid the use of value types in their applications.

10.3 Defining Value Types in IDL

A value type in IDL can be thought of as a hybrid of a `struct` and an interface. Like a `struct`, the value type can have data members, and passes all of its enclosed data across an interface. Like an interface, a value type defines a set of operations that can be called upon objects of that type. In addition, value types support inheritance from other value types as well as interfaces, allowing for the definition of complex models.

For example, here is a definition for a simple value type:

```
valuetype Person {
```

```
  public string name;
  public long   balance;

  factory create(in string name);

  void debit(in long amt);
};
```

The state members of a value type require the `private` or `public` keyword to indicate the accessibility of the generated accessors and mutators. The `factory` keyword is used to denote a special operation that will be used to construct a `Person`. A value type can contain more than one factory declaration, and the factory can have any name.

Note *It is often unnecessary to declare factory methods in IDL, because you can simply use normal C++ constructors to create the value type instances.*

The value type can be used with an IDL operation:

```
interface Transaction {
  void update(in Person p);
};
```

As the `Person` object is passed via the `update()` operation, the entire object (including its state) is copied from the client to the server. This means that a `Person` object exists in both the client and server, and invocations in a particular process only affect the state of the object local to that process.

10.4 A Value Type Example

In this section we show an example of using value types that is based on the Messenger example introduced in Chapter 3.

10.4.1 Implementing Value Types

The first step in implementing value types is to use the IDL compiler to generate the usual skeleton and stub C++ files from an IDL file defining one or more value types and interfaces that use them. The IDL compiler generates all the code required for value type support. It will generate the C++ classes

Message and OBV_Message (OBV stands for object-by-value) as well as POA_Messenger. The IDL compiler also generates Message_var, Message_out, and Message_ptr types like those generated for interfaces. The complete source code for this example can be found in the source code distribution in $TAO_ROOT/DevGuideExamples/ValueTypes/Messenger.

10.4.1.1 The valuetype IDL

The IDL for our example defines a simple Message value type which we pass through a modified version of our Messenger interface. We provide attributes and operations that will presumably give access to the internal data. This example is meant to illustrate available features more than correct design. We also forgo specification of a factory interface, because we create our factory from the default base class.

```
valuetype Message {
  typedef sequence<string> AddrList;

  private AddrList addrs_;
  private string user_;
  private string subject_;
  private string text_;

  AddrList getAddresses();
  void addAddress (in string address);
  attribute string user;
  attribute string subject;
  attribute string text;
  void print ();
};

interface Messenger {
  boolean send (inout Message msg);
};
```

10.4.1.2 Implementing the Value Type Class

Implementing a value type class is analogous to implementing a servant class for an interface. The IDL compiler generates an OBV_Message class from which our implementation class must inherit. The OBV_Message class itself inherits from the generated Message class and includes the state member accessor/mutator functions. Think of the OBV_Message class as a partial implementation of the abstract Message class.

Note *The* `private` *and* `public` *qualifiers for state members do not directly map to the C++* `private` *and* `public` *keywords. For all state members the IDL compiler generates pure virtual accessor and modifier functions. For public members these functions map to public C++ functions. For private members these functions map to protected C++ functions.*

It is up to us to provide implementations for the value type state members and operations. In our example, we need to provide implementations for the `user`, `subject`, and `text` attributes, as well as the `getAddresses()` and `addAddress()` operations.

Additionally, all value type objects use intrusive reference counting. The `ValueBase` class, from which all value types derive, defines pure `virtual _add_ref()`, `_remove_ref()`, and `_refcount_value()` functions. The easiest way to provide a correct implementation for these is to mix in the `CORBA::DefaultValueRefCountBase` type. This is analogous to mixing in `RefCountServantBase` for servant classes.

Here is a portion of the code for our `Message` implementation class:

```
// Message_i.h
class MessageImpl : public virtual OBV_Message
                  , public virtual CORBA::DefaultValueRefCountBase
{
public:
  MessageImpl (const char* address, const char* user, const char* subject,
               const char* txt);

  virtual char* user();
  virtual void user(const char*);
  // ...
};


// Message_i.cpp
MessageImpl::MessageImpl (const char* address, const char* user,
                          const char* subject, const char* txt)
  : OBV_Message(Message::AddrList(), user, subject, txt)
{
  addAddress(address);
}

char* MessageImpl::user()
{
  return CORBA::string_dup(user_());
}
```

```
void MessageImpl::user(const char* s)
{
  user_(s);
}
```

Notice that, in our constructor, we use the `OBV_Message` constructor to initialize most of the attributes.

10.4.1.3 Implementing the Value Type Factory

A value type factory is required to allow the ORB to create and demarshal value type objects that are included in GIOP `Request` and `Reply` messages. This factory knows how to create value type objects of a specific type that are registered with the ORB.

We must implement a factory class derived from the `ValueFactoryBase` defined in TAO. If we had not specified any operations or attributes in our value type IDL, then a complete factory class called `Message_init` would have been generated automatically. If we had defined any factories in the IDL, then a base class called `Message_init` would have been generated, and we would have to provide implementation for each of its virtual methods. Here is the definition for the factory:

```
#include <tao/Valuetype/ValueFactory.h>

// Message_i.h
class MessageFactory : public virtual CORBA::ValueFactoryBase
{
public:
  static void register_new_factory(CORBA::ORB& orb);
  virtual CORBA::ValueBase* create_for_unmarshal();
};
```

We add the static `register_new_factory()` function to make it easier to register the factory with the ORB. We call this function from both the server and client `main()` functions. The `MessageFactory` definition is as follows:

```
// Message_i.cpp
void MessageFactory::register_new_factory (ORB& orb)
{
  CORBA::ValueFactoryBase_var mf = new MessageFactory;
  CORBA::String_var id = ::_tc_Message->id();
  orb.register_value_factory(id.in(), mf.in());
}
```

```
CORBA::ValueBase* MessageFactory::create_for_unmarshal ()
{
  return new MessageImpl;
}
```

The `register_new_factory()` operation simply associates the Repository ID of our factory class with an instance of our factory class. Later, when a `Message` object is received, the ORB will use the `create_for_unmarshal()` operation to create a local implementation for the `Message` object.

A value type factory only needs to be registered with orbs that need to unmarshal value types. Often this means that you can forgo this registration on the client side, although in our example the Message object is passed through an inout parameter, and therefore both sides must register a factory.

10.4.2 Using Value Types

A value type is typically referenced through a `_var` smart pointer that manages memory allocated for a value type in a manner similar to the `_var` smart pointer types used for CORBA object proxies. You can call the operations on a value type by simply calling the corresponding member function on the value type object. Public state members are accessed and modified using member functions with the same name as the state member. In our example, the client creates a `Message`, then sends it to the server. The server simply calls the `Message::print()` function, then uses the same `Message` object to reply to the client.

Here is our implementation class for the Messenger interface:

```
//  Messenger_i.h
class  Messenger_i : public virtual POA_Messenger
{
public:

  virtual CORBA::Boolean send_message (Message*& msg);
};


// Messenger_i.cpp
CORBA::Boolean Messenger_i::send_message (Message*& msg)
{
  // print the message sent from the client
  msg->print();
```

```
  // populate the return message
  msg->user("Son");
  msg->addAddress("Mom");
  msg->addAddress("Dad");
  std::ostringstream out;
  CORBA::String_var subject = msg->subject();
  out << "RE: " << subject.in();
  msg->subject(out.str().c_str());
  msg->text("Ok. I'm on my way.");

  return true;
}
```

10.5 An Example using Value Types as Events

It can be convenient to pass value types as events using the TAO Event Service, RT Event Service, or Notification Service.

The complete source code for this example can be found in the source code distribution in `$TAO_ROOT/orbsvcs/DevGuideExamples/ValueTypes/Notify`.

10.5.1 Event Example IDL

```
valuetype MyEvent
{
  public string name;
  public long kind;
  private CORBA::LongSeq payload;

  void dump();
  long size();
  void add_long(in long n);
};
```

10.5.2 Event Example Supplier

The following is an excerpt from an example showing how to use a value type with the Notification Service `CosEvent` interfaces. The supplier encapsulates the MyEvent value type in an Any before pushing it to the Notification Service channel using the `PushConsumer` interface.

```
...
MyEvent_var event_ = new MyEvent_i("TestName", 42);
...
```

```
bool push_next_event() {
    try {
      if (! connected_) {
        std::cout << "Trying to push when disconnected." << std::endl;
        return false;
      }
      std::cout << "+" << std::flush;

      ++event_count_;

      Any a;
      a <<= event_;
      consumer_->push(a);

      if (event_count_ >= num_events_ && num_events_ > 0) {
        std::cout << "Supplier stopping after sending "
                  << event_count_ << " events." << std::endl;
        disconnect();
      } else {
        schedule_next_event(EVENT_DELAY);
      }
      return true;
    } catch (CORBA::Exception& e) {
      std::cerr << "TestSupplier::push_next_event() exception: " << e
                << std::endl;
    }
    return false;
}
```

10.5.3 Event Example Consumer

The following is an excerpt from the consumer code, showing how the value type is extracted from the interface. The consumer code must register a value factory with the ORB, or the extraction will trigger an `CORBA::UNKNOWN` system exception.

```
...
virtual void push(const Any& a)
  {
    MyEvent* vt;
    a >>= vt;

    std::cout << std::endl
              << "Received MyEvent name=" << vt->name()
              << ", kind=" << vt->kind()
              << ", size=" << vt->size()
              << std::endl;
```

```
    vt->dump();

    if ( ++ event_count_ >= num_events_ && num_events_ > 0) {
      std::cout << "Consumer disconnecting after receiving "
                << event_count_ << " events." << std::endl;
    }
  }
...
// In main() after ORB_init()
CORBA::ValueFactoryBase_var factory = new MyEventFactory;
String_var id = _tc_MyEvent->id();
orb->register_value_factory(id.in(), factory.in());
...
```

10.6 Value Types and Inheritance

TAO supports multiple inheritance from `abstract` value types (discussed later in this section), and single inheritance from regular non-abstract value types. TAO also supports single inheritance from non-abstract interfaces and multiple inheritance from abstract interfaces via the `supports` keyword.

An abstract interface can be used for defining parameters that can be passed either object references or values. Resulting operation invocations on the abstract interface parameter can result in remote CORBA invocations or local value type execution.

10.6.1 Regular Value Types Inheritance

You can use inheritance between regular (also known as stateful or non-abstract) value types as you might have done with a C++ `struct` in the past, and it can be used to support a common interface as with C++ inheritance. For example, in our application, there might be other value types that we wish to print:

```
valuetype Printable
{
  void print();
};

valuetype Message : Printable
{
  // ...
};
```

```
valuetype Document : Printable
{
  // ...
};
```

By moving the `print()` operation from our `Message` type to a `Printable` base type, we are able to add a new type that can print. This will allow `Message` and `Document` to be passed to any IDL operation that takes a `Printable` type. However, you must still implement the `print()` operation for `Message` and `Document`, because value type inheritance does not provide implementation inheritance.

Of course, we can also have base types that contain data. Here is an example of several value types that would have been extremely tedious to access with the CORBA `struct` mapping:

```
valuetype Party
{
  public string name;
  public string address;
};

valuetype Person : Party
{
  public string birth_date;
};

valuetype Employee : Person
{
  public string date_of_hire;
};

valuetype Manager : Employee
{
  typedef sequence<Employee> Reports;
  public Reports reports;
};
```

If we had implemented these as nested `structs`, we would have had code that called such things as `mgr.employee.person.party.name = "Stan"`. Instead, we simply use `mgr.name()`.

10.6.1.1 Truncatable

When deriving one stateful value type from another, you can specify the `truncatable` keyword. This keyword indicates that you are willing to let the ORB instantiate a value of the base value type with a value instance of the derived value instance. This would occur when an ORB is passed an instance of a value type that it cannot instantiate but it can instantiate an instance of the base value type. Because this substitution can modify the behavior and even meaning of the passed value, the truncatable keyword is required for each derivation where you want to allow the ORB to make these substitutions. Here is the previous example with the truncatable keyword:

```
valuetype Party
{
  public string name;
  public string address;
};

valuetype Person : truncatable Party
{
  public string birth_date;
};

valuetype Employee : truncatable Person
{
  public string date_of_hire;
};

valuetype Manager : truncatable Employee
{
  typedef sequence<Employee> Reports;
  public Reports reports;
};

interface InterfaceType {
  void litigate (in Party p);
};
```

When a client passes an `Employee` value type to the `litigate()` operation, the server's ORB is free to instantiate the passed value as either an `Employee`, `Person`, or `Party` value instance, depending on which types it has knowledge of.

10.6.2 Abstract Value Type Inheritance

A value type may be defined as abstract. This allows the value type to be used as a base class in combination with another value type. An abstract value type may not contain any state members. However, it can contain operations and attributes. A value type may inherit from one non-abstract value type, and many abstract value types. Using abstract value types, we could add another base type to our `Message` value type from the previous example:

```
valuetype Named {
  public name;
};

abstract valuetype Printable {
  void print();
};

valuetype Message : Printable, Named {
  // ...
};
```

By adding the `abstract` keyword, we are able to derive `Message` from both `Named` and `Printable`.

10.6.3 Interface Inheritance

Inheriting a value type from an interface allows that the resulting value to be passed by both a reference (using the interface type) or as a value (using the value type). For example, modify the above example to make printable an interface:

```
interface Printable {
  void print();
};

valuetype Message : supports Printable {
  // ...
};

interface MyInterface {
  void pass_by_value (in Message msg);
  void pass_by_reference (in Printable p);
};
```

When calling `pass_by_value()`, the client is passing the value to the server. If the server calls `print()` using the parameter, it results in a local call to print() on the value in the server's process.

When calling `pass_by_reference()`, the client is passing a CORBA object reference to the server. If the server calls `print()` using the parameter, it results in a remote invocation on the value in the client's process.

10.6.4 Abstract Interface Inheritance

In the previous section, the interface designer specifies whether a given parameter is a CORBA object or a value type. The use of abstract interfaces, as described in this section, allow this decision to be left until run-time. An abstract interface defines a type that can be either a CORBA object or a value type. Modifying the previous example to use an abstract interface, results in the following:

```
abstract interface Printable {
  void print();
};

interface ConcretePrintable : Printable{
};

valuetype Message : supports Printable {
  // ...
};

interface MyInterface {
  void pass_flexible (in Printable p);
};
```

When `pass_flexible()` is passed a `Message` value, it is marshalled and passed to the server. Subsequent calls to `print()` on it are handled by the server's value.

When `pass_flexible()` is passed an object reference to a `ConcretePrintable` value, the reference is passed to the server. Subsequent calls to `print()` result in a remote CORBA invocation on that reference and are handled by that object's server.

10.7 Value Boxes

Value boxes are a simplification to the IDL syntax and C++ mapping for value types that define no inheritance, no operations, and a single attribute. Here are some sample value box definitions in IDL:

```
valuetype ValueBoxString string;

typedef sequence<string> StringSeq;
valuetype ValueBoxStringSeq StringSeq;
```

Any IDL type can be used in a value box. Value boxes cannot be forward declared, so forward declarations always refer to non-boxed value types. Value boxes may not be derived from or derive from anything else.

Value boxes lead to a number of simplifications in their C++ mapping (as compared to non-boxed value types):

- C++ classes representing value boxes are concrete and application developers do not need to derive from and implement there own types.
- All value box classes are always reference counted.
- Value box classes do not need or use factories.
- All value boxes supply `_boxed_in()`, `_boxed_inout()` and `_boxed_out()` member functions that return the appropriate type when a given value box is passed as a parameter of the corresponding mode.

There is some variation in the interface to the C++ class representing a value box. For details, refer to Section 4.19.7 of the C++ Language Mapping, Version 1.2.

10.8 TAO Compliance

TAO supports many of the basic features of value types, including:

- Inheriting one value type from another
- The use of abstract value types
- Inheritance from IDL interfaces.

- Most of the basic operations available to what *Pure CORBA* refers to as *regular value types*.
- Value boxes (also called boxed values).
- The `truncatable` keyword and truncatable inheritance.
- `eventtype` keyword.

TAO's support of value types does *not* include:

- Custom marshaling.
- `ValueBase::_copy_value().`
- `PortableServer::ValueRefCountBase`.
- `ValueBase::_add_ref`() returning `ValueBase`*.

Work is continuing on OBV features in TAO, and some of the above features may be implemented within the near future. However, value types as they currently exist in TAO are a functional and valuable tool that you can take advantage of right now.

CHAPTER 11

Smart Proxies

11.1 Introduction

Smart proxies are a TAO-specific feature that allows customization of proxy behavior. Some other ORB implementations provide similar functionality, but there is no standard for smart proxies.

A smart proxy is an alternative class to the default proxy generated by the TAO IDL compiler. The purpose is to provide the client application developer the ability to extend the default behaviors. It is written by the client application developer and may contain member functions that never send requests to the target object.

A proxy supplies a client application with an interface to a target CORBA object that allows the client to access operations on the object in a location-transparent manner (as if the remote object resides in the client's address space). When the TAO IDL compiler is invoked on an IDL interface, it generates a proxy (stub) class that has the same name as the interface. For each operation and attribute in the interface, the TAO IDL compiler generates a corresponding member function in the proxy class. A client application instantiates a proxy object at run time when it narrows an object reference.

For example, in the following code fragment, adapted from the familiar Messenger example from Chapter 3, a generic object reference (`CORBA::Object`) is narrowed to a more-specific type (`Messenger`), yielding a proxy to a Messenger object:

```
// Obtain the Messenger's object reference.
CORBA::Object_var obj = orb->string_to_object("file://Messenger.ior");

// Narrow it to type Messenger.
Messenger_var messenger = Messenger::_narrow(obj.in());
```

For more information on the role of proxies, see *Advanced CORBA Programming with C++,* 2.5.4.

To invoke an IDL operation or access an attribute on a target object, a client application calls the corresponding member function of the proxy class. This member function (via the client ORB) marshals the invocation into a request message and sends it to the server. The function then waits for the reply (unless it is a `oneway` or AMI function). When the reply arrives, the function (again via the client ORB) demarshals it and returns the reply values (the `return` value, plus the `out` and `inout` parameters) to the client application.

For each IDL interface, the TAO IDL compiler generates a *proxy factory* class that is used to construct the proxy object when an object reference is narrowed. The proxy class static member function `_narrow()` calls the factory's `create_proxy()` member function, which in turn calls the proxy class constructor.

For most client applications, the *default proxy class* generated by the TAO IDL compiler is adequate. However, some applications can benefit from the customization of proxy behavior. TAO provides a way for an application developer to write a custom proxy class, known as a *smart proxy* class, that is used by the client application in lieu of the default proxy class. Smart proxies are not part of the CORBA specification, but since they are a client-side-only issue, they do not affect interoperability between different ORBs. Because the member functions of the smart proxy class have exactly the same signatures as those of the default class, no changes are required in the calls made by the client application on the proxy object. The added functionality of the smart proxy class is entirely transparent to the client application.

11.2 Smart Proxy Use Cases

The following use cases provide some insight into how smart proxies can be used.

- *Client-side caching*—The use of client-side caching is widespread in distributed computing. There are at least two advantages to caching information in the client:
  - Minimization of access time.
  - Minimization of the number of remote calls needed.

  A common example is the caching of web pages by Internet browsers. By caching web pages as they are accessed, the browser allows repeated access to these pages without making repeated remote calls to the web server. This is advantageous to the user, who can access repeated pages more quickly, and to the web server, because its load is reduced.

  The caching of pricing information can be quite useful in a purchasing system. When the client is only providing a price to a customer, it is often sufficient to provide this information from a client-side cache, rather than making a remote call. In the simplest case, the system can be set up to change prices only at specified times of the day. The smart proxy object can then cache pricing information and update it at those specified times. Remote calls are then only required during updates and purchases.

  In a system that uses remote operations for time-consuming calculations that are often repeated, the smart proxy can cache the results of these operations and return them to the client application instead of making repeated remote calls.

  None of the above cases requires changes to the target object's interface. There are cases, however, where the efficiency of the system can be improved by adding additional operations to the interface. One example is the case of a client that frequently accesses multiple attributes of a target object. If an operation that accesses all of the attributes at once is added to the interface, the smart proxy can call this operation to get all the information at once, cache the information on the client-side, and provide the information to the client application through the individual attribute functions. If the operation is added to a derived interface, then the original interface need not be altered and can be provided to clients that do not use

smart proxies. (Another way to achieve similar behavior would be to redesign the interface to return a *value type*; see Chapter 10.)

- *Executing a sequence of operations*—If a client repeatedly makes the same sequence of operation invocations on one or more target object(s), the client application can be provided with a much simpler interface by building this functionality into a smart proxy and providing the client application with a single operation that encapsulates the entire sequence.
- *Choosing between target objects*—If more than one target object is available to serve a particular client request, the smart proxy can implement functionality that chooses which, among a number of possible target objects, to send the request to. One use of such functionality is to provide a form of load balancing by always sending the request to a lightly-loaded server.
- *Batch processing*—If a client application frequently makes a set of changes to a particular target object, a “batch” operation can be added to the target interface that makes all the changes at one time. The client will continue to make individual function calls on the smart proxy, but the smart proxy will cache these calls and combine them into a single request, thus reducing the number of remote calls. (Care must be taken if the smart proxy is used by multiple threads, of course.) If the batch operation is added to a derived interface, then the original interface need not be altered and can be provided to clients that do not use smart proxies.
- *Logical target object*—If a set of functions is to be provided to a client application by a number of target objects (with different interfaces), a smart-proxy class can be written to provide a single logical view of this collection of objects, whether they are legacy objects or not-yet-defined objects. The smart-proxy member functions contain the functionality required to call the correct operation on the correct target object. If new target operations are added later, only the smart proxy needs to be modified. The changes will be transparent to the client.

11.3 TAO’s Smart Proxy Framework

This section describes the classes that make up TAO’s smart-proxy framework, the responsibilities of each class, and how these classes interact. There are six C++ classes in the framework, but developers are only

responsible for writing two of them; the other four are provided in TAO or are generated by the IDL compiler. If you would like to get started writing and using smart proxy classes, you may skip now to 11.4 and refer to this section as needed.

Figure 11-1 shows the six classes that make up the TAO smart proxy framework.

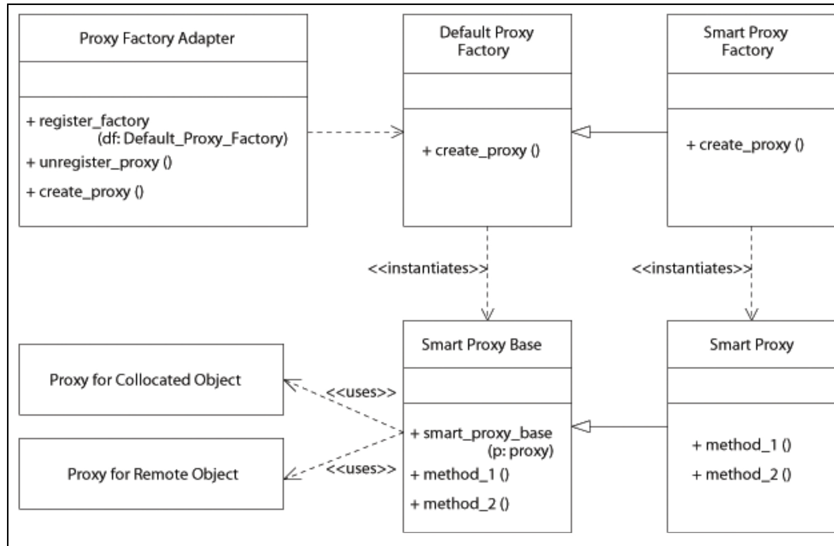


Figure 11-1 TAO's Smart Proxy Framework

The `TAO_Smart_Proxy_Base` class is part of the TAO source code (see `$TAO_ROOT/tao/SmartProxies/Smart_Proxies.h`). For each interface, the TAO IDL compiler generates a smart proxy base class that inherits from `TAO_Smart_Proxy_Base`. The smart proxy base classes are discussed in 11.3.3.

Invoking the TAO IDL compiler on an interface named `MyInterface` causes the classes `MyInterface`, `MyInterface_out`, and `MyInterface_var` to be generated. If the TAO IDL compiler is invoked with the `-Gsp` option on this same interface, three additional classes are generated that support smart proxies. They are:

- `TAO_MyInterface_Proxy_Factory_Adapter`
- `TAO_MyInterface_Default_Proxy_Factory`
- `TAO_MyInterface_Smart_Proxy_Base`

These three classes are discussed in 11.3.1 through 11.3.3.

Two classes are written by the application developer to provide application-specific behavior for the smart-proxy factory and the smart proxy. Your smart-proxy factory class will inherit from the default proxy-factory class generated by the TAO IDL compiler, and your smart proxy class will inherit from the smart-proxy base class, also generated by the IDL compiler. These classes are discussed in 11.4.

11.3.1 The Proxy Factory Adapter

The proxy-factory adapter uses a proxy factory to create proxies. It provides a consistent interface for both the default and developer-written proxy factories. It "adapts" the various proxy-factory interfaces to a common one. The proxy-factory adapter contains the following three member functions:

- `register_proxy_factory()`
- `unregister_proxy_factory()`
- `create_proxy()`

It also contains the `proxy_factory_` member variable that stores a pointer to the proxy factory currently in use. The type of this pointer is the type of the default proxy factory. For the interface `MyInterface`, the type of this pointer is `TAO_MyInterface_Default_Proxy_Factory`.

The proxy-factory adapter class is instantiated as an `ACE_Singleton` object. For the interface `MyInterface`, it is defined by a `typedef` as the singleton class `TAO_MyInterface_Proxy_Factory_Adapter`. The three member functions of this class are discussed below.

- `register_proxy_factory()`

  This function takes a pointer to a proxy factory as a parameter and stores it in the `proxy_factory_` member variable. The default proxy-factory constructor calls this function to register itself with the adapter. A developer-written proxy-factory constructor will implicitly call this function because it inherits from the default factory, and thus implicitly calls the default factory constructor.

- `unregister_proxy_factory()`

  This function deletes the currently-registered proxy-factory object, then sets the `proxy_factory_` member variable to zero. It is called by the

member function `register_proxy_factory()` prior to storing a new proxy factory pointer in the `proxy_factory_` member variable. It is also called by the smart-proxy-base `get_proxy()` member function. It is necessary to unregister the currently-registered proxy-factory object before registering a new one to avoid getting into an infinite loop.

- `create_proxy()`

  When this function is called, if the `proxy_factory_` member variable points to a proxy factory, then that factory's `create_proxy()` member function is called to create a new proxy object.

  If the `proxy_factory_` member variable is zero when this function is called (i.e., no factory object is currently registered), the default proxy-factory constructor is called. This constructor will register itself with the adapter by calling the adapter's `register_proxy_factory()` member function. Once the registration process is completed, the `proxy_factory_` member variable points to the default proxy factory. The factory's `create_proxy()` function is then called.

11.3.2 The Default Proxy Factory

The default proxy factory serves both as the base class for developer-written proxy factories and as the default factory to be used in the absence of a developer-written factory. Its `create_proxy()` member function simply returns the proxy pointer that is passed to it as an argument. Because this function is called only by the proxy-factory adapter's `create_proxy()` member function with an argument of a pointer to the default proxy, it always returns a pointer to the default proxy.

11.3.3 The Smart Proxy Base Classes

The `TAO_Smart_Proxy_Base` class is the base class for all TAO-IDL-compiler-generated smart-proxy base classes. Its constructor takes as an argument a pointer to the default proxy object that it stores in the member variable `base_proxy_`.

When the TAO IDL compiler is invoked with the `-Gsp` option, it generates a smart-proxy base class that inherits from both the default proxy class and the `TAO_Smart_Proxy_Base` class. For an interface named `MyInterface`, the `TAO_MyInterface_Smart_Proxy_Base` class is generated. This class contains:

- The member variable proxy_ that stores a pointer to the default proxy.
- The member function get_proxy() that returns the value of proxy_.
- Overridden member functions of the default proxy class that represent operations and attributes.

11.3.3.1 The overridden member functions

Each member function in the default proxy that represents an operation or attribute of the interface is overridden by a function in the smart-proxy base class generated by the TAO IDL compiler. The overriding function first calls the get_proxy() member function to get a pointer to the default proxy, then uses this pointer to call the overridden member function. For a default proxy function op(arg), the body of the overriding function in the smart proxy base class is:

```
{
  return this->get_proxy()->op(arg);
}
```

11.3.3.2 The get_proxy() function

The get_proxy() function returns a pointer to the default proxy. Because the value of the proxy_ member variable is zero upon the first invocation of this function, a pointer to the default proxy must be obtained. This is accomplished by narrowing the pointer held by the base_proxy_ variable, which is inherited from the TAO_Smart_Proxy_Base class. For the MyInterface default base class, the definition of get_proxy is:

```
MyInterface_ptr
TAO_MyInterface_Smart_Proxy_Base::get_proxy (void)
{
  // Obtain the real proxy stored in <base_proxy_>
  if (CORBA::is_nil (this->proxy_.in()))
    {
     TAO_MyInterface_PROXY_FACTORY_ADAPTER::instance()->unregister_proxy_factory);
      this->proxy_ = ::MyInterface::_unchecked_narrow(this->base_proxy_.in());
    }
    return this->proxy_.in();
}
```

11.3.4 An Overview of the Smart Proxy Creation Process

The key players in the smart proxy creation process are the *proxy factory*, the *proxy-factory adapter*, and the `_unchecked_narrow()` function. The proxy-factory adapter class and the default proxy-factory class are created along with the default proxy class (the standard proxy class that is always generated) when the TAO IDL compiler is invoked with the `-Gsp` option. A smart-proxy factory class that inherits from the default proxy-factory class is then written by the application developer.

11.3.4.1 The proxy factory and the proxy-factory adapter

The default proxy factory and the smart-proxy factory each contains a function named `create_proxy()`. The default proxy factory version of this function returns a pointer to the default proxy object, whereas the smart-proxy factory version of this function returns a pointer to a smart-proxy object. Neither function is called directly by the application to create a proxy. Instead, the proxy-factory adapter provides its own version of `create_proxy()` that chooses which factory to use when creating a proxy.

The proxy-factory adapter maintains a proxy-factory pointer as a data member. At the time a smart-proxy factory is constructed, a pointer to this factory is loaded into the adapter's data member, replacing any previously-constructed factory. When the adapter's `create_proxy()` function is called, it uses this pointer to call the `create_proxy()` function on the factory. If the pointer value is zero (because no smart-proxy factory has been constructed), the adapter will construct a default factory, then call `create_proxy()` on it.

11.3.4.2 The _unchecked_narrow() function

During the narrowing of an object reference, the `_unchecked_narrow()` function is called to provide a pointer to a proxy object. This function first loads a pointer to either the default proxy or a collocated object into the variable `proxy`. If the object reference is being narrowed to the `MyInterface` type, then the `_unchecked_narrow()` function will contain the following code:

```
return TAO_MyInterface_PROXY_FACTORY_ADAPTER::instance ()->create_proxy (proxy);
```

The proxy factory adapter singleton is instantiated and its `create_proxy()` function is called with `default_proxy` as its argument. If a smart proxy

factory has already been constructed, the adapter calls `create_proxy()` on that factory. Otherwise, the default proxy factory will be constructed and its `create_proxy()` function will be called. In either case a pointer to the created proxy is ultimately returned by `_unchecked_narrow()`.

Note that there is a singleton instance of a proxy factory adapter for each interface type.

11.4 Writing and Using Smart Proxy Classes

Using smart proxies in TAO involves the following steps:

- Compile the IDL interface(s) using the `-Gsp` option.
- Define the smart proxy class.
- Define the smart-proxy factory class.
- Instantiate a smart-proxy factory object.

11.4.1 The TAO IDL Compiler -Gsp Option

Compiling the IDL interface with the `-Gsp` option generates the additional classes needed to support the smart proxy feature. Of these classes, the two that are most relevant to using smart proxies are the smart-proxy base class and the default proxy factory. Your smart-proxy class will inherit from the generated smart-proxy base class. You will also need to create a smart-proxy factory class that inherits from the generated default proxy-factory class.

When the TAO IDL compiler is invoked on `MyInterface` with the `-Gsp` option, the generated smart-proxy base class and the default proxy-factory class are:

- `TAO_MyInterface_Smart_Proxy_Base`
- `TAO_MyInterface_Default_Proxy_Factory`

These classes are found in `MyInterfaceC.h` and `MyInterfaceC.cpp` along with the `MyInterface` proxy class.

The `-Gsp` option will automatically be added to the `tao_idl` options when your MPC project inherits from the `smart_proxies` base project. In fact, it also adds the `TAO_SmartProxies` library to the set of libraries linked into the MPC project.

When a project involves multiple interfaces, some of which are used with smart proxies and some of which are not, less code will be generated if the interfaces that are not used with smart proxies are compiled separately without the `-Gsp` option. The following is an MPC example which shows how to use the `-Gsp` option with just the smart proxy related interfaces.

```
project: taoexe, smart_proxies {
  IDL_Files {
    with_smart_proxies.idl
  }
  IDL_Files {
    commandflags -= -Gsp
    without_smart_proxies.idl
  }
  Source_Files {
    main.cpp
    with_smart_proxiesC.cpp
    without_smart_proxiesC.cpp
  }
}
```

Note *When an interface is compiled with the* `-Gsp` *option, the generated code uses the* `TAO_Smart_Proxy_Base` *class. Since this class is defined in the file* `$TAO_ROOT/tao/SmartProxies/Smart_Proxies.h`*, this file is automatically included by the generated code.*

11.4.2 The Smart Proxy Class

Your smart-proxy class should inherit from the smart-proxy base class generated by the IDL compiler for your interface. You can name your smart-proxy class anything you want. For example, you might define a smart-proxy class called `Smart_MyInterface_Proxy` for the interface `MyInterface` as follows:

```
class Smart_MyInterface_Proxy : public TAO_MyInterface_Smart_Proxy_Base
{ ... };
```

Your smart-proxy class may override any of the operations defined in the base class. You only need to override the operations for which you want to provide custom behavior in your smart-proxy class. The default behavior of these operations defined in the base class is to simply delegate to a standard proxy

for the interface. The signatures of these operations follow the standard IDL-to-C++ mapping rules for interface operations.

11.4.3 The Smart Proxy Factory Class

Proxy objects are created by a proxy factory. You need to implement a factory to generate instances of your smart-proxy class. Your smart-proxy factory class should inherit from the proxy-factory class generated by the IDL compiler for your interface. You can name your proxy factory class anything you want. For example, you might define a smart proxy factory class called `Smart_MyInterface_Proxy_Factory` for the interface `MyInterface` as follows:

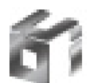
```
class Smart_MyInterface_Proxy_Factory :
  public TAO_MyInterface_Default_Proxy_Factory
{
  public:
    virtual MyInterface_ptr create_proxy (
      MyInterface_ptr proxy);
  ...
};
```

Your factory's `create_proxy()` function will be called by the proxy-factory adapter whenever a new proxy is needed (e.g., when an object reference is narrowed to the interface proxy type). Your implementation of the `create_proxy()` function should return a pointer to a proxy object. For example, it might create a new instance of your smart-proxy class or return a pointer to a singleton instance of your smart-proxy class.

You may provide any other functionality you need in your smart proxy factory class. For example, it may allocate resources it needs in its constructor and deallocate them in its destructor.

11.4.4 Instantiating a Smart Proxy Factory Object

When you want to use smart proxies, your client code should create an instance of your smart-proxy factory class before obtaining a proxy. When your smart-proxy factory is created, it will register itself with the proxy factory adapter, thereby replacing the adapter's pointer to the default proxy factory. Then, when a proxy is needed (e.g., when an object reference is narrowed to the interface proxy type), the proxy-factory adapter will call your proxy factory's `create_proxy()` member function to obtain a proxy.



All you need to do in your client code is instantiate your proxy factory. Its base-class constructor contains the necessary logic to register your proxy factory with the proxy-factory adapter. For example, here is code to create and register a smart-proxy factory for the interface `MyInterface`:

```
// Create an instance of the smart proxy factory.
Smart_MyInterface_Proxy_Factory * factory =
  new Smart_MyInterface_Proxy_Factory();
// Obtain object references and narrow them as usual...
```

You should always instantiate your smart-proxy factory on the heap. The proxy-factory adapter assumes ownership of it and deletes it in `unregister_proxy_factory()`. *Do not attempt to delete it yourself.*

You can, of course, create more than one type of smart-proxy factory and/or smart proxy for a given interface. Simply instantiate the factory you want to use at run time. Only one proxy factory for a given interface type can be registered with that interface's proxy-factory adapter at a time. Instantiating a new factory will cause the previous factory to be unregistered (and deleted) from the adapter before the new one is registered.

11.5 Linking Your Application

A Smart Proxy enabled application must be linked with the `TAO_SmartProxies` library. This library contains the `TAO_Smart_Proxy_Base` class from which all generated smart proxy classes inherit.

You can easily add this library (and the `tao_idl -Gsp` option) by adding the `smart_proxies` base project to your MPC project inheritance list. You will have to manually add the `TAO_SmartProxies` library to your list of libraries if you are not using MPC.

11.6 A Smart Proxy Example

The `Messenger` example introduced in Chapter 3 is extended here to use the smart proxy feature. We add a simple logging operation to the example through the introduction of a `Logger` interface. The `Logger` interface has one operation, `log_message()`, that writes a message to a log file. Our smart

proxy's send_message() operation calls log_message() to log information about usage of the Messenger both before and after it calls the default proxy's send_message() member function. The complete source code for this example can be found in the TAO source code distribution in $TAO_ROOT/DevGuideExamples/SmartProxies.

11.6.1 The Messenger Interface

The Messenger interface is unchanged from the original example:

```
interface Messenger
{
  boolean send_message(in string user_name,
                       in string subject,
                       inout string message);
};
```

When the TAO IDL compiler is invoked on this interface with the -Gsp option, it generates the following classes in the files MessengerC.h and MessengerC.cpp:

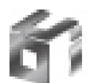
- The class Messenger is the normal proxy class for the Messenger interface.
- The class TAO_Messenger_Smart_Proxy_Base inherits from both the Messenger class and the TAO_Smart_Proxy_Base class.
- The class TAO_Messenger_Default_Proxy_Factory creates default proxy objects and serves as the base for the Smart_Messenger_Proxy class that we use in this example.
- TAO_Messenger_PROXY_FACTORY_ADAPTER is the proxy factory adapter singleton.

11.6.2 Implementation of the Messenger Interface

Recall that the use of smart proxies is strictly a client-side issue; the server is unaware of the existence of smart proxies. Therefore, the implementation of the Messenger interface is unchanged from the original example and is not shown here.

11.6.3 The Logger Interface

The Logger interface is:



```
// Logger.idl
interface Logger
{
  boolean log_message (in string message);
};
```

11.6.4 Implementation of the Logger Interface

To implement the `Logger` interface, we first generate starter code by invoking the TAO IDL compiler on the `Logger` interface with the `-GI` option. Next, we rename the generated `LoggerI.{h,cpp}` files to `Logger_i{h,cpp}` and add our Logger's implementation logic. In the code shown below, the added implementation code is shown in **bold** type.

Note *Smart proxies are not being used with the* `Logger` *interface, so the* `-Gsp` *option does not need to be used when compiling it since doing so will add unnecessary code to* `LoggerC.h` *and* `LoggerC.cpp`.

The following code is from `Logger_i.h`:

```
#include "LoggerS.h"
#include <fstream>

class  Logger_i : public virtual POA_Logger
{
public:
  Logger_i (void);
  virtual ~Logger_i (void);
private:
  std::ofstream log_file; // Output file stream to which messages are logged.
  time_t log_time;        // Needed for creating a time stamp.
  char* log_time_string;  // The time stamp string.

  virtual CORBA::Boolean log_message (const char * message);
};
```

The following code is from the implementation of `class Logger_i` found in `Logger_i.cpp`:

```
#include "Logger_i.h"

Logger_i::Logger_i (void)
```

```
{
  log_file.open("Logger.txt");
}

Logger_i::~Logger_i (void)
{
  log_file.close();
}

CORBA::Boolean Logger_i::log_message (const char * message)
{
  ACE_OS::time(&log_time);
  log_time_string = ACE_OS::ctime(&log_time);
  // Replace carriage return with string delimiter.
  log_time_string[24] = '\0';
  log_file << log_time_string << " " << message << std::endl;
  return true;
}
```

11.6.5 The Smart Proxy Factory

In this example, the smart-proxy factory class for the `Messenger` interface is named `Smart_Messenger_Proxy_Factory`. It inherits from the `TAO_Messenger_Default_Proxy_Factory` class that is generated by the IDL compiler. The factory class is defined in the file `Smart_Messenger_Proxy.h` as follows:

```
#include "MessengerC.h"
#include "LoggerC.h"

class Smart_Messenger_Proxy_Factory : public TAO_Messenger_Default_Proxy_Factory
{
  public:
    Smart_Messenger_Proxy_Factory(CORBA::ORB_ptr orb);
    virtual Messenger_ptr create_proxy (
      Messenger_ptr proxy);
  private:
    Logger_var logger_;
};
```

The constructor, found in the file `Smart_Messenger_Proxy.cpp`, is implemented as follows:

```
#include "Smart_Messenger_Proxy.h"
#include <iostream>
#include <stdexcept>
```

```
Smart_Messenger_Proxy_Factory::Smart_Messenger_Proxy_Factory(
  CORBA::ORB_ptr orb)
{
  std::cout << "Creating smart proxy factory" << std::endl;
  // Convert the contents of the Logger.ior file to an object reference.
  CORBA::Object_var obj = orb->string_to_object("file://Logger.ior");
  if (CORBA::is_nil(obj.in())) {
    throw std::runtime_error (
      "Smart_Messenger_Proxy_Factory::CTOR: Nil Logger reference");
  }

  // Narrow the object reference to a Logger object reference.
  logger_ = Logger::_narrow(obj.in());
  if (CORBA::is_nil(logger_.in ())) {
    throw std::runtime_error (
      "Smart_Messenger_Proxy_Factory::CTOR: Not a Logger object reference");
  }
}
```

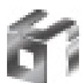
The following definition overrides the `create_proxy()` virtual member function from the base class:

```
Messenger_ptr
Smart_Messenger_Proxy_Factory::create_proxy (
  Messenger_ptr proxy)
{
  Messenger_ptr smart_proxy = 0;
  if (CORBA::is_nil(proxy) == 0)
    smart_proxy = new Smart_Messenger_Proxy(proxy, logger_.in());
  return smart_proxy;
}
```

11.6.6 The Smart Proxy Class

In this example, the smart-proxy class for the `Messenger` interface is named `Smart_Messenger_Proxy`. It inherits from the `TAO_Messenger_Smart_Proxy_Base` class that is generated by the IDL compiler. The `Smart_Messenger_Proxy` class is defined as follows:

```
#include "MessengerC.h"
#include "LoggerC.h"

class Smart_Messenger_Proxy : public TAO_Messenger_Smart_Proxy_Base
{
  public:
    Smart_Messenger_Proxy (Messenger_ptr proxy, Logger_ptr logger);
```



```
    virtual CORBA::Boolean send_message(
      const char* user_name,
      const char* subject,
      char*& message);
  private:
    Logger_var logger_;
};
```

In the constructor, the `Messenger` proxy passed in is used to initialize the `TAO_Smart_Proxy_Base` base class. In addition, the object reference of the `Logger` object is duplicated and stored in a private data member of type `Logger_var`.

```
#include "Smart_Messenger_Proxy.h"
#include <iostream>

Smart_Messenger_Proxy::Smart_Messenger_Proxy(
  Messenger_ptr proxy, Logger_ptr logger)
    : TAO_Smart_Proxy_Base(proxy),
      logger_(Logger::_duplicate(logger))
{
  std::cout << "Creating smart proxy" << std::endl;
}
```

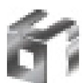
Now we override the `send_message()` operation in the smart proxy class. When the `MessengerClient` invokes `send_message()`, the smart proxy calls `log_message()` on the `Logger` proxy object both before and after it calls `send_message()` on the `Messenger` object. Note that it delegates the call to `send_message()` to its base class.

```
CORBA::Boolean
Smart_Messenger_Proxy::send_message (
  const char* user_name,
  const char* subject,
  char*& message)
{
  logger_->log_message ("Before send_message()");
  CORBA::Boolean ret_val =
    TAO_Messenger_Smart_Proxy_Base::send_message (user_name, subject, message);
  logger_->log_message ("After send_message()");
  return ret_val;
}
```

11.6.7 The MessengerClient

The only changes needed in the original `MessengerClient` are to `#include` the smart proxy's header file and to create an instance of the smart-proxy factory before obtaining and using a `Messenger` object reference. Here is the modified `MessengerClient` program:

```
#include "MessengerC.h"
#include "Smart_Messenger_Proxy.h"
#include <iostream>
#include <stdexcept>

int main (int argc, char* argv[])
{
  try {
    // Initialize the ORB
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Create a smart proxy factory. This will register it with the
    // smart proxy factory adapter so it will be used to create
    // Messenger proxies. Note that it is created on the heap, but is
    // otherwise unused here.
    new Smart_Messenger_Proxy_Factory (orb.in());

    // Convert the contents of the Messenger.ior file to an object reference.
    CORBA::Object_var obj = orb->string_to_object ("file://Messenger.ior");
    if (CORBA::is_nil(obj.in())) {
      std::cerr << "Nil Messenger reference" << std::endl;
      return 1;
    }

    // Narrow the object reference to a Messenger object reference.
    Messenger_var messenger = Messenger::_narrow(obj.in());
    if (CORBA::is_nil(messenger.in())) {
      std::cerr << "Not a Messenger object reference" << std::endl;
      return 1;
    }

    // Create a message and send it to the Messenger.
    CORBA::String_var message = CORBA::string_dup ("Hello!");
    messenger->send_message ("TAO User", "TAO Test", message.inout());
    std::cout << "Message was sent" << std::endl;

    // Release resources.
    orb->destroy();
  }
  catch (CORBA::Exception& e) {
    std::cerr << "Caught a CORBA exception: " << e << std::endl;
    return 1;
```



```
  }
  catch (std::exception& ex) {
    std::cerr << ex.what() << std::endl;
    return 1;
  }

  return 0;
}
```

11.6.8 The MessengerServer

Recall that the use of smart proxies on the client side is completely transparent to the server side. Thus, the `MessengerServer` is unchanged from the original example and is not shown here.

11.6.9 The LoggerServer

The `LoggerServer` is very simple. In fact, it is identical to the `MessengerServer` except that every instance of the name "`[Mm]essenger`" in the `MessengerServer` is replaced with the name "`[Ll]ogger`" in the `LoggerServer`.

```
#include "Logger_i.h"
#include <iostream>
#include <fstream>

int
main(int argc, char* argv[])
{
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Get a reference to Root POA.
    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

    // Activate the POA manager.
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Create a Logger_i servant.
    PortableServer::Servant_var<Logger_i> logger_servant = new Logger_i();

    // Register the servant with the RootPOA, obtain its object reference,
    // stringify it, and write it to a file.
   PortableServer::ObjectId_var oid = poa->activate_object(logger_servant.in());
```

```
    CORBA::Object_var logger_obj = poa->id_to_reference(oid.in());
    CORBA::String_var str = orb->object_to_string(logger_obj.in());
    std::ofstream iorFile("Logger.ior");
    iorFile << str.in() << std::endl;
    iorFile.close();
    std::cout << "IOR written to file Logger.ior" << std::endl;

    // Accept requests from clients.
    orb->run();

    // Release resources.
    orb->destroy();
  }
  catch (CORBA::Exception&) {
    std::cerr << "Caught a CORBA exception." << std::endl;
    return 1;
  }
  return 0;
}
```

11.6.10 Running the Programs

Compile the `LoggerServer`, `MessengerServer`, and `MessengerClient` programs. Then, start the `LoggerServer` and `MessengerServer` before running the `MessengerClient`.

11.6.11 The Output

When you run the `MessengerServer`, you will see the following output:

```
IOR written to file Messenger.ior
```

When you run the `LoggerServer`, you will see the following output:

```
IOR written to file Logger.ior
```

When you run the `MessengerClient`, you will see the following output from the client:

```
Creating smart proxy factory
Creating smart proxy
Message was sent
```

and the following output from the `MessengerServer`:

```
Message from: TAO User
Subject:      TAO Test
Message:      Hello!
```

In addition, the `Logger.txt` file will contain something similar to the following:

```
Mon Sep 15 12:07:57 2003 Before send_message()
Mon Sep 15 12:07:57 2003 After send_message()
```

CHAPTER 12

Local Interfaces

12.1 Introduction

TAO implements the concept of "local" interfaces as described in the CORBA 3.1 specification (OMG Document formal/08-01-04, 7.8.7) and C++ mapping (OMG Document formal/08-01-09, 4.37). The above specifications define a standard set of behaviors for *locality-constrained* objects (cannot be invoked externally or remotely). Previously, pseudo-IDL (PIDL) was used to define operations that the ORB supported but were intended only for the local application. PIDL has been deprecated in favor of standard IDL with interfaces specified as local.

Note *The file extension* `PIDL` *is still used on many files in the TAO distribution. Many of these files are now true IDL. Some are still pseudo-IDL.*

With the inclusion of local interfaces, specifications can avoid creation of special pseudo-objects that are never called remotely, such as the POA and ORB interfaces. Local interfaces permit you to define portable local classes that are independent of any ORB-specific optimizations for local access (e.g.,

interceptors are not called, POA policies are not honored). Local objects are important for server-only components. They help improve performance and minimize memory footprint by avoiding the overhead of ORB mediation. In addition, the use of a local interface permits the consistent disposition of objects because locality-constrained objects and remote objects share a uniform method of definition (IDL).

A local interface is specified in the IDL with the keyword `local` before the keyword `interface`. An instance of a local interface is a local object. A standard IDL interface is an *unconstrained* interface. Local interfaces and objects differ from unconstrained interfaces and objects in several ways:

- The ORB does not mediate any invocation on a local object. Implementations of local interfaces are responsible for providing the parameter copy semantics that clients expect.
- A local interface may inherit from other local or unconstrained interfaces. However, an unconstrained interface may not inherit from a local interface.
- Local types cannot be marshaled, externalized, or invoked from another process. References to local objects cannot be converted to strings through `CORBA::ORB::object_to_string()`. An attempt to do so raises a `CORBA::MARSHAL` system exception with the minor code set to 4.
- A local object is valid only in the process in which it is instantiated.
- There is no concept of an “object id” for a local object. Its identity is implementation specific (e.g., pointer or reference).
- Neither the Dynamic Invocation Interface (DII) nor Asynchronous Method Invocation (AMI) is supported on local objects.
- Instances of certain local objects that are part of the OMG specification (e.g., POA) are obtained via `resolve_initial_references()`.

12.2 C++ Mapping for LocalObject

A locality-constrained class is derived from both `CORBA::LocalObject` and the class mapping the interface. The class `CORBA::LocalObject` is used as a base class for locality-constrained implementations. It is derived from `CORBA::Object`. The `CORBA::LocalObject` class implements the

following `CORBA::Object` pseudo-operations by throwing the `CORBA::NO_IMPLEMENT` exception.

- `get_interface()`
- `get_domain_managers()`
- `get_policy()`
- `get_client_policy()`
- `set_policy_overrides()`
- `get_policy_overrides()`
- `validate_connection()`
- `get_component()`
- `repository_id()`

The `CORBA::LocalObject` class provides implementations of the following `CORBA::Object` pseudo-operations:

- `non_existent()`—always returns FALSE.
- `hash()`—returns a consistent hash value for the lifetime of the object.
- `is_equivalent()`—returns TRUE if the references refer to the same `CORBA::LocalObject` implementation.
- `is_a()`—returns TRUE if the LocalObject derives from or is itself the type specified by the logical_type_id argument.
- get_orb()—The default behavior is to throw the `CORBA::NO_IMPLEMENT` exception. Certain local objects will, when specified by the specification, return their associated ORB. Examples include POAs, current objects, and portable interceptors.

12.3 Changing Existing Interfaces to Local Interfaces

With the adoption of local interfaces, the specification changed the following into local interfaces:

- `CORBA::Current`
- All the interfaces in the `DynamicAny` module.

- All the interfaces in the `PortableServer` module.

The following CORBA messaging interfaces have become local interfaces:

- `CORBA::PolicyManager`
- `CORBA::PolicyCurrent`
- `CORBA::Pollable`
- `CORBA::DIIPollable`
- `CORBA::PollableSet`
- All interfaces in the `Messaging` module that inherit from `CORBA::Policy`

12.4 Example: ServantLocator

The POA determines which servant is associated with a particular request by calling upon the application's Servant Manager. An application registers a Servant Manager with a POA. There are two types of Servant Managers, depending upon whether or not the POA retains the associations of objects to servants in its Active Object Map. The `ServantRetentionPolicy` is used to determine if the POA retains servant activations in the Active Object Map (`RETAIN`) or does not (`NON_RETAIN`). When this policy is set to `RETAIN`, the Servant Manager must *activate* the servant associated with the object and implement the `PortableServer::ServantActivator` interface. When the `ServantRetentionPolicy` is set to `NON_RETAIN`, the Servant Manager must *locate* the servant associated with the object and must implement the `PortableServer::ServantLocator` interface.

The following example modifies the `Messenger` example from Chapter 3. A Servant Locator is used to locate the servant associated with the `Messenger` object when a request is invoked on this object. The complete source code for this example can be found in the TAO source code distribution in `$TAO_ROOT/DevGuideExamples/LocalObjects/ServantLocator`.

12.4.1 The Messenger Locator Implementation

The following implementation of the `Messenger_Locator_i` class will find the `Messenger` servant that incarnates the target object of the request. The `PortableServer::ServantLocator` interface provides two operations:

preinvoke() and postinvoke(). The preinvoke() operation is invoked by the POA to bind a servant to the target CORBA object when an incoming request is received. The postinvoke() operation is invoked by the POA to release the servant once the request has been fulfilled. The following code shows the definition of the Messenger_Locator_i implementation class:

```
#include <tao/corba.h>
#include <tao/PortableServer/PortableServer.h>

class Messenger_Locator_i :
  public PortableServer::ServantLocator,
  public CORBA::LocalObject
{
 public:
  Messenger_Locator_i ();

  // Preinvoke function
  virtual PortableServer::Servant preinvoke (
    const PortableServer::ObjectId& oid,
    PortableServer::POA_ptr poa,
    const char* operation,
    void*& cookie);

  // Postinvoke function
  virtual void postinvoke (
    const PortableServer::ObjectId& oid,
    PortableServer::POA_ptr poa,
    const char* operation,
    void* cookie,
    PortableServer::Servant servant);

};
```

The Messenger_Locator_i class inherits from both its skeleton class, PortableServer::ServantLocator, and CORBA::LocalObject. To implement a PortableServer::ServantLocator, you must inherit from both PortableServer::ServantLocator and CORBA::LocalObject.

The preinvoke() operation converts the ObjectId to a string. If the ID is valid, a new servant is created and returned. The cookie allows the ServantLocator to pass data from an invocation of preinvoke() to the corresponding postinvoke() invocation. The following code shows the implementation of the preinvoke() operation:

```
PortableServer::Servant
Messenger_Locator_i::preinvoke (
```

```
  const PortableServer::ObjectId& oid,
  PortableServer::POA_ptr,
  const char*,
  void*& cookie )
{
  // Get the ObjectId in string format.
  CORBA::String_var oid_str = PortableServer::ObjectId_to_string (oid);

  std::cout << "preinvoke called..." << oid_str << std::endl;

  // Check if the ObjectId is valid.
  ACE_CString cstr(oid_str.in());
  if (cstr == "Messenger") {
    // Create the required servant
    PortableServer::ServantBase_var servant = new Messenger_i ();

    // Set a flag so that we know to delete it in postinvoke().
    cookie = (void *)1;

    return servant._retn();
  }
  else {
    throw CORBA::OBJECT_NOT_EXIST ();
  }
}
```

In the `postinvoke()` operation, the servant is simply destroyed:

```
void
Messenger_Locator_i::postinvoke (
  const PortableServer::ObjectId&,
  PortableServer::POA_ptr,
  const char*,
  void* cookie,
  PortableServer::Servant servant)
{

  std::cout << "postinvoke called..." << std::endl;

  // Delete the servant as it is no longer needed.
  if (cookie != 0) {
    delete servant;
  }
}
```

12.4.2 The Server Implementation

We now modify the original `MessengerServer.cpp` to use our newly-created `Messenger_Locator_i` class.

First, we replace `#include “Messenger_i.h”` with `#include “MessengerLocator_i.h”`.

We create a new POA (the `childPOA`) with the `USE_SERVANT_MANAGER` value for the `RequestProcessingPolicy` and `NON_RETAIN` for the `ServantRetentionPolicy` as follows:

```
// Create the policies and assign them for the child POA
CORBA::PolicyList policies (3);
policies.length (3);

policies [0] = rootPOA->create_id_assignment_policy (
  PortableServer::USER_ID);
policies [1] = rootPOA->create_request_processing_policy (
  PortableServer::USE_SERVANT_MANAGER);
policies [2] = rootPOA->create_servant_retention_policy (
  PortableServer::NON_RETAIN);

// Create the POA with these policies
PortableServer::POA_var childPOA =
  rootPOA->create_POA ("childPOA", mgr.in(), policies);

// Destroy the policy objects
for (CORBA::ULong i = 0; i != policies.length(); ++i)
      policies[i]->destroy();
```

Now that we have a POA that can support Servant Managers, we need to create an instance of a `ServantLocator` and set it as the Servant Manager for the `childPOA`:

```
// Create our Messenger’s ServantLocator.
PortableServer::ServantLocator_var locator = new Messenger_Locator_i;

// Set the Servant Manager with the childPOA.
childPOA->set_servant_manager (locator.in());
```

The `ServantLocator` is now registered with the `childPOA`. We now create a Messenger reference via the `childPOA` with the user-defined `ObjectId`:

```
// Get the object id for the user-created ID in the childPOA
PortableServer::ObjectId_var child_oid =
  PortableServer::string_to_ObjectId ("Messenger");
```

```
// Create the object without creating a servant.
CORBA::Object_var messenger_obj =
  childPOA->create_reference_with_id (child_oid.in(), ::_tc_Messenger->id());
```

As before, we write the object reference as a stringified IOR to the file `Messenger.ior`:

```
// Put the object reference into an IOR string
CORBA::String_var str = orb->object_to_string (messenger_obj.in());

// Write the IOR string to a file
std::ofstream iorFile ("Messenger.ior");
iorFile << str.in ();
iorFile.close ();
std::cout << "IOR written to the file Messenger.ior." << std::endl;
```

We then handle incoming requests from clients:

```
// Accept requests from clients.
orb->run();
```

12.4.3 An Example Using a Local Object

In this section, we discuss a rather contrived example, where the `Messenger` interface itself is defined as `local`:

```
// Messenger.idl
local interface Messenger
{
  boolean send_message (
    in string user_name,
    in string subject,
    inout string message);
};
```

The `Messenger` interface is now locality-constrained. The ORB will not mediate requests to instances of the `Messenger` class. The overhead associated with a call to `send_message()` is on the order of one virtual function call, making possible performance improvements of large magnitude over collocated operation invocations on normal CORBA objects.

The complete source code for this example can be found in the TAO source code distribution:
`$TAO_ROOT/DevGuideExamples/LocalObjects/Messenger`.

The definition of the `Messenger_i` class becomes:

```
// Class Messenger_i
class  Messenger_i :
  public virtual Messenger,
  public virtual CORBA::LocalObject
{
public:
  //Constructor
  Messenger_i (void);

  //Destructor
  virtual ~Messenger_i (void);

  virtual CORBA::Boolean send_message (
    const char* user_name,
    const char* subject,
    char*& message
  );

};
```

Note *The TAO IDL compiler's* `-GI` *option should not be used to automatically generate implementation-class starter code for IDL files containing local interfaces.*

The definition of the `Messenger_i` class is identical to the earlier example with the unconstrained interface:

```
#include "Messenger_i.h"

// Implementation skeleton constructor
Messenger_i::Messenger_i (void)
{
}

// Implementation skeleton destructor
Messenger_i::~Messenger_i (void)
{
}
```

```
CORBA::Boolean Messenger_i::send_message (
  const char* user_name,
  const char* subject,
  char*& message
  )

{
  // my implementation
  std::cout << "Message from: " << user_name << std::endl;
  std::cout << "Subject:      " << subject << std::endl;
  std::cout << "Message:      " << message << std::endl;
  CORBA::string_free(message);
  message = CORBA::string_dup("Thanks for the message.");
  return true;
}
```

The server and client are, by definition, collocated for this example. The code below shows the construction and use of a `Messenger` object:

```
#include "Messenger_i.h"
#include <iostream>

int main (int argc, char * argv[])
{
  try {
    // Construct a Messenger object and use it "as if" it's a corba object.
    // Put it into CORBA object reference
    // comparable to activation, narrow, etc.
    Messenger_var messenger(new Messenger_i);

    // Send a message to the Messenger object.
    CORBA::String_var message = CORBA::string_dup ("Hello!");
    messenger->send_message("TAO User", "TAO Test", message.inout());

    // Print the Messenger's reply.
    std::cout << "Reply: " << message.in() << std::endl;
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Caught CORBA::Exception : " << ex << std::endl;
    return 1;
  }

  return 0;
}
```

Recall that if you initialize a `_var` reference with a `_ptr` reference, as shown here:

```
Messenger_var messenger (new Messenger_i);
```

the `_var` takes ownership (without incrementing the reference count on the proxy) and eventually calls `CORBA::release()` on the underlying `_ptr`. The usage of this `_var` is equivalent to the `_var` of unconstrained objects.

12.4.4 Reference Counting and Local Objects

Local objects can use reference counting to manage their life cycles just as unconstrained CORBA objects. The class `CORBA::LocalObject` defines two virtual functions:

- `_add_ref()`: This member function is called when the reference is duplicated.
- `_remove_ref()`: This member function is called when the reference is released.

Beginning in the Version 1.2 C++ Mapping specification (OMG Document formal/08-01-09, 4.37), the default is that all local objects have reference counting enabled. `CORBA::LocalObject` supplies default implementations of `_add_ref()` and `_remove_ref()` with this behavior. To avoid this, simply override these member functions in your local object implementations.

Older versions of the C++ mapping (and TAO), do not have reference counting enabled by default and users must implement it themselves or use the TAO-specific base class `TAO_Local_RefCounted_Object` in place of `ORBA::LocalObject`. This class is no longer needed in TAO 2.2a, but is still supported for backward compatibility.

Note *When porting applications that implement local objects to TAO 2.2a, it is important to ensure that local object reference counts are properly managed. Improperly managed reference counts can lead to system crashes.*

CHAPTER 13

IOR Table

13.1 Introduction

When a client sends a request to a server, the object key portion of the interoperable object reference (IOR) of the target object is included in the GIOP request message header. The server's ORB uses the object key to demultiplex the request to the appropriate target servant. Typically, the ORB looks up the POA via the POA name portion of the object key, then the POA locates the servant via the ObjectId portion of the object key.

The CORBA specification defines the `corbaloc` object URL scheme to provide for convenient exchange of human-readable IORs. Here is an example of a `corbaloc` object URL:

```
corbaloc:iiop:malory:5555/ObjectKeyString
```

See 17.13.36 for more information on `corbaloc` object URLs.

Unfortunately, object keys are often not human-readable, making it difficult for users to construct simple `corbaloc` object URLs such as the example above. Users need a way to specify a simple object key string in `corbaloc`

object URLs and servers need a way to map these simple object key strings to target objects. This chapter describe's TAO's IOR Table feature that allows a server to map simple object key strings to the actual IORs of target objects.

13.2 IOR Table

The IOR Table is a TAO-specific local object that allows a server to expose an object reference as a simple `corbaloc` object URL by mapping simple object key strings to stringified IORs. The IOR Table provides a form of indirect binding similar to that provided by the TAO Implementation Repository (ImR), and can be used by any server. Many TAO services, such as the Naming Service, Notification Service, and Implementation Repository support such indirect binding.

The TAO `IORTable::Table` local interface is defined in `$TAO_ROOT/tao/IORTable/IORTable.pidl`. It provides simple `bind()`, `rebind()`, and `unbind()` operations. A portion of the interface definition is shown here:

```
module IORTable
{
  exception AlreadyBound {};
  exception NotFound {};

  local interface Table
  {
    void bind (in string object_key,in string IOR)
      raises (AlreadyBound);
    void rebind (in string object_key, in string IOR);
    void unbind (in string object_key)
      raises (NotFound);

    // ...
  };
```

Your server can access the ORB's IOR Table by calling `resolve_initial_references("IORTable")`. You then use the `bind()` operation to bind a simple object key string to a stringified IOR, as shown in this example:

```
#include <tao/IORTable/IORTable.h>
...
```

```
// Initialize the ORB, get the RootPOA, activate servants, and
// generate object references as usual (not shown).

// Stringify your object references for binding in the IORTable.
CORBA::String_var messenger_str1 = orb->object_to_string (ior1.in());
CORBA::String_var messenger_str2 = orb->object_to_string (ior2.in());

// Get the IORTable from the ORB.
CORBA::Object_var table_obj = orb->resolve_initial_references("IORTable");
IORTable::Table_var table = IORTable::Table::_narrow(table_obj.in());

// Bind your objects to simple object key strings.
tbl->bind("Messenger1", messenger_str1);
tbl->bind("Messenger2", messenger_str2);
```

The simple object key strings can also indicate the persistent POA name in which the object is activated (if not the Root POA), as shown here:

```
tbl->bind("MessengerService/Messenger1", messenger_str1);
tbl->bind("MessengerService/Messenger2", messenger_str2);
```

Now, if clients know the simple object key string of the object they want to access, and the endpoint on which the server is listening, they can use simple `corbaloc` ObjectURLs such as the following:

```
corbaloc:iiop:malory:5555/Messenger1
```

The client would normally provide this ObjectURL to the ORB's `string_to_object()` operation, as follows:

```
CORBA::Object_var obj =
  orb->string_to_object ("corbaloc:iiop:malory:5555/Messenger1");
Messenger_var myobj = Messenger::_narrow (obj.in());
```

In this case, we assume the server was stared using and endpoint specification such as:

```
-ORBListenEndpoints iiop://malory:5555
```

and that the object reference bound to "`Messenger1`" in the server implements the `Messenger` interface.

Typically, the first request sent to the object is an invocation of the implicit `is_a()` operation that results from a call to the static `_narrow()` function as

shown in the above example. When the request reaches the server's ORB, it matches the simple object key string from the ObjectURL to a binding in the IORTable. It then returns a `LOCATION_FORWARD` reply containing the IOR that was bound to the simple object key string to the client. The client's ORB automatically reinvokes the request on this returned IOR. This second invocation will be demultiplexed to the target object as usual. The client will continue to use the returned IOR for subsequent requests.

The IORs bound in the IORTable can be of any valid format accepted by `string_to_object()`, including `IOR:`, `corbaloc`, and `file://`. (See 17.13.36 for more information on valid IOR formats.) Clients will be forwarded to the location indicated by the returned IOR using a `LOCATION_FORWARD` reply. Note that a client could be forwarded to a different server than that from which it received the `LOCATION_FORWARD` reply, depending upon the profiles in the returned IOR.

13.3 Locator

In addition to mapping simple object key strings to IORs as described above, you can also interact with the IORTable by registering a Locator object. A Locator is a user-defined local object that the IORTable can use to find objects. If the table is unable to find a binding for the object key string in an incoming request, it will call a function in your Locator object, passing the object key string. The Locator can use any user-defined mechanism to return a valid IOR. For example, you could use this mechanism to support case-insensitive IORTable lookup.

The `IORTable::Locator` local interface is defined in `$TAO_ROOT/tao/IORTable/IORTable.pidl`. It provides a simple `locate()` operation that takes an object key string and returns a stringified IOR. You register your Locator with the IORTable via the table's `set_locator()` operation. These interfaces are shown here:

```
module IORTable
{
  local interface Locator;  // forward declaration

  exception NotFound {};

  local interface Table
  {
```

```
    // Other operations shown previously...

    void set_locator (in Locator the_locator);
  };

  local interface Locator
  {
    string locate (in string object_key)
      raises (NotFound);
  };
};
```

To use a Locator with the IORTable, simply create a class that derives from `IORTable::Locator` and `CORBA::LocalObject` and override the `locate()` method to return a stringified IOR for the given object key string. For example:

```
#include <tao/IORTable/IORTable.h>

class MessengerLocator :
  public IORTable::Locator,
  public CORBA::LocalObject
{
  // Our Locator will use an internal map of key strings to stringified IORs.
  std::map<string, string> map_;

public:
  MessengerLocator ()
  {
    // Initialize the map of keys and stringified IORs (not shown).
  }
  // Override IORTable::Locator::locate().
  virtual char* locate (const char* key)
  {
    return map_[key];
  }
};
```

For more information on implementing local interfaces, see Chapter 12.

Register your locator with the IORTable by calling `set_locator()` in your server's startup code:

```
class MessengerServer
{
  MessengerLocator locator_;
public:
  void run();
```

```
};

void MessengerServer::run()
{
  // Initialize the ORB, get the RootPOA, activate servants, and
  // generate object references as usual (not shown).

  // Get the IORTable from the ORB.
  CORBA::Object_var obj = orb->resolve_initial_references("IORTable");
  IORTable::Table_var tbl = IORTable::Table::_narrow(obj.in());

  // Bind some objects to simple object key strings (as before).
  tbl->bind("Messenger1", messenger_str1);
  tbl->bind("Messenger2", messenger_str2);

  // Use our Locator to resolve any object key string not bound in the table.
  tbl->set_locator(&locator_);

  // ...
}
```

13.4 IOR Refresh

A new feature added to OCI TAO 2.2a with patch 7, IOR Refresh addresses the problem of servers started before the host's network interfaces are completely initialized. When enabled, this feature will refresh the list of IP addresses contained in an IOR bound to the IORTable.

The problem is illustrated with the following example. A TAO server using a defaulted IIOP listen endpoint argument or one for which the specified IP address implies "all interfaces," queries the host networking infrastructure to obtain a list of available IP addresses. This happens at POA initialization time, or ORB initialization, if a listen endpoint argument is used. If sometime later a new interface is enabled, for example a wifi association is made, clients using IORs published by the server are unable to reach the server through the new interface. Clients using CORBALOC IORs composed with the new interface's IP address or host name will reach the IORTable in the server, but will not be able to use the forwarded IOR.

The solution to this problem is to ensure the forwarded IOR contains as up to date list of endpoints as possible when requested by a client. This is done automatically by the IORTable when the refresh feature is enabled.

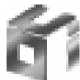
The TAO IORTable interface definition has been expanded to include a switch for enabling refresh.

```
module IORTable
{
  local interface Table
  {
    // Other operations shown previously...

    void refresh (in boolean enable);

  };
```

After resolving the `IORTable` reference, use `IORTable::refresh (true)` to enable IOR refresh. As client requests cause individual IORs to be updated, the binding is updated. Once the network topology is stable and the bindings are all updated, the active refresh can be disabled.

Note *As of TAO 2.2a patch 7, there is no way to force IORs to be refreshed through the IORTable interface. There is also no way to detect or be notified of network interface state changes.*

See Section 18.2.12 to see how to enable the IOR Refresh feature via a service configuration directive.



CHAPTER 14

Using Pluggable Protocols

14.1 Introduction

With the adoption of CORBA version 2.0, the OMG defined a wire protocol for inter-ORB communication. This protocol is defined as a messaging layer, known as the General Inter-ORB Protocol (GIOP), combined with at least one transmission protocol, TCP/IP. The mapping of GIOP on to TCP/IP is known as the Internet Inter-ORB Protocol (IIOP). The CORBA specification also defines a protocol known as the DCE Environment Specific Inter-ORB Protocol (DCE ESIOP). To be compliant with the standard, all ORB vendors must supply an implementation of IIOP. Vendors may additionally supply ESIOPs to enable inter-ORB communication using varied IPC mechanisms.

Like many CORBA implementations, TAO provides several ESIOPs. This chapter describes how to use and configure the different protocols. TAO additionally provides a programming framework so that others may produce alternative inter-ORB protocols. Section 14.18, “Developing Pluggable Protocols” describes how to develop your own ESIOP and integrate it with TAO.

Note *What has been historically referred to as a "pluggable protocol" in TAO is really a pluggable transport. TAO does not currently support pluggable messaging (i.e., only GIOP is supported), nor is it possible to choose alternative marshaling schemes (i.e., only CDR is supported). However, for historical reasons, all of the pluggable transports described in this chapter are referred to as pluggable protocols.*

14.2 Protocol Introduction

Each inter-ORB protocol consists of three basic components: messaging, marshaling, and transport. A given ESIOP consists of concrete definitions of all three of these. As an example, the IIOP protocol, as defined by the OMG, uses GIOP messaging, CDR marshaling, and the TCP/IP transport. Chapter 9 of Part 2 of the CORBA 3.1 specification (OMG Document formal/08-01-06, Chapter 17) provides details of all the layers of inter-ORB protocols. Further insight into these protocols may be found in *IIOP Complete: Understanding CORBA and Middleware Interoperability* by William Ruh, Thomas Herron, and Paul Klinker (1999).

14.2.1 Messaging

All of TAO's protocols use the General Inter-ORB Protocol (GIOP) as their messaging layer. TAO does not currently provide a way to replace the messaging layer.

Part 2 of the CORBA 3.1 Core specification (OMG Document formal/08-01-06) defines GIOP as a description of the messages that clients and servers exchange, and many other details of their interaction. The messages are defined as IDL structures in Chapter 9 of this specification.

Although GIOP is defined as transport-independent, it does make several transport assumptions that restrict the use of transports. The main assumptions are that the transport:

- is connection-oriented.
- connection initiation is similar to TCP (the server can publish a network address at which it's listening in an IOR).
- is reliable according to the TCP definition (bytes are delivered in order, at most once, and are acknowledged).

- can be viewed as a byte stream (no message size limitations, fragmentation, or alignments).
- provides notification of disorderly connection loss.

This does not mean that the transport must directly adhere to these features, just that the transport layer shows this behavior to the messaging layer.

14.2.2 Marshaling

The marshaling layer defines the transformation of inter-ORB messages into and out of a transmission (wire) format. The Common Data Representation (CDR), performs this role for all of TAO's provided protocols. TAO does not currently provide a way to replace the marshaling layer. CDR is able to account for variable byte ordering on different hardware architectures, efficiently align data types, and transfer any data type defined in IDL.

14.2.3 Transport

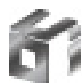
The most common CORBA protocol, IIOP, is GIOP mapped to TCP/IP. The transport layer defines connection establishment and inter-ORB message transmission. The transport layer also defines how connection "endpoints" are represented in the profiles of Interoperable Object References (IORs).

When protocols utilize low-level transports that do not support the messaging layer's transport assumptions, the transport layer of the protocol must simulate the behavior that meets these assumptions. For example, transports that are not connection oriented (like UDP or shared memory) must simulate connections.

TAO's pluggable protocol framework provides an easy way to replace the transport layer. When you write a pluggable protocol, you are writing a new transport layer. Future work may provide for replacement of the other layers.

14.3 Protocols Provided with TAO

The rest of this chapter discusses the inter-ORB protocols supplied with TAO, how to use them in your application, and how to control their behavior. The main motivator for using these protocols is typically higher performance, although security or reduced bandwidth consumption may also motivate their use. Since CORBA allows multiple profiles in an object reference, it is



possible to access the same CORBA object using more than one protocol. The protocols provided with TAO are shown in Table 14-1.

Table 14-1 Available Pluggable Protocols

`IIOP`	(**default**) Internet Inter-ORB Protocol. This protocol is required for CORBA 2.0+ compliance (found in `-lTAO`).
`UIOP`	Local IPC (i.e., Unix domain) socket-based transport protocol (found in `-lTAO_Strategies`).
`SHMIOP`	Shared memory transport protocol (found in `-lTAO_Strategies`).
`DIOP`	Datagram (UDP) transport protocol (found in `-lTAO_Strategies`).
`SSLIOP`	IIOP over Secure Sockets Layer (SSL) session protocol (found in `-lTAO_SSLIOP`). This protocol requires the freely available OpenSSL library.
`SCIOP`	SCTP Inter-ORB Protocol (found in `-lTAO_Strategies`). An SCTP-based protocol, currently only for UNIX platforms.
`HTIOP`	HTTP Tunneling Inter-ORB Protocol (requires `-lTAO_HTIOP` and `-lACE_HTBP`). An asymmetric protocol for communication across a firewall using HTTP tunnels.
`MIOP/UIPMC`	Multicast Inter-ORB Protocol (MIOP) Unreliable IP Multicast (UIPMC) protocol (found in `-lTAO_PortableGroup`).
`ZIOP`	This is an IIOP-variant with compression of the messages (found in `-lTAO_ZIOP`).
`COIOP`	Collocated-Only Inter-ORB Protocol (found in `-lTAO_Strategies`). Intended for use in systems without network interfaces.

The following sections describe each of these protocols in detail, including the exact steps needed to use the protocol. In general, the steps needed to use each existing protocol are:

1. Build the library supplying the protocol.
2. Load and initialize the protocol. TAO’s pluggable protocols are all defined as dynamically-loadable services. A factory object creates the components that collaborate to implement the protocol.
3. Identify the protocol to the ORB. This allows the ORB to request the creation of protocol components from each of the identified factories.
4. Supply any required endpoint information to be used by the ORB initialization run-time code.

14.4 Building the Protocol Libraries

As noted in Table 14-1, the base TAO library includes IIOP.

The SSLIOP components are built in a separate ORB service library, which depends on a third-party security package, as described in 14.10.

The UIPMC protocol is included as part of TAO's Portable Group implementation in the `TAO_PortableGroup` library. This library is located in `$TAO_ROOT/orbsvcs/orbsvcs/PortableGroup`.

HTIOP depends upon the HTBP library for the communication between the inside and outside peer through a firewall. This library is located in `$ACE_ROOT/protocols/ace/HTBP`. The HTIOP library is located in `$TAO_ROOT/orbsvcs/orbsvcs/HTIOP`.

ZIOP is included in its own library, `TAO_ZIOP`. The source for this library is in `$TAO_ROOT/tao/ZIOP`.

The remaining protocols are all available as part of the `TAO_Strategies` library. This library is located in `$TAO_ROOT/tao/Strategies`. The default makefile or project file for TAO will build the strategy library in the course of building the base TAO library. Separate make and project files are available if you wish to build the strategy library individually. Note that COIOP is, by default, not built. See 14.15 for details about performing a COIOP-enabled build.

14.5 Loading Pluggable Protocols

To use a protocol in TAO, you must declare its protocol factory by using the ACE Service Configurator (see 18.6.13). This factory instantiates such things as the acceptor and connector responsible for establishing connections. Declaring a protocol factory is done through options passed to the ORB's resource factory (Chapter 18) in the service configurator's (Chapter 16) configuration file. The resource factory option used is `-ORBProtocolFactory`, described in 18.6.13 Further details about individual protocol factories are given below.

Protocol factories are dynamically loadable. If you wish to use a non-default protocol, or one which is loaded automatically as a result of using a specific resource factory, you must direct the service configurator to load the protocol

factory object, in addition to supplying the argument to the resource factory. A `dynamic` directive, as shown in 16.3.2, is used to load the protocol factory code. Specific directives for loading the various protocol factories are shown below.

Pay close attention to the syntax for the dynamic directive. It is tricky, and incorrect syntax is the cause of most problems encountered when dynamically loading protocols.

14.6 IIOP

The Internet Inter-ORB Protocol is the most widely used CORBA protocol, and is required by the CORBA standard. IIOP specifies how GIOP messages are sent, such that invocations may be made from client to server over a TCP/IP network. IIOP, along with TAO's other protocols, is implemented as a pluggable protocol. This enables IIOP to be combined with other protocols, or excluded if it is not needed.

IIOP is the default protocol for TAO. If no action is taken to specifically declare an alternative protocol, IIOP will be automatically available. This also means that an endpoint will be selected for any TAO server that does not otherwise specify its endpoint(s).

14.6.1 Loading and Declaring the Protocol

The IIOP protocol is included in the TAO library. If IIOP is the only protocol used, no action is required to load and declare the protocol. However, if IIOP is to be used along with other protocols, then it must be explicitly declared, along with the other protocols. As mentioned in 14.5, this is done through an argument to the resource factory. The name of the IIOP protocol factory is "`IIOP_Factory`," as shown in the following example:

```
static Resource_Factory "-ORBProtocolFactory IIOP_Factory ... "
```

14.6.2 Address Definition

If your TAO server must always be available on a well-known address, you must specify its endpoint(s) as arguments to `CORBA::ORB_init()`.

Often, programs pass command-line arguments to `CORBA::ORB_init()`, so you can also specify endpoints on the command line.

You can also use an *implicit endpoint declaration* with IIOP-based servers. These endpoints choose a port, possibly from a constrained range, and listen on that same port across multiple network interfaces.

Occasionally, it is desirable to limit the server to a particular network interface, with or without regard to a particular port. Explicit declaration of an endpoint for TAO servers is done with the `-ORBListenEndpoints` option. See 17.13.14 for a full explanation of this option.

When making endpoint specifications, IIOP uses the prefix `iiop://` followed by an optional host name, port number, and any per-endpoint options.

A canonical, or fully-qualified, name should be used whenever supplying a host name to avoid possible inconsistencies in mapping between the host name and the IP address actually used.

Here is a sample IIOP endpoint:

```
server -ORBListenEndpoints iiop://example.ociweb.com:12345
```

Occasionally, IIOP is used in environments where DNS is not available, or is not desirable. In those situations, the IP address of the host is used in place of the host name. For example:

```
server -ORBDottedDecimalAddresses 1 -ORBListenEndpoints
iiop://192.168.1.10:3000
```

When you set `-ORBDottedDecimalAddresses 1`, TAO places the IP address in the profile. Otherwise, TAO places whatever is specified on the command line in the profiles.

Communication between processes on the same host may be optimized by using the loopback interface. This is achieved by using the reserved loopback IP address `127.0.0.1`.

According to RFC 1912 and several other RFCs, the name `localhost` should always resolve to the reserved loopback address `127.0.0.1`. Thus, it is permissible to use `localhost` to identify the host portion of an endpoint. However, some machines do not have name resolution properly configured, so it is safest to always specify `127.0.0.1`.

TAO's IIOP pluggable protocol tries to find all IP-based network interfaces. With no endpoint specification, TAO listens on all interfaces on an ephemeral port, and publishes profiles for all the interfaces it can find.

TAO usually gets this right, but finding IP interfaces requires different techniques and APIs on different platforms, so there's always a possibility TAO might miss a more exotic interface.

You can optionally specify only a port, with profiles published for all interfaces TAO can find:

```
server -ORBListenEndpoints iiop://:12345
```

Or, you can specify just a host name, allowing for port selection from the ephemeral range:

```
server -ORBListenEndpoints iiop://example.ociweb.com:
```

14.6.3 IIOP Options

Endpoints may be extended with optional values added to the end of each declaration. In general, these options are supplied in the form "name=value.". Multiple options can be fed by separating them with an '&' e.g. "name=value&name=value"

14.6.3.1 portspan

The `portspan` option constrains port selection for an IIOP endpoint to a user-specified limited range. To use this feature, a base port value is specified as shown above. The "span," or number of ports, including the base, is given as the `portspan` value. For example:

```
server -ORBListenEndpoints iiop://example.ociweb.com:12345/portspan=5
```

This instructs the server to open an endpoint on the specified interface on any port from 12345 and 12349.

The `portspan` option is particularly useful if a server or collection of services is to be accessed through a firewall.

14.6.3.2 hostname_in_ior

This option is used to explicitly specify the host name used in IORs. The server ORB will not validate the specified host name. This option allows the same host name to be used in the IORs of objects from multiple servers. This technique can be used for redundancy or as a load-balancing strategy.

For example, two separate servers could be started on separate hosts with the same hostname_in_ior option value, as follows:

```
# Start a server on host 10.20.1.100:
server -ORBListenEndpoints iiop://10.20.1.100:12345/hostname_in_ior=malory

# Start another copy of the server on host 10.20.1.200:
server -ORBListenEndpoints iiop://10.20.1.200:12345/hostname_in_ior=malory
```

In the above example, the IORs generated by both servers would have the host name `malory` encoded within their IIOP profiles.

14.6.3.3 reuse_addr

This option allows users to set the SO_REUSEADDR socket option on an endpoint. By default, this behavior is disabled and applications may fail to open an endpoint that was used by a another recently terminated application. Enabling it, by setting its value to 1, bypasses the TCP TIME_WAIT and can be used to open an endpoint on a port still in TIME_WAIT state. A typical use case for this option is when a fixed endpoint is used and the application cannot tolerate the start-up delay that is necessary to avoid the TIME_WAIT.

Note *This option is not recommended for the general use-case. Setting SO_REUSEADDR has been observed to cause unexpected side-effects on some platforms (e.g. Solaris 5.7 x86 allows programs run as same or different users to bind to the same port when SO_REUSEADDR is set by all users).*

Here is an example of enabling this option:

```
server  -ORBListenEndpoints iiop://example.ociweb.com:12345/reuse_addr=1
```

14.7 UIOP

UIOP stands for the Unix Inter-ORB Protocol. The TAO UIOP specifies how GIOP messages are sent, such that invocations may be made from client to server over Unix Domain Sockets, also known as local IPC. Local IPC often offers better performance for messaging between processes on the same host, while still presenting application developers a sockets-based interface. Though Unix Domain Sockets are supported on many platforms, a few platforms, such as Win32, pSOS and QNX, do not support them.

Unlike IIOP, UIOP is not available by default. When available, servers will present UIOP profiles only when you explicitly specify a UIOP endpoint.

14.7.1 Loading and Declaring the Protocol

The UIOP factory and its components are included in the `TAO_Strategies` library. It is possible to directly link the strategies library to your application by adding `TAO_Strategies` to the list of libraries in your makefile or project settings. UIOP, along with the other protocols included in the strategies library, may be made available by default as well by including the following TAO-specific header in your application's main source file:

```
#include <tao/Strategies/advanced_resource.h>
```

Including this file makes all of the protocols supplied in the strategies library available to the application.

You may want to limit the available protocols, which you can do by specifying the UIOP protocol factory name, `UIOP_Factory`, as an argument to the `-ORBProtocolFactory` option when initializing the resource factory. For example:

```
static Advanced_Resource_Factory "-ORBProtocolFactory UIOP_Factory"
```

Either the default Resource Factory or the Advanced Resource Factory may be used when the strategies library is statically linked.

If you also want to use IIOP, you will need to include `IIOP_Factory` as an additional argument to `-ORBProtocolFactory`.

You can also load the UIOP protocol dynamically through the service configurator. This removes the need to include the TAO-specific header or

explicitly link the strategies library. However, the TAO_Strategies library must exist in your platform's library search path (e.g., LD_LIRBARY_PATH on may Unix systems). You also must use the following dynamic directives in your svc.conf file (see 16.3.2 for more information):

```
dynamic UIOP_Factory Service_Object*
   TAO_Strategies:_make_TAO_UIOP_Protocol_Factory () ""

dynamic Advanced_Resource_Factory Service_Object*
   TAO_Strategies:_make_TAO_Advanced_Resource_Factory ()
   "-ORBProtocolFactory UIOP_Factory"
```

Once the UIOP protocol factory is loaded, the ORB will be able to communicate via this protocol.

14.7.2 Address Definition

For TAO servers to generate UIOP profiles for inclusion in object references, an endpoint must be explicitly declared using the -ORBListenEndpoints option. See 17.13.14 for more general information on this option.

UIOP Endpoints are composed of the prefix uiop:// followed by an optional Unix domain rendezvous point. Unix domain rendezvous points appear as directory entries in the local file system. As such, they are represented as a file path, such as /tmp/my_uiop_endpoint. To use this rendezvous point as a UIOP endpoint, the declaration is:

```
-ORBListenEndpoints uiop:///tmp/my_uiop_endpoint
```

You can cause TAO to create an *ephemeral* rendezvous point (similar to ephemeral ports in TCP) by using an empty endpoint declaration. You still must explicitly declare the endpoint:

```
-ORBListenEndpoints uiop://
```

TAO generates UIOP rendezvous points using a facility like the system function tempnam() or tmpnam(), depending on your platform. It places rendezvous points in the temporary directory defined for the environment, usually /tmp, and generates a unique name prefixed with "TAO". These ephemeral rendezvous points are removed upon normal shutdown of the ORB.

14.7.3 Guidelines

The rendezvous point for UIOP is any valid path and filename that the ORB has permission to read from and write to. However, UIOP rendezvous points have the same restrictions as local IPC. The following guidelines will help you use TAO's UIOP pluggable transport protocol successfully:

- *Limit the length of the endpoint.* For the greatest platform portability of your code, ensure that rendezvous points are no longer than 99 characters, including path and filename. This limit may be greater on specific platforms, but the POSIX.1g standard dictates a limit of no more than 100 characters, including null terminator. Endpoints longer than what the platform supports are truncated. and a warning is issued.
- *Use absolute paths whenever possible.* This avoids the trouble of ensuring that the client and server are started in a particular directory. For instance, declaring a relative endpoint, such as `-ORBListenEndpoints uiop://myuiop`, causes a local IPC rendezvous point called `myuiop` to be created in the current working directory. If the client is not started in the same directory, it cannot locate the rendezvous point, and fails to communicate with the server. On the other hand, using an absolute path insures that the client knows exactly where to find the rendezvous point.

  Relative paths can also expose one to security risks through man-in-the-middle attacks.
- *Insure accessibility.* It is up to the user to make sure that a given UIOP rendezvous point is accessible by both the server and the client.

14.8 SHMIOP

SHMIOP stands for the Shared Memory Inter-ORB Protocol. The TAO SHMIOP specifies the proper encoding of IORs, and how GIOP messages are sent, such that invocations may be made from client to server via shared memory.

TAO's SHMIOP implementation uses ACE memory-mapped files as the shared-memory substrate. It does not use the System V Unix shared-memory IPC mechanism.

Like UIOP, SHMIOP is useful only between processes that can share memory, typically limited to same-host processes. This protocol is only

available on platforms that support shared memory. Though supported on many platforms, it is unavailable on a few, such as some single process real-time operating systems. This protocol can provide a performance advantage over IIOP, and is available on some platforms that do not support UIOP, such as Win32.

An interesting feature of TAO's SHMIOP protocol is how it solves the notification problem. A requirement of pluggable protocols is to generate I/O events. This is solved in SHMIOP by the use of a TCP/IP connection through the loopback interface, ensuring that the ORB receives timely notification of data availability.

14.8.1 Loading the Protocol

The SHMIOP factory and its components are in the `TAO_Strategies` library. To directly link the strategies library to your application, add `TAO_Strategies` to the list of libraries in your makefile or project settings. SHMIOP, along with the other protocols included in the strategies library, may be made available by default by including the following TAO-specific header in your application's main source file:

```
#include <tao/Strategies/advanced_resource.h>
```

By including this file, you make all of the protocols supplied in the strategies library available to your application.

To limit the available protocols, specify the SHMIOP protocol factory name, `SHMIOP_Factory`, as an argument to the `-ORBProtocolFactory` option when initializing the resource factory. For example:

```
static Advanced_Resource_Factory "-ORBProtocolFactory SHMIOP_Factory"
```

Either the default Resource Factory or the Advanced Resource Factory may be used when the strategies library is statically linked.

If you also want to use IIOP, then include `IIOP_Factory` as an additional argument to `-ORBProtocolFactory`.

You can also load the SHMIOP protocol dynamically through the service configurator, which makes it unnecessary to include the TAO-specific header or explicitly link the strategies library. However, the `TAO_Strategies` library must exist in your platform's library search path (e.g.,

LD_LIRBARY_PATH on many Unix systems). You also must use the following dynamic directives in your svc.conf file (see 16.3.2 for more information):

```
dynamic SHMIOP_Factory Service_Object*
   TAO_Strategies:_make_TAO_SHMIOP_Protocol_Factory () ""

dynamic Advanced_Resource_Factory Service_Object*
   TAO_Strategies:_make_TAO_Advanced_Resource_Factory ()
   "-ORBProtocolFactory SHMIOP_Factory"
```

14.8.2 Configuring the SHMIOP Factory

The SHMIOP factory takes two options that globally affect the transports created by it. These options control the name and the size of the memory-mapped file that is used for virtual-memory backing storage.

- -MMAPFilePrefix *prefix*. This option sets the prefix on the backing store filename. This prefix may be any string that is valid as the start of a filename. If this option is omitted, then "MEM_Acceptor_" is used by default. The remainder of the filename, appended to the prefix, is the decimal representation of the port number used for I/O event notification.
- -MMAPFileSize *size*. This option sets the size of the backing store file. The size value is in bytes. The default size is 10k bytes.

  The largest message size that can be handled by SHMIOP is the size of the backing store. The minimum size of the map file should be at least the size of the message, including message headers and other overhead. You should reserve a file size equal to the expected maximum size of a request plus the expected maximum size of a reply, times a safety factor:

  ```
  (MAX_REQUEST_SIZE + MAX_REPLY_SIZE) * SAFETY_FACTOR
  ```

  For example, if you expect the maximum request size in your application to be 4k bytes, the maximum reply size to be 16k bytes, and you assume a safety factor of two, you would have:

  ```
  (4k + 16k) * 2 = 40k
  ```

You specify these as options to the SHMIOP factory in the svc.conf file. For example:

```
static SHMIOP_Factory "-MMAPFilePrefix server_shmiop -MMAPFileSize 40960"
```

14.8.3 Address Definition

For TAO servers to generate SHMIOP profiles for inclusion in object references, an endpoint must be explicitly declared using the `-ORBListenEndpoints` option. See 17.13.14 for more information on this option.

SHMIOP endpoints are composed of the prefix `shmiop://` followed by an optional port number. For example:

```
-ORBListenEndpoints shmiop://12345
```

The port number specifies the TCP port used for notification of new data. TAO uses an ephemeral port if you do not specify a port.

Although SHMIOP endpoints do not accept a host name in the declaration, the canonical name of the host appears in the profile, which is embedded in object references. This name is used to prevent processes on other hosts from attempting to use that SHMIOP profile.

14.9 DIOP

DIOP stands for Datagram Inter-ORB Protocol and is a UDP-based transport protocol. In TAO this protocol is only partially implemented and as such, has some restrictions that will be covered later in this section. The DIOP implementation uses connectionless UDP sockets, and therefore is intended for use as a lower-overhead protocol for certain classes of applications.

The original motivation for this protocol came from applications that used only `oneway` operations.

14.9.1 Loading the Protocol

The DIOP factory and its components are included in the `TAO_Strategies` library. To directly link the strategies library to your application, add `TAO_Strategies` to the list of libraries in your makefile or project settings. DIOP, along with the other protocols included in the strategies library, may be made available by default by including the following TAO-specific header in your application's main source file:

```
#include <tao/Strategies/advanced_resource.h>
```

By including this file, you make all of the protocols supplied in the strategies library available to your application.

To limit the available protocols, specify the DIOP protocol factory name, `DIOP_Factory`, as an argument to the `-ORBProtocolFactory` option when initializing the resource factory. For example:

```
static Advanced_Resource_Factory "-ORBProtocolFactory DIOP_Factory"
```

Either the default Resource Factory or the Advanced Resource Factory may be used when the strategies library is statically linked.

You can also load the DIOP protocol dynamically through the service configurator, making it unnecessary to include the TAO-specific header, or explicitly link the strategies library. However, the `TAO_Strategies` library must exist in your platform's library search path (e.g., `LD_LIRBARY_PATH` on many Unix systems). You also must use the following `dynamic` directives in your `svc.conf` file (see 16.3.2 for more information):

```
dynamic DIOP_Factory Service_Object*
   TAO_Strategies:_make_TAO_DIOP_Protocol_Factory () ""

dynamic Advanced_Resource_Factory Service_Object*
   TAO_Strategies:_make_TAO_Advanced_Resource_Factory ()
   "-ORBProtocolFactory DIOP_Factory"
```

Once you have the DIOP protocol factory loaded, the ORB will be able to communicate via this protocol.

14.9.2 Address Definition

For TAO servers to generate DIOP profiles for inclusion in object references, an endpoint must be explicitly declared using the `-ORBListenEndpoints` option. See 17.13.14 for more information on this option.

Endpoints for DIOP are composed of the prefix `diop://` followed by a host and port combination, similar to IIOP endpoints. An example of a DIOP endpoint is:

```
-ORBListenEndpoints diop://example.ociweb.com:12345
```

14.9.3 Notes and Restrictions

Keep the following points in mind when using the DIOP protocol:

- There are no connections, and therefore no state; requests from different clients all arrive at the *same* socket!
- The thread-per-connection concurrency model is supported.
- Only `oneway` operation invocations are supported.

DIOP was developed for applications satisfying the following assumptions:

- UDP communications are nearly 100 percent reliable (e.g., IP over ATM). Even under less reliable conditions, DIOP can be used by restricting the application to use `oneway` operations for both requests and responses in combination with application-level time outs.
- TCP is inappropriate, due to its sluggishness on sudden disconnections (e.g., it must be possible to “hot swap” CPU cards without impacting an ORB’s communications with a particular CPU). For this reason, no state is kept on the client side of the DIOP protocol.
- No GIOP message (and therefore no IDL signature) has a size greater than `ACE_MAX_DGRAM_SIZE` (4 kilobytes). Thus, no data sent via DIOP can be larger than 4 kilobytes.

  Note that support for fragmented messages, as specified in GIOP 1.2, may alleviate this restriction, and may be added to a future DIOP implementation.

14.10 SSLIOP

SSLIOP stands for the Secure Sockets Layer (SSL) Inter-ORB Protocol. This protocol is defined by the OMG as part of the CORBA Security Service specification. SSLIOP uses GIOP as a messaging protocol and SSL as the transport protocol. It is a drop-in replacement for IIOP, providing secure communication between hosts.

TAO's SSLIOP pluggable protocol implementation supports both the standard IIOP transport protocol and the secure IIOP over SSL transport protocol. No changes were made to the core TAO source code to provide SSL support, nor does TAO contain any security-related hooks. Details about TAO’s SSLIOP and other security-related issues are covered in Chapter 27, particularly 27.3.

14.10.1 Loading the Protocol

The SSLIOP factory and its components are included in a separate `TAO_SSLIOP` library. See 27.5 for details on building and using this library. Specifically, 27.6.2 covers the various initialization options that are accepted by the SSLIOP Factory.

14.10.2 Address Definition

SSLIOP endpoints are nearly identical to IIOP endpoints. The same prefix, `iiop://` is used, along with host name and port number. The only difference is that SSLIOP takes a separate per-endpoint option, `ssl_port`, that allows explicit specification of both the secure and insecure endpoints. The `portspan` option of the IIOP connector is not used by SSLIOP. Here is an example of an endpoint supplying both an insecure and a secure port:

```
-ORBListenEndpoints iiop://example.ociweb.com:12000/ssl_port=12001
```

14.10.3 Hostname in IOR Option

The `hostname_in_ior` option, which was described earlier in 14.6.3.3 as an IIOP option, is also available for SSLIOP endpoints.

14.10.4 Portspan Option

The portspan option, which was described earlier in 14.6.3.1 as an IIOP option, is also available for SSLIOP endpoints. When used in conjunction with ssl_port as shown below:

```
-ORBListenEndpoints
iiop://example.ociweb.com:12000/ssl_port=12001&portspan=5
```

it instructs the server to open an endpoint on the specified interface on any port from 12001 and 12006.

14.10.5 Reuse Address Option

The `reuse_addr` option, which was described earlier in 14.6.3.3 as an IIOP option, is also available for SSLIOP endpoints.

14.11 MIOP/UIPMC

MIOP stands for Multicast Inter-ORB Protocol and UIPMC stands for Unreliable IP Multicast. UIPMC underlies the OMG's Multicast Inter-ORB Protocol, defined in the MIOP specification. (OMG Document ptc/01-11-08 is the Final Adopted Specification.) The current MIOP version is 1.0. The MIOP protocol wraps a GIOP request in a MIOP *Packet* and transmits it via an underlying connectionless transport protocol (e.g., UDP/IP). An MIOP Packet is defined as the MIOP *Packet Header* information, which represents the state information for a single MIOP Packet, as well as the raw GIOP data (body) contained in the rest of the MIOP Packet. An MIOP Packet will be sent and later reassembled on the receiving side. MIOP Packets are the atomic pieces that comprise a *Packet Collection*. A Packet Collection is comprised of one or more MIOP Packets and is defined as a complete, packaged, GIOP request message or request message fragment. Only GIOP request messages and associated request message fragments are allowed in an MIOP Packet in a Packet Collection.

Note *MIOP packet reassembly is not yet supported in TAO. Thus, the maximum request size that can be transmitted via MIOP/UIPMC is currently limited to roughly 5-6k bytes depending upon the platform.*

TAO's MIOP support enables servants to receive requests sent to multicast addresses. A GroupId that identifies the multicast group is created and associated with one or more servants. The UIPMC pluggable protocol is used to send and receive multicast requests. MIOP/UIPMC is currently usable only for `oneway` operations. Servers (receivers) join an IP Multicast group and listen on the same *group id* for requests.

See `$TAO_ROOT/orbsvcs/tests/Miop/McastHello` for an example that uses MIOP/UIPMC.

14.11.1 Loading the Protocol

The UIPMC factory and its components are included in the `TAO_PortableGroup` library. You can load the UIPMC protocol dynamically through the service configurator. The `TAO_PortableGroup` library must exist in your platform's library search path (e.g.,

LD_LIRBARY_PATH on many Unix systems). You also must use the following dynamic directives in your svc.conf file (see 16.3.2 for more information):

```
dynamic UIPMC_Factory Service_Object*
TAO_PortableGroup:_make_TAO_UIPMC_Protocol_Factory() ""

static Resource_Factory "-ORBProtocolFactory IIOP_Factory -ORBProtocolFactory
UIPMC_Factory"
```

Once you have the UIPMC protocol factory loaded, the ORB will be able to communicate via this protocol. The UIPMC protocol factory must be loaded and configured on both the client and server sides. In addition, the server side must add the following directive to load the Portable Group Loader service object.

```
dynamic PortableGroup_Loader Service_Object*
TAO_PortableGroup:_make_TAO_PortableGroup_Loader() ""
```

14.11.2 Address Definition

The MIOP specification provides the following corbaloc Object URL syntax for specifying an MIOP profile:

```
<corbaloc> = "corbaloc:"<obj_addr_list>["/"<key_string>]
<obj_addr_list> = [<obj_addr> ","]* <obj_addr>
<obj_addr> = <prot_addr>
<prot_addr> = <iiop_prot_addr> | <miop_prot_addr>
<miop_prot_addr> = <miop_prot_token><miop_addr>
<miop_prot_token> = "miop"
<iiop_prot_token> = "iiop"
<miop_addr> = <version><group_addr>[;<group_iiop>]
<version> = <major> "." <minor> "@" | empty_string
<group_addr> = <group_id>"/"<ip_multicast_addr>
<group_iiop> = <iiop_prot_token>":"<version> <hostname>":"\
               <port> "/" <objecy_key>
<ip_multicast_addr> = <classD_IP_address> | <IPv6_address> ":" <port>
<classD_IP_address> = "224.0.0.0" - "239.255.255.255"
<port> = number (default to be defined)
<group_id> = <group component version>"-"<group_domain_id>"-"
             <object_group_id>["-"<object group reference version>]
<group component version> = <major> "." <minor>
<group_domain_id> = string
<object_group_id> = unsigned long long
<object group reference version> = unsigned long
<major> = number (default 1)
<minor> = number (default 0)
```

An example of a valid MIOP corbaloc Object URL (taken from the example in `$TAO_ROOT/orbsvcs/tests/Miop/McastHello`) is:

```
corbaloc:miop:1.0@1.0-TestDomain-1/225.1.1.225:1234;
```

Here, the group id is `1.0-TestDomain-1` and the IP multicast address is 225.1.1.225 at port 1234. (Any valid multicast address will do.) Multiple servers using the MIOP/UIPMC protocol can share this same group id and address; all the servers will be able to receive client requests directed to that group.

The `-ORBListenEndpoints` option is not used to specify MIOP profiles as with TAO's other pluggable protocols. Instead, servers should generate an object reference from the group's corbaloc Object URL (using `CORBA::ORB::string_to_object()`), then use the POA's new `create_id_for_reference()` operation to generate a `PortableServer::ObjectId` from a group's object reference. This object id is then associated with servants via the POA's usual `activate_object_with_id()` or similar operation. The following server code fragment shows how the above steps can be accomplished:

```
// ORB and POA initialization code not shown.

// Set up the object group's corbaloc Object URL.
CORBA::String_var group_url =
  CORBA::string_dup ("corbaloc:miop:1.0@1.0-TestDomain-1/225.1.1.225:1234");

// Generate a group object reference.
CORBA::Object_var group_reference = orb->string_to_object (group_url.in());

// Create an Object Id for the group reference.
PortableServer::ObjectId_var id =
  poa->create_id_for_reference (group_reference.in ()

// Create and activate a servant.
PortableServer::Servant_var<Messenger_i> servant = new Messenger_i();
poa->activate_object_with_id (id.in(), servant.in());

// Activate the POAManager and accept client requests as usual.
```

Clients can simply convert the group's corbaloc Object URL to an object reference using `CORBA::ORB::string_to_object()`, then invoke oneway operations on the object reference as usual. Since only oneway operations are

supported, clients cannot use a normal `_narrow()` function to narrow the object reference since doing so may cause a two-way `_is_a()` operation to be invoked; instead, clients should use `_unchecked_narrow()`. The following client code fragment shows the steps we have just described:

```
// ORB initialization code not shown.

// Set up the object group's corbaloc Object URL.
CORBA::String_var group_url =
  CORBA::string_dup ("corbaloc:miop:1.0@1.0-TestDomain-1/225.1.1.225:1234");

// Generate a group object reference.
CORBA::Object_var group_reference = orb->string_to_object (group_url.in());

// Narrow the object reference (use _unchecked_narrow() since _narrow() may
// result in a two-way _is_a() operation invocation).

Messenger_var messenger = Messenger::_unchecked_narrow (group_reference.in ());

if (CORBA::is_nil (messenger.in())) {
  std::cerr << "Could not narrow group reference." << std::endl;
  ACE_OS::exit(-1);
}

// Invoke oneway operations on the group reference.
```

14.11.3 Notes and Restrictions

Keep the following points in mind when using MIOP:

- There are no connections, and therefore no state; requests from different clients are all sent to the *same* multicast address and will be received by all servers that have joined the multicast group!
- The thread-per-connection concurrency model is supported.
- Only `oneway` operation invocations are supported.
- IP multicast communications are unreliable, therefore MIOP/UIPMC should not be used where reliable transmission of requests/data is required.
- Since TAO's support of MIOP does not yet support message fragmentation and MIOP Packet reassembly, request sizes are limited to 5-6 kilobytes, depending upon the platform.

14.12 HTIOP

HTIOP stands for HTTP Tunneling Inter-ORB Protocol. It is a TAO-specific protocol that allows inter-ORB communications to be transmitted across a firewall. It does this by layering GIOP messages over HTTP packets.

HTIOP support in TAO consists of two libraries:

- The `ACE_HTBP` library, found in `$ACE_ROOT/protocols/ace/HTBP`, implements the low-level HTBP protocol. This protocol allows for communications to take place through a firewall between "inside" and "outside" peers. HTBP supports both "direct" and "indirect" connections through the firewall. The default is indirect, which requires the use of an HTTP proxy. HTBP can be used independently of TAO.
- The `TAO_HTIOP` library, found in `$TAO_ROOT/orbsvcs/orbsvcs/HTIOP`, implements HTIOP on top of HTBP.

HTIOP is an *asymmetric protocol*. This means that applications inside the firewall must be configured differently than applications outside the firewall. Inside peers are the only ones that may initiate connections.

Note *If a peer-to-peer relationship is desired wherein CORBA invocations flow in either direction, then bi-directional GIOP must be used. See 6.4 for more information on bi-directional GIOP.*

One of the challenges imposed by the use of an HTTP proxy is that, if a connection is idle for long enough, it may be closed. This means the inside peer must establish a new connection by sending another request. Since the outside peer cannot open a connection, it must queue messages destined to the inside peer until the inside peer opens a new connection. Once a connection has been established, all the queued messages are sent.

A second challenge is that, while a proxy will open multiple TCP/IP connections to a server (the outside peer), when faced with multiple simultaneous HTTP requests, it will reuse those connections at will to forward any subsequent requests. This means that a socket is associated with a particular HTIOP session for only one HTTP request/reply cycle.

Apart from loading the protocol and declaring an endpoint, no modifications to application code are required to use HTIOP.

HTBP isolates HTIOP from differences in HTTP proxies by defining a filter class that encapsulates the particular characteristics of the proxy that is being used and that is responsible for marshaling and demarshaling binary data. At the time of this writing, HTBP has a single filter class that works with the *Squid* HTTP proxy. More information on the Squid web proxy cache can be found via <http://www.theaceorb.com/references/>. This filter may also be used as a null filter when the inside and outside peers are directly connected (i.e., a proxy is not being used), which is mainly useful during testing.

14.12.1 Loading the Protocol

HTIOP is loaded via the service configurator. Since HTIOP is an asymmetric protocol, the configurations are different for the server and client sides. Since the server resides outside the firewall, its configuration is quite straightforward. Here is an example of a server-side configuration:

```
dynamic HTIOP_Factory Service_Object *
  TAO_HTIOP:_make_TAO_HTIOP_Protocol_Factory () ""

static Resource_Factory "-ORBProtocolFactory HTIOP_Factory"
```

The `HTIOP_Factory` protocol factory accepts the following initialization parameters:

- `-config <filename>`—Specifies the name of a text file that contains an HTBP-specific configuration. See Table 14-2 for a list of possible HTBP configuration parameters. The `-config` parameter is optional.
- `-env_persist <filename>`—Specifies the name of a memory mapped file that contains a previously saved HTBP configuration, or to which a new HTBP configuration will be saved. If both `-config` and `-env_persist` are specified, the memory mapped file will be loaded first, then the text file will be interpreted. The new configuration will then be persisted in the memory mapped file.
- `-win32_reg`—If this parameter is provided, the HTBP configuration will be saved to the Windows registry (in the HTBP section) rather than to a memory mapped file. This feature is available only on Windows.
- `-inside (`**`-1`**`|0|1)`—Explicitly declare whether the peer's endpoint is inside or outside the firewall. The default setting of `-1` causes the

HTIOP_Factory to use the ACE_HTBP_Environment's proxy_host setting to determine if it the peer is inside or outside the firewall. An explicit setting of 1 means the peer's endpoint is inside the firewall; an explicit setting of 0 means it is outside the firewall.

You only need to explicitly specific if the peer is inside or outside the firewall when you are testing without a real HTTP proxy. As stated previously, HTIOP is an asymmetric protocol: only peers inside the firewall can initiate connections; peers outside the firewall cannot initiate connections to inside peers.

- -proxy_port <port>—Specifies the host name of the HTTP proxy through which all requests will be sent.
- -proxy_host <hostname>—Specifies the port number on which the HTTP proxy is listening and through which all requests will be sent

HTBP is configured independently of the HTIOP protocol factory. The HTBP configuration contains a single [htbp] section that contains a list of name/value pairs to configure HTBP. Table 14-2 lists the configuration parameters HTBP accepts:

Table 14-2 HTBP Configuration Parameters

proxy_host	The host name of the HTTP proxy through which all requests will be sent. This value overrides the value set by the corresponding protocol factory option.
proxy_port	The port number on which the HTTP proxy is listening and through which all requests will be sent. This value overrides the value set by the corresponding protocol factory option.

In the following example, we show the configuration of an HTIOP client in which a proxy is not being used. Note that the client must explicitly specify that it is an inside peer.

```
dynamic HTIOP_Factory Service_Object *
  TAO_HTIOP:_make_TAO_HTIOP_Protocol_Factory () "-inside 1"
static Resource_Factory "-ORBProtocolFactory HTIOP_Factory"
```

In this example, we show a client configuration that is using a proxy. The HTBP configuration parameters are specified in a separate configuration file called HTBP_Config.txt.

```
dynamic HTIOP_Factory Service_Object *
  TAO_HTIOP:_make_TAO_HTIOP_Protocol_Factory () "-config HTBP_Config.txt"
```

```
static Resource_Factory "-ORBProtocolFactory HTIOP_Factory"
```

Here we show possible contents of the `HTBP_Config.txt` file.

```
[htbp]
proxy_port=3128
proxy_host=proxy.acme.com
```

14.12.2 Address Definition

For TAO servers to generate HTIOP profiles for inclusion in object references, an endpoint must be explicitly declared using the `-ORBListenEndpoints` option. See 17.13.43 for more information on this option.

HTIOP endpoints are composed of the prefix `htiop://`, followed by the host name and port number. For example:

```
-ORBListenEndpoints htiop://malory.acme.com:12345
```

14.12.3 HTIOP Options

14.12.3.1 hostname_in_ior

This option is used to explicitly specify the host name used in IORs. For example:

```
-ORBListenEndpoints htiop://malory.acme.com:12345/hostname_in_ior=www.acme.com
```

See 14.6.3.2 for more information on the `hostname_in_ior` option.

14.13 SCIOP

SCIOP stands for SCTP Inter-ORB Protocol. It layers the TAO pluggable protocol framework atop the Stream Control Transmission Protocol (SCTP).

SCTP is defined in Internet Engineering Task Force (IETF) Requests for Comments (RFC) 2960 as the transport layer to carry signaling messages over IP networks. According to the RFC, “SCTP is a reliable transport protocol

operating on top of a connectionless packet network such as IP. It offers the following services to its users:

- acknowledged error-free non-duplicated transfer of user data,
- data fragmentation to conform to discovered path MTU size,
- sequenced delivery of user messages within multiple streams, with an option for order-of-arrival delivery of individual user messages,
- optional bundling of multiple user messages into a single SCTP packet, and
- network-level fault tolerance through supporting of multi-homing at either or both ends of an association.

The design of SCTP includes appropriate congestion avoidance behavior and resistance to flooding and masquerade attacks."

TAO's implementation of SCIOP depends upon an external implementation of the SCTP APIs. Currently, the only supported SCTP implementation is the Linux Kernel Stream Control Transmission Protocol (`lksctp`). `lksctp` is available in the Linux kernel source tree in versions 2.5.36 and later. For this reason, TAO's SCIOP support is currently limited to the Linux platform.

14.13.1 SCTP Association

SCTP provides a means for each SCTP endpoint to provide the other endpoint (during association startup) with a list of transport addresses (i.e., multiple IP addresses in combination with an SCTP port) through which that endpoint can be reached and from which it will originate SCTP packets. The association spans transfers over all of the possible source/destination combinations that can be generated from each endpoint's lists. Each GIOP connection is mapped to a single SCTP association. At startup, a *primary* path is selected, which is indicated in the IOR profile. This primary path is used for normal communication. Using SCIOP, a TAO application is able to access the fault tolerance capabilities of the SCTP protocol.

14.13.2 Loading the Protocol

The SCIOP implementation is included in the `TAO_Strategies` library. It can be loaded and configured either dynamically or statically.

SCIOP can be dynamically loaded via the service configurator. An example configuration file is shown below:

```
dynamic SCIOP_Factory Service_Object*
  TAO_Strategies:_make_TAO_SCIOP_Protocol_Factory () ""
dynamic Advanced_Resource_Factory Service_Object*
  TAO_Strategies:_make_TAO_Advanced_Resource_Factory ()
  "-ORBProtocolFactory SCIOP_Factory"
```

See 14.7.1 for information on statically loading the `TAO_Strategies` library.

14.13.3 Address Definition

SCIOP addresses must encapsulate the SCTP association information. Multiple host names separated by a '+' are allowed for a single endpoint. As with TAO's other pluggable protocols, the `-ORBListenEndpoints` option is used to create an SCIOP endpoint. SCIOP endpoints are composed of the prefix `sciop://`, followed by one or more host names and a port number. Some sample SCIOP endpoint specifications are shown below:

```
-ORBListenEndpoints sciop://malory:12345
-ORBListenEndpoints sciop://malory+arthur+gawain:12345
```

14.13.3.1 hostname_in_ior

This option is used to explicitly specify the host name used in IORs. For example, the following option:

```
-ORBListenEndpoints sciop://malory+arthur+gawain:12345/hostname_in_ior=host4
```

will cause the ORB to encode `host4` in the SCIOP profiles of the IORs that it generates (i.e., `host4` will be the *primary* host for the service).

See 14.6.3.2 for more information on the `hostname_in_ior` option.

14.13.3.2 portspan

The portspan option, which was described earlier in 14.6.3.1 as an IIOP option, is also available for SCIOP endpoints.

14.14 ZIOP

The ZIOP protocol is based on the proposed GIOP Compression specification (OMG Document mars/08-12-20). This specification defines a compression mechanism for GIOP. The ZIOP protocol in TAO is an experimental proof-of-concept implementation of this proposed standard. It utilizes TCP/IP using the open source zlib compression library.

The ZIOP protocol is not built unless TAO is built with `TAO_HAS_ZIOP` explicitly set to 1. Set this macro at the top of your config.h, rebuild TAO, and you are ready to experiment with this protocol. Once built with ZIOP, TAO's use of compression is controlled by the ZIOP policies defined in 14.14.1.

See the test in `$TAO_ROOT/tests/ZIOP` for an example that uses this protocol.

14.14.1 ZIOP Policies

ZIOP defines four policies that can be used to configure the ORB's use of the protocol. These are set on the client and server as described in 6.3.1.

14.14.1.1 Compression Enabling Policy

This policy takes a boolean value and must be enabled for both the client and server for the ORB to use compression between the two. TAO's default value of this policy is to disable compression.

14.14.1.2 Compressor Level List Policy

This policy takes as its value a sequence of structures that each take a pair of CompressionId/CompressionLevel values in order of preference. Each CompressorId identifies a unique compression algorithm or library. The client or server will go through its compressor level list until it finds a CompressorId that the other supports. This is then the selected compression algorithm and the lowest specified level between the two is used. TAO's default value of this policy, when no list is specified, is to support ZLIB compression at level 0.

14.14.1.3 Compression Low Value Policy

This policy takes a long value that specifies a message length. The ORB will not compress any messages with application data length fewer than the

specified length. TAO’s default value of this policy is 0, meaning that the ORB will attempt to compress all messages with application data.

14.14.1.4 Compression Min Ratio Policy

This policy takes a long value that is the minimum compression ratio that must occur before a compressed message is used. TAO does not currently support this option and all compressed messages are sent, regardless of compression ratio.

14.15 COIOP

COIOP stands for Collocated-Only Inter-ORB Protocol. This protocol is intended for use when TAO is used in systems with no network interfaces enabled. In this use case, TAO should be built with COIOP enabled via `TAO_HAS_COIOP` (this protocol is not built unless explicitly enabled) and other protocols should be disabled. For example, the following could be added to the top of your `config.h`:

```
#define TAO_HAS_IIOP 0
#define TAO_HAS_UIOP 0
#define TAO_HAS_DIOP 0
#define TAO_HAS_SHMIOP 0
#define TAO_HAS_COIOP 1
```

This would build TAO with COIOP support enabled and without IIOP, UIOP, DIOP, or SHMIOP support. Because COIOP is the only protocol available, no factory configuration is necessary. Simply link your application with the `TAO_Strategies` library and COIOP will be used for all CORBA invocations.

If the CORBA object is located in the same process as the client invocation, then it proceeds using the TAO’s collocation optimizations. If the CORBA object is located in another process or collocation optimization is disabled, a `CORBA::Transient` exception is thrown.

14.16 Combining Protocols

You may use multiple protocols simultaneously with TAO. Using the `-ORBListenEndpoints` option, you can instruct the ORB to listen on multiple endpoints for different protocols. For example, to instruct the IIOP protocol to listen on an ephemeral port on `192.168.1.6`, and the SHMIOP protocol to listen on port `12345`, use the following command:

```
-ORBListenEndpoints iiop://192.168.1.6 -ORBListenEndpoints shmiop://12345
```

or alternatively:

```
-ORBListenEndpoints iiop://192.168.1.6;shmiop://12345
```

Creating multi-protocol endpoints, as in the example above, requires the service configurator to load and initialize the protocol-specific factory objects. The IIOP factory is initialized by default unless another protocol is explicitly loaded via the service configurator. Therefore, the example above requires both IIOP and SHMIOP factories to be statically or dynamically initialized. The dynamic directives for this are shown below:

```
dynamic IIOP_Factory Service_Object*
  TAO_Strategies:_make_TAO_IIOP_Protocol_Factory () ""
dynamic SHMIOP_Factory Service_Object*
  TAO_Strategies:_make_TAO_SHMIOP_Protocol_Factory () ""

dynamic Advanced_Resource_Factory Service_Object*
  TAO_Strategies:_make_TAO_Advanced_Resource_Factory ()
    "-ORBProtocolFactory IIOP_Factory -ORBProtocolFactory SHMIOP_Factory"
```

When a multi-endpoint server generates an IOR, that IOR will contain a profile for each protocol in the order in which they appear on the command line. Without using the capabilities of real-time CORBA, there is no standard way for the client to specify at run-time which protocol to use for a particular invocation. A TAO server and a TAO client, however, can leverage the fact that the client connection code iterates over profiles in the IOR in the order in which they appear. Thus, for the endpoints specified above, the client ORB first tries to establish a connection and send a request using IIOP and, if that fails, tries SHMIOP.

Reversing the order of the endpoints on the server's command line causes the client ORB to first attempt to use SHMIOP, then IIOP:

```
-ORBListenEndpoints shmiop://12345 -ORBListenEndpoints iiop://192.168.1.6
```

The standard and more portable way to accomplish this is through the use of the real-time-CORBA protocol selection policies. An example of this can be found in `$TAO_ROOT/tests/RTCORBA/Client_Protocol` and in 8.3.12.

14.17 TAO and IPv6

TAO's support for IPv6 has progressed steadily through the last few releases. When TAO is built to support IPv6, each of this section's protocols that uses IP addresses in its endpoint can also use IPv6 addresses. By default, TAO's IPv6 support is not enabled.

14.17.1 Building TAO with IPv6 Support

The easiest way to enable IPv6 support is to add the `ipv6` feature to your `default.features` file:

```
// $ACE_ROOT/bin/MakeProjectCreator/default.features
ipv6=1
```

After making this modification, you should regenerate your build files using MPC. As an alternative to changing `default.features`, you could add `ACE_HAS_IPV6` to the top of your `$ACE_ROOT/ace/config.h` file:

```
#define ACE_HAS_IPV6 1

// remainder of config.h...
```

After either of these you will need to rebuild your TAO and your applications. See Appendix A for further details about configuring ACE/TAO builds.

14.17.2 Using IPv6 with TAO

In order to distinguish between IPv4 and IPv6 address, IPv6 addresses must be placed within square brackets. Here is an example of a Naming Service server

specifying an IPv6 endpoint and a client using that endpoint to construct a corbaloc to find that service:

```
tao_cosnaming -ORBListenEndpoints iiop://[01ef::1]:1234
tao_nslist -ORBInitRef NameService=
```

There are a number of IPv6-related ORB_init() options that become available when TAO is built with IPv6 enabled. See 17.9 for details on these IPv6-related options.

14.18 Developing Pluggable Protocols

This section is intended as a brief introduction to how you can develop your own pluggable protocols for TAO. The best source for up to date information about the APIs required are the source code of the various pluggable protocols that are included in the TAO distribution.

14.18.1 Background

As with the use of the existing TAO protocols, the main motivator for using custom protocols is typically higher performance, although reduced bandwidth consumption, security, or custom network hardware may also motivate their use. If the requirements of your application are such that IIOP or one of the other protocols provided with TAO is not sufficient, then you can take full advantage of TAO's open pluggable-protocol framework to architect your own transport for seamless use with the rest of the TAO ORB.

Note *What has been historically referred to as a "pluggable protocol" in TAO is really a pluggable transport. TAO does not currently support pluggable messaging (i.e., only GIOP is supported), nor is it possible to choose alternative marshaling schemes (i.e., only CDR is supported). However, for historical reasons, pluggable transports in TAO are referred to as pluggable protocols.*

Generally, as a CORBA application programmer, you need not be concerned with the internal workings of the ORB, beyond the configuration options provided to you by the ORB vendor. When developing pluggable protocols however, you need to be familiar with the patterns, mechanisms, and APIs

used by the ORB internal code, so that you can quickly and effectively add your own transport. After covering the requirements that any transport must satisfy to be used for GIOP messaging, this section outlines the basic steps you need to take to develop your custom protocol and integrate it with TAO.

14.18.2 Pluggable Protocol Requirements

This section discusses the preconditions that must be met by any candidate transport protocol considered for use with TAO.

14.18.2.1 Transport Behavior

Any transport protocol used in inter-ORB communication must be both connection-oriented and reliable. Basic TCP/IP is a representative example, because it fulfills both of these requirements. UDP is an example of a protocol that does not fulfill these minimum requirements, being neither connection-oriented nor reliable. Of course, a connection-oriented, reliable layer could be built on top of UDP.

Additionally, the candidate protocol must have a way to multiplex I/O events, using either the Unix `select()` system call or its Win32 equivalent, `WaitForMultipleObjects()`. Such event notification may be supplied through an alternate channel if the native transport mechanism does not support it. For example, the shared-memory transport protocol, SHMIOP, uses a socket on the loopback interface to notify the peer connection that data is available for reading. As an extreme example, it may be possible to use some other stimulus, such as signals or semaphores, for notification, but integrating such a mechanism with TAO is difficult and beyond the scope of this book.

14.18.2.2 Identifier Tags

Transport protocols are identified with a magic number, which is defined by the OMG. Throughout the following class descriptions, all of the constructors defined by the TAO pluggable-protocol framework take a `CORBA::ULong` value as an argument. This value represents the protocol's tag, which is incorporated into the profile section of an IOR, and therefore must be unique across all CORBA vendors. To obtain a tag assignment when constructing a new protocol, it is necessary to register the protocol with the OMG. Any value may be used as the tag, but failure to register it leaves your code liable to

inconsistent behavior when interacting with clients or servers produced by others.

14.18.3 Details of the Pluggable-Protocol Framework

There are a small number of classes that you need to define and implement to enable your protocol to work with the TAO pluggable-protocol framework. This framework allows you to add new protocols without changing TAO itself by providing an interface between TAO and your custom protocol. As long as your protocol classes conform to expectations and patterns of the framework, the task of producing a TAO pluggable protocol can be as simple as following a recipe. Using existing protocols as models for your protocol can make your job much easier.

The TAO pluggable-protocol framework provides abstract base classes from which your protocol-specific classes must derive. Table 14-3 shows the framework components that must be implemented for each protocol, the base classes that define each interface, and the header files in which the base classes are defined.

Table 14-3 Pluggable-Protocol Classes and Files

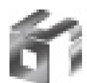
Component	Base Class	Base-Class Header File
Acceptor	`TAO_Acceptor`	`$TAO_ROOT/tao/Transport_Acceptor.h`
Connector	`TAO_Connector`	`$TAO_ROOT/tao/Transport_Connector.h`
Connection Handler	`TAO_Connection_Handler`	`$TAO_ROOT/tao/Connection_Handler.h`
Endpoint	`TAO_Endpoint`	`$TAO_ROOT/tao/Endpoint.h`
Profile	`TAO_Profile`	`$TAO_ROOT/tao/Profile.h`
Protocol Factory	`TAO_Protocol_Factory`	`$TAO_ROOT/tao/Protocol_Factory.h`
Transport	`TAO_Transport`	`$TAO_ROOT/tao/Transport.h`

The source code for the derived types is typically divided into several files, which are usually named for the protocol and component expressed within. For instance, IIOP-related classes are in files such as `IIOP_Acceptor.h` and `IIOP_Connector.h`, located in the `$TAO_ROOT/tao` directory. Other protocol implementations, such as UIOP and SHMIOP, are located in the `$TAO_ROOT/tao/Strategies` directory. SSLIOP implementations are located in the `$TAO_ROOT/orbsvcs/orbsvcs/SSLIOP` directory.

The rest of this section is devoted to describing the interfaces that must be implemented by any pluggable protocol. As is typical with any complex programming exercise, simply fulfilling the interface is not sufficient for providing a robust implementation.

There are seven fundamental classes included in a TAO pluggable protocol. These classes are described in general below and in detail in the following sections:

- **The Acceptor** provides the passive endpoint in servers that opens new connections as requested by clients. Within the context of the pluggable-protocol framework, there are several distinct acceptor classes:
  - The abstract base-class `TAO_Acceptor` serves as the interface supplied by the framework. In the following discussion it is referred to as the "the TAO acceptor."
  - A protocol-specific acceptor class that you provide to implement this interface. It is referred to as the "pluggable acceptor," or simply, "your acceptor." The name "`TAO_*_Acceptor`" is used to refer to any protocol-specific acceptor type, of which there are several.
  - The class that is actually responsible for accepting new connections. This class is commonly the `ACE_Strategy_Acceptor` template class. It is referred to as the "base acceptor."
  - In many protocols, there is another acceptor class, which is one of the template parameters to the `ACE_Strategy_Acceptor` class, and is referred to as the "peer acceptor." This acceptor provides an extra level of indirection that leverages the existing ACE framework classes that assist in implementing the protocol-specific behavior. Examples of these classes include `ACE_SOCK_Acceptor` (for IIOP) and `ACE_MEM_Acceptor` (for SHMIOP). Later sections show how to leverage the existing ACE framework classes to support your protocol.
- **The Connector** provides an active endpoint for creating connections. Within the context of the pluggable-protocol framework, there are several distinct connector classes:
  - The abstract base-class `TAO_Connector`, which serves as the interface supplied by the framework. In the following discussion it is referred to as "the TAO connector."



- A protocol-specific connector class that you provide to implement this interface as dictated by the TAO-connector base class. It is referred to either as the "pluggable connector" or "your connector." The name "`TAO_*_Connector`" is used to refer to any protocol-specific connector type, of which there are several.
- The class that is responsible for establishing new connections. This class is commonly the `ACE_Strategy_Connector` template class, and is referred to as "the base connector."
- In many protocols, there is another acceptor class, which is one of the template parameters to the `ACE_Strategy_Connector` class, and is referred to as the "peer connector." This connector provides an extra level of indirection that leverages the existing ACE framework classes that assist in implementing the protocol-specific behavior. Examples of these classes include `ACE_SOCK_Connector` (for IIOP) and `ACE_MEM_Connector` (for SHMIOP). Later sections show how to leverage the existing ACE framework classes to support your protocol.

• **The Connection Handler** manages the interface between TAO and the actual communication protocol as managed by the transport abstraction. Your protocol implementation must include a subclass of the base-class `TAO_Connection_Handler`. There is only one connection-handler class that is used by both clients and servers.

• **The Endpoint** represents a single point of contact for the server, and is the smallest unit of addressing information necessary for a client to connect to a server. Your protocol implementation must include a subclass of the `TAO_Endpoint` base class and must contain the addressing information specific to your custom protocol.

• **The Profile** is used to communicate protocol information between the various objects within an application. This information is encoded within an IOR that is used to communicate endpoint information from servers to clients. Your protocol implementation must include a subclass of the `TAO_Profile` base class. It must also contain the necessary protocol-specific information to be encoded into an IOR, exported to clients, decoded by clients, and used to contact the server via the custom protocol.

- **The Protocol Factory** is responsible for constructing acceptors and connectors. Your protocol implementation must include a subclass of the `TAO_Protocol_Factory` base class.
- **The Transport** works in conjunction with the connection handler to provide an interface to the higher-level messaging layer of TAO, and a level of indirection down to the send and receive mechanics of the concrete protocol.

To learn more about these objects and their interfaces, refer to the header file comments from the framework's base classes. Refer to the concrete protocol implementations of each them for sample implementations.

CHAPTER 15

Multithreading with TAO

15.1 Introduction

TAO is used in many application domains, across a wide variety of platforms and operating systems, and often subject to wide-ranging performance requirements. Often, overall throughput of an application can be improved by using multithreaded programming techniques. For example, multithreading can be useful in the following situations:

- The application is characterized by long-running operations that are largely independent of one another.
- The application must support a large number of clients whose requests must be executed concurrently.
- Data and resource sharing among operations is minimal and easily identifiable.
- Operations can be executed in parallel.
- Servant implementations are thread-safe (reentrant) (i.e., can be safely executed concurrently by different threads) and appropriate

synchronization mechanisms (e.g., mutex locks) are used to protect critical regions of code from concurrent execution.

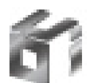
- Legacy and third-party code and libraries used by servant implementations are also thread-safe; or if they are not they are protected from concurrent execution.

Note *This chapter assumes the reader is somewhat familiar with the fundamental concepts of multithreaded programming. For more information, see Chapter 5 of C++ Network Programming: Volume 1 (C++NPv1) and Chapter 2 of Threads Primer.*

15.1.1 Road Map

This chapter describes several aspects of multithreading for TAO applications, including application programming interfaces (both CORBA-compliant and TAO-specific), policies, configuration options, and design choices. While the chapter is intended to be read in its entirety, you may find benefit in reading certain sections independently.

- Section 15.2, “Overview of Client/Server Roles in CORBA,”defines the client and server roles in CORBA and reviews the stages of request invocation by clients and request processing by servers.
- Section 15.3, “Multithreading in the Server,” describes configuration and behavior for a thread in the server role.
    - Section 15.3.1, “Event Handling,” and Section 15.3.3, “Shutdown and Destruction,” describe ORB operations for providing one or more threads to the ORB, shutting down the ORB, and destroying the ORB. TAO’s ORB supports the thread-related operations defined by the CORBA specification and provides certain TAO-specific extensions.
    - Section 15.3.4, “Request Processing,” describes how TAO chooses a thread for dispatching a request and describes the POA threading models and associated policy settings that affect concurrent processing of requests. It also describes the various server concurrency models provided by TAO and the options and mechanisms to configure them, including the motivating factors for each concurrency model and the consequences of using each one.



This section includes examples of applications using the various concurrency models.

- Section 15.4, “Multithreading in the Client,” describes configuration and behavior for a thread in the client role. We provide guidelines for configuring TAO clients to achieve intended behavior while avoiding common pitfalls that can arise from incorrectly applying TAO’s concurrency-related configuration options.
    - Section 15.4.1, “Establishing a Connection to the Server,” describes TAO’s *connection strategies*, by which a TAO client will wait for a connection to a server to be established.
    - Section 15.4.2, “Multiplexing Requests on a Connection,” describes how a single connection can be used for multiple simultaneous requests.
    - Section 15.4.3, “Flushing Requests to the Server,” describes TAO’s *flushing strategies*, by which a TAO client thread will write request messages to the transport layer.
    - Section 15.4.4, “Waiting for a Reply from the Server,” describes TAO’s *wait strategies*, the client-side concurrency strategies affecting both how a client waits for a reply to come from a server and what types of activities it might do in the meantime.
    - Section 15.4.5, “Optimizing Performance of Collocated Objects,” describes TAO’s collocation optimizations, through which a client can configure trade-offs between performance and behavior for collocated objects.

15.2 Overview of Client/Server Roles in CORBA

Usually, a discussion of multithreading with respect to CORBA applications is oriented toward server-side behavior. Indeed, CORBA servers typically benefit most from the use of multithreading techniques. However, some multithreading issues are relevant to client behavior as well.

In practice, few CORBA applications are either *pure* clients (making only outbound requests and receiving replies) or *pure* servers (only processing inbound requests and returning replies). Hybrid applications that play the role of both server and client, depending upon the semantics of a particular task or

object interaction, are typical. Therefore, in many places in this chapter, to clarify the discussion of TAO's strategies associated with client/server behavior, the following definitions apply:

- A *client thread* is a thread playing the role of a client; i.e., invoking a request, waiting for a reply.
- A *server thread* is a thread playing the role of a server; i.e., available to the ORB to handle connections from clients, process inbound requests, send replies, etc.

Note that a given thread can switch between these two roles. For example, while processing an inbound request, a server thread may invoke an outbound request on another CORBA object, thereby switching to the client role. It may remain in the client role until the reply to that outbound request is received. It then switches back to the server role to prepare and send the reply to the original inbound request.

Likewise, under certain conditions a client thread may temporarily switch to the server role. For example, while waiting for a reply to an outbound request, a thread may be made available to the ORB for processing incoming client connections and requests, then return to the client role to handle its reply when it arrives.

It is important to note that by invoking a request on another CORBA object, the application controls when a thread switches from the server role to the client role. However, the configuration of the ORB at run time influences whether a thread's role can change in the opposite direction, from the client role to the server role. It may only happen under certain conditions, such as when a single-threaded hybrid application must respond to client connections and requests even while it is waiting for a reply to an outbound request.

15.2.1 The Client's Role: Overview of Request Invocation

Invoking a synchronous CORBA request can be viewed as a four-step process:

1. *A client thread invokes an operation defined in the target object's IDL interface.*

   The client invokes the operation on a client-side proxy or stub. The proxy prepares a CORBA request message to send to the target object. For our purposes, we assume the target object is located in a different address space (and possibly on a different host) than the client.

2. *The client-side ORB sends the request message to the target object.*

   The client requires a connection to transmit the request message. The client-side ORB may already have established a connection to the server. If not, a connection must be established to transmit the client's request.

3. *The client waits for the reply.*

   How the client thread waits for the reply depends upon the wait strategy configured at run-time. In some cases, the thread may be made available for handling inbound CORBA requests or other events until its reply is received. If so, the thread may temporarily switch to the server role.

4. *The client receives and processes the reply.*

   The client-side ORB receives the reply message sent from the server and returns it to the proxy. The proxy demarshals the reply and returns to the client application code that initially invoked the operation. The client thread then continues.

The steps above assume the invocation was a synchronous (two-way) request on a remote CORBA object. Different steps may be followed in the case of either a `oneway` request, an *Asynchronous Method Invocation* (AMI), or an invocation on a *collocated* object. Policies that affect invocation behavior for oneway requests, as well as the technique of AMI, are described in Chapter 6. Strategies for controlling thread behavior for invocations on collocated objects are discussed later in this chapter in 15.4.5.

15.2.2 The Server's Role: Overview of Request Processing

Processing a CORBA request can be generalized as a five-step process:

1. *The ORB binds the request to a thread.*

   A request message sent to a server remains queued in the transport's input buffer until a thread is available to process it. How the ORB selects a thread to process the request depends upon several factors, such as how the application developer has chosen to make threads available to the ORB and strategies configured at run-time that affect threading behavior. Often, the same thread is used throughout the processing of the request.

2. *The ORB dispatches the request to a POA.*

   After selecting a thread to process the request, the ORB identifies the target POA and invokes the POA to process the request.

3. *The POA dispatches the request to an application object (servant).*

To dispatch a request to a servant, a POA applies a threading model to enforce constraints on concurrent processing and a request processing policy to identify and invoke the target object, or servant. In addition, some POAs may employ customized servant dispatching strategies. The application developer chooses the threading model and request processing policy used by the POA.

4. *The servant executes and returns.*

   The server thread executes application code to satisfy the request. Depending upon the application, this code may switch to the client role and invoke outbound requests on other CORBA objects. If so, it then must wait for its reply, in which case it may switch back (temporarily) to the server role, as described in 15.2.1. Eventually, the request that was invoked on this servant is satisfied and the application code returns.

5. *The server-side ORB creates a reply message and sends it to the client.*

   In the case of a synchronous (two-way) request, the return from the servant causes a reply message to be generated and sent back to the client.

The CORBA specification (OMG Document formal/08-01-04) defines operations to enable CORBA servers to process requests. For example, a server thread calls `CORBA::ORB::run()` to run the ORB's event loop, thus making that thread available to the ORB for dispatching requests. In addition, the specification defines POA policies that applications can use to enforce constraints on concurrent request processing and control how the POA chooses the target object for each request.

Often, the operations and policies defined in the CORBA specification are insufficient for application developers to effectively control and manage the request processing behavior of their CORBA applications. For example:

- A server may need to provide multiple threads to efficiently process incoming requests, perhaps from multiple clients, in a timely manner.
- A single-threaded server that also acts in a client role may need to be able to respond to incoming requests while it is waiting for a reply to an outbound request to avoid a possible deadlock situation.
- A middle-tier server in a three-tier architecture may be used to break up large blocks of incoming data into smaller blocks that can be processed independently. This middle-tier may delegate processing of these smaller blocks of data to back-end servers, gather together the individual results

from the back-end servers, then formulate and return a single reply to the originating client.

- A server may need to de-couple threads allocated to receiving client requests from threads used to process and reply to such requests.

In fact, the CORBA specification gives ORB implementers a great deal of latitude in how they deal with threading and concurrency issues. Therefore, TAO provides several strategies that applications can use to control concurrency and request processing by servers. Some of these strategies can be completely configured at run time; others require use of specific APIs in code. We discuss several such strategies later in this chapter.

TAO's Custom Servant Dispatching (CSD) framework, gives application developers the ability to insert custom code that performs the actual servant dispatching of step 3 above, in an application-specific way. One common use of this is to select a particular thread to process a request. See 15.3.9 for more details about CSD.

In addition, TAO provides a technique called *Asynchronous Method Handling* (AMH) that allows an application to delegate processing of a synchronous request and transmission of a reply to a thread other than the one from which the servant's method was invoked. For more information on Asynchronous Method Handling, please see Chapter 7.

15.3 Multithreading in the Server

The server's role in a distributed application is a passive one, as the server waits for requests from clients. The server-side ORB has several operations to control the server's handling of requests from clients. The ORB's thread-related operations are used to do the following:

- Provide one or more threads to an ORB to handle events.
- Shut down an ORB.
- Destroy an ORB.

The relevant portions of the ORB's interface definition are:

```
// IDL
module CORBA
{
```

```
  // Portions of the CORBA module omitted.

  interface ORB
  {
    // Portions of the ORB interface omitted.

    void run();
    void shutdown( in boolean wait_for_completion );
    void destroy();
    boolean work_pending();
    void perform_work();
   };
};
```

The relevant operations can be summarized as follows:

- `CORBA::ORB::run()` provides a thread to the ORB to handle events. In appropriate circumstances, this operation can be called from multiple threads to provide more than one thread to the ORB.
- `CORBA::ORB::perform_work()` provides a thread to the ORB for a single *unit of work*. The definition for a unit of work is left to ORB implementers; often it is a single invocation. The `perform_work()` operation is best used in conjunction with `CORBA::ORB::work_pending()` to implement a polling loop that interleaves ORB processing with other activities.
- `CORBA::ORB::shutdown()` instructs an ORB to halt processing. This operation is typically invoked just prior to the ORB's destruction.
- `CORBA::ORB::destroy()` destroys an ORB and releases its resources so they can be reclaimed by the application.

The remainder of the section elaborates on each operation.

15.3.1 Event Handling

There are two CORBA-compliant operations for handling requests from clients. `The CORBA::ORB::run()` operation provides a self-contained CORBA event loop. The `CORBA::ORB::perform_work()` operation is designed to act as part of a larger event loop that interleaves non-CORBA behavior with a CORBA event loop. Both of these operations are CORBA-compliant, but provide TAO-specific extensions.

In addition, a TAO server can interleave non-CORBA behavior with CORBA events by registering an event handler with TAO's Reactor. Accessing TAO's

Reactor is discussed in more depth in 15.3.2. Further information on the Reactor is available in Chapter 3 of *C++ Network Programming: Volume 2 (C++NPv2)* by Douglas C. Schmidt and Stephen D. Huston.

15.3.1.1 CORBA::ORB::run()

`CORBA::ORB::run()` provides a thread to the ORB to handle events. In appropriate circumstances, this operation can be called from multiple threads to provide more than one thread to the ORB. This is a blocking operation from the thread's perspective; each thread that calls `run()` becomes dedicated to the ORB. The `run()` operation does not return until the ORB has shut down. More information on providing multiple threads to the ORB is available in the discussion of the thread-pool concurrency model in 15.3.6.

TAO Extensions

TAO provides overloaded versions of `CORBA::ORB::run()` that accept an `ACE_Time_Value` indicating the maximum amount of time the ORB should run its event loop before returning. The declarations of these functions, in C++, are as follows (found in `$TAO_ROOT/tao/ORB.h`):

```
// C++
namespace CORBA
{
  class ORB
  {
    // Portions of the ORB class definition omitted.

  public:
    void run (ACE_Time_Value &tv);
    void run (ACE_Time_Value *tv);

  };
};
```

`ACE_Time_Value`, defined in `$ACE_ROOT/ace/Time_Value.h`, expresses time in seconds and microseconds. For more information on `ACE_Time_Value`, please see Chapter 3 of *C++ Network Programming: Volume 2 (C++NPv2)* by Douglas C. Schmidt and Stephen D. Huston.

If a time value is passed to the ORB's `run()` method, it will block until the ORB has shut down or until it has completed processing events that are initiated within the specified time-out period. If `run()` returns before the time-out period has expired (e.g., due to ORB shut down, the process receives

a signal, etc.), the value of the `ACE_Time_Value` parameter will have been reduced by the amount of time spent in the call. In the second form of the function above, a pointer value of `0` specifies an infinite time-out period.

Note *Using these overloaded versions of* `CORBA::ORB::run()` *reduces the portability of your application code to other ORB implementations.*

Note *You should not use these overloaded versions of* `CORBA::ORB::run()` *in more than one thread unless each thread passes its own* `ACE_Time_Value` *instance. If multiple threads call* `run()` *with the same time value instance, they may concurrently modify the time value instance, leading to unexpected behavior.*

15.3.1.2 CORBA::ORB::perform_work()

`CORBA::ORB::perform_work()` provides a thread to the ORB for a single *unit of work*. This is a non-blocking operation from the calling thread's perspective; the thread can perform tasks unrelated to the ORB between calls to `perform_work()`. The definition for a unit of work is left to ORB implementers; often it is a single invocation.

The `perform_work()` operation is best used in conjunction with `CORBA::ORB::work_pending()` to implement a polling loop that interleaves ORB processing with other activities, such as servicing another external interface. However, this can lead to jittery performance compared with calling `CORBA::ORB::run()` in its own thread. The boolean return value from `work_pending()` indicates whether or not an ORB has an immediate need for a thread to perform ORB-related activities.

TAO Extensions

TAO provides overloaded versions of `CORBA::ORB::perform_work()` that accept an `ACE_Time_Value` indicating the maximum amount of time the ORB should stay in `perform_work()` before returning. The declarations of these functions, in C++, are as follows (found in `$TAO_ROOT/tao/ORB.h`):

```
namespace CORBA
{
  class ORB
  {
```

```
    // Portions of the ORB class definition omitted.

  public:
    void perform_work (ACE_Time_Value &tv);
    void perform_work (ACE_Time_Value *tv);

  };
};
```

Note *In TAO, when* `CORBA::ORB::perform_work()` *is called without a time value, the ORB may handle multiple events before returning. These events may include input events (such as inbound requests or replies), output events (such as outbound requests or replies), timers, or signals. There is no guarantee that* `perform_work()` *will return after dispatching just one event.*

Note *The CORBA specification states that only the main thread should use* `work_pending()` *and* `perform_work()`*. There is no such restriction in TAO.*

Guidelines for Application Developers

For an application to behave in the server role, at least one thread must be made available to the ORB by calling `CORBA::ORB::run()` or by periodically calling `CORBA::ORB::perform_work()`. A thread from which `run()` is called becomes dedicated to the ORB. A thread that calls `perform_work()` can be used to perform other tasks in addition to ORB-related tasks.

Typical CORBA server architectures include:

- A single-threaded server that has all of its activities initiated via CORBA invocations typically calls `run()` from the main thread after the application is initialized.
- A multithreaded server that has all of its activities initiated via CORBA invocations typically calls `run()` from each thread.
- A single-threaded server that has processing tasks unrelated to CORBA invocations has three options:

1. Implement a polling loop, using `work_pending()` and `perform_work()`, to interleave CORBA invocations with other processing tasks. A typical polling loop is similar to the following:

   ```
   for(;;)
   {
     if(orb->work_pending())
     {
       orb->perform_work();
     }
     // do other tasks
   }
   ```

   In most cases, `perform_work()` should be called only when `work_pending()` returns `true` to prevent the application from blocking in the absence of CORBA invocations. However, TAO allows time-bounded calls to `perform_work()` and thus removes the need to call `work_pending()`. This use of `perform_work()` is shown in the following code fragment:

   ```
   for(;;)
   {
     // wait for incoming requests for a maximum of 100 milliseconds
     ACE_Time_Value tv(0, 100000); // 100 msec == 100000 usec
     orb->perform_work(tv);
     // do other tasks
   }
   ```

   This call to `perform_work()` blocks until either a unit of work is processed or at least one second passes, whichever comes first. The time spent in `perform_work()` is not deterministic, as is discussed below.

2. Implement a time-bounded `CORBA::ORB::run()` loop, using TAO's `ACE_Time_Value`-based extensions. For example:

   ```
   for(;;)
   {
     // wait for incoming requests for 100 milliseconds
     ACE_Time_Value tv(0, 100000); // 100 msec == 100000 usec
     orb->run(tv);
     // do other tasks
   }
   ```

This call to `run()` blocks until at least one second passes. The time spent in `run()` is not deterministic.

3. Register an event handler with TAO's reactor and handle both ORB and non-CORBA events with `orb->run()`, thus interleaving ORB events with events from other sources. This is discussed in more detail below.

- A multithreaded server that has tasks unrelated to CORBA invocations has two options:

    1. Dedicate each thread to either performing ORB-related activities using `orb->run()`, or to other activities using the appropriate mechanism.
    2. Dedicate a group of threads to the ORB using `orb->run()` and use a polling loop to interleave ORB-related and other activities on another thread or group of threads.
    3. Register an event handler with TAO's reactor and handle both ORB and non-CORBA events with multiple threads using `orb->run()`, thus interleaving ORB events with events from other sources. This is discussed in more detail below.

    Whether or not to use multiple threads for activities unrelated to the ORB depends upon the nature of those activities and the applicable concurrency constraints. When a polling loop is used in multithreaded applications, it is often executed on the main thread after other threads that are dedicated to the ORB have been spawned.

The run time of `perform_work()` is not deterministic. Using `perform_work()` to interleave CORBA invocations with other processing activities can be tricky**.** Numerous factors, including long-running CORBA requests, outbound CORBA requests that are issued during the processing of an inbound CORBA request, timers, and signals influence the run time of `perform_work()`. As described above, TAO extends `perform_work()` to permit specification of a maximum run time, but typically this is not sufficient to place an absolute limit on `perform_work()`'s run time. For example, when a synchronous outbound request is issued during the processing of an inbound request, the run time for the inbound request becomes a function of the outbound request's run time, which is beyond the scope of the local ORB.

15.3.2 Interleaving ORB Events with Events from Other Sources

Often, an application that plays the role of a CORBA server must also be responsive to events from other sources, such as a graphical user interface or a socket-based messaging or data distribution infrastructure. These applications may need to interleave ORB events (inbound client connections and requests, outbound requests and replies) with these non-CORBA events. As discussed above, there are several options for accomplishing this interleaving of events. For example, the application could:

- Use non-blocking ORB event handling operations (`work_pending()` and `perform_work()`) to implement a polling loop. See the discussion above for more information.
- Dedicate separate threads for CORBA request processing and for processing events from other sources. See the discussion above for more information.
- Register its own custom event handlers with the ORB's reactor and handle both ORB and non-CORBA events with `orb->run()`. For example:

```
// get TAO's reactor
ACE_Reactor* reactor = orb->orb_core()->reactor();

// ... create an event handler, register it with the Reactor ...

// handle CORBA and non-CORBA events
orb->run();
```

  More information on creating an event handler and registering it with the reactor is available in Chapter 3 of *C++ Network Programming: Volume 2 (C++NPv2)* and in Chapter 7 of *The ACE Programmer's Guide (APG)*.

- Use one of the special resource factory and reactor types provided by TAO that are designed for integrating with other event sources. For example, see 18.3 for information on integrating TAO with the Qt GUI toolkit, 18.4 for information on integrating TAO with the X Window System XtIntrinsics toolkit, and 18.7.6 for information on using the WFMO reactor with TAO.
- Implement a custom reactor type that integrates CORBA events with events from other sources.

15.3.3 Shutdown and Destruction

The `CORBA::ORB::shutdown()` and `CORBA::ORB::destroy()` operations halt the ORB's processing and release the ORB's resources, respectively. Both of these operations are CORBA-compliant; neither has TAO-specific extensions. This sections contains descriptions of both operations and guidelines for successful execution of shutdown and destruction.

15.3.3.1 CORBA::ORB::shutdown()

`CORBA::ORB::shutdown()` instructs an ORB to halt processing. This operation is typically invoked just prior to the ORB's destruction. If the `wait_for_completion` parameter is `true`, this operation blocks until the ORB has concluded request processing and other object adapter related activities, and all object adapters have been destroyed. If `wait_for_completion` is `false`, then `shutdown()` may return before the ORB completes the process of shutting down.

Note *The ORB continues normal processing activities while it is shutting down, so there may be a significant delay after* `shutdown()` *is called before the ORB actually stops processing. For example, if* `shutdown()` *is called with* `wait_for_completion` *equal to* `true`*, the ORB waits for all threads that are processing ORB events to exit before returning.*

Exceptions

The following exceptions may be raised by the ORB's thread-related operations or by circumstances arising from the use of those thread-related operations:

- `work_pending()` or `perform_work()` called on an ORB after it has been shut down raises `CORBA::BAD_INV_ORDER`.
- `shutdown()` with `wait_for_completion` equal to `true` called from a thread processing a CORBA request raises `CORBA::BAD_INV_ORDER` with an OMG minor code of 3 and a completion status of `COMPLETED_NO`.
- Any operation other than `duplicate()`, `release()`, or `is_nil()` invoked on an ORB after it has shut down, or invoked on an object reference obtained through an ORB that has since shut down, raises `CORBA::BAD_INV_ORDER` with an OMG minor code of 4. Scenarios may arise in which operations other than those cited here may be called while

an ORB is shutting down. Application developers should be prepared to handle this exception when such scenarios may occur because the time required to shut down an ORB is not predictable.

15.3.3.2 CORBA::ORB::destroy()

`CORBA::ORB::destroy()` destroys an ORB and releases its resources so they can be reclaimed by the application. This operation initiates the shutdown process when called on an ORB that has not been shut down. If `destroy()` initiates the shutdown process, then it blocks until the ORB has shut down as if `shutdown()` had been called with `wait_for_completion` equal to `true`.

Exceptions

The following exceptions may be raised by the ORB's thread-related operations or by circumstances arising from the use of those thread-related operations:

- Any operation invoked on an ORB after its destruction raises `CORBA::OBJECT_NOT_EXIST`.
- `destroy()` called from a thread processing an invocation raises `CORBA::BAD_INV_ORDER` with an OMG minor code of 3.

Guidelines for Application Developers

Terminating a CORBA server gracefully is often a complicated matter because the ORB continues normal processing while it shuts down. An application dedicated to processing requests can be signaled to shut down via a CORBA request. An application that performs activities unrelated to CORBA requests can be signaled to shut down via a CORBA request as well as any other interfaces that are serviced outside the scope of CORBA requests.

If signaled to shut down via a CORBA interface, `shutdown()` can be invoked with `wait_for_completion` equal to `false` from the thread that processes the request; invoking `shutdown()` with `wait_for_completion` equal to `true` would raise an exception in this situation. If signaled to shut down outside the scope of a CORBA request, the responding thread can invoke `shutdown()` with `wait_for_completion` equal to `true` or `false`; `wait_for_completion` equal to `true` in this case will block the calling thread until the ORB has shut down.

Once `shutdown()` has been called, all calls to `run()` will return once the ORB has shut down. However, the time required to shut down an ORB depends upon the volume of client activity when shutdown is requested. In extreme cases, a complete cessation of client activity may be necessary to allow the ORB to shut down.

Here are some options, both graceful and ungraceful, for shutting down a CORBA-based application:

- The application can be killed with the appropriate platform-specific mechanism. This is the simplest solution but it is not graceful and may have undesirable side-effects.
- `CORBA::ORB::shutdown()` can be called with `wait_for_completion` equal to `false` from a thread processing a CORBA request that was invoked via some maintenance or administrative interface. All calls to `CORBA::ORB::run()` return once the ORB has shut down and thus release the threads dedicated to the ORB. Some mechanism should be used to prevent the main thread from exiting before the other threads exit.

  The application may continue to run for a relatively long time. If this is unacceptable, another mechanism should be used to forcibly terminate the application.
- The application can block the main thread on a condition variable or sempahore until the application is signaled to shut down. (In this case, the main thread is not used for ORB-related activities.) Once unblocked, the main thread calls `CORBA::ORB::shutdown()` to terminate ORB-related processing. If `shutdown()` is called with `wait_for_completion` equal to `false`, some other mechanism should be used to prevent the main thread from exiting before all other threads have exited.

  This option provides some additional flexibility. After the main thread initiates shutdown, it might then block until all other threads have exited or a specified time interval has elapsed. If the other threads exit before the time interval elapses, the main thread conducts any other cleanup activities and then exits. If the time period elapses, the main thread performs some, perhaps not all, of the usual clean-up activity and then exits causing a forced termination.

The next section contains more information on gracefully shutting down a TAO server.

15.3.3.3 Gracefully Shutting Down a TAO Server

In practice, CORBA applications are often designed for 24/7 operation. If a server is not running, the system is non-functional. Still, there are often situations in which a server needs to be *gracefully* shut down (e.g., to replace a running server with a newer version or to move a running server to a different host). By a *graceful* shutdown, we mean something slightly more elegant than, for example, sending the process a kill signal.

Gracefully shutting down a TAO server can be tricky. An application could do one of the following:

- Use non-blocking ORB event handling operations (`work_pending()` and `perform_work()`) to implement a polling loop as described in . An application-specific flag could be used to indicate when the loop should be terminated.

  The following code fragment shows how this could be done:

```
bool done = false;
do
{
  // wait for incoming requests for a maximum of 100 milliseconds
  ACE_Time_Value tv(0, 100000); // 100 msec == 100000 usec
  orb->perform_work(tv);
  // do other tasks
  if ( /* some exit condition exists */ )
    done = true;
}
while (!done);
```

- Use `CORBA::ORB::run()` in one or more threads to process CORBA requests, then invoke `CORBA::ORB::shutdown()` from a thread that is not being used to process CORBA requests. For example, the application could provide one thread that monitors console input for an application-specific command such as “`quit`.” If the `wait_for_completion` parameter to `shutdown()` is `true`, request processing could continue for some time after the call the `shutdown()`.
- Call `CORBA::ORB::shutdown()` within the context of processing a request. For example, the application could implement an interface that has a `shutdown()` operation, such as the following:

```
interface MyAppAdmin
{
```

```
  oneway void shutdown ();
};
```

One possible implementation of the `MyAppAdmin::shutdown()` operation could be as follows:

```
void MyAppAdmin_i::shutdown ()
{
  // shut down the ORB (stored in a CORBA::ORB_var variable orb_)
  orb_->shutdown (false);
}
```

Note that the application passes `false` for the `wait_for_completion` parameter to `CORBA::ORB::shutdown()` as required when `shutdown()` is called within the context of a request; otherwise, the ORB raises the `CORBA::BAD_INV_ORDER` system exception with an OMG minor code of 3.

- Register a timer (derived from `ACE_Event_Handler`) with the ORB's reactor (or a separate reactor) and call `CORBA::ORB::shutdown()` in the timer's `handle_timeout()` method.

  The following code fragment shows how such a timer class could be implemented:

```
class MyAppShutdownTimer : public ACE_Event_Handler
{
public:
  // Constructor
  MyAppShutdownTimer (CORBA::ORB_ptr orb)
    : orb_(CORBA::ORB::_duplicate(orb)) { }

  // Receive timeout events from the Reactor
  virtual int handle_timeout (const ACE_Time_Value &current_time,
                              const void *act)
  {
    orb_->shutdown (false);
    return 0;
  }

private:
  CORBA::ORB_var orb_;
};
```

  The following code fragment shows how to create and schedule a timer with the ORB's reactor:

```
   // Portions of int main(int, char *argv[]) omitted.

// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

// Intervening code omitted.

// Create a shutdown timer.
MyAppShutdownTimer * timer = new MyAppShutdownTimer (orb.in());

// Schedule the timer to shutdown the ORB in 30 seconds.
ACE_Time_Value timeout (30,0);
orb->orb_core()->reactor()->schedule_timer(
  timer, 0, timeout);

// Run the ORB's event loop.
orb->run();
orb->destroy();
```

- Register a signal handler (derived from `ACE_Event_Handler`) with either the ORB's reactor, a separate reactor, or directly via `ACE_Sig_Handler`. It is probably not safe to call `CORBA::ORB::shutdown()` in the signal handling context, so the signal handler must somehow notify another thread to actually call `shutdown()`. One way to do this is via the reactor's `notify()` member function.

Note *Signal handlers cannot be registered with the thread-pool reactor.*

- Use the overloaded version of `CORBA::ORB::run()` or `CORBA::ORB::perform_work()` that processes events for a specified time interval.

An example showing many of the above techniques for gracefully shutting down a TAO server can be found in the `$TAO_ROOT/DevGuideExamples/Multithreading/GracefulShutdown` directory.

15.3.4 Request Processing

A CORBA request is processed in five stages:

1. The ORB binds the request to a thread.
2. The ORB dispatches the request to a POA.
3. The POA dispatches the request to an application object (servant).
4. The servant completes the request and returns.
5. The ORB sends a reply to the client.

Strategies employed by a multithreaded ORB as it receives and dispatches requests determine the extent to which requests may be processed concurrently. Since the CORBA specification does little to address how an ORB should select a thread for dispatching requests, different ORB implementations employ different concurrency models for request dispatching. TAO employs the following concurrency models for request dispatching:

- single-threaded — all requests are processed on a single thread.
- thread-per-connection — a separate thread is spawned to process all requests received via a distinct network connection. At least one thread must be available to handle incoming client connections.
- fixed thread-pool — request processing is distributed amongst a group (pool) of threads. The threads in the pool are supplied by the application.
- dynamic thread-pool — request processing is distributed amongst a pool of threads. The threads in the pool are spawned by the ORB. The pool may grow or shrink based on the server load.

Each concurrency model has its strengths and weaknesses, which motivates understanding an application's request processing characteristics to choose the most appropriate strategy or combination of strategies.

The remainder of this section describes the POA's threading models. See Chapter 11 of *Advanced CORBA Programming with C++* for more information on the Portable Object Adapter. See 15.3.6 for a complete discussion of TAO's threading models.

15.3.5 The POA's Threading Models

The POA threading models establish concurrency constraints that are imposed during request processing in a multithreaded environment. These constraints further qualify the extent to which CORBA requests may be processed concurrently.

A POA's threading model is determined by its `ThreadPolicy` value. The CORBA specification defines three standard `ThreadPolicy` values:

- `ORB_CTRL_MODEL`—requests are dispatched to application objects from the thread with which the ORB invoked the POA. Concurrent servant upcalls may occur if the ORB's strategies allow requests to be dispatched concurrently. `ORB_CTRL_MODEL` is the default `ThreadPolicy` setting.
- `SINGLE_THREAD_MODEL`—the POA dispatches requests to application objects sequentially. Concurrent servant upcalls cannot occur within the scope of one `SINGLE_THREAD_MODEL` POA.
- `MAIN_THREAD_MODEL`—requests for all main-threaded POAs are dispatched sequentially. Concurrent servant upcalls cannot occur within the scope of all `MAIN_THREAD_MODEL` POAs.

Note *TAO does not support the* `MAIN_THREAD_MODEL` *POA ThreadPolicy value.*

15.3.5.1 ORB-Controlled Model

An ORB-controlled POA places no constraints on concurrent requests; the POA effectively abdicates responsibility for threading and concurrent request processing to the ORB. Figure 15-1 shows servants activated in POAs using the ORB-controlled threading model.

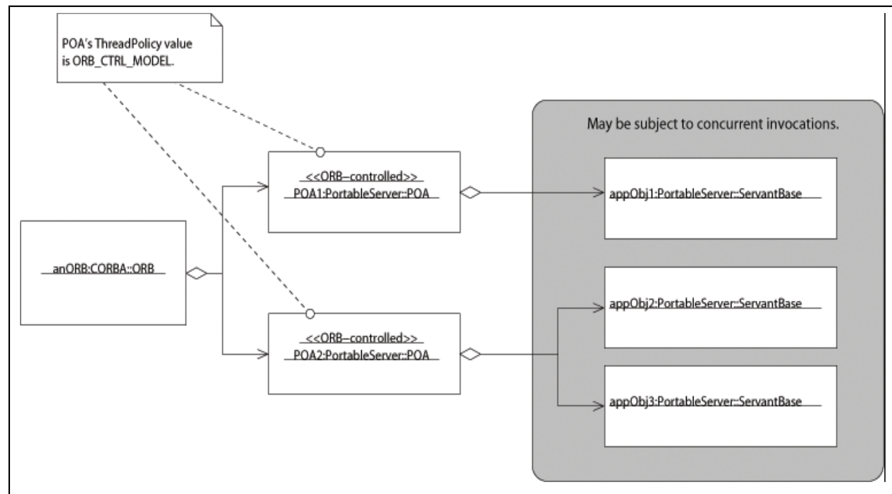


Figure 15-1 ORB-Controlled Threading Model

In the ORB-controlled threading model, concurrent servant upcalls may occur at the POA level in multithreaded environments if the ORB dispatches requests concurrently. Application objects activated in an ORB-controlled POA in a multithreaded environment should be multithread safe. Ensuring that those application objects are multithread safe is the responsibility of the application developer.

15.3.5.2 Single Thread Model

A single-threaded POA constrains request processing such that concurrent upcalls cannot occur at the POA level, even in multithreaded environments. Figure 15-2 shows servants activated in POAs using the single-thread model.

A multithreaded ORB may dispatch concurrent requests to a single-threaded POA, but the subsequent servant upcalls will occur sequentially; thus an application object activated in one single-threaded POA will not be subject to concurrent requests. However, an application object activated in multiple single-threaded POAs may be subject to multiple concurrent requests in a multithreaded environment, as shown in Figure 15-3. Therefore, activating an application object in multiple POAs is not recommended.

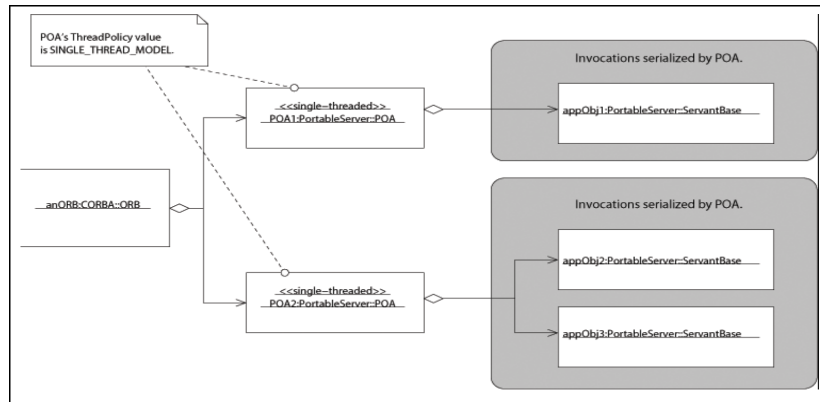


Figure 15-2 Single-Thread Threading Model

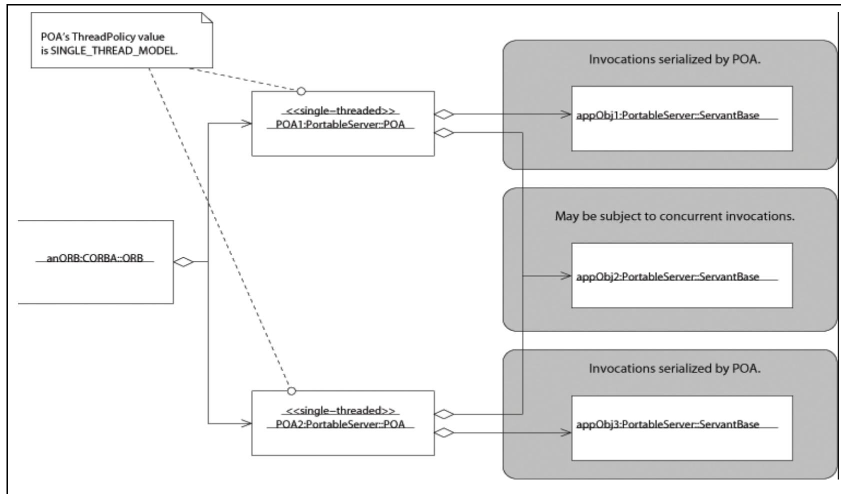


Figure 15-3 Servant Activated in Multiple Single-threaded POAs

15.3.5.3 Main Thread Model

In this model, requests are dispatched to all main-thread POAs sequentially. This model effectively serializes requests dispatched by main-thread POAs at the ORB level. Figure 15-4 shows servants activated in POAs using the main-thread model.

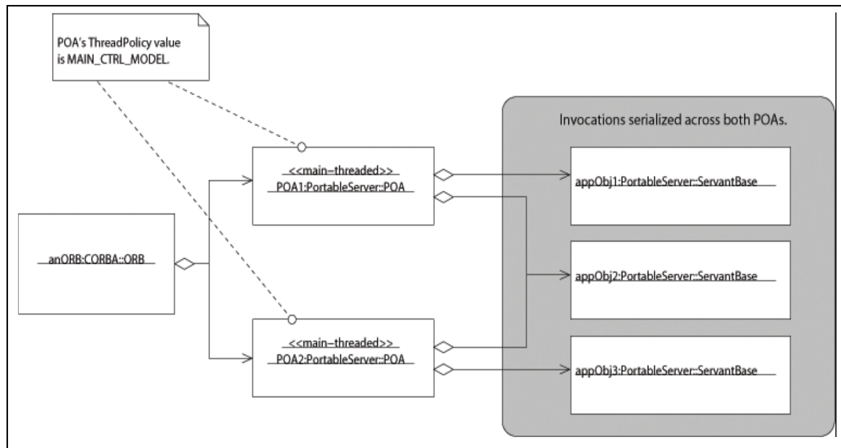


Figure 15-4 Main-Thread Model

An application object activated by a main-thread POA will not be subject to concurrent requests. The main-thread model is also applicable in processing environments where some code must execute on the main thread. In such environments, the main-thread model insures that servant upcalls are processed on that thread. However, the application must make the main thread available to the ORB by calling either `CORBA::ORB::run()` or `CORBA::ORB::perform_work()`.

Note *TAO does not support the* `MAIN_THREAD_MODEL` *POA ThreadPolicy value.*

15.3.5.4 Configuring a POA's Threading Model

A POA's thread policy, like all other POA policies, can be assigned only when the POA is created. In most cases, the default `ThreadPolicy` of `ORB_CTRL_MODEL` is desired, so no intervention is required on the part of the developer. If a single-threaded environment is desired, simply give the ORB exactly one thread.

However, if you'd like to create a POA using the `SINGLE_THREAD_MODEL`, the remainder of this section demonstrates how to do that.

The relevant portions of the `PortableServer` module and the POA's interface are:

```
module PortableServer {

  // Portions of the PortableServer module omitted.

  const CORBA::PolicyType THREAD_POLICY_ID = 16;

  enum ThreadPolicyValue {
    ORB_CTRL_MODEL,
    SINGLE_THREAD_MODEL,
    MAIN_THREAD_MODEL
  };

  local interface ThreadPolicy : CORBA::Policy {
    readonly attribute ThreadPolicyValue value;
  };

  local interface POA {

    // Portions of interface POA omitted.
```

```
    POA create_POA(
      in string adapter_name;
      in POAManager a_POAManager,
      in CORBA::PolicyList policies
      ) raises (AdapterAlreadyExists, InvalidPolicy);

    ThreadPolicy create_thread_policy(
      in ThreadPolicyValue value);

  };
};
```

The following code fragment demonstrates how to create a single-threaded POA:

```
  // Portions of int main(int, char *argv[]) omitted.

  // Initialize the ORB.
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

  // Get a reference to the RootPOA.
  CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow (obj.in());

  // Create and populate a policy list.
  CORBA::PolicyList policies(1);
  policies[0] = poa->create_thread_policy(PortableServer::SINGLE_THREAD_MODEL);

  // Use the RootPOA’s POAManager.
  PortableServer::POAManager_var mgr = poa->the_POAManager ();

  // Create a child POA.
  // Threading model is single-threaded.
  // All other policies assume their default values.
  PortableServer::POA_var st_poa = poa->create_POA("ST POA", mgr.in(), policies);

  // Release memory allocated to the policy list.
  policies[0]->destroy();
```

Guidelines for Application Developers

The ORB-controlled threading model (`ORB_CTRL_MODEL`) is the default, and usually the desired, threading model. It allows application developers to make the most of an ORB’s concurrent processing capabilities and performance optimization strategies. An application using this threading model can also take advantage of fine-grained contention management techniques to resolve application objects’ contention for shared resources and services. However,

this model obligates application developers to assume concurrent requests will occur, and to identify and resolve points of contention that may arise during concurrent CORBA requests.

The following scenarios might motivate use of the ORB-controlled threading model for a CORBA-based application that is subject to concurrent requests:

- The application's mission is to provide access to and resolve contention for a shared resource or service, i.e. the application is effectively a contention manager so using an ORB-controlled threading model does not add additional complexity.
- The application serializes and dispatches events, perhaps received from disparate sources, to discrete event processors (application objects) so multiple events can be processed concurrently. As with the previous scenario, applications such as this are obligated to resolve contention.
- The application is a CORBA interface to an existing system that permits concurrent invocations.
- Acceptable performance can only be achieved with concurrent processing and fine-grained contention management.
- The application is stateless, i.e. the context carried with each CORBA request is sufficient to process the request.

The single-threaded model (`SINGLE_THREAD_MODEL`) can simplify an application's design and implementation by guaranteeing that an application object will not be subject to concurrent invocations from a single POA. Applied within its constraints, this model shifts some responsibility for contention management away from the application.

However, this mechanism can only resolve contention for distinct servants arising from concurrent requests; it cannot resolve contention for services and resources shared by application objects. Resolving contention for commonly-used services and resources remains the application's responsibility.

The single-threaded model is appropriate when:

- The likelihood of concurrent requests is acceptably low.
- The increased latency caused by the serialization of requests is acceptable.

- The potential performance improvement resulting from concurrent request processing does not justify the added complexity associated with fine-grained contention management.
- An application depends upon software libraries or legacy code that effectively prohibits concurrent processing.

The main-thread model (`MAIN_THREAD_MODEL`) is applicable in two circumstances:

- An application's environment mandates that certain portions of code execute only on the main thread. However, the application is obligated to make the main thread available to the ORB by calling `run()` or `perform_work()` on the ORB from the main thread.
- An application requires protection from concurrent requests more rigorous than that provided by the single-thread model.

This model imposes the most stringent constraints upon an application and likely reduces a multithreaded ORB's efficiency. Application developers may want to compare performance of this threading model with that of a single-threaded ORB before committing to this model.

Note *TAO does not support the* `MAIN_THREAD_MODEL` *POA ThreadPolicy value.*

The following section describes usage of the default `ORB_CTRL_MODEL` thread policy with TAO's threading models.

15.3.6 TAO's ORB Threading Models

The ORB-controlled threading model (`ORB_CTRL_MODEL`) allows application developers to make the most of an ORB's concurrent processing capabilities and performance optimization strategies by enabling the application developer to take advantage of TAO's server-side concurrency models. TAO's concurrency models are relevant only to a POA configured with the default `ORB_CTRL_MODEL` thread policy.

In this section, we discuss TAO's various concurrency models. Concurrency in TAO is controlled by various factors, including how the ORB is configured at run time, how many threads are in the process, and the specific behavior of each thread.

TAO provides three concurrency models affecting how a server thread receives and processes requests:

- reactive
- thread-per-connection
- thread-pool

Note *In addition to these, TAO's support of Real-Time CORBA provides the ability to associate a thread pool with a POA; see 8.3.7 for more information on using RT CORBA's thread-pool policy.*

The remainder of this section discusses each of these concurrency models in detail. For each concurrency model, we present the motivation for its use, configuration options to enable it, and consequences of its use. We have provided a working example for each concurrency model.

15.3.6.1 Reactive Concurrency Model

By default, TAO servers process incoming client connections and requests *reactively*. Typically, a single thread (often the main thread) is dedicated to processing client requests by calling `CORBA::ORB::run()`. Other threads may also exist in the process, but are dedicated to performing other tasks.

In the reactive concurrency model, the ORB uses an ACE Reactor to receive and dispatch client requests. The default reactor type used in TAO is the thread-pool reactor, but the ORB can be configured at run time to use a different reactor type (see 15.3.11 and 18.7.6). Assuming that the ORB's `run()` operation is called from only one thread, that single thread will be used for receiving and dispatching all requests.

Note *If a client thread makes an outbound request, that thread may be used to process inbound requests while waiting for its reply. When this occurs, the thread is said to be in a "nested upcall." See 15.4.4 and 20.2.3 for more details.*

Note *More information on the ACE Reactor Framework and ACE Reactor implementations can be found in Chapters 3 and 4 of C++ Network*

Programming: Volume 2 (C++NPv2) and Chapter 7 of The ACE Programmer's Guide (APG).

Figure 15-5 illustrates a server using the reactive concurrency model with a single thread. All incoming requests are received and processed on the same thread. Subsequent requests, either from the same client or a different client, are blocked waiting for the thread to complete processing the previous request.

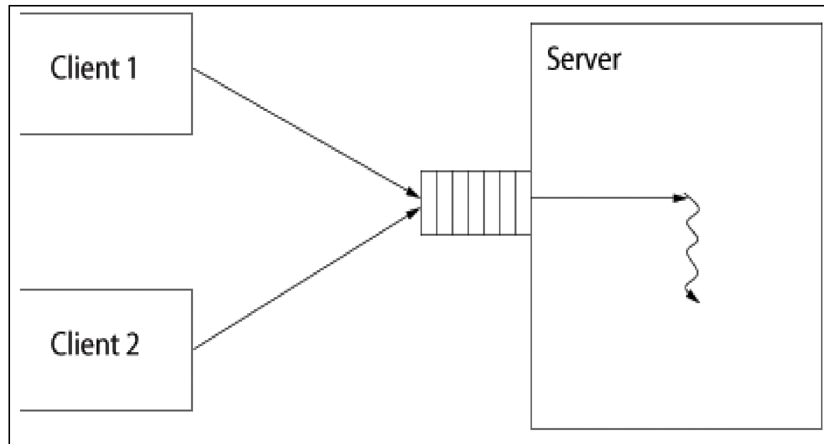


Figure 15-5 Reactive concurrency model with single thread

Motivation

The reactive concurrency model is useful in the following situations:

- The application is a single-threaded server in which request processing times remain relatively fixed.
- Since this model is simple to configure and program, it is often used during initial development and testing of interface implementations.

Configuration

The reactive concurrency model is selected by default in TAO; no specific action is necessary to configure it. However, it can be specified explicitly by supplying the following configuration option to the default server strategy factory:

```
static Server_Strategy_Factory "-ORBConcurrency reactive"
```

See 19.3.5 for more information on using this option.

Consequences

The consequences of using the single-threaded reactive concurrency model are as follows:

- Requests are processed in the order received.
- Subsequent requests may be blocked until the thread completes processing the previous request. This can lead to unbounded latencies.
- Since requests are not processed concurrently, developers may be able to simplify synchronization control in their implementation code. However, synchronization issues should not be completely ignored in case multithreading is used elsewhere in the application or added later.
- Certain locks in the ORB and reactor may be disabled to improve throughput.
- Scalability is limited; as the number of simultaneous client requests increases, so does the per-request latency.

Example

The following example shows a typical single-threaded server using the reactive concurrency model. Configuration of the reactive concurrency model has been specified explicitly even though it is the default model. In this example, we have also configured the single-threaded reactor type (`-ORBReactorType select_st`), the single-threaded wait strategy (`-ORBWaitStrategy st`), and disabled some locks to attempt to maximize overall throughput. See 15.3.11 for more information on the single-threaded reactor. See 15.4.4.2 for more information on the single-threaded, wait-on-reactor wait strategy.

The server source code for this example is the Messenger server from Chapter 3. We show only the file `MessengerServer.cpp`, which contains the code for the server's `main()` function. Only one thread exists in the server and that thread is dedicated to processing client connections and requests by calling `CORBA::ORB::run()`. The source code for this example is in the `$TAO_ROOT/DevGuideExamples/Multithreading/Reactive` directory.

```
#include "Messenger_i.h"
#include <iostream>
#include <fstream>
```

```
int main(int argc, char* argv[])
{
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Get a reference to the RootPOA.
    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

    // Activate the POAManager.
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Create a servant.
   PortableServer::Servant_var<Messenger_i> messenger_servant = new Messenger_i;

    // Register the servant with the RootPOA, obtain its object
    // reference, stringify it, and write it to a file.
    PortableServer::ObjectId_var oid =
      poa->activate_object(messenger_servant.in());
    CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
    CORBA::String_var str = orb->object_to_string(messenger_obj.in());
    std::ofstream iorFile("Messenger.ior");
    iorFile << str << std::endl;
    iorFile.close();
    std::cout << "IOR written to file Messenger.ior" << std::endl;

    // Accept requests from clients.
    orb->run();
    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "CORBA exception: " << ex << std::endl;
    return 1;
  }

  return 0;
}
```

The service configurator file, shown below, contains directives to configure the reactive concurrency model, use the single-threaded select reactor type, disable certain locks, and use the single-threaded wait-on-reactor wait strategy. These options are selected to improve overall throughput. Each directive should appear on a single line.

```
# server.conf file for single-threaded reactive server.
```

```
static Server_Strategy_Factory "-ORBConcurrency reactive -ORBPOALock null"
dynamic Advanced_Resource_Factory Service_Object *
TAO_Strategies:_make_TAO_Advanced_Resource_Factory () "-ORBReactorType select_st
-ORBInputCDRAllocator null"
static Client_Strategy_Factory "-ORBProfileLock null -ORBWaitStrategy st"
```

In the above example, we use the `dynamic` directive to configure the Advanced Resource Factory so it can be loaded and configured from the `TAO_Strategies` library dynamically. Alternatively, we can statically link the application with the `TAO_Strategies` library, insert the line:

```
#include <tao/Strategies/advanced_resource.h>
```

prior to `main()`, and use the `static` directive to configure the Advanced Resource Factory:

```
static Advanced_Resource_Factory "-ORBReactorType select_st
-ORBInputCDRAllocator null"
```

See 18.5 for more information on using and configuring TAO's Advanced Resource Factory.

15.3.6.2 Thread-per-Connection Concurrency Model

The simplest multithreaded server configuration in TAO is the *thread-per-connection* concurrency model. In this model, the ORB automatically spawns a new thread for each new client connection. Each spawned thread is dedicated to processing requests that arrive on a given connection. The thread reads and processes those requests serially. When the connection is closed, the thread associated with that connection exits and all resources allocated by the ORB associated with that thread are released. At least one thread (other than those that are dedicated to processing requests) must remain available to the ORB to handle new connections; usually, this is accomplished by calling `CORBA::ORB::run()` in the main thread.

Figure 15-6 shows a multithreaded server using the thread-per-connection concurrency model.

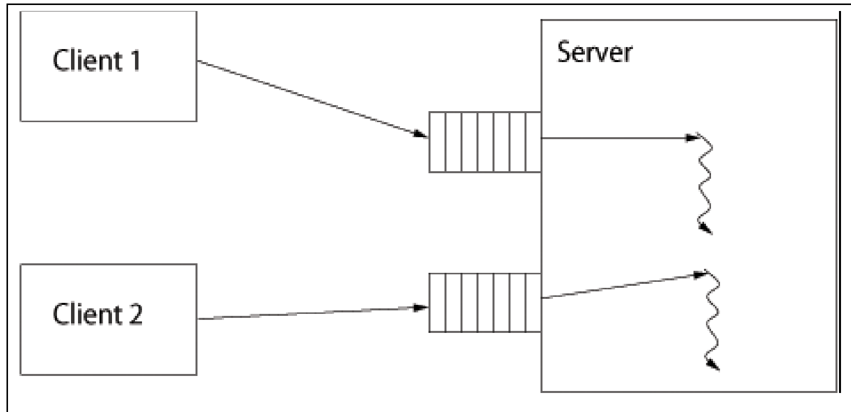


Figure 15-6 Thread-per-connection concurrency model

Motivation

The thread-per-connection concurrency model is useful in the following situations:

- Multiple client connections are active simultaneously.
- Request processing is computationally intense.
- Requests arriving over separate connections are largely decoupled from one another.

Configuration

The thread-per-connection concurrency model is specified by supplying the following configuration option to the default server strategy factory:

```
static Server_Strategy_Factory "-ORBConcurrency thread-per-connection"
```

See 19.3.5 for more information on using this option.

Consequences

The consequences of using the thread-per-connection concurrency model are as follows:

- The total number of threads created and managed by the ORB is one plus the number of simultaneous client connections.

- Requests arriving on different connections are processed concurrently.
- Requests arriving on a given connection are processed serially, always by the same thread.
- Requests from one client are not blocked by requests from another client.
- Subsequent requests from one client connection may be blocked until the thread dedicated to that connection completes processing the previous request.
- Since requests arriving on different connections may be processed concurrently, developers may have to protect critical regions of code with locks. However, depending upon the application, it may be possible to reduce locking overhead. For example, if processing of requests on different connections is completely decoupled, with no shared code or shared state, locking may not be necessary.
- Threads that are spawned by the ORB for each new connection do not use a reactor for receiving and dispatching inbound requests; they simply block waiting to read. These threads may need to awaken periodically to check whether the ORB has been shut down. The default server strategy factory's `-ORBThreadPerConnectionTimeout` option can be used to configure the thread wake-up interval. See 19.3.10 for more information on this option.
- All threads spawned by the ORB process requests at the same priority. Applications that need to process different requests at different priorities should use the features provided by the RT CORBA thread-pool model. See 8.3.7 for more information.
- Scalability is limited; as the number of simultaneous incoming connections increases, so does the number of threads in the server.

Note *A single client may open multiple simultaneous connections to a server ORB, depending upon the number of client threads that invoke requests and the configuration of the ORB's transport multiplexing strategy* (`-ORBTransportMuxStrategy`). *Thus, using the thread-per-connection concurrency model, it is possible for multiple server threads to be created for processing requests from a single client. See 15.4.2 and 20.3.6 for more information on the* `-ORBTransportMuxStrategy` *option.*

Example

In this section, we describe an example of a typical multithreaded server using the thread-per-connection concurrency model. No application code changes are required for creating and managing threads; all thread management is performed by the ORB. However, the application's servant implementations may need to provide locking around critical regions of code.

The server source code for the example is the Messenger server from Chapter 3. The main thread is dedicated to processing client connections by calling `CORBA::ORB::run()`. The ORB automatically spawns a new thread for each new connection. Since no changes are required to the server's `main()` function, we do not show the file `MessengerServer.cpp`.

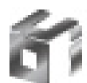
The source code for this example is in the `$TAO_ROOT/DevGuideExamples/Multithreading/ThreadPerConnection` directory.

The server's service configurator file, shown below, contains directives to configure the ORB to use the thread-per-connection concurrency model. It also shows how the thread-per-connection time-out interval (in milliseconds) can be optionally configured at run time. In this example, we wake up the server's threads on a one-second interval so they can check if the ORB has been shut down. Each directive should appear on a single line.

```
# server.conf file for thread-per-connection server.
static Server_Strategy_Factory "-ORBConcurrency thread-per-connection
-ORBThreadPerConnectionTimeout 1000"
```

15.3.6.3 Thread-pool Concurrency Model

In the thread-pool concurrency model, a pool of threads is made available to the ORB for handling incoming client connections and processing inbound requests. The developer takes responsibility for creating the threads and calling `CORBA::ORB::run()` or `CORBA::ORB::perform_work()` on the same ORB instance from each thread. Connections are established and requests are dispatched onto threads in the pool by the thread-pool reactor, which is described in 15.3.11.3. The thread-pool reactor allows multiple threads to handle events concurrently. If a thread is available to handle an event, then the event is dispatched right away; otherwise, it is held until a thread becomes ready.



The thread-pool reactor implements the *Leader/Followers* architectural pattern described in *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* (POSA2), by Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. The leader-followers model allows multiple threads to share a single reactor. One thread becomes the "leader" and runs the reactor's event loop. The leader thread receives and processes the next event, such as an incoming client connection or inbound request. Other threads in the pool are "followers." As soon as the leader thread receives an event, one of the follower threads becomes the new leader. When a thread finishes processing an event, it rejoins the leader-followers group and waits to become the leader thread again.

Since TAO uses the thread-pool reactor by default, any TAO server can be easily configured to use a thread pool by simply spawning a number of threads and calling `CORBA::ORB::run()` on the same ORB instance in each thread.

Figure 15-7 shows a server using the thread-pool concurrency model. In this server, four threads have been spawned by the application and made available to the ORB for request processing.

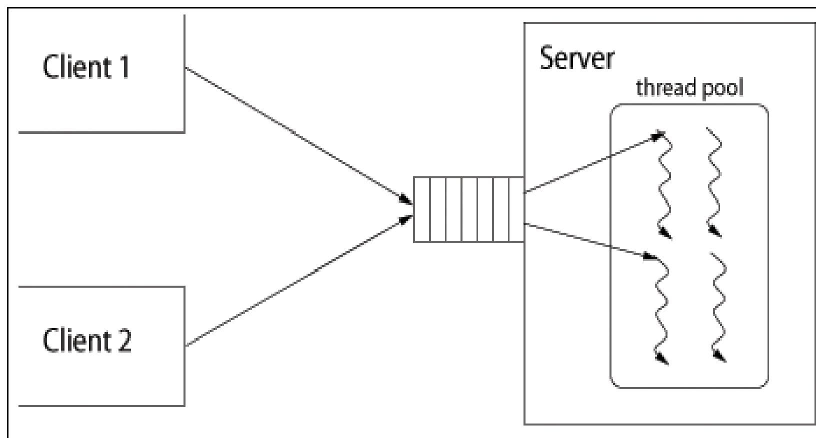


Figure 15-7 Thread-pool concurrency model with four threads

Motivation

The thread-pool concurrency model is useful in the following situations:

- Multiple client connections can be active simultaneously.
- Multiple threads can be dedicated to processing requests.

- The number of active client connections can be large, thereby making the thread-per-connection model undesirable due to the large number of threads that would be spawned by the ORB.
- Multiple concurrent requests may be received on a single connection (e.g., from a multithreaded client that is multiplexing requests on a single connection) and need to be processed concurrently.
- Request processing is computationally intense.

Configuration

Since TAO uses the thread-pool reactor and the reactive concurrency model by default, no special configuration options are required.

Consequences

The consequences of using the thread-pool concurrency model are as follows:

- The total number of threads used for processing requests is bounded and can be controlled by the server developer.
- The server developer must take responsibility for creating threads and running the ORB's event loop in each thread.
- Requests are processed concurrently, regardless of the connection on which they arrive.
- Requests from one client will not be blocked by requests from another client.
- Subsequent requests from one client connection will not necessarily be blocked waiting for processing of a previous request on that connection to complete.
- Since requests may be processed concurrently, developers may have to protect critical regions of code with locks.
- Since there is no way to predict which thread in the pool will be used to process a given request, all threads in the pool should be run at the same priority. Applications that need to process different requests at different priorities should use the features provided by the RT CORBA thread-pool model. See 8.3.7 for more information.
- Scalability is improved, especially for servers with large numbers of clients, not all of which are sending requests simultaneously.

Example

The following example shows a typical multithreaded server using the thread-pool concurrency model. Since the application is responsible for creating threads, application code changes are required to use the thread-pool model. In addition, the application's servant implementations may need to provide locking around critical regions of code.

The server source code for this example is based on the Messenger server from Chapter 3. The main thread creates multiple threads and dedicates each to processing requests by calling `CORBA::ORB::run()`. The ORB dispatches requests onto these threads using the ACE thread-pool reactor. The server's `main()` function is implemented in the file `MessengerServer.cpp`. The source code for this example is in the `$TAO_ROOT/DevGuideExamples/Multithreading/ThreadPool` directory.

```
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

// 1. Define a "task" class for implenting the thread-pool threads.
#include <ace/Task.h>

class ORB_Task : public ACE_Task_Base
{
public:
  ORB_Task (CORBA::ORB_ptr orb)
  : orb_(CORBA::ORB::_duplicate(orb)) { }
  virtual ~ORB_Task () { }
  virtual int svc ()
  {
    this->orb_->run();
    return 0;
  }
private:
  CORBA::ORB_var orb_;
};

// 2. Establish the number of threads.
static const int nthreads = 4;

int main(int argc, char* argv[])
{
  try {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

```
    // Get a reference to the RootPOA.
    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

    // Activate the POAManager.
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Create a servant.
    PortableServer::Servant_var<Messenger_i> messenger_servant =
      new Messenger_i();

    // Register the servant with the RootPOA, obtain its object
    // reference, stringify it, and write it to a file.
    PortableServer::ObjectId_var oid =
      poa->activate_object(messenger_servant.in());
    CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
    CORBA::String_var str = orb->object_to_string(messenger_obj.in());
    std::ofstream iorFile("Messenger.ior");
    iorFile << str << std::endl;
    iorFile.close();
    std::cout << "IOR written to file Messenger.ior" << std::endl;

    // 3. Create and activate threads for the thread pool.
    ORB_Task task (orb.in());
    int retval = task.activate (THR_NEW_LWP | THR_JOINABLE, nthreads);
    if (retval != 0) {
      std::cerr << "Failed to activate " << nthreads << " threads." << std::endl;
      return 1;
    }

    // 4. Wait for threads to finish.
    task.wait();

    // Clean up.
    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "CORBA exception: " << ex << std::endl;
    return 1;
  }

  return 0;
}
```

The code sections numbered 1-4 in the above example have been added to provide a thread pool for request processing by the ORB. Each step is further described here:

1. *Define a "task" class for implementing the thread-pool threads.*

   Threads for the thread pool can be created using the platform's normal threading APIs (e.g., `pthread_create()` on platforms that support POSIX threads). However, ACE provides an object-oriented abstraction for threads known as *task*. In this section of the code, we define a new task type, called `ORB_Task`, that is dedicated to running the ORB's event loop. We can initialize an `ORB_Task` with an object reference to the ORB, then activate several threads. Each thread enters the task's `svc()` method, in which we simply call `run()` on the ORB to make that thread available to the ORB for handling connections and requests. If the ORB is shut down, then all the threads exit the ORB's event loop, causing each `svc()` method to return and each thread to exit.

2. *Establish the number of threads.*

   We define a constant to represent the number of threads we want in the thread pool. With a few simple code changes, this value could be provided as a command-line parameter to allow the number of threads in the thread pool to be dynamically configured at run time.

3. *Create and activate threads for the thread pool.*

   In this section of the code, we create an instance of our `ORB_Task` class, then create a number of threads using the task's `activate()` method. Each new thread invokes the task's `svc()` method. The thread creation flags are a bitwise `OR` of the values `THR_NEW_LWP` and `THR_JOINABLE`. The `THR_NEW_LWP` flag means a new light-weight process (LWP) is created for each new thread to increase the concurrency level of the thread pool. `THR_JOINABLE` means we can later join these threads and wait for each one to exit using the task's `wait()` method.

4. *Wait for threads to finish.*

   Since all ORB request processing is taking place in the threads activated via the `ORB_Task`, the main thread continues. To keep it from exiting (and possibly causing the process to exit), we wait for all the threads in the pool to exit. After the last thread in the thread pool has exited, we can proceed to destroy the ORB and clean up resources.

Note *More information on the ACE Task Framework can be found in Chapter 6 of C++ Network Programming: Volume 2 (C++NPv2) and Chapter 15 of The ACE Programmer's Guide (APG).*

15.3.7 Dynamic Thread Pool Strategy

The Dynamic thread pool strategy provides flexibility in managing a thread pool for an ORB or for individual or groups of POAs. Dynamic thread pools may be configured by an API at run time or may be externally configured using the service configurator framework. The dynamic thread pools described in this section are similar to those defined by the RT-CORBA thread pool specification, except that there is no way to set priority. If you are familiar with RT-CORBA's specification, TAO's dynamic thread pool strategy defines a single lane when applied to an ORB.

15.3.7.1 Dynamic Thread Pool Settings

Dynamic thread pool configuration provides control over attributes such as an initial number of threads, minimum and maximum numbers of threads, the amount of time a thread must be idle before it is removed from the pool, and how much stack should be allocated to each running thread.

It is possible to define multiple thread pools within a single application to accommodate multiple ORBs or multiple POAs. Once defined thread pools are then attached to the ORB or POA(s) that use them. The values described here are used to configure any dynamic thread pools. This section focuses on the thread pool itself, applying the thread pool to an ORB or a POA is discussed in later sections.

Minimum Pool Threads - the default is -1 indicating no minimum. The minimum thread count controls the expiration of idle threads. If the minimum thread count is left at -1, threads in the pool will never expire, and eventually the pool will grow to reach its maximum number of threads. Set the minimum number of threads to a positive number less than the maximum number of threads to have idle threads time out and release resources down to this minimum number.

Maximum Pool Threads - the default is -1 meaning it is effectively unlimited. Setting the maximum is to limit the number of concurrent requests that can be handled. Reaching this limit has different side effects depending on whether the thread pool is applied to an ORB or a POA. On a POA, a

request received that cannot be dispatched is queued based on the constraints imposed by the Request Queue Depth setting described below. On an ORB, a dynamic thread pool at the maximum thread count behaves the same as a fixed thread pool, recursively handling incoming events as needed.

Initial Pool Threads - the default is 5 if the minimum count is not set. The default will not exceed a maximum if one is set. This setting controls the number of threads allocated when the pool is applied to an ORB or a POA. The initial thread count should be between the minimum and maximum thread counts. These initial threads are subject to the same lifespan constraints as any dynamically allocated thread, expiring on idle time out if necessary.

Thread Stack Size - the default is `ACE_DEFAULT_THREAD_STACKSIZE` bytes. On most systems this macro is the default stack size per thread set in the kernel of the operating system. This is equivalent to setting this value to 0. Any override of this default should be the number of bytes desired per thread on the stack.

Thread Idle Time (timeout) - the default is 60 (seconds). This value determines how long a thread should be allowed to be idle before it exits and is removed from the pool. Any use of the thread within the timeout period resets the timeout when it returns to a non-busy state and is available for another request. An important exception to this behavior is when the Minimum Pool Threads (see above) is set to -1. In this case the Thread Idle Time is ignored altogether and any thread initialized in the pool at startup or added to the pool after startup will live until the process is shut down.

Request Queue Depth - the default is 0 (requests) meaning that the queue is unbounded. This value only affects thread pools applied to POAs. A POA thread pool uses a queue to hold incoming requests when there are no threads available at that time to handle the request. An unbounded queue will continue to cache requests until it runs out of memory. Setting this value to a non-zero request count will cause the queue to reject new requests, instead raising a CORBA::TRANSIENT exception with a minor code of DISCARDING to the client.

15.3.7.2 Initializing a Dynamic Thread Pool via API

For an application specified thread pool, you must first include the Dynamic Thread Pool library in your project settings. This is easily done using MPC by adding “`dynamic_tp`” to your project’s dependency list. If you are not using

MPC to manage your project build files, the library you need to link to is TAO_Dynamic_TP, with the appropriate platform-specific modifiers. Additional libraries are required to support applying the thread pool to POAs.

The configuration values described above are encapsulated in a structure, which is used to initialize strategy objects. The TAO_DTP_Definition structure is defined along with the constructor defaults.

```
struct TAO_DTP_Definition
{
  int min_threads_;
  int init_threads_;
  int max_threads_;
  size_t stack_size_;
  ACE_Time_Value timeout_;
  int queue_depth_;

  // Create explicit constructor
  TAO_DTP_Definition() :
    min_threads_(-1),
    init_threads_(5),
    max_threads_(-1),
    stack_size_(ACE_DEFAULT_THREAD_STACKSIZE),
    timeout_(60,0),
    queue_depth_(0)
{}
```

As this structure is only used to hold sets of values, you are free to create one on the stack, and reuse it as needed, neither the ORB nor POA thread pool strategy objects retain ownership of a supplied definition structure. Usage of an initialized TAO_DTP_Definition is shown below.

15.3.7.3 Initializing a Dynamic Thread Pool via Configuration

Dynamic thread pool definitions may be configured outside of the application, giving a measure of control to users of the application. As we will show later, externally supplied definitions may be used by application code, or by externally defined ORB or POA thread pool strategy. As with any service objects, the dynamic thread pool objects may be statically or dynamically linked. Dynamic linking requires no prior application configuration, so it is shown here. Using the service object to supply a thread pool definition takes one additional parameter not required through the API. That is a name, since multiple thread pool definitions may be supplied.

In order for the service configurator to be able to manage thread pool definition objects, a service object wrapper known as `DTP_Config` is used. This service object is part of the TAO Dynamic TP library and has parameters that match the `TAO_DTP_Definition` members. A mapping of parameters to definition structure members is shown in Table 15-1 below. Here are a couple of example configurations.

```
dynamic DTP_Config Service_Object * TAO_Dynamic_TP:_make_TAO_DTP_Config()
"-DTPName POA1 -DTPMin 3 -DTPInit 5 -DTPMax 7 -DTPQueue 100"

static DTP_Config "-DTPName ORBTP -DTPMin 1 -DTPTimeout 10 -DTPStack 256"
```

:The first example shows a dynamically loaded `DTP_Config` object named “POA1” which starts out with 5 threads and floats between 3 and 7. When the limit of 7 threads is reached, up to 100 pending requests will be queued before requests start getting rejected.

The second example shows a statically linked `DTP_Config` object named “ORBTP” with a minimum of 1 thread that is allowed to grow unbounded, but idle threads are reaped after 10 seconds. The threads are given 256 KB stacks too.

Table 15-1 DTP_Config parameters to TAO_DTP_Definition members

DTP_Config parameters	TAO_DTP_Definition member
DTPName	Name of the configuration set, not defined in Definition structure.
DTPMin	min_threads_
DTPInit	init_threads_
DTPMax	max_threads_
DTPTimeout	timeout_
DTPStack	stack_size_
DTPQueue	queue_depth_

Note that there is no special meaning for names supplied to `DTP_Config` objects. As shown below, the name is used to identify a particular configuration set is applied to a POA or an ORB.

15.3.8 Applying Dynamic Thread Pools to an ORB

Now that you know how to configure a dynamic thread pool, now you will see how to apply it to an ORB. Recall that an application may have multiple ORBs, each may have a thread pool, ORB thread pools cannot be shared. The dynamic thread pool configuration enhances the Thread Pool concurrency

model described in section 15.3.6.3 above. The steps involved are to first define a `DTP_Config` object in a service configuration file, then reference it by supplying `-ORBDynamicThreadPoolName <DTP_Name>` to the ORB via `ORB_init()`. The name you supply matches the name of the desired configuration object. Although thread pool instances are not shared, definitions can be. As long as multiple ORBs have access to the same service configuration gestalt (see sections 17.3 and 17.13.29) the same `DTP_Config` can be used by each to define its own dynamic thread pool.

An important side effect of using dynamic thread pools is that the thread pool is activated during `ORB_init()` and does not require a call to ORB::run(). An application may call ORB::run() which would contribute the main thread to the thread pool, however based on the thread pool definition, this additional thread may time out and exit the run loop which would return it to main, possibly terminating the process. To avoid that fate, you must obtain a reference to the thread pool object and wait on that. If that is the only source of threads in the application, you can also achieve the same result waiting on the global `ACE_Thread_Manager` object. The following example shows how to wait on the dynamic thread pool rather than waiting on the ORB.

```
#include "tao/ORB_Core_TSS_Resources.h"
#include "tao/ORB_Core.h"
#include "tao/Dynamic_TP/DTP_Thread_Pool.h"
//...
   TAO_ORB_Core_TSS_Resources &tss = *orb->orb_core ()->get_tss_resources ();
   TAO_DTP_Thread_Pool *pool = static_cast <TAO_DTP_Thread_Pool *> (tss.lane_);
   if (pool == 0) {
     orb->run ();
   }
   else {
     pool->wait();
   }
```

First you include the header for the dynamic thread pool object. Then at the point where you would logically call `ORB::run()` you get a reference to the thread pool through the ORB core. A thorough discussion of the API used above is beyond the scope of this text, but here is a brief description. The TAO ORB interface has a hook to get a reference to its implementation core, which then has a helper object for managing resources. The “`lane_`” resource is a `void *` pointer to whatever manages thread lanes for the ORB. If this can be cast to a dynamic thread pool reference, then you want to wait on that. If it cannot, that indicates the dynamic thread pool was not initialized, and

therefore waiting should be done on the ORB as usual. Or perhaps you can report an error and terminate.

15.3.9 Custom Servant Dispatching

TAO's default servant dispatching strategy simply invokes the appropriate servant in the thread context that processed the incoming request message. The other threading models and strategies discussed in this section determine which thread processes a given incoming request.

TAO's Custom Servant Dispatching (CSD) framework allows application developers to define their own dispatching strategy and dispatch the request as appropriate to that application. The source code for the CSD framework is located in the `$TAO_ROOT/tao/CSD_Framework` directory. TAO also includes a reference strategy implementation, the Thread Pool CSD Strategy in the `$TAO_ROOT/tao/CSD_ThreadPool` directory.

Applications define their own CSD strategy by deriving a strategy class from the `TAO::CSD::Strategy_Base` class, creating instances of these strategy objects, and attaching the strategy object to the POA that they want to use that strategy. Any subsequent requests processed by that POA are then dispatched to the associated CSD strategy object. The CSD strategy object is then free to pass the request to whichever thread it wants. For example, TAO's Thread Pool CSD Strategy queues each request for processing by a pool of worker threads.

For more information about defining your own CSD strategy, the source code for the Thread Pool CSD Strategy provides a complete example of what needs to be done and may provide a starting point for other implementations.

For information about using the Thread Pool CSD Strategy, see the examples and tests under `$TAO_ROOT/examples/CSD_Strategy` and `$TAO_ROOT/tests/CSD_Strategy_Tests`.

15.3.10 Dynamic Thread Pooling in POAs

The Custom Servant Dispatch framework serves as the basis for implementing dynamic thread pool capabilities for POAs in TAO. The CSD base framework employs an interceptor point in the core that enables the association of created POAs with strategies that are developed externally from the core TAO libraries. These strategies can control the behavior of invocations to the

servants assigned to the POA. The Dynamic Thread Pool POA Strategy is one such strategy that is discussed here.

15.3.10.1 Dynamic Thread Pool POA Strategy

The Dynamic Thread Pool POA Strategy controls two different resources when associated with one or more POAs as shown in Figure 15-8. The first resource is a request queue that is used to accept calls coming in from the ORB being directed to an associated Servant. A second resource is a pool of threads that have been created by the instantiation of the strategy that sit waiting (listening) on the queue ready to service calls coming in on the queue.

Figure 15-8Dynamic Thread Pool POA Strategy

The Dynamic Thread Pool POA Strategy allows for the control of the request queue depth allowable by the application programmer for any associated POA. Likewise the strategy controls several attributes of the thread pool (discussed later) the allows the programmer to tailor the dynamics of how the application should govern execution resources. When a thread is signaled to handle a request coming in on the queue it dispatches the request to the appropriate servant specified in the request.

Dynamic Thread Pool Configuration

Each Dynamic Thread Pool POA Strategy instance must be configured with a set of attributes that modify its behavior when dispatching requests to one or more servants. An instance of a `TAO_DTP_POA_Strategy` object contains:

- a request queue that is used to queue requests coming from the TAO ORB and
- a thread pool containing threads that will listen for requests arriving at the request queue.

A Dynamic Thread Pool Configuration holds attributes that control both the request queue and the thread pool. The following rules apply to the configuration of a strategy.

Using the Dynamic Thread Pool POA Strategy in Your Application

The general flow for implementing dynamic thread pooling for POAs in an application is as follows:

1. Create each POA
2. Create a Dynamic Thread Pool Configuration
3. Create a Dynamic Thread Pool Strategy using this configuration
4. Associate the Dynamic Thread Pool Strategy with one or more POAs

Creating and configuring a dynamic thread pool can be performed using two methods:

1. Manual creation in an application
2. Dynamic creation using a svc.conf entry as described in Chapter 16.

The primary purpose of each method is to create the configured strategy and associate it with the targeted POA(s).

Method 1: Manually Creating a Dynamic Thread Pool POA Strategy

An application using the manual method must add the following include:

```
#include "tao/Dynamic_TP/DTP_POA_Strategy.h"
```

Below is an excerpt demonstrating the remainder of the method. This example applies the features of a dynamic thread pool in lines 15 - 32 with the remainder of the application being standard TAO CORBA implementation. A description of pertinent elements follows.

```
1. int ACE_TMAIN (int argc, ACE_TCHAR *argv[])
2. {
```

```
3.   try {
4.     // Initialize the ORB.
5.     CORBA::ORB_var orb = CORBA::ORB_init( argc, argv );
6.
7.     if (parse_args (argc, argv) != 0)
8.         return 1;
9.
10.    //Get reference to the RootPOA.
11.    CORBA::Object_var obj = orb->resolve_initial_references( "RootPOA" );
12.    PortableServer::POA_var poa = PortableServer::POA::_narrow( obj.in() );
13.
14.    // Create a configuration structure and set the values
15.    TAO_DTP_Definition tp_config;
16.    tp_config.min_threads_ = 1;        // Set low water mark to 1 thread
17.    tp_config.init_threads_ = 3;       // Start 3 threads to start
18.    tp_config.max_threads_ = -1;       // Create threads as needed (no limit)
19.    tp_config.queue_depth_ = -1;       // Allow infinite queue depth
20.    tp_config.stack_size_ = (64 * 1024); // Set 64K thread stack size
21.    tp_config.timeout_ = ACE_Time_Value(30,0);  // Expire after 30 sec idle
22.
23.    // Create the dynamic thread pool servant dispatching strategy object
24.    TAO_DTP_POA_Strategy * dtp_strategy =
25.                new TAO_DTP_POA_Strategy(&tp_config, false);
26.    // Apply the strategy to a POA.
27.    if (dtp_strategy.apply_to(poa.in()) == false)
28.      {
29.        ACE_ERROR((LM_ERROR,
30.                   "Failed to apply CSD strategy to root poa.\n"));
31.        return -1;
32.      }
33.    // Activate the POAManager.
34.    PortableServer::POAManager_var mgr = poa->the_POAManager();
35.    mgr->activate();
36.      <...remainder of application not shown...>
```

Line 15 introduces the `TAO_DTP_Definition` structure which holds the configuration desired for the thread pool and the request queue used by the new strategy. Although each member of the structure is set in this example, it is important to note that this structure is implemented with a constructor to set defaults for each member element. Therefore it is not necessary to set all values of the structure if the defaults are suitable for the application.

The example sets the Minimum Pool Threads to 1 (line 16) with an Initial Pool Threads set to 3 (line 17). Because the Thread Idle Time (line 21) is set to 30 seconds, two of the three initial pool threads will expire after 30 seconds of idle unless they are used in a call. The low water mark of 1 for minimum pool threads will ensure at least 1 thread stays active in the pool. If the 3 threads all become active before the 30 second idle and since the Max Pool Threads is set

to -1 (line 18), the example application will see new threads created as needed with each having a 30 second idle timeout and a thread stack size of 64K as shown in line 20.

Now that a `TAO_DTP_Definition` is ready for use, a new `TAO_DTP_POA_Strategy` object is created (line 24-25) with a reference to the newly created configuration (`&tp_config`). This strategy is now ready to be used by one or more POAs. `TAO_DTP_POA_Strategy` has an `apply_to()` method that performs the association of a strategy instance with a POA as shown on line 27.

If an application creates more than one POA and the developer desires that a single dynamic thread pool configuration is shared among multiple POAs, then the application should call the `apply_to()` method to associate each additional POA with a single `TAO_DTP_POA_Strategy` instance making it a shared resource. Another instance of a `TAO_DTP_POA_Strategy` can be created and associated with POA(s) separately with its apply_to() method if the developer wants dedicated thread pool resources to be available to individual POAs rather than sharing resources. It should be noted that the association with the `apply_to()` method is somewhat lazy in that internal association occurs when the POA is activated as shown in line 35.

For reference, the `TAO_DTP_Definition` structure is described in detail in section 15.3.7.1.

Method 2: Dynamically Creating a Dynamic Thread Pool POA Strategy

The second method of applying a dynamic thread pool POA strategy can be executed without modifying a TAO application. The key to method 2 is in preparing a service configuration file to define a thread pool then associate it with one or more POAs. For example:

```
dynamic DTP_Config Service_Object * TAO_Dynamic_TP:_make_TAO_DTP_Config()
  "-DTPName POA1 -DTPMin 1 -DTPMax 5"
dynamic DTP_POA_Loader Service_Object *TAO_Dynamic_TP:_make_TAO_DTP_POA_Loader()
  "-DTPPOAConfigMap RootPOA,MyPOA2,MyPOA3:POA1"
```

The first entry defines the contents of a `DTP_Config` object (as described in section 15.3.7.3) with a minimum thread count of 1 and a maximum thread count of 5. This `DTP_Config` object is named “POA1” allowing it to be referred to by the second directive. That directive is defining a `DTP_POA_Loader` object, which is responsible for associating a thread pool with one or more POAs. The POA loader takes one parameter,

-`DTPPOAConfigMap`, which takes repeated arguments to define separate thread pool groups. The argument for this parameter is a comma separated list of POA names, then a colon (':'), then a name of a `DTP_Config` object. In the example shown above, The above example would assume that three POAs (RootPOA, MyPOA2, MyPOA3) would share the resources of a dynamic thread pool instance configured with a configuration entry named 'POA1'.

The following extension of the above example demonstrates how to configure an application with multiple POAs each with separate (dedicated) dynamic thread pools.

```
dynamic DTP_Config Service_Object * TAO_Dynamic_TP:_make_TAO_DTP_Config()
"-DTPName POA1 -DTPMin -1 -DTPInit 5 -DTPMax -1 -DTPTimeout 60 -DTPStack 0
-DTPQueue 0"
dynamic DTP_Config Service_Object * TAO_Dynamic_TP:_make_TAO_DTP_Config()
"-DTPName POA2 -DTPMin 1 -DTPInit 5 -DTPMax -1 -DTPTimeout 30 -DTPStack 0
-DTPQueue 0"
dynamic DTP_Config Service_Object * TAO_Dynamic_TP:_make_TAO_DTP_Config()
"-DTPName POA3 -DTPMin 8 -DTPInit 15 -DTPMax 30 -DTPTimeout 60 -DTPStack 0
-DTPQueue 0"
dynamic DTP_POA_Loader Service_Object *
TAO_Dynamic_TP:_make_TAO_DTP_POA_Loader() "-DTPPOAConfigMap RootPOA:POA1
MyPOA2:POA2 MyPOA3:POA3"
```

Each of the POAs in the last line are mapped to separate named configurations in the first three entries. As shown, the POA name and the POA configuration name is separated by a colon ' : ' again, but each pair is separated by a space.

Note *If you wish to see the Dynamic Thread Pool POA Strategy configuration values being used in your application, start your application with a command line parameter of -ORBDebugLevel 5 and your output will reflect the usage of the configuration.*

15.3.10.2 MPC Projects Configured for POA Dynamic Thread Pools

A Make Project Creator (MPC) project needs to import two configurations to enable your build project to use the described capability. In the below, the `csd_framework` and `dynamic_tp` configurations are included on the project line. This will configure your build environment to include the proper libraries.

```
project(*Server): taoserver, csd_framework, dynamic_tp {
  exename = MessengerServer
  after += *idl
  Source_Files {
    Messenger_i.cpp
```

```
    MessengerServer.cpp
  }
  Source_Files {
    MessengerC.cpp
    MessengerS.cpp
  }
  IDL_Files {
  }
}
```

15.3.11 Configuring TAO's Reactor

Under the covers, TAO uses a *reactor* to handle events from clients. A reactor is a object-oriented layer over an operating system's event handling functions.

The reactor separates the detection of events from the handling of those events. Applications may register event handlers with the reactor whereby they are associated with various handles or other sources of events (e.g., timers, signals). When an event occurs on one of these handles or other event sources, the reactor dispatches the event to the associated event handler. The ORB uses a reactor to accept new connections in a socket context and to respond to incoming and outgoing data.

Note *For more information on the reactor, see Chapters 3 and 4 of C++ Network Programming: Volume 2 (C++NPv2) and Chapter 7 of The ACE Programmer's Guide (APG).*

There are several different reactor implementation types available to the ORB. The Advanced Resource Factory's `-ORBReactorType` option allows run-time configuration of the type of reactor that the ORB uses to handle events. See 18.7.6 for more information on using this option. Here we discuss how the selection of reactor type affects the ORB's threading behavior. We discuss only the single-threaded select reactor, multithreaded select reactor, and thread-pool reactor implementations.

15.3.11.1 Single-threaded Select Reactor

As described in 15.3.6.115.3.6, single-threaded applications that are dedicated to receiving and processing CORBA requests can benefit from using the single-threaded select reactor type (`-ORBReactorType select_st`). Since

all request processing is performed on a single thread, these applications may also safely disable certain locks. The single-threaded select reactor uses the `select()` system call to monitor I/O handles for input and output availability. It dispatches all inbound client connections and requests onto a single thread. If the thread is busy, new connections and requests are blocked in the transport layer until the thread re-enters the reactor's event loop.

The thread re-enters the reactor's event loop when it has completed processing each connection or request. If the application makes an outbound (client role) request within the context of processing an inbound (server role) request, then it may enter the reactor's event loop to await its reply (depending upon the wait strategy); while waiting for its reply in this fashion, the thread may be used by the reactor to dispatch additional inbound requests or other events. See 15.4.4 for more information on client-side wait strategies.

Use the Advanced Resource Factory's `-ORBReactorType select_st` option to configure the single-threaded select reactor type, as shown here:

```
dynamic Advanced_Resource_Factory Service_Object *
TAO_Strategies:_make_TAO_Advanced_Resource_Factory () "-ORBReactorType
select_st"
```

The above directive should appear on one line in the application's service configurator file.

15.3.11.2 Multithreaded Select Reactor

Like the single-threaded select reactor, the multithreaded select reactor uses the `select()` system call to monitor I/O handles for input and output availability. However, this reactor type also employs locks to enforce synchronization between threads. Only one thread is allowed to enter the reactor's event loop at a time. When an event is dispatched onto a thread, that thread retains ownership of the reactor (i.e., holds the reactor's token) and another thread is not allowed to enter into the reactor's event loop until the owner thread finishes dispatching *all* active event handlers. While a thread holds the reactor token, other threads cannot get into `select()`. A single thread may recursively call the reactor's `handle_events()` method without deadlocking.

In summary, the `select_mt` reactor type, while safe to use in multithreaded servers, performs poorly compared to the thread-pool reactor due to its coarse level of synchronization. Use of the `select_mt` reactor will likely result in

more jittery performance compared to the thread-pool reactor. Multithreaded applications should use the thread-pool reactor, which is discussed below.

15.3.11.3 Thread-Pool Reactor

The default reactor type in TAO is the thread-pool (`tp`) reactor. The underlying implementation of the thread-pool reactor uses the `ACE_TP_Reactor` type, a specialization of the `ACE_Select_Reactor` that is designed to support thread-pool-based event dispatching. Since the select reactor receives events via the `select()` system call, only one thread can be blocked in the reactor's `handle_events()` method at a time. The thread-pool reactor overcomes this limitation by taking advantage of the fact that events reported by `select()` are held (e.g., in transport-level input buffers) if not acted upon immediately. The thread-pool reactor keeps track of which event handler was most recently activated, releases the reactor's internal lock (thereby allowing another thread to enter the event loop), then dispatches the event to the event handler outside the context of the lock.

To simplify synchronization within event handlers, the thread-pool reactor ensures that a given event handler cannot be called by multiple threads simultaneously. It does this by automatically suspending an activated event handler before making the upcall onto its event-handling code (e.g., the `handle_input()` method), then resuming the event handler after the upcall completes.

Note *The* `ACE_TP_Reactor` *implements the Leader/Followers architectural pattern described in Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (POSA2). More information on the ACE_TP_Reactor can also be found in Chapter 4 of C++ Network Programming: Volume 2 (C++NPv2).*

Synchronization among event processing threads in the thread-pool reactor is far more efficient than in the multithreaded select (`select_mt`) reactor. Any TAO server that uses the thread-pool concurrency model, as described in 15.3.615.3.6.3, should use the thread-pool reactor. Since the thread-pool reactor is the default reactor type, no explicit configuration options are necessary. However, the thread-pool reactor type can be specified explicitly via the Advanced Resource Factory's `-ORBReactorType tp` option, as shown here:

```
dynamic Advanced_Resource_Factory Service_Object *
TAO_Strategies:_make_TAO_Advanced_Resource_Factory () "-ORBReactorType tp"
```

The above directive should appear on one line in the application's service configurator file.

15.3.12 Flushing Replies to the Client

When a server thread sends a reply to a client, the reply message is marshaled for writing to the transport. The ORB uses a resource factory to obtain a strategy for flushing these messages. In some cases, messages prepared for writing may be queued and written to the transport at a later time. For more information on *flushing strategies*, please see the discussion in 15.4.3.

15.4 Multithreading in the Client

The client-side ORB has several configuration options that affect the client's threading behavior. In this section, we describe the effects of various configuration options under various conditions and provide guidelines on making appropriate configuration choices to meet your application's needs.

As described in 15.2, a thread may behave as a client (invokes requests, waits for replies) or a server (processes requests, sends replies). Under certain conditions, a thread may switch between the client and server roles. The discussion of concurrency in this section pertains to a thread in the role of a client.

In the remainder of this section, we describe options for controlling the following ORB strategies:

- *Establishing a connection to the server*

  When a client thread invokes a request on a remote CORBA object, the client-side ORB must have a transport-level connection to the target object's ORB over which to send the request message. TAO's *connect strategy* determines how the client thread waits for the connection to be completed.

- *Multiplexing requests on a connection*

When a client thread invokes a request on a remote CORBA object, the ORB may be able to reuse an existing connection or it may have to establish a new connection. It is possible configure TAO's *transport multiplexing strategy* to multiplex several simultaneous requests on a single connection.

- *Flushing requests to the server*

  When a client thread sends a request or a server thread sends a reply, the request or reply message is first marshaled for writing to the transport. Messages prepared for writing may be queued and written to the transport at a later time as determined by TAO's *flushing strategy*.

- *Waiting for a reply from the server*

  When a client thread invokes a synchronous request on a remote CORBA object, it must wait for the reply. The way in which the client thread waits for its reply and the activities the thread may perform in the meantime is determined by TAO's *wait strategy*.

- *Optimizing performance of collocated requests*

  When a client thread invokes an operation on a collocated CORBA object (i.e., an object that is implemented within the same address space as the client), the invocation may be optimized to bypass several layers of marshaling, networking, demultiplexing, demarshaling, and dispatching logic by configuring TAO's *collocation optimization*.

15.4.1 Establishing a Connection to the Server

When a client thread invokes a request on a remote CORBA object, the client-side ORB must have a transport-level connection to the target object's ORB over which to send the request message. The ORB may be able to reuse an existing connection, or it may have to establish a new connection. Establishing a new connection can be time-consuming. TAO's default client strategy factory provides a *connect strategy* that determines how the client thread waits for the connection to be completed.

The default client strategy factory provides three possible connect strategies:

- leader-follower (`lf`)

  In the leader-follower connect strategy, the connecting thread joins the leader-follower group of threads to be awakened when the connection has been completed.

While waiting for the connection to complete, the thread may become the leader and may be used to process inbound requests or other events. Thus, with the leader-follower connect strategy, it is possible for a client thread to temporarily switch to the server role.

TAO uses the leader-follower connect strategy by default.

- `reactive`

  The reactive connect strategy behaves similarly to the leader-follower connect strategy, except that the connecting thread uses the ORB's reactor instead of joining the leader-follower group of threads. It provides non-blocking connect behavior with less locking overhead than the leader-follower connect strategy.

  The thread registers an event handler with the reactor, then runs the reactor's event loop. The event handler is invoked by the reactor when the connection is complete. The reactive connect strategy should only be used in single-threaded or pure client applications.

- blocking (`blocked`)

  In the blocking connect strategy, the connecting thread simply blocks for the duration of the connection attempt. This option should only be used in multithreaded environments or in environments where network congestion is unlikely to be a factor.

The connect strategy can be configured at run time by setting the default client strategy factory's `-ORBConnectStrategy` option in the application's service configurator file. This option accepts one of three values: `lf` (leader-follower), `reactive`, or `blocked`. The default setting for this option is `lf`.

In the example below, we show the directive to select the `blocked` strategy.

```
static Client_Strategy_Factory "-ORBConnectStrategy blocked"
```

See 20.3.2 for more information on using the `-ORBConnectStrategy` option.

Note *The* `blocked` *connect strategy is required when you use the wait-on-read wait strategy (*`-ORBWaitStrategy rw`*). See 15.4.4.1 for more information on wait-on-read.*

15.4.2 Multiplexing Requests on a Connection

When a client thread invokes a request on a remote CORBA object, the client-side ORB must have a transport-level connection to the target object's ORB over which to send the request message. The ORB may be able to reuse an existing connection, or it may have to establish a new connection.

If a reply is expected, the request is said to be *pending* on the connection. Under some conditions, only one request may be pending on a connection at a time. For example, if the client is using the wait-on-read wait strategy as described in 15.4.4.1, there cannot be more than one pending request on a single connection. In other cases, it is possible to multiplex requests on a single connection. Multiplexing requests on a single connection can optimize utilization of connection-related resources between a client and a server.

The default client strategy factory provides two possible *transport multiplexing strategies*:

- `exclusive`

  An exclusive transport uses a connection to service a single request and receive a reply before making it available for another request. A multithreaded client using the exclusive transport strategy will open a new connection for each concurrent request.

- multiplexed (`muxed`)

  The multiplexed strategy allows multiple concurrent requests to share a single connection, using asynchronous callbacks to handle the distribution of the replies.

  TAO uses the multiplexed transport multiplexing strategy by default.

The transport multiplexing strategy can be configured at run time by setting the default client strategy factory's `-ORBTransportMuxStrategy` option in the application's service configurator file. This option accepts one of two values: `muxed` (requests can be multiplexed on a connection) or `exclusive` (each request requires exclusive access to a connection). The default setting for this option is `muxed`.

In the example below, we show the directive to select the exclusive transport multiplexing strategy.

```
static Client_Strategy_Factory "-ORBTransportMuxStrategy exclusive"
```

See 20.2.2 for more information on using the `-ORBTransportMuxStrategy` option.

Note *The* `exclusive` *transport multiplexing strategy is required when you use the wait-on-read wait strategy (*`-ORBWaitStrategy rw`*). See 15.4.4.1 for more information on wait-on-read.*

15.4.3 Flushing Requests to the Server

When a client thread sends a request or a server thread sends a reply, the request or reply message is first marshaled for writing to the transport. The ORB then uses a resource factory to obtain a strategy for flushing these messages. In some cases, messages prepared for writing may be queued and written to the transport at a later time, according to the *flushing strategy*. The default resource factory provides three possible message flushing strategies:

- leader-follower (`leader_follower`)

  In the leader-follower flushing strategy, the thread sending the message will first attempt a non-blocking write operation to the transport. If the write operation succeeds, the thread will continue. If the non-blocking write operation could not be completed, the message is queued and the thread joins the leader-follower group of threads to be awakened when it can flush its message to the transport.

  While waiting to flush queued messages, the thread may become the leader and may be used to process inbound requests or other events. Thus, with the leader-follower flushing strategy, it is possible for a client thread to temporarily switch to the server role, or for a server thread flushing a reply message to be used to process additional requests.

  TAO uses the leader-follower flushing strategy by default.

- `reactive`

  The reactive flushing strategy behaves similarly to the leader-follower flushing strategy, except that a sending thread uses the ORB's reactor instead of joining the leader-follower group of threads. It provides non-blocking flushing with less locking overhead than the leader-follower flushing strategy.

  First, a non-blocking write operation to the transport is attempted. If it is not successful, then the thread queues the message, registers an event

handler with the reactor, and runs the reactor's event loop. When writing to the transport is possible, the thread flushes its queued message.

The reactive flushing strategy should only be used in single-threaded or pure client applications.

- `blocking`

  In the blocking flushing strategy, a thread sending a message simply blocks until it has successfully written its message to the transport. While blocked waiting to write, the thread will not be available to the ORB for connection handling or request processing. The blocking flushing strategy should only be used in pure client applications or in environments where network congestion is unlikely to be a factor.

The flushing strategy can be configured at run time by setting the default resource factory's `-ORBFlushingStrategy` option in the application's service configurator file. This option accepts one of three values: `leader_follower`, `reactive`, or `blocking`. The default setting for this option is `leader_follower`.

In the example below, we show the directive to select the `blocking` strategy.

```
static Resource_Factory "-ORBFlushingStrategy blocking"
```

See 18.6.7 for more information on using the `-ORBFlushingStrategy` option.

Note *The* `blocking` *flushing strategy is required when you use the wait-on-read wait strategy (*`-ORBWaitStrategy rw`*). See 15.4.4.1 for more information on wait-on-read.*

15.4.4 Waiting for a Reply from the Server

When a client thread invokes a synchronous request on a remote CORBA object, it must wait for a reply. The way in which the client thread waits for its reply and what it might do in the meantime is determined by the application's *wait strategy*.

By default, TAO attempts to use all available threads efficiently. This includes client threads that have sent a request to a server and are waiting for a reply. While waiting, a client thread may be "borrowed" by the ORB to handle

incoming requests, performing a *nested upcall*. If an application acts in both the client and server roles, then nested upcalls are possible.

This “borrowing” of a client thread waiting for a reply has side effects that may be surprising. First, it is possible for even a single-threaded server to process more than one request at the same time. Second, any thread known by the ORB may play the role of a client thread or a server thread.

This is TAO’s default behavior for a number of reasons. First, it avoids many potential deadlock situations. For example, suppose a single-threaded client provides a callback object to a server. After the client sends a request to the server, the server sends a callback request back to the client. If the client's only thread is waiting for the reply and cannot handle incoming requests, then a deadlock occurs. TAO’s default behavior, on the other hand, allows the client ORB to process the callback request and send a reply to the server, freeing the server ORB to send its reply back to the client.

Second, this behavior gives the application complete control over thread creation and destruction. Some ORB implementations spawn a new thread to handle each incoming request. TAO does not do this.

A nested upcall happens in the context of an ORB. In a process with more than one ORB, a client thread that makes a request through an ORB is not asked to handle a nested upcalls for an object in another ORB.

This behavior is configurable. TAO provides several wait strategies affecting what a client thread might do while waiting for a reply. They are as follows:

- wait-on-read
- wait-on-reactor
- wait-on-leader-follower
- wait-on-leader-follower-no-upcall (*experimental*)

TAO uses the wait-on-leader-follower wait strategy by default.

15.4.4.1 Wait-on-Read Wait Strategy

In the *wait-on-read* wait strategy (also known as “receive-wait”), when a client thread invokes a synchronous request, it simply blocks waiting to read the reply message from the underlying transport. When the reply message arrives, the thread becomes unblocked and processing continues.

Figure 15-9 shows a client using the wait-on-read wait strategy

While a client thread is blocked waiting for its reply, it is not available to the ORB for handling incoming connections or requests. In some situations, this behavior can lead to deadlocks.

For example, in a hybrid application (that plays both server and client roles), if all threads are acting in the client role and are currently blocked waiting for replies, the application is unable to respond to incoming requests from other clients. If a client thread invokes a request on a server that is also using the

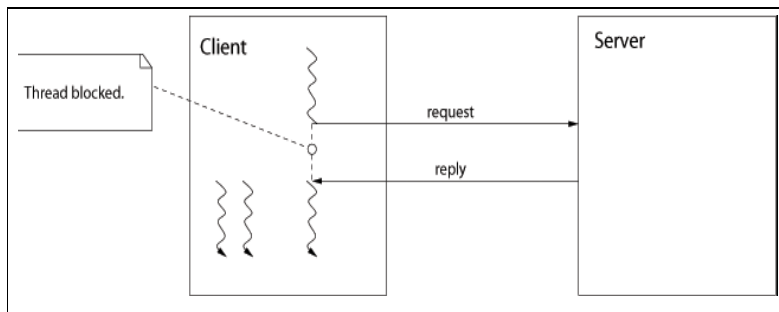


Figure 15-9 Client using the wait-on-read wait strategy

wait-on-read wait strategy, and the server thread that is handling that request switches to the client role and calls back to our application, threads in both

processes are now blocked waiting for their replies and neither can make progress. This situation is shown in Figure 15-10.

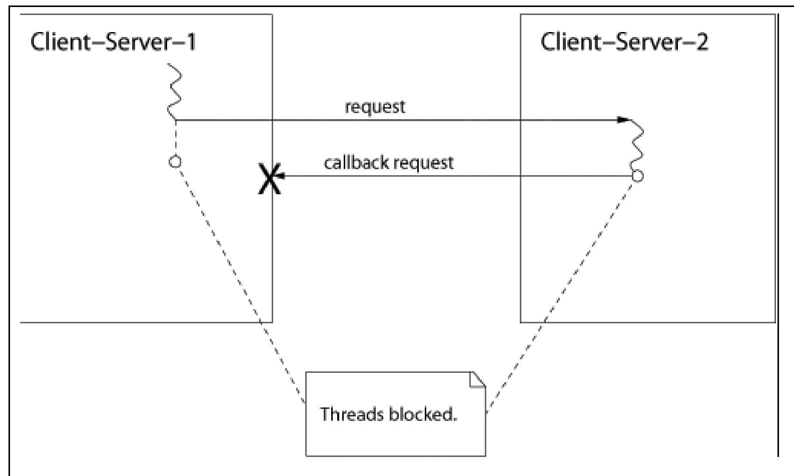


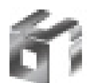
Figure 15-10 Deadlock created by using wait-on-read wait strategy

In the situation shown in Figure 15-10, the deadlock results because there are insufficient threads available to the ORB (in the process shown as “Client-Server-1”) for processing inbound requests that arrive while client threads are blocked waiting for replies. Applications can avoid this problem by making sure there is always at least one thread available to the ORB via `CORBA::ORB::run()` or `CORBA::ORB::perform_work()`. This can be accomplished using the thread-per-connection or thread-pool (with a sufficient number of threads in the pool) server concurrency models described in previous sections.

Motivation

The wait-on-read wait strategy is useful in the following situations:

- The application is a *pure* client application.
- All outbound request invocations are oneway requests with no replies.
- The application must ensure that client threads that are waiting on replies to outbound synchronous requests cannot be used by the ORB for processing inbound requests (i.e., nested upcalls cannot be allowed).
- Requests cannot be multiplexed across a single connection (i.e., the transport multiplexing strategy is configured as `exclusive`).



- Request messages must be completely written to the transport (i.e., the flushing strategy is configured as `blocking`).

Configuration

The wait-on-read wait strategy is specified by supplying the following configuration options to the default client strategy factory:

```
static Client_Strategy_Factory "-ORBWaitStrategy rw -ORBTransportMuxStrategy
exclusive -ORBConnectStrategy blocked"
static Resource_Factory "-ORBFlushingStrategy blocking"
```

Each of the above directives should appear on one line in the application's service configurator file.

See 20.3.8, 20.3.6, 20.3.2, and 18.6.7 for more information on using these options.

Note *The* `exclusive` *transport multiplexing strategy, the* `blocked` *connect strategy, and the* `blocking` *message flushing strategy are required when using the wait-on-read (*`rw`*) wait strategy.*

Consequences

The consequences of using the wait-on-read wait strategy are as follows:

- Client threads that are waiting for replies cannot be "stolen" by the ORB and used to process incoming requests; nor can they be used to handle other reactor events, such as timers.
- An application that uses the wait-on-read wait strategy may not use Asynchronous Method Invocation (AMI). Since the connection handler is not registered with the ORB's reactor, the ORB will not be able to dispatch the reply when it arrives.
- An application that uses the wait-on-read wait strategy may have a slightly smaller footprint than it would if it used the other wait strategies because there is no need for the ORB to create a reply dispatcher to handle the reply message. On the other hand, since requests cannot be multiplexed across connections, the application could experience an increase in footprint since multiple connections may be needed when sending concurrent requests to the same target server.

15.4.4.2 Wait-on-Reactor Wait Strategy

In the *wait-on-reactor* wait strategy (also known as "reactive"), a client thread waits for its reply by registering its connection handler as an event handler with the ORB's reactor, then running the reactor's event loop (`ACE_Reactor::handle_events()`) until the reply message arrives. When the reply is received, the connection handler is invoked by the reactor, the reply is read and demarshaled, and execution continues after the point of invocation.

Motivation

The wait-on-reactor wait strategy is useful in the following situations:

- The application is a *pure* client application.
- The application is single-threaded and acts as both a client and a server.
- Nested upcalls are possible.

Configuration

The wait-on-reactor wait strategy is specified by supplying the following configuration option to the default client strategy factory:

```
static Client_Strategy_Factory "-ORBWaitStrategy st"
```

See 20.3.8 for more information on using this option.

Consequences

The consequences of using the wait-on-reactor wait strategy are as follows:

- A single-threaded pure client application can still handle other reactor events (e.g., timers) while waiting for replies.
- In single-threaded applications, the wait-on-reactor wait strategy still allows the application to handle inbound requests while it is waiting for replies, but the locking overhead is less than with the default wait-on-leader-follower wait strategy.

15.4.4.3 Wait-on-Leader-Follower Wait Strategy

In the *wait-on-leader-follower* wait strategy (also known as "multithreaded reactive"), a client thread waits for its reply using the ORB's *leader-follower* model, based on the *Leader/Followers* architectural pattern described in

Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (POSA2), by Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann.

The leader-follower model allows multiple threads to share a single reactor. One thread becomes the “leader” and runs the reactor’s event loop. The leader thread receives and processes the next event; other threads are “followers.” As soon as the leader thread receives an event, one of the follower threads becomes the new leader.

In the wait-on-leader-follower wait strategy, it is possible for a client thread waiting for a reply to be used by the ORB to handle incoming requests or other events. Thus, nested upcalls are possible in this configuration. When a client thread begins waiting for its reply, it is likely to become a follower in the leader-followers group of threads. It is awakened when one of the following occurs:

- Its reply is received.
- It is selected to become the leader and run the ORB’s reactor’s event loop.

If the reply is received before the waiting thread becomes the leader, the thread remains in the client role and does not have to temporarily switch to the server role. However, if the application is experiencing heavy load or an insufficient number of other threads are dedicated to the ORB (by calling `CORBA::ORB::run()` or `CORBA::ORB::perform_work()`), the probability that the waiting thread will temporarily switch to the server role is increased.

When the reply to the outbound request arrives, the thread waiting on the reply eventually breaks out of the leader-followers group and returns to its client role.

Motivation

The wait-on-leader-follower wait strategy is useful in the following situations:

- The application is either single-threaded or multithreaded and acts as both a client and a server.
- Nested upcalls are possible.

Configuration

The wait-on-leader-follower wait strategy is the default wait strategy used in TAO. It can be specified explicitly by supplying the following configuration option to the default client strategy factory:

```
static Client_Strategy_Factory "-ORBWaitStrategy mt"
```

See 20.3.8 for more information on using this option.

The wait-on-leader-follower wait strategy should be used with the thread-pool reactor type. Since that is the default reactor type in TAO, no additional configuration options are necessary to select it. See 15.3.11 for more information on the thread-pool reactor. See 18.7.6 for more information on options to configure the ORB's reactor type.

Consequences

- In the wait-on-leader-follower wait strategy, it is possible for a client thread to be "stolen" by the ORB to handle inbound requests while waiting for a reply.
- A client thread should not hold a lock when invoking a request. If the thread is used to handle an inbound request while waiting for its reply, and the processing of that request results in an attempt to acquire the same lock, a deadlock could occur.
- Recursive nested upcalls may lead to unbounded stack growth, which will eventually crash the application.

  For example, consider two processes, A and B, both using the wait-on-leader-follower wait strategy. Suppose A invokes a synchronous operation on B, which results in B invoking a synchronous operation on A, resulting in a nested upcall. If that nested upcall results in another invocation on B, which results in another invocation on A, and so on, one or both processes will eventually exhaust all available stack space.

15.4.4.4 Wait-on-Leader-Follower-No-Upcall Wait Strategy

The *wait-on-leader-follower-no-upcall* wait strategy is an experimental strategy that combines features of the wait-on-read and wait-on-leader-follower wait strategies. A client thread waits for its reply using the ORB's *leader-follower* model, but no nested upcalls are permitted

on the client thread while it waits for its reply. The waiting thread may handle non-upcall reactor events such as time-outs and connection-related events. Other threads operate normally.

Note *The wait-on-leader-follower-no-upcall wait strategy is an **experimental** wait strategy. It has not been exercised in a wide variety of use cases. Please use caution and due-diligence in testing your application's behavior with this option if you decide to use it.*

The wait-on-leader-follower-no-upcall wait strategy was motivated by the need to recognize connections opened in the client role and closed by the server. Using wait-on-read, the closure is not recognized until another invocation is made through that connection. A closed wait-on-read connection sits in a `CLOSE_WAIT` state, consuming a file descriptor.

With this strategy, a client thread waiting for a reply enters the leader-followers group of threads just as it does in the wait-on-leader-followers wait strategy. When the client thread becomes the leader, the reactor may ask it to handle a nested upcall. The client thread, however, defers the nested upcall back to the reactor for processing by another thread. The client thread does handle other reactor events such as connection establishment and time-out events.

In the wait-on-leader-follower-no-upcall wait strategy, it is not possible for a client thread waiting for a reply to be used by the ORB to handle incoming requests. In other words, nested upcalls are not possible. However, the client has more flexibility in its selection of connect strategy, transport multiplexing strategy, and flushing strategy.

Motivation

The wait-on-leader-follower-no-upcall wait strategy is useful in the following situations:

- The application is a hybrid client/server application with predominately server-side activity.
- The application manages connections that are opened by the client and closed by the server.

Note *Other use cases have not been thoroughly tested.*

Configuration

The wait-on-leader-follower-no-upcall wait strategy can be specified by supplying the following configuration option to the default client strategy factory:

```
static Client_Strategy_Factory "-ORBWaitStrategy mt_noupcall"
```

See 20.3.8 for more information on using this option.

The wait-on-leader-follower-no-upcall wait strategy should *only* be used with the thread-pool reactor type. Since that is the default reactor type in TAO, no additional configuration options are necessary to select it. Note that no warnings are given if one tries to use this option with an incompatible reactor type. See 15.3.11 for more information on the thread-pool reactor. See 18.7.6 for more information on options to configure the ORB's reactor type.

Consequences

- A client thread that is waiting for a reply cannot be "stolen" by the ORB and used to process incoming requests; however, it can be used to handle other reactor events, such as time-outs and connection-related events.
- A deadlock is possible. Because a client thread cannot be "stolen" by the ORB to handle inbound requests while waiting for a reply, a thread-pool-based server may "run out of threads" if all of its threads are waiting for replies.
- The client thread is still called by the reactor to handle requests, but it defers the requests back to the reactor. Therefore, a client that waits a long time for a reply with lots of other inbound activity might result in significantly increased CPU usage.
- In situations where a high percentage of threads are waiting for replies and thus cannot handle inbound requests, it has been observed that the reactor may fail to re-dispatch a request deferred to it by the waiting client thread.
- The wait-on-leader-follower-no-upcall wait strategy has not been thoroughly tested with Asynchronous Method Invocation (AMI). The client thread may not necessarily handle an AMI callback, but instead may defer it to other threads.

- Unlike the wait-on-read strategy, connection establishment does not have to be blocking (i.e. the connect strategy does not have to be configured as `blocked`).
- Unlike the wait-on-read strategy, requests can be multiplexed across a single connection (i.e., the transport multiplexing strategy does not have to be configured as `exclusive`).
- Unlike the wait-on-read strategy, request messages do not have to be completely written to the transport (i.e., the flushing strategy does not have to be configured as `blocking`).

15.4.5 Optimizing Performance of Collocated Objects

When a client thread invokes an operation on a collocated CORBA object (i.e., an object that is implemented within the same address space as the client), the invocation may be optimized to bypass several layers of marshaling, networking, demultiplexing, demarshaling, and dispatching logic (see 17.13.4).

Since collocated invocations do not follow the usual ORB request dispatching path, the server logic (servant implementation code) is executed on the same thread as the client invocation; that is, the client thread is “stolen” by the server. For example, even if an application is using the thread-pool concurrency model described in 15.3.615.3.6.3, a request invoked on a collocated object will not be subject to dispatching via the ORB’s thread pool.

Usually, the collocation optimization is desirable. Optimization of collocated invocations can result in dramatically increased performance and decreased latency compared to normal or ORB-mediated invocations. However, in the case of a hybrid server/client application, if the collocation optimization is allowed, a client thread making an invocation on a collocated object cannot be made available to the ORB for handling incoming client connections or requests, regardless of the effective wait strategy or other strategies that normally affect client thread behavior. If other threads in the process are not available to the ORB, remote client connection attempts or requests could be blocked until a thread becomes available.

Also, in the case of a real-time CORBA application, the client thread may not be running at the priority at which the request should be processed, possibly leading to priority inversions.

Note *However, if the application is using RT CORBA, TAO uses a special real-time collocation resolver to determine whether the target object is collocated with the invoking thread. The real-time collocation resolver considers not only the address space and ORB of the target object, but also the POA, POA thread pool policy, thread pool id, thread pool lane, and effective priority model policy in making this determination. The result is that invocations on collocated objects in RT CORBA applications attempt to avoid priority inversions by not violating constraints imposed by priority and threading models. See Chapter 8 for more information on RT CORBA and 17.13.11 for information about an option to disable TAO's real-time collocation resolution mechanism.*

TAO provides an ORB initialization option to control the scope within which collocation optimizations are allowed. The `-ORBCollocation` option can have one of the following values:

- `global`

  Any object whose implementation resides in the same address space as the client is considered collocated. TAO uses `global` collocation by default.

- `per-orb`

  An object is considered collocated only if its implementation resides in the same address space as the client and the servant is accessed through the same ORB as that which the client is using to make the invocation. (This setting results in different behavior from `global` only in applications that use more than one ORB instance.)

- `no`

  The collocation optimization is disabled; invocations on objects implemented within the same address space as the client follow the same request dispatching path as invocations from remote clients.

See 17.13.4 for more information on using the `-ORBCollocation` option.

TAO also provides an ORB initialization option to specify the collocation strategy to use. Once the ORB has determined that the target of a request is collocated with the invoking client, the collocation strategy determines how the collocated invocation will be carried out. The `-ORBCollocationStrategy` can have one of the following values:

- `thru_poa`

Collocated invocations go through the POA and therefore respects the POA's current state and request demultiplexing and processing policies. TAO uses the `thru_poa` collocation strategy by default.

- `direct`

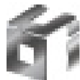
  Collocated invocations are carried out directly on the target servant, bypassing the POA completely. The servant must be in memory.

See 17.13.5 for more information on using the `-ORBCollocationStrategy` option.

The `-ORBAMICollocation` option can be used to disable collocation for only AMI-based invocations. See 17.13.2 for more information on using the `-ORBAMICollocation` option.

15.5 Summary

Multithreading is important to many distributed applications, including applications that use CORBA. This chapter has discussed several aspects of multithreading with CORBA and TAO, including application programming interfaces, policies, configuration options, and design choices. Multithreading issues have been discussed from both server and client viewpoints.



Part 3

Run-time Configuration of TAO

CHAPTER 16

Configuring TAO Clients and Servers

16.1 Introduction

Though it is possible to use TAO *out-of-the-box* as a general purpose ORB, many applications will have special needs that are not met by TAO's default configuration. For example, in some applications or operating environments, you may need to control the concurrency and locking strategies used when processing requests. In other cases, you may want to control how TAO behaves when a client invoking a request is collocated with the target object of the request. In addition, your operating environment may impose restrictions on your application's use of resources such as memory, communication protocols, buffers, and endpoints.

TAO's flexible and open architecture allows a high degree of run-time configuration for meeting the needs of a wide variety of applications and operating environments. TAO employs various design patterns to achieve this degree of flexibility.

16.1.1 Road Map

In this chapter, we introduce the concepts and techniques involved in configuring TAO at run time.

- In 16.2, we introduce the fundamental *patterns, components, and techniques* used for run-time configuration of TAO clients and servers.
- In 16.3, we introduce the *ACE Service Configurator* that TAO uses to dynamically configure its own components.
- In 16.4, we present *command line options* that control how TAO uses the service configurator.
- In 16.5, we describe details of using the *ACE Service Configurator framework*.
- In 16.6, we describe the ACE *XML Service Configurator*.
- In 16.7, we discuss *service objects*, including how to create your own *dynamically-loaded services*.
- In 16.8, we discuss the *ACE Service Manager*, which allows *remote administration* of the service configurator.

To fully understand and take advantage of the ACE Service Configurator, read Chapter 5 of *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks* (C++NPv2) and Chapter 19 of *The ACE Programmer's Guide (APG)*. See also the Component Configurator design pattern (75) in *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* (POSA2).

16.2 Patterns and Components for Configuring TAO Clients and Servers

In this section, we describe the primary design patterns and components TAO uses for run-time configuration.

16.2.1 Factories

TAO relies on *object factories* to construct the required strategy and resource objects. An object factory is similar to a traditional factory in the following ways:

- A traditional factory constructs and delivers products in response to the orders it receives, whereas an object factory constructs and delivers objects in response to requests.
- A traditional factory specializes in the products it constructs (e.g., an automobile factory may construct many varieties of cars and trucks, but will not construct bricks or children's toys), whereas an object factory specializes in constructing the types of objects defined by its interface.
- A traditional factory uses supplied materials to construct products, whereas an object factory uses supplied parameters to construct objects.

16.2.2 Strategies

A *strategy* defines a means of obtaining a goal. In a server, for instance, one goal is to match incoming requests with the servants responsible for handling those requests; this process is called *demultiplexing*, or *demux* for short. TAO defines a number of request demultiplexing strategies for accomplishing this goal, each with its own idiosyncrasies and side effects. A server strategy factory is an object factory used in TAO that can produce a request demultiplexing strategy. Using dynamic configuration, it is possible to change at run time the way TAO demultiplexes requests to servants with no impact on application code.

16.2.3 Resources

A *resource* is a tool or source of supply, such as a buffer for containing the product of converting data from one format to another or a reactor used to detect the presence of new requests from the input source. TAO uses a resource factory to produce resources used by the ORB at run time.

16.2.4 TAO's Default Strategy and Resource Factories

TAO organizes its strategies and resources into three groups, each group being served by one of the following default object factories:

- The *Resource Factory* controls creation of configurable resources used by the ORB core. Most of the resources required by the ORB core are fixed, but you have some flexibility in the choice of a reactor, the selection of communication protocols, and the behavior of CDR allocators.
- The *Server Strategy Factory* produces configurable elements utilized by the object adapter, such as request demultiplexing strategies.

- The *Client Strategy Factory* produces configurable elements that optimize object stub operations on the client side, such as concurrency within the client application, or multiple requests sharing a communication channel.

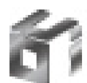
16.2.5 Specializing TAO's Factories

Factories define interfaces for obtaining objects, and implementations supply the object instances. The default implementations supplied with TAO permit run time tuning via options set in the *ACE Service Configurator*. Sophisticated users, or those with special concerns about issues such as consistency or footprint, can tune at compile- or link-time by implementing specialized factories. The interfaces to the various configuration factories, and the options controlling the default implementations of these factories, are provided in subsequent chapters.

16.3 The ACE Service Configurator

The ACE Service Configurator is a framework that supports the creation of dynamically-configured applications. Applications may be comprised of different components, depending on commands—known as *directives*—provided to the framework. Directives processed by the service configurator direct it to load or unload service objects, start and stop the execution of these objects, and configure the state of statically-linked objects. By default, the service configurator reads directives from a file called `svc.conf` that it finds in the current directory. It is possible to declare alternative configuration files on the application's command line, and even to supply directives directly to the service configurator. The service configurator provides additional services to the application, such as daemonizing a process or enabling the output of debug information from the ACE and TAO classes. Table 16-3 shows the ORB configuration options used to override the default behavior of the service configurator in TAO applications.

The pattern implemented by the ACE Service Configurator is described in detail in Chapter 5 of *C++ Network Programming, Volume 2* (C++NPv2) and Chapter 19 of *The ACE Programmer's Guide*.



16.3.1 Using the *static* Directive

The service configurator provides a means for altering the configuration of an application at run time. Without shutting down the application, new service objects may be added dynamically from a library, and existing service objects may be removed. The directives most important for configuring TAO are those that supply initialization options to the statically-linked default factories. The `static` directive supplies initialization to service objects that are statically linked. The following line shows the form of a `static` directive:

```
static service_name "options"
```

For example, the following directive could be used to supply options to TAO's statically-linked `Resource_Factory` initialization function at run time.

```
static Resource_Factory "options"
```

The components of the *static* directive are shown in Table 16-1.

Table 16-1 *Static* Directive Components

Component	Description
`static`	The service configurator directive. It must appear as shown.
`service_name`	The name by which the service object is registered with the service configurator. The name is `Resource_Factory` in the example above.
`options`	The list of options supplied to the service object's initialization function. The options must be quoted if more than one whitespace-separated option is to be supplied to the service object.

All service objects are registered with the service configurator by name. This name is used to locate the object either to supply it with initialization parameters or (with other directives) to control its state.

TAO's default factories register themselves with the service configurator by name as well. The default resource factory name is `Resource_Factory`, the default server strategy factory name is `Server_Strategy_Factory`, and the default client strategy factory name is `Client_Strategy_Factory`. You supply initialization options to these factories using the following directives:

```
static Resource_Factory "-ORBoption value ..."
static Client_Strategy_Factory "-ORBoption value ..."
static Server_Strategy_Factory "-ORBoption value ..."
```

Note *When you use the default configuration factories, you only need to supply directives to the service configurator if you want to apply option values other than the defaults. The default configuration factories properly initialize without the use of directives.*

16.3.2 Using the *dynamic* Directive

When developing an application, using the dynamic configuration capabilities of the service configurator, you use the `dynamic` directive to trigger the loading and initialization of service objects from libraries. The following line shows the form of the `dynamic` directive:

```
dynamic service_name base_object_type library:factory_function() "options"
```

For example, to replace the default resource factory with one of your own defined in `libmylib.so` (on Unix) or `mylib.dll` (on Win32), the directive is:

```
dynamic Resource_Factory Service_Object * mylib:_make_resources() "param ..."
```

The components of the *dynamic* directive are shown in Table 16-2.

Table 16-2 *Dynamic* Directive Components

Component	Description
`dynamic`	The service configurator directive. It must appear as shown.
`service_name`	The name used to register the newly-created service object with the service configurator. In the example above, the name is `Resource_Factory`.
`base_object_type`	The basic type returned by the factory function. For general service objects, use `Service_Object*`.

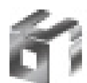


Table 16-2 *Dynamic* Directive Components

Component	Description
library	The name of the library containing the code for the service object. The form of the library name is the same as the name supplied to the compiler/linker via the `-l` option. Similarly, the library must reside in a directory accessible to the application (e.g, in `LD_LIBRARY_PATH` on Unix and in `PATH` on Windows). In the example above, the library name is `mylib` and the actual library would be `libmylib.so` on Solaris or `mylib.dll` on Windows.
factory_function()	The name of the function called by the service configurator to create a new instance of the service object. It must conform to a particular interface, described in 16.7.3. Supplied macros simplify the creation of the factory function. In the example above, the function name is `_make_resources()`.
options	The list of options supplied to the newly-created service object's initialization function. The options must be quoted if more than one whitespace- separated option is to be supplied to the service object.

TAO does not require the use of any dynamically-loaded service objects. However, it is possible that you may use them because of application requirements, or merely personal or corporate preference. For example, the advanced resource factory, which allows configuration of advanced options, is often loaded dynamically. (See 18.5 for information on using the advanced resource factory.) You may also want to use alternative communications protocols. These alternative protocols may be supplied by creating new pluggable protocol factories, then loading them as dynamic service objects. The use of alternative protocols is explored in greater detail in Chapter 14.

16.4 Service Configurator Control Options

Table 16-3 lists options that influence the behavior of the service configurator used in TAO. These options duplicate many of the common options available to applications built using the Service Configurator pattern. If a client or server application is based on this pattern, these options are redundant.

Table 16-3 Service Configurator ORB Initialization Options

Option	Section	Description
`-ORBDaemon`	17.13.7	Instruct the process to run as a daemon.

Table 16-3 Service Configurator ORB Initialization Options

Option	Section	Description
`-ORBSvcConf` *`config_file_name`*	17.13.63	Specify an alternate service configurator file name.
`-ORBSvcConfDirective` *`directive`*	17.13.64	Pass a single directive directly to the service configurator.
`-ORBServiceConfigLoggerKey` *`logger_key`*	17.13.58	Specify where to write ORB logging output.
`-ORBSkipServiceConfigOpen`	17.13.60	Skips loading of a service configurator file for this ORB.

16.5 The ACE Service Configurator Framework

This section presents details of the ACE Service Configurator framework. It is not necessary to explicitly use this framework when developing TAO-based applications. However, the framework is useful when developing modular applications that can benefit from dynamic configuration.

The ACE Service Configurator framework allows decisions about system configuration to be deferred until the deployment phase of a project, rather than during the design phase. Deployment personnel configure service objects into processes and assemble a running system. Processes may contain many of these service objects, or the service objects may be spread across several processing elements; the choice is made at deployment time. Service objects may be active (running in their own thread) or reactive (registering with a reactor to handle events), and a process may contain any combination thereof.

To successfully deploy a system based on the service configurator framework, the following two fundamental goals must be achieved:

- Service objects must fit into the framework (see 16.7).
- Service objects must be loaded into the system (see 16.5.1).

16.5.1 Loading Service Objects

The service configurator loads services for you by initializing an `ACE_Service_Config` object. Arguments are passed to the service configurator using an array of strings, similar to the way command line arguments are passed to the main function of an application. A very common

use case for the service configurator is to simply pass the `argc` and `argv` parameters of `main()` to the service configurator's `open()` function. The example below shows a generic main program that loads services, then runs them:

```
#include <ace/Service_Config.h>

int main(int argc, char** argv)
{
  // open() reads svc.conf, creates services, initializes
  ACE_Service_Config configurator;
  configurator.open(argc,argv);

  // run event loop to handle client requests
  ACE_Reactor::instance()->run_event_loop();

  // a multithreaded program can use
  // ACE_Thread_Manager::instance()->wait();
}
```

In the above example, the service configurator is opened, using the program's command line arguments as initialization parameters. The service configurator locates a file containing directives, then loads and initializes service objects as indicated by these directives. In 16.4, we describe the ORB initialization options that provide some control over the service configurator. Table 16-4 shows the arguments processed by the service configurator in its `open()` function.

Table 16-4 Service Configurator Command Line Arguments

Argument	Description
`-f` *`file`*	Supply a file containing directives other than the default `svc.conf`. Multiple `-f` arguments may be supplied to chain configuration files.
`-s` *`signal`*	Supply an alternative signal for causing the service configurator to reprocess the directives file. By default, `SIGHUP` is used.
`-d`	Turn on debugging in ACE and TAO objects.
`-n`	Ignore static directives.
`-y`	Process static directives. The default behavior. It overrides `-n`.
`-b`	Turn the containing process into a daemon.
`-S` *`directive`*	Supply a directive to the service configurator directly. The `-S` argument may be repeated to supply multiple directives to the service configurator.

Table 16-4 Service Configurator Command Line Arguments

Argument	Description
`-k` *logger_key*	Specify where to write ORB logging output.

Synopsis In addition to the `open()` function, the service configurator provides some additional functions as part of a control interface. Though not complete, the class definition below shows the useful functions available to application developers. Other functions in the public interface to `ACE_Service_Config` are for use by other elements of the framework:

```
class ACE_Export ACE_Service_Config
{
public:
   static int open (int argc,
                    ACE_TCHAR *argv[],
                    const ACE_TCHAR *logger_key = ACE_DEFAULT_LOGGER_KEY,
                    int ignore_static_svcs = 1,
                    int ignore_default_svc_conf_file = 0,
                    int ignore_debug_flag = 0);
   static int open (const ACE_TCHAR program_name[],
                    const ACE_TCHAR *logger_key = ACE_DEFAULT_LOGGER_KEY,
                    int ignore_static_svcs = 1,
                    int ignore_default_svc_conf_file = 0,
                    int ignore_debug_flag = 0);
   static int close (void);
   static void reconfigure (void);
   static int process_directive (const ACE_TCHAR directive[]);
}
```

16.5.2 Opening and Closing the Service Configurator

There are two forms of the `open()` function. The first accepts a list of command line arguments. The second accepts only a program name. For both forms, the remaining parameters are optional. Table 16-5 shows the arguments used by these two functions.

Table 16-5 `ACE_Service_Config::open()` Arguments

Argument	Type	Description
argc	`int`	Number of arguments in the `argv` list. Usually the same value supplied as `argc` to `main()`.

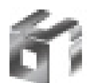


Table 16-5 `ACE_Service_Config::open()` Arguments

Argument	Type	Description
argv	ACE_TCHAR*[]	List of command line arguments. Usually the same value supplied as `argv` to `main()`. `argv[0]` is usually the program name.
program_name	const ACE_TCHAR[]	Alternate first argument: used when the command line is not to be interpreted by the service configurator.
logger_key	const ACE_TCHAR*	Rendezvous point for connection to the ACE Logger daemon. Default depends on platform type (platforms that support stream pipes use `/tmp/server_daemon`. Others use `localhost:20012`).
ignore_static_svcs	int	Indicates that the service configurator should not process any static services. Default is `1` (do not process static services).
ignore_default_svc_conf_file	int	Indicates the service configurator should not read the default service configuration file, `svc.conf`, even if no other file is specified using the `-f` command line option. Default is `0` (attempt to read from `svc.conf` unless overridden).
ignore_debug_flag	int	Indicates that the service configurator should ignore any debug settings. If non-zero, the application is responsible for setting `ACE_Log_Msg::priority_mask` appropriately. Default is `0` (do not ignore debug settings).

Calling `ACE_Service_Config::close()` shuts down all services and deletes all memory allocated by the service configurator.

16.5.3 Commanding the Service Configurator

The remaining `ACE_Service_Config` functions are used by an application to command the service configurator. These functions provide internal access to the service configurator similar to the external interface available through the use of signals and service configuration files. Another way to command

the service configurator is to use the service manager (see 16.8). The service manager uses these functions among others to allow remote configuration of the service configurator.

The following function instructs the service configurator to reprocess all directives both in service configuration files and on the command line:

```
static void reconfigure (void);
```

Calling `ACE_Service_Config::reconfigure()` is the same as signalling the process by using whatever signal the service configurator is set to monitor (e.g., `SIGHUP`).

You may provide a single directive to the service configurator using:

```
static int process_directive (const ACE_TCHAR directive[]);
```

16.5.4 Additional Directives

In addition to the `static` and `dynamic` directives introduced in 16.3, the service configurator also processes the directives `remove`, `suspend`, and `resume`.

The *remove* Directive

Part of the dynamic nature of the service configurator is the ability to remove a currently-loaded service object. The `remove` directive causes the service configurator to call the object's finalizer, then remove the instance of the object from the repository. Typically, this would not be done during process start-up, but might be part of a service configuration file. For example, you might modify the service configuration file by adding a `remove` directive, the send a `SIGHUP` signal to the process to force the service configurator to re-read the file and process the directives therein. Another possibility is to use the service manager to process the directive remotely (see 16.8).

The form of the remove directive is:

```
remove service_name
```

For example, to remove a service previously loaded, using the name `my_service`, the directive is:

```
remove my_service
```

The *suspend* Directive

A running service may be halted for a period of time, without removing the service from the process, by suspending the service. A suspended service may be resumed later. The `suspend` directive causes the service configurator to call the `suspend()` function on the associated service object if the service object is not already suspended.

The form of the `suspend` directive is:

```
suspend service_name
```

For example, to suspend a running service named `my_service`, the directive is:

```
suspend my_service
```

The *resume* Directive

A suspended service may be resumed by placing the `resume` directive in the service configuration file, then triggering the reprocessing of the configuration file via a signal. The ability to suspend and resume services is useful for remote administration of the service configurator. You can also pass the `resume` directive remotely using the service manager (see 16.8).

The form of the `resume` directive is:

```
resume service_name
```

For example, to resume a suspended service named `my_service`, the directive is:

```
resume my_service
```

16.6 XML Service Configurator

By default, ACE uses the *classic* service configurator (described in 16.5). However, an XML front end to the service configurator is also provided. It can be enabled by defining the `ACE_HAS_XML_SVC_CONF` macro in

`$ACE_ROOT/ace/config.h`. When this macro is enabled, the `ACE` library will be built with support for the XML service configurator and will not support the classic service configurator format. A script, `$ACE_ROOT/bin/svcconf-convert.pl`, is provided that will convert the classic service configurator files into XML-based service configurator files.

16.6.1 Service Configurator DTD

The file `$ACE_ROOT/ACEXML/apps/svcconf/svcconf.dtd` contains the Service Configurator DTD. It is shown here for convenience:

```
<!-- $Id: svcconf.dtd,v 1.1.1.1 2005/01/03 19:35:39 chad Exp $   -->
<!-- Document Type Definition for XML ACE Service Config files -->

<!-- An ACE_Svc_Conf document contains zero or more entries    -->
<!--   The entries are processed in the order they appear      -->
<!--   in the ACE_Svc_Conf file.                               -->
<!ELEMENT ACE_Svc_Conf (dynamic|static|suspend|resume|remove|stream|streamdef)*>

<!-- Streams are separate into two elements.  One defines how  -->
<!-- the stream should be constructed and the other defines    -->
<!-- what to do with it.  The identity of a stream is defined  -->
<!-- in the first dynamic/static element.  -->
<!ELEMENT streamdef ((dynamic|static),module)>
<!-- @@ Do we ever need to suspend/resume/remove modules when  -->
<!--    constructing a stream?  Should we leave only dynamic   -->
<!--    and static here?  -->
<!ELEMENT module (dynamic|static|suspend|resume|remove)+>

<!-- A 'stream' element controls the stream object -->
<!-- @@ Likewise, we are reusing the 'module' element here.    -->
<!--    Do we ever need to insert new modules into a stream?   -->
<!--    Nanbor: I guess we can do that.  -->
<!ELEMENT stream (module)>
<!ATTLIST stream id IDREF #REQUIRED>

<!-- A 'dynamic' entry. -->
<!-- @@ The kind of attributes the corresponding initializer   -->
<!--    should take seems to be determined by the 'type'       -->
<!--    attribute.  Should we further partition the dynamic    -->
<!--    element definition into several elements?  E.g. into   -->
<!--    dyn_service_object/dyn_module/dyn_stream?              -->
<!--    Nanbor: Will that be too confusing? -->
<!ELEMENT dynamic (initializer)>
<!ATTLIST dynamic id ID #REQUIRED
                  status (active|inactive) "active"
                  type (module|service_object|stream) #REQUIRED>
```

```
<!-- Initializing function for dynamic entry.  -->
<!ELEMENT initializer EMPTY>
<!ATTLIST initializer init CDATA #REQUIRED
                      path CDATA #IMPLIED
                      params CDATA #IMPLIED>

<!-- A 'static' entry takes an ID attribute and an optional -->
<!-- parameter lists.  -->
<!ELEMENT static EMPTY>
<!ATTLIST static id ID #REQUIRED
                 params CDATA #IMPLIED>

<!-- A 'suspend' entry takes an ID attribute.  -->
<!ELEMENT suspend EMPTY>
<!ATTLIST suspend id IDREF #REQUIRED>

<!-- A 'resume' entry takes an ID attribute.  -->
<!ELEMENT resume EMPTY>
<!ATTLIST resume id IDREF #REQUIRED>

<!-- A 'remove' entry takes an ID attribute.  -->
<!ELEMENT remove EMPTY>
<!ATTLIST remove id IDREF #REQUIRED>
```

16.6.2 XML Service Configurator Syntax

In this section, we discuss the `dynamic`, `static`, `remove`, `suspend`, and `resume` entries of the XML configurator syntax.

16.6.2.1 Dynamic

The `dynamic` example discussed in 16.3.2 is as follows:

```
dynamic Resource_Factory Service_Object * mylib:_make_resources() "param ..."
```

The equivalent in XML format is shown below:

```
<dynamic id="Resource_Factory" type="Service _Object">
  <initializer path="mylib" init="_make_resources()" params="param ..."/>
</dynamic>
```

16.6.2.2 Static

The `static` example discussed in 16.3.1 is as follows:

```
static Resource_Factory "-ORBoption value ..."
```

The equivalent in XML format is shown below:

```
<static id="Resource_Factory" params="-ORBoption value ..."/>
```

16.6.2.3 Remove

The `remove` example discussed in 16.5.4 is as follows:

```
remove my_service
```

The equivalent in XML format is shown below:

```
<remove id="my_service"/>
```

16.6.2.4 Suspend

The `suspend` example discussed in 16.5.4 is as follows:

```
suspend my_service
```

The equivalent in XML format is shown below:

```
<suspend id="my_service"/>
```

16.6.2.5 Resume

The `resume` example discussed in 16.5.4 is as follows:

```
resume my_service
```

The equivalent in XML format is shown below:

```
<resume id="my_service"/>
```

16.7 Service Objects

The class `ACE_Service_Object` is the base class for all objects that can be loaded by the service configurator. Class `ACE_Service_Object` inherits

from both ACE_Shared_Object and ACE_Event_Handler, as shown in Figure 16-1. These base classes provide the behavior required to catalog the service object in a repository, and to provide basic interaction with a reactor.

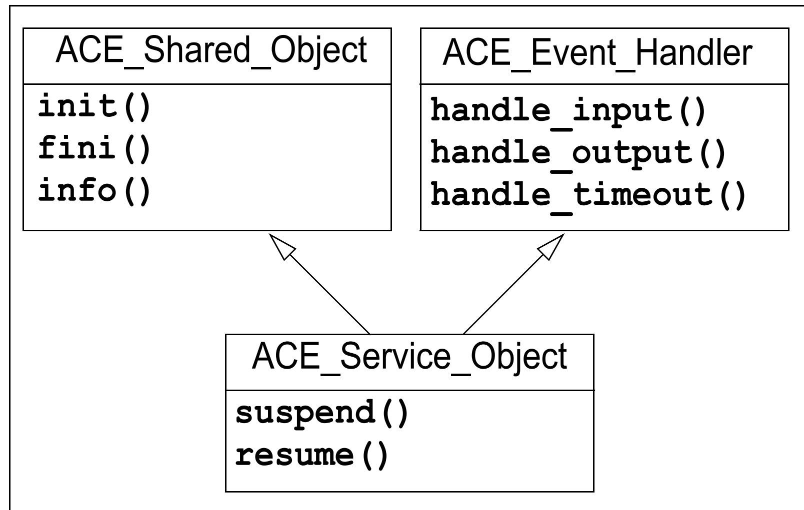


Figure 16-1 ACE Service Object Inheritance

16.7.1 Interface Definition

All of the virtual functions of a service object's interface have default *no-op* implementations. The class definition below shows the interface to the service object, including some of the inherited elements:

Synopsis

```
class ACE_Export ACE_Service_Object :
  public ACE_Event_Handler, public ACE_Shared_Object
{
  public:
    ACE_Service_Object (ACE_Reactor * = 0);
    virtual ~ACE_Service_Object (void);
    virtual int suspend (void);
    virtual int resume (void);

    // From ACE_Shared_Object
    virtual int init (int argc, ACE_TCHAR *argv[]);
    virtual int fini (void);
    virtual int info (ACE_TCHAR **info_string, size_t length = 0) const;

    // From ACE_Event_Handler
    virtual int handle_input (ACE_HANDLE fd = ACE_INVALID_HANDLE);
```

```
    virtual int handle_output (ACE_HANDLE fd = ACE_INVALID_HANDLE);
    virtual int handle_exception (ACE_HANDLE fd = ACE_INVALID_HANDLE);
    virtual int handle_timeout (const ACE_Time_Value &current_time,
                                const void *act = 0);
    virtual int handle_close (ACE_HANDLE handle, ACE_Reactor_Mask close_mask);
};
```

The functions shown above are examined in the following subsections.

16.7.2 Service Initialization and Finalization

Dynamic loading and discarding of service objects occurs throughout a program's lifetime. Externally, this is facilitated by using the `dynamic` and `remove` directives. A dynamically-loaded object is initialized through a call to its `init()` function. When dynamically loading an object, the service configurator first locates and opens the shared library that contains the object's factory function. Then the service configurator invokes the function, creating a new instance of the object. Finally, the service configurator calls the new object's `init()` function, supplying the initialization parameters from the service configuration directive. The initialization signature is:

```
virtual int init (int argc, ACE_TCHAR *argv[]);
```

As shown above, `init()` takes arguments supplied with the `dynamic` directive in an array as parameters, similar to the program's `main()` function. Statically-loaded objects are also initialized through a call to `init()`, using arguments supplied with a `static` directive. If initialization is successful, `init()` returns zero. The function returns `-1` if initialization could not complete.

The service configurator calls the finalizer, *not the destructor*, when processing a `remove` directive. The destructor is called for dynamic objects only if they are resident in the process when the process is terminated. The finalizer signature is:

```
virtual int fini (void);
```

Although `fini()` returns a result code, the service configurator framework does not currently check the value.

16.7.3 Service Creation

The service configurator creates new objects by invoking factory functions. A conforming factory function may have any name, but it must conform to the following interface:

```
typedef void (*ACE_Service_Object_Exterminator)(void *);
ACE_Service_Object * factory (ACE_Service_Object_Exterminator *gobbler);
```

The factory must return a pointer to a (typically new) instance of the service object type in question, and must also initialize the `gobbler` pointer to point to a function that conforms to the `ACE_Service_Object_Exterminator` function interface. This exterminator function is called when it is appropriate to delete the object.

ACE_Static_Svc_Descriptor

For statically-linked service objects, it is the application's responsibility to create an instance of the object and to register the object with the service configurator. To do this, a factory function, as shown above, must still be used, and an object of type `ACE_Static_Svc_Descriptor` must be created and provided to the service configurator.

The `ACE_Static_Svc_Descriptor` class shown below is a wrapper around all the information needed by the service configurator to use static service objects:

```
class ACE_Static_Svc_Descriptor
{
  public:
    const ACE_TCHAR *name_;
    int type_;
    ACE_SERVICE_ALLOCATOR alloc_;
    u_int flags_;
    int active_;
    void dump (void) const;
    int operator== (ACE_Static_Svc_Descriptor &) const;
    int operator!= (ACE_Static_Svc_Descriptor &) const;
};
```

Since `ACE_Static_Svc_Descriptor` attributes are publicly accessible, no constructor is needed, other than the default. These attributes are described below:

- *name_* is a string representing the unique identifier used to register the service with the service configurator. Note that the type is declared as a `const ACE_TCHAR *`, rather than a `const char *`. The macro `ACE_TCHAR` enables building on platforms requiring wide character strings.
- *type_* is a value indicating the type of wrapper used by the service configurator. For service objects, use the macro ACE_SVC_OBJ_T to initialize the type.
- *alloc_* is a pointer to the factory function used to create an instance of the service object. This function must return a pointer to the appropriate type of object, usually a pointer to an `ACE_Service_Object`. As with other "boilerplate" functions used in conjunction with the service configurator, there is a macro defined that becomes a service object factory function. The name of the factory function is represented by the macro `ACE_SVC_NAME(`*`service_object_class_name`*`)`.
- *flags_* is used to communicate destruction semantics for the service object and its container. There are two values used to set *flags_*, `DELETE_OBJ` and `DELETE_THIS`, defined in the class `ACE_Service_Type`.

  `DELETE_THIS` instructs the service configurator to remove the container of the service object when the service is shut down. `DELETE_OBJ` instructs the service configurator to delete the service object as well by invoking the exterminator supplied by the factory function, after calling `fini()`. Typically, the container should always be deleted by specifying `DELETE_THIS`, and if a service object is created in the factory function, it should be deleted as well by specifying `DELETE_OBJ`. These values may be combined via bitwise `OR` for assignment to *flags_*, as in:

  ```
  ACE_Service_Type::DELETE_THIS | ACE_Service_Type::DELETE_OBJ
  ```

- *active_* is set to a non-zero value to indicate that the service should be run in a separate thread by having its `activate()` function called or being registered with the reactor.

Helper Macros

ACE defines macros in `$ACE_ROOT/ace/Global_Macros.h` to simplify the creation of factory and exterminator functions, as well as static object descriptors.

Use the following macros in your service object's class header file to declare the static functions and classes used to create and manage service objects:

- `ACE_STATIC_SVC_DECLARE_EXPORT(EXPORT_NAME,SVC_CLASS)` must be included when your service is to be statically linked with your application. This macro declares a class with a constructor that will ensure an instance of your service is created. If necessary (e.g., in Windows DLLs), the name of the class is exported for visibility outside the library in which it is defined. The `EXPORT_NAME` parameter is the prefix for the export directive ("`_Export`" will automatically be appended to this prefix). The export macro has no effect on platforms where it is not needed. The `SVC_CLASS` parameter is the name of your service object class.

Note *A header file defining the export macro can be automatically generated using the Perl script* `$ACE_ROOT/bin/generate_export_file.pl`*. The script provides instructions for using it.*

- `ACE_STATIC_SVC_DECLARE(SVC_CLASS)` is similar to `ACE_STATIC_SVC_DECLARE_EXPORT`, except that it does not include the macro to export the symbol. You can use this macro on platforms that do not need to export symbols from DLLs. The `SVC_CLASS` parameter is the name of your service object class.
- `ACE_FACTORY_DECLARE(EXPORT_NAME,SVC_CLASS)` declares the uniquely-named factory function that the service configurator uses to create an instance of your service class. The parameters are the same as for `ACE_STATIC_SVC_DECLARE_EXPORT`.

The following macros are used in the source files to complete the definition of functions and classes that are declared by the above macros:

- `ACE_STATIC_SVC_DEFINE(SVC_CLASS,NAME,TYPE,FN,FLAGS,ACTIVE)` is used to statically initialize an `ACE_Static_Svc_Descriptor`. The `SVC_CLASS` parameter is the name of your service object class. The remaining parameters are described as the fields of the `ACE_Static_Svc_Descriptor` structure.
- `ACE_FACTORY_DEFINE(EXPORT_NAME,SVC_CLASS)` defines the body of the factory function declared with `ACE_FACTORY_DECLARE`. This factory function creates an instance of your service object and returns a

pointer to it as a pointer to an `ACE_Service_Object`. `ACE_FACTORY_DEFINE` also defines a function to clean up your service object. The parameters to this macro are the same as for `ACE_FACTORY_DECLARE`.

- `ACE_SVC_NAME(SVC_CLASS)` expands to the name of the service object factory function. This macro is useful when supplying to the service configurator a pointer to the factory function.
- `ACE_STATIC_SVC_REQUIRE(SVC_CLASS)` automatically registers your service with the service configurator by adding to the service configurator's service repository the service descriptor created with the `ACE_STATIC_SVC_DEFINE` macro. This macro also creates a static instance of your service class to ensure that the service is registered before `main()`.

16.7.3.1 Dynamic Service Example

The following example shows a service object that can be loaded dynamically. The code compiles to a shared library that is then loaded when the service configurator processes the `dynamic` directive for this shared object:

```
//declare the service
class My_Service: public ACE_Service_Object
{
  public:
    int init (int argc, ACE_TCHAR *argv[]);
    int fini (void);
};

//declare the service factory
ACE_FACTORY_DECLARE (My_Lib, My_Service);

// define the service
int My_Service::init (int argc, ACE_TCHAR *argv[]) {
  // parse the args and get things initialized
  return 0;
}

int My_Service::fini (void) {
  // service is shut down
  return 0;
}
// define the factory
ACE_FACTORY_DEFINE (My_Lib, My_Service)
```

16.7.3.2 Static Service Example

The following example shows the modifications (in **boldface**) from the previous example required to statically instantiate an object and register it with the service configurator when the service is to be statically linked to the application.

```
//declare the service
class My_Service: public ACE_Service_Object
{
  public:
    int init (int argc, ACE_TCHAR *argv[]);
    int fini (void);
};

//declare the static service descriptor
ACE_STATIC_SVC_DECLARE_EXPORT (My_Lib, My_Service);

//declare the service factory
ACE_FACTORY_DECLARE (My_Lib, My_Service);

// define the service
int My_Service::init (int argc, ACE_TCHAR *argv[]) {
  // parse the args and get things initialized
  return 0;
}

int My_Service::fini (void) {
  // service is shut down
  return 0;
}

// define the static service descriptor
ACE_STATIC_SVC_DEFINE (
  My_Service,                      // our service class
  "My_Static_Service",             // the name used to register the service
  ACE_SVC_OBJ_T,                   // use the service object container
  &ACE_SVC_NAME(My_Service),       // a reference to the factory function
  ACE_Service_Type::DELETE_THIS| // delete the container when done
  ACE_Service_Type::DELETE_OBJ,  // delete the service object when done
  0);                            // this object is not active

// define the factory
ACE_FACTORY_DEFINE (My_Lib, My_Service);

// register the service descriptor with the service configurator
ACE_STATIC_SVC_REQUIRE(My_Service);
```

16.7.4 Service Information

Services are expected to provide information about themselves by overloading the `info()` function. The signature of the `info()` function is:

```
virtual int info (ACE_TCHAR **info_string, size_t length = 0) const;
```

The arguments are a pointer to a string buffer and the buffer's length. If the provided `info_string` is a null pointer, the `info()` function creates a string buffer. Otherwise, `info()` copies the information to that buffer using the provided `length` as a limit. The result should be the length of the contents in the buffer, or `-1` if there is an error.

16.7.5 Service State

Services may be suspended for a time, then later resumed. To support this behavior, the service object must overload the following two functions:

```
virtual int suspend (void);
virtual int resume (void);
```

The service configurator calls `suspend()` when it processes a `suspend` directive. The service object is expected to enter into a state in which it does not respond to inputs. For reactor-based services, this usually means unregistering from the reactor. For active services, the service suspends its associated thread. A suspended service does not have to finalize itself, as it will stay resident in the process.

You reactivate a suspended service with `resume()`, called by the service configurator when it processes a `resume` directive. A suspended service should continue to operate normally after being resumed.

16.7.6 Obtaining Services

When building a dynamically-configured application, it is useful to be able to locate services by name in a manner similar to using the Naming Service. The service configurator framework provides this capability by using the following template for looking up service objects of a particular type by name:

Synopsis

```
template <class TYPE>
class ACE_Dynamic_Service : public ACE_Dynamic_Service_Base
{
public:
  static TYPE* instance (const ACE_TCHAR *name);
```

```
};
```

The function `instance()` will return a pointer to a service object of the appropriate type registered with the service configurator, using the specified name. If no object with that name is found, then `instance()` will return a null pointer.

Care must be taken that the service object associated with the name is really of the specified type. Due to the lack of run-time type safety in some C++ environments, incorrect casting can occur.

The following example uses `ACE_Dynamic_Service` to obtain a pointer to a `My_Service` object:

```
My_Service* svc_ptr;
svc_ptr = ACE_Dynamic_Service<My_Service>::instance("my_service");
```

16.8 ACE Service Manager

The service manager is a service object that is statically linked with ACE. It registers itself with the service configurator using the name `ACE_Service_Manager`. The service manager allows users to control the service configurator via a telnet connection. Using the service manager, it is possible to send remotely any directive accepted by the service configurator. Other capabilities include signaling the service configurator to reprocess its service configurator file, and getting a list of available service objects. The service manager is configured by the service configurator. Table 16-6 shows the options that may be supplied to the service manager.

Table 16-6 ACE Service Manager Command Line Arguments

Option	Definition
`-d`	Set the service manager to debug mode.
`-p` *portno*	Specify the port number for listening. Default is 10000.
`-s` *signal*	Specify an alternative signal to trigger a reconfiguration. Default is `SIGHUP`.

The following example shows how to supply these options to the service manager using the static directive:

```
static ACE_Service_Manager "-d -p 3911"
```

When initialized, the service manager will listen on the specified TCP port for incoming connections. When a connection is made, a single command is accepted from the client, the result (if any) is returned, and the connection is closed. Table 16-7 shows the commands the service manager will accept and its action in response (commands are case sensitive).

Table 16-7 ACE Service Manager Commands

Command	Action
`directive`	Takes any single service configurator directive (`dynamic`, `suspend`, `resume`, etc.) and processes it using `ACE_Service_Config::process_directive()`.
`help`	Iterate through the list of service objects registered with the service configurator and call the `info()` function for each. Returns the collection of object names and information strings.
`reconfigure`	Causes the service configurator to reprocess the directives in the service configuration file.

The following example shows how to send commands/directives to the service manager in a process running on a host named `malory` listening on port `3911`:

```
$ echo help | telnet malory 3911
$ echo reconfigure | telnet malory 3911
$ echo "suspend my_service" | telnet malory 3911
```

16.9 Summary

The ACE Service Configurator framework provides a mechanism for designing systems composed of collections of services and to defer decisions about service configuration until system deployment. Further, services may be dynamically configured, meaning they may be loaded or unloaded at any time during the life of the host process.

CHAPTER 17

ORB Initialization Options

17.1 Introduction

Many of the common behaviors of the ORB can be controlled at run time by passing options to the ORB initialization function, `CORBA::ORB_init()`. ORB initialization options are commonly passed into the program from the command line, using the `argc` and `argv` parameters made popular by C and UNIX. However, you can also create and pass these ORB initialization options programmatically.

You can use ORB initialization options to control the following ORB behaviors:

- Service configurator behavior.
- Quantity of debugging information output.
- Optimizations applied during request transfer and processing.
- Connection management and protocol selection behavior.
- Use of the Implementation Repository.

All of the ORB initialization command line options are of the form:

```
-ORBoption [arguments...]
```

Some examples are:

```
-ORBDebug
-ORBRcvSock 1024
-ORBListenEndpoints iiop://localhost:9999
```

You can pass several options on a single command line, for example:

```
./server -ORBDebug -ORBCollocation global -ORBCollocationStrategy thru_poa
```

In addition, you can supply some options more than once on the same command line. Most ORB initialization options apply to both clients and servers.

The initialization function parses these options in a case-insensitive manner (e.g., `-ORBCollocationStrategy` and `-ORBcollocationstrategy` are equivalent), removes them from the `argv` list, then decrements `argc`. Since the command line may also contain options that are specific to your application, you may find it easier to defer your own application-specific command line processing until *after* calling `CORBA::ORB_init()`.

This chapter explains all of the ORB initialization options and describes the appropriate context for their use.

Note *The* `ORBId`*,* `ORBInitRef`*,* `ORBDefaultInitRef`*,* `ORBListenEndpoints`*,* `ORBNoProprietaryActivation`*, and* `ORBServerId` *options described in this chapter are part of the standard CORBA specifications. All other options are TAO-specific.*

17.2 Interface Definition

The interface for `CORBA::ORB_init()` is defined by the OMG CORBA specification as follows:

```
// C++
namespace CORBA {
  static ORB_ptr ORB_init(
    int& argc, char** argv, const char* orb_identifier = "");
```

```
};
```

The first two arguments to `CORBA::ORB_init()` are the familiar `argc` and `argv` parameters popularized by C and UNIX for passing command line arguments to `main()`. You may frequently pass command line arguments directly from `main()` to `CORBA::ORB_init()` via `argc` and `argv`. The third argument is an ORB identifier that defaults to an empty string. The first time a particular ORB identifier is used within a process as an argument to `CORBA::ORB_init()`, an ORB with that name will be created. Subsequent calls within the same process, using this same identifier, will return a reference to the previously-created ORB. You can create more than one ORB instance in a process by calling `CORBA::ORB_init()` multiple times, using a different `orb_identifier` value each time.

An alternate method for passing `orb_identifier` to `CORBA::ORB_init()` is to pass it in the `argv` list as the argument for the `-ORBId` option (i.e., `-ORBId orb_identifier`). If both methods are used, a directly-passed non-empty value will take precedence over the value of the `-ORBId` option. In either case, if `orb_identifier` refers to a previously-created ORB, all other arguments passed via the `argv` list will be ignored because the ORB has already been initialized.

As stated previously, `CORBA::ORB_init()` parses arguments from the `argv` list and "consumes" any arguments it recognizes that begin with `-ORB`. This means the value of `argc` and the contents of `argv` may be modified by `CORBA::ORB_init()`. If you want to preserve command line arguments, you should make a copy of the `argv` array *before* calling `CORBA::ORB_init()`. An alternative to using the command line arguments is to construct your own argument list programmatically (e.g., to pass a unique set of arguments to each ORB instance in your application).

Note *The TAO implementation of* `CORBA::ORB_init()` *assumes that* `argv[0]` *contains the program name and begins parsing at* `argv[1]`*. So, if you construct your own argument list, be sure to provide a "dummy" argument for* `argv[0]`.

Here is an example showing how to create your own argument list for `CORBA::ORB_init()` programmatically:

```
// Set up argv[] array of -ORB options
char* argv[] = {
  "dummy",                          // argv[0] is skipped
  "-ORBId", "MyORB",                // Provide a unique ORB id
  "-ORBDebug",                      // Enable debug messages
  "-ORBDebugLevel", "6",            // Set debug level
  "-ORBListenEndpoints",
  "iiop://myhost:9999",             // Specify ORB's listening endpoint
  0                                 // argv[] should end with a null value
};

// Set value of argc based on actual contents of argv[]
int argc = (sizeof(argv)/sizeof(char*)) - 1;

//Pass the arguments to CORBA::ORB_init()
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

The first time you call `CORBA::ORB_init()` with a given ORB identifier, a new instance of the class `TAO_ORB_Core` is created. `TAO_ORB_Core` is the TAO implementation class for the `CORBA::ORB` interface. You will find its class definition in `$TAO_ROOT/tao/ORB_Core.h`. Each `TAO_ORB_Core` instance encapsulates several resources and components used by the ORB core, such as an acceptor, a root POA, a resource factory, a server strategy factory, and a client strategy factory. The resource factory is discussed in Chapter 18, the server strategy factory in Chapter 19, and the client strategy factory in Chapter 20.

Since each call to `CORBA::ORB_init()`, using a unique ORB identifier, creates a new `TAO_ORB_Core` instance, implementing an ORB-per-thread concurrency model is as simple as calling `CORBA::ORB_init()` once from each thread, making sure to supply a unique ORB identifier for each call. For example, the thread identifier could be used to generate a unique ORB identifier. In `$TAO_ROOT/performance-tests/Cubit/TAO/MT_Cubit`, you will find sample code for creating an application with an ORB-per-thread concurrency model.

Since each ORB core instance encapsulates an acceptor, each ORB you create will be listening on one or more endpoints. These endpoints may be specified using the `-ORBEndpoint` or `-ORBListenEndpoints` option or with the `TAO_ORBENDPOINT` environment variable. Otherwise, the ORB creates an endpoint for each protocol known to the resource factory. The default configuration of the resource factory causes the ORB to create only an IIOP endpoint. For more details on endpoints, see the discussion of the

-ORBListenEndpoints option in 17.13.43. For more details on the supported protocols, see the discussions of TAO's pluggable protocols in Chapter 14 and the resource factory in Chapter 18.

You can clean up all the resources allocated to the ORB core during initialization by calling `CORBA::ORB::destroy()`. For example:

```
// Intialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// ...use the ORB...

// Release resources.
orb->destroy();
```

17.3 Controlling Service Configurator Behavior

TAO uses the *ACE Service Configurator* framework to support dynamic creation and configuration of its components. See 16.3 for more information on how the service configurator is used in TAO.

Table 17-1 lists the options that influence the behavior of the ORB's service configurator. They duplicate many of the common options available to applications built using the service configurator framework. Client and server applications based on the service configurator framework may either use these options during ORB initialization or use the corresponding one-letter options supplied to `ACE_Service_Config:open()`. Each option is described in more detail in the referenced section.

Table 17-1 Service Configurator Control Options

Option	Section	Description
`-ORBDaemon`	17.13.7	Instruct the process to run as a daemon.
`-ORBGestalt` *context_name*	17.13.29	Specifies the service configurator gestalt to use for this ORB (global, local, shared, or current).
`-ORBIgnoreDefaultSvcConfFile 0|1`	17.13.31	Tells the ORB to ignore the file svc.conf if one happens to be present in the current directory.
`-ORBServiceConfigLoggerKey` *logger_key*	17.13.58	Specify where to write ORB logging output.

Table 17-1 Service Configurator Control Options

Option	Section	Description
`-ORBSvcConf` *`config_file_name`*	17.13.63	Specify an alternate service configurator file name.
`-ORBSvcConfDirective` *`directive`*	17.13.64	Pass a single directive directly to the service configurator.
`-ORBSkipServiceConfigOpen`	17.13.60	Skips loading of a service configurator file for this ORB.

17.4 Controlling Debugging Information

Often, during application development and testing, you will want to have fine-grained control over the amount and type of debugging information you receive from the application. You may not always have a debugger available, such as `gdb` or `dbx`, to debug a running application. For example, a debugger may not be available in an embedded environment.

TAO can provide debugging information at several levels of granularity. Table 17-2 lists options that influence the amount and type of debugging information generated by an application. Each option is described in more detail in the referenced section.

Table 17-2 Debugging Control Options

Option	Section	Description
`-ORBDebug`	17.13.8	Instruct the process to print debug messages from the ACE Service Configurator component.
`-ORBDebugLevel` *`level`*	17.13.9	Set the maximum tolerance for debugging messages reported from TAO.
`-ORBLogFile` *`file`*	17.13.44	Redirect all `ACE_DEBUG` and `ACE_ERROR` output to a file.
`-ORBObjRefStyle` *`style`*	17.13.52	Specify the format used to print Interoperable Object References (IORs).
`-ORBVerboseLogging` *`n`*	17.13.71	Controls the amount of status data printed on each line of the debug log. Higher numbers generate more output.

17.5 Optimizing Request Processing

Often it is possible to achieve better performance and reduce latency by optimizing certain stages of request processing in the ORB. Table 17-3 lists options to control various optimizations during request processing. Each option is described in more detail in the referenced section.

Table 17-3 Request Processing Optimization Options

Option	Section	Description
`-ORBAMICollocation` *enabled*	17.13.2	Controls the use collocation optimizations for AMI invocations.
`-ORBCDRTradeoff` *maxsize*	17.13.3	Control the trade-off strategy between copy and no-copy marshaling of octet sequences.
`-ORBCollocation` *type*	17.13.4	Specify the use and type of collocated object optimization.
`-ORBCollocationStrategy` *strategy*	17.13.5	Specify the collocated object strategy.
`-ORBNegotiateCodesets` *enabled*	17.13.48	Specify whether codeset negotiation should be used at all.
`-ORBNodelay` *enabled*	17.13.49	Sets the `TCP_NODELAY` option that disables the Nagle algorithm.
`-ORBRcvSock` *receive_buffer_size*	17.13.56	Specify the size, in bytes, of the socket receive buffer.
`-ORBSingleReadOptimization` *enabled*	17.13.59	Specify whether or not the ORB will use a single read operation when reading request messages.
`-ORBSndSock` *send_buffer_size*	17.13.61	Specify the size, in bytes, of the socket send buffer.
`-ORBStdProfileComponents` *enabled*	17.13.62	Specify whether or not the ORB generates the optional standard profile components in IORs.
`-ORBDisableRTCollocation` *boolean*	17.13.11	Controls whether collocation optimization decisions in RT CORBA applications use the real-time collocation resolver

17.6 Connection Management and Protocol Selection

ORBs send and receive requests and replies using various messaging and transport protocols. Each protocol has its own concept of an endpoint. Table 17-4 lists options to manage connections and to control protocol selection within your application. Each option is described in more detail in the referenced section.

Table 17-4 Connection Management and Protocol Selection Options

Option	Section	Description
`-ORBAcceptErrorDelay` *seconds*	17.13.1	Defines a delay to use after an accept fails before attempting a new accept.
`-ORBDottedDecimalAddresses` *enable*	17.13.13	For IP addresses under IIOP, use dotted decimal notation rather than host name. Other protocols use a suitable character representation of a numeric address.
`-ORBEndpoint` *endpoint*(s)	17.13.15	Deprecated in favor of `-ORBListenEndpoints`.
`-ORBEnforcePreferredInterfaces` *enforce*	17.13.16	Specifies whether specification for `-ORBPreferredInterfaces` option needs to be enforced.
`-ORBIIOPClientPortBase` *base*	17.13.32	Set the base of a range of ports a client is limited to using.
`-ORBIIOPClientPortSpan` *span*	17.13.33	Set the span of a range of ports a client is limited to using.
`-ORBLaneEndpoint` *laneid endpoint(s)*	17.13.40	Deprecated in favor of `-ORBLaneListenEndpoints`.
`-ORBLaneListenEndpoints` *laneid endpoint(s)*	17.13.41	Provide a collection of endpoints for a specific RTCORBA thread pool lane.
`-ORBListenEndpoints` *endpoint*(s)	17.13.43	Specify that the ORB is to listen for requests on the specified *endpoint*(s).
`-ORBNoServerSideNameLookups` *enabled*	17.13.51	Allows disabling of server side host name look-ups.
`-ORBParallelConnectDelay` *msecs*	17.13.53	Sets the connect delay when the `-ORBUseParallelConnects` option is specified.
`-ORBUseParallelConnects` *enabled*	17.13.69	Specify whether or not interfaces in a profile are connected to serially or in parallel.

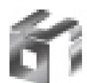


Table 17-4 Connection Management and Protocol Selection Options

Option	Section	Description
`-ORBPreferredInterfaces` *list*	17.13.55	Affects how network interfaces are selected on multi-homed hosts.
`-ORBUseSharedProfile` *enabled*	17.13.70	Specify whether or not the ORB should combine multiple endpoints into a single profile.

17.7 Socket Configuration Options

The options in this section control various options and attributes of sockets and other communication resources used by the ORB. Table 17-5 lists the options related to configuring these resources. Each option is described in more detail in the referenced section.

Table 17-5 Socket-Related Options

Option	Section	Description
`-ORBDontRoute` *enabled*	17.13.12	Set the SO_DONTROUTE option on the TCP sockets used.
`-ORBIPHopLimit` *enabled*	17.13.37	Specifies the TTL (IPv4) or hop limit (IPv6) for datagrams sent over a socket.
`-ORBIPMulticastLoop` *enabled*	17.13.38	Specifies the multicast loop option when using MIOP.
`-ORBKeepAlive` *enabled*	17.13.39	Set the SO_KEEPALIVE option on the TCP sockets used.
`-ORBLingerTimeout` *timeout*	17.13.42	Set the linger timeout on a TCP socket before closing it.

17.8 Service Location Options

The options in this section assist in locating specific services or with the general `resolve_initial_references()` mechanism. Table 17-6 lists the

options related to service location. Each option is described in more detail in the referenced section.

Table 17-6 Service Location Options

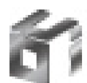
Option	Section	Description
`-ORBDefaultInitRef` *`URL_prefix`*	17.13.10	Specifies default URL prefix to apply when resolving initial object references.
`-ORBImplRepoServicePort` *`port`*	17.13.35	Specify the *`port`* on which the Implementation Repository Service listens for multicast requests.
`-ORBInitRef` *`ObjectID=IOR`*	17.13.36	Specify an object reference for an initial service.
`-ORBMulticastDiscoveryEndpoint` *`endpoint`*	17.13.46	Specify the *`endpoint`* on which the Naming Service listens for multicast requests.
`-ORBNameServicePort` *`port`*	17.13.47	Specify the *`port`* on which the Naming Service listens for multicast requests.
`-ORBTradingServicePort` *`port`*	17.13.65	Specify the *`port`* on which the Trading Service listens for multicast requests.

17.9 IPv6-Related Options

TAO provides a number of options related to its support of IPv6. Table 17-7 lists the ORB initialization options related to IPv6. These options are only available when TAO is built with IPv6 support enabled, through either the MPC `ipv6` feature or `ACE_HAS_IPV6`. Each option is described in more detail in the referenced section.

Table 17-7 IPv6-Related Options

Option	Section	Description
`-ORBConnectIPV6Only` *`enabled`*	17.13.6	Specifies whether to only use and support IPv6 connections.
`-ORBPreferIPV6Interfaces` *`enabled`*	17.13.54	When connecting to a server, specify whether to prefer IPv6 endpoints over IPv4 endpoints.
`-ORBUseIPV6LinkLocal` *`enabled`*	17.13.67	Specify whether IPv6 link local addresses should be used.



17.10 Multiple Invocation Retry Options

In cases such as connection timing issues or a load balancing server is temporarily not able to handle requests, it may be desirable to have the ORB transparently retry an invocation a maximum number of times until a successful invocation is made. To support this, a set of options can be used to cycle over the base and location forwarded profiles until an invocation is successful. When cycling over profiles and the first base profile is to be used, a user-defined delay is made before retrying.

Note *For backward compatibility, the `-ORBForward*` options described in 17.11 are retained. However, if any option in this section is used, then any `-ORBForward*` option given in 17.11 is ignored.*

These options can be used in the following situations:

- Initial connection - If the connection to a server would result in a `CORBA::TRANSIENT` exception being thrown, cycle over profiles no more than the value of the `-ORBForwardOnTransientLimit` option before throwing the exception.
- Waiting for a reply from the server - When using the IIOP protocol, on many platforms if zero bytes is read as the server's reply, this could indicate that the server may have had a disorderly shutdown or the connection to the server was abruptly terminated. In this case cycling through the profiles to retry the request will not exceeding the value of `-ORBForwardOnReplyClosedLimit`. If for example the Implementation Repository and Activator are used to start the server then this cycling could be used to give the server enough time to re-launch. When this option is used the `-ORBForwardOnTransientLimit` option should also be used to avoid exceptions being thrown when trying to reestablish a connection.

Note *Note that because it is not known if the request was sent, this could result in the request being processed more than one by the server. Therefore this option should be used with care. This is why, although a `CORBA::COMM_FAILURE` exception is normally thrown in this case, this option is kept distinct from `-ORBForwardOnCommFailureLimit` discussed below.*

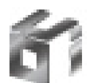
- Server sends an exception as a reply - A reply is received and it is checked if it is an exception. If the exception is of type `COMM_FAILURE` with completion status of `COMPLETED_NO`, then the profiles will be cycled no more than the value of option `-ORBForwardOnCommFailureLimit` until a successful reply is received. Similar retries can be applied to `TRANSIENT` exceptions (`-ORBForwardOnTransientLimit`), `OBJECT_NOT_EXIST` exceptions (`-ORBForwardOnObjectNotExistLimit`), and `INV_OBJREF` exceptions (`-ORBForwardOnInvObjrefLimit`).

These options can also be used in the list of Client_Strategy_Factory options in the service configurator file described in 16.3.1. Using the service configurator file can help insure these options are uniformly applied across multiple clients by having them use the same configurator file.

The retry options are given in the table below.

Table 17-8 Multiple Invocation Retry Options

Option	Section	Description
`-ORBForwardDelay` *`int msecs`*	17.13.17	Defines the time to delay while cycling through profiles when the first base profile is to be tried.
`-ORBForwardOnCommFailureLimit` *`limit`*	17.13.23	Use this option to cycle through profiles when a server replies to a request with a `COMM_FAILURE` exception. The number of retries will not exceed limit.
`-ORBForwardOnInvObjrefLimit` *`limit`*	17.13.24	Cycle through profiles when a server replies to a request with a `INV_OBJREF` exception. The number of retries will not exceed limit.
`-ORBForwardOnObjectNotExistLimit` *`limit`*	17.13.25	Cycle through profiles when a server replies to a request with a `OBJECT_NOT_EXIST` exception. The number of retries will not exceed limit.
`-ORBForwardOnReplyClosedLimit` *`limit`*	17.13.26	Cycle through profiles when it has been detected that a connection is closed when reading a server reply.



Option	Section	Description
-ORBForwardOnTransientLimit *limit*	17.13.27	Cycle through profiles when establishing a connection with a server or when a server replies to a request with a TRANSIENT exception.

17.11 Implementation Repository Options

Table 17-9 lists client options that are either directly or indirectly related to using a server with the Implementation Repository. Each option is described in more detail in the referenced section.

Note *The Multiple Invocation retry options discussed in 17.10 can also impact the behavior of clients and servers when using the Implementation Repository..*

Table 17-9 Implementation Repository Options

Option	Section	Description
-ORBForwardInvocationOnObjectNotExist *enabled*	17.13.18	Controls the handling of the OBJECT_NOT_EXIST exception after location forwards.
-ORBForwardOnceOnObjectNotExist *enabled*	17.13.19	Limits the retries of forwarding attempts after an OBJECT_NOT_EXIST exception
-ORBForwardOnceOnCommFailure *enabled*	17.13.20	Limits the retries of forwarding attempts after a COMM_FAILURE exception
-ORBForwardOnceOnTransient *enabled*	17.13.21	Limits the retries of forwarding attempts after a Transient exception
-ORBForwardOnceOnInvObjref *enabled*	17.13.22	Limits the retries of forwarding attempts after an INV_OBJREF exception

Table 17-9 Implementation Repository Options

Option	Section	Description
`-ORBIMREndpointsInIOR` *`enabled`*	17.13.32	Controls whether IMR or the server's endpoints should be used in the persistent POA IORs.
`-ORBServerId` *`server_id`*	17.13.57	A CORBA 3 option to uniquely identify a server to an IMR.
`-ORBUseIMR` *`enabled`*	17.13.66	Enables the use of the IMR for persistent POAs.

17.12 Miscellaneous Options

Table 17-10 lists certain miscellaneous ORB initialization options. Each option is described in more detail in the referenced section.

Table 17-10 Miscellaneous Options

Option	Section	Description
`-ORBFTSendFullGroupTC` *`enabled`*	17.13.28	Controls what is passed as the `IOP::FT_GROUP_VERSION` service context to IOGRs.
`-ORBId` *`orb_name`*	17.13.30	Sets the name of an ORB to *`orb_name`*.
`-ORBNoProprietaryActivation`	17.13.50	A CORBA 3 option to specify that a server should avoid the use of any proprietary activation framework (e.g., registration with an Implementation Repository) upon start up.
`-ORBMaxMessageSize` *`bytes`*	17.13.45	Controls GIOP fragmentation for messages that TAO sends by specifying a maximum message size.
`-ORBUseLocalMemoryPool` *`enabled`*	17.13.68	Specifies whether TAO should use a local memory pool or the platform's default memory allocator for allocating its internal memory needs.

17.13 Option Descriptions

The remainder of this chapter describes the individual ORB initialization options that may be passed, either programmatically or from the command line, to the ORB initialization function, `CORBA::ORB_init()`.

17.13.1 ORBAcceptErrorDelay *seconds*

Values for *seconds*

5 (default)	Be default, the ORB will wait five seconds after a transient failure before attempting to accept new connections.
> 0	Delay in seconds before attempting to accept new connections in the event that a transient error occurs.
0	A value of zero means that another attempt should not be made and no new connections will be accepted after a transient error.

Description When a transient error, such as running out of file handles, causes the ORB to fail to accept new connections, this option controls how long the ORB waits before attempting to accept new connections.

This option affects IIOP, SSLIOP, SCIOP, SHMIOP, and UIOP.

Usage Some rare applications may want to set this option to optimize their behavior when running out of file descriptors. Most applications should use the default.

Example The following example shows how to use the `-ORBAcceptErrorDelay` command line option to specify an accept delay of 30 seconds:

```
myserver -ORBAcceptErrorDelay 30
```

17.13.2 ORBAMICollocation *enabled*

Values for *enabled*

`1 (default)`	Enables collocation optimization for AMI invocations. The servant is called on the calling thread.
`0`	Disables collocation optimization for AMI invocations. The invocation is performed as a remote call and the servant is called on one of the ORB's processing threads.

Description By default, AMI invocations to collocated objects result in the servant being called on the client's thread. Because this essentially forces the AMI call to be synchronous, many applications find this undesirable. This option allows the user to directly control the use of collocation optimizations for AMI when they are otherwise enabled through `-ORBCollocation`.

Usage Use this option to disable AMI collocation optimizations, when a client requires all AMI calls to be truly asynchronous, but doesn't want to disable all collocation optimizations.

See Also 15.4.5, 17.13.4, 17.13.5

Example The following example shows how to use the `-ORBAMICollocation` command line option to disable collocation for AMI invocations:

```
myserver -ORBAMICollocation 0
```

17.13.3 ORBCDRTradeoff *maxsize*

Values for *maxsize*

`256 (default)`	The actual default is defined in `$ACE_ROOT/ace/OS.h` as `ACE_DEFAULT_CDR_MEMCPY_TRADEOFF`.
`>= 0`	The maximum octet sequence size that can utilize the current message block.

Description Excessive data copying is a significant source of memory management overhead during request processing. Resizing internal marshaling buffers multiple times when encoding large operation parameters leads to excessive data copying.

TAO minimizes unnecessary data copying by keeping a linked list of Common Data Representation (CDR) buffers. Operation arguments are marshaled into buffers allocated from thread-specific storage (TSS). The

buffers are linked together to minimize data copying. Gather-write I/O system calls, such as `::writev()`, can then write these buffers atomically without requiring multiple OS calls, unnecessary data allocation, or copying.

However, if an octet sequence is small and the last buffer in the linked list contains enough unused space for this sequence, copying into that buffer is more efficient than allocating additional buffers and appending them to the linked list.

If the length of the octet sequence is smaller than *maxsize* and there is room in the current message block for it, it will be copied there. This option is useful when applications can predict the octet sequence length and can therefore marshal without copying. By carefully choosing the value of *maxsize*, you can increase marshaling speed, avoid extra pointers and message blocks, and reduce the overall message size.

Usage Both server and client applications may use this option. The value of maxsize should be chosen to achieve a balance between message block overhead and the copying of octet sequences.

See Also *Applying Optimization Principle Patterns to Real-time ORBs*, 3.2, available via <http://www.theaceorb.com/references/>.

Example The following example shows how to use the `-ORBCDRTradeoff` command line option to specify a CDR data copy trade-off value of `1024`:

```
myserver -ORBCDRTradeoff 1024
```

17.13.4 ORBCollocation *is_allowed*

Values for *is_allowed*

`global` (default)	Objects within the same address space, even if from different ORBs, are considered collocated.
`yes`	*Deprecated* - means the same thing as `global`.
`per-orb`	Only objects from the same ORB are considered collocated.
`no`	Assume collocation is not possible and avoid testing for it.

Description When a client invokes an operation on a CORBA object whose servant is in the same address space (i.e., the same process), the client and servant are said to be collocated. TAO optimizes collocated client/servant configurations by generating collocation stubs for the client. These stubs bypass several layers

of marshaling, networking, demultiplexing, demarshaling, and dispatching logic, resulting in dramatically increased performance and decreased latency. This option controls in which situations TAO will use the collocation stubs.

Usage Normally, allowing for collocation is desirable. When the collocation optimization is turned off, *every* request, even on collocated objects, involves calling through the ORB and into the kernel, at least to the level of a loopback connection. However, if you are certain an application cannot use collocation, you can explicitly disallow the collocation optimization to avoid the slight overhead of determining whether a client and servant are collocated. You may also direct the IDL Compiler to not generate the collocation stubs and thus achieve a smaller memory footprint (see 4.13 for details).

Both server and client applications may use this option.

Note *Collocation may not be desirable for some real-time applications since collocated function invocations are run in the client's thread-of-control and this can cause priority inversions under some circumstances.*

See Also 17.13.5, 17.13.11

Example The following example shows how to use the `-ORBCollocation` command line option to disallow the collocation optimization.

```
myserver -ORBCollocation no
```

In addition, for an example that utilizes this option, see `$TAO_ROOT/performance-tests/Cubit/TAO/IDL_Cubit/collocation_test.cpp`.

17.13.5 ORBCollocationStrategy *strategy*

Values for *strategy*

`thru_poa` (default)	TAO uses a collocated object implementation that respects the POA's current state and policies.
`direct`	Invocations on collocated objects become direct calls to the servant without checking the POA's status.

Description As described in 17.13.4, using the collocation optimization allows requests to be dispatched more directly to collocated servants, bypassing several layers of marshaling, networking, demultiplexing, demarshaling, and dispatching logic.

This option controls which of two collocation stubs is used by clients when the collocation optimization is applied.

The default is `thru_poa`, which will deliver the request through the POA with which the servant is registered. The `direct` strategy will directly deliver the request from the stub to the servant, bypassing the POA. This is a TAO-specific extension and will behave differently in certain situations as described below.

Usage If the `thru_poa` strategy is used, a *safe* collocated stub is used to handle operation invocations on a collocated object. Though not as fast as a direct virtual function call, these safe collocated stubs are still very efficient, especially compared to normal operation invocations on collocated objects that must go through demarshaling and the loopback interface. Invoking an operation via the safe collocated stub ensures that:

- The servant's ORB has not been shut down.
- The thread-safety of all ORB and POA operations is maintained.
- The POA that manages the servant still exists.
- The POA Manager of this POA is queried to make sure upcalls are allowed to be performed on the POA's servants.
- The servant for the collocated object is still active.
- The `POACurrent`'s context is set up for this upcall.
- The POA's threading policy is respected.

Using the `direct` strategy optimizes for the common case and ensures that performance is the same as for a direct virtual function call. Invoking an operation via the direct collocated stub causes the following non-standard behaviors:

- The `POACurrent` is not set up.
- Interceptors are bypassed.
- POA Manager state is ignored.
- Servant Managers are not consulted.
- Etherealized servants can cause problems.
- Location forwarding is not supported.
- The POA's `Thread_Policy` is circumvented.

Note *Using the* `direct` *collocation strategy is not CORBA-compliant. It may also be less reliable than the* `thru_poa` *strategy for the reasons cited above. However, direct invocations on collocated objects may be desirable in some real-time applications with very stringent latency requirements.*

See Also 17.13.4, 17.13.11

Example The following example shows how to use the `-ORBCollocationStrategy` command line option to allow direct operation invocations on collocated objects, thereby bypassing the POA altogether:

```
myserver -ORBCollocationStrategy direct
```

In addition, for an example that utilizes this option, see

```
$TAO_ROOT/performance-tests/Cubit/TAO/IDL_Cubit/collocation_test.cpp.
```

17.13.6 ORBConnectIPV6Only *enabled*

Values for *enabled*

`0` (default)	Does not force exclusive use of IPv6.
`1`	Forces TAO to exclusively use IPv6.

Description This option is only available when TAO is built with IPv6 support enabled, through either the MPC `ipv6` feature or `ACE_HAS_IPV6`.

Enabling this option on the server side, tells TAO to only allow IPv6 interfaces as endpoints, only include IPv6 interfaces in IOR profiles, and either prevent or block IPv6 to IPv4 connections. Enabling this option on the client side causes IPv4 interfaces in IORs to be ignored and only IPv6 interfaces to be used.

This option affects IIOP, DIOP, and MIOP.

Usage Specify this option when your application must only use IPv6. Most applications would benefit from inter-operation with IPv4 by leaving it disabled.

See Also 17.13.54, 17.13.67

Example The following example shows how to use the `-ORBConnectIPV6Only` command line option to restrict TAO to only using IPv6 interfaces:

```
myserver -ORBConnectIPV6Only 1
```

17.13.7 ORBDaemon

Description Use `-ORBDaemon` to instruct the process to run as a daemon. In a UNIX environment, this means several things:

- The TTYs (terminals) associated with the process, including `stdin`, `stdout`, and `stderr`, are closed.
- The process will ignore `SIGHUP` signals, so signals generated from the terminal on which the process was launched will not affect it.
- The working directory of the daemon process is set to the root directory, so the file system from which it was launched can be safely unmounted while it is running
- The `umask` settings of the daemon process are cleared, so the permission bits in the inherited file mode creation mask do not affect the permission bits of new files created by the process.

Using `-ORBDaemon` on the command line is equivalent to passing the -`b` option to the `open()` function of the `ACE_Service_Config` class.

Usage Both server and client applications may use this option, but it is most applicable to long-running servers that need to run in the background, disconnected from any terminals. Since it is a command line option, you can still interact with these applications during development and testing, then use the option upon deployment. It uses mechanisms already available through the supplied service configurator, saving you the work of implementing this behavior yourself.

This option is effective only in environments where `ACE::fork()` is implemented. This excludes, for example, Win32 and VxWorks.

Example The following example shows how to use the `-ORBDaemon` command line option to *daemonize* a server process:

```
myserver -ORBDaemon
```

17.13.8 ORBDebug

Description The `-ORBDebug` option enables the printing of ACE and TAO debug messages generated by the *ACE Service Configurator* framework. You can use this option on the command line to enable the printing of additional debug information during development and testing, then disable the printing of such information (by not including the option on the command line) during deployment. Using `-ORBDebug` on the command line is equivalent to passing the `-d` option to the `open()` function of the `ACE_Service_Config` class.

Usage Both server and client applications may use this option. It affects the printing of debugging information from the TAO ORB core, connection handling, and protocol handling code.

See Also 16.4, 17.13.9

Example The following example shows how to use the `-ORBDebug` command line option to enable the output of debugging information from a server process.

```
myserver -ORBDebug
```

17.13.9 ORBDebugLevel *level*

Values for *level*

`0` (default)	No optional debugging messages are printed.
`> 0`	Debugging messages with a threshold less than *`level`* will be printed.

Description Sets the maximum tolerance for debugging messages reported from TAO. The default is to not print optional debugging messages.

Usage Use this option for fine-grained control over the amount of debugging information printed from applications. The environment variable `TAO_ORB_DEBUG` supersedes this option. Both server and client applications may use this option.

Nearly all messages controlled by the debug level have a threshold of `0`. Thus, choose a debug level of `1` to see most debugging messages.

GIOP commands (sent and received) will dump information about messages if the debug level is greater than `4`, and will include the message contents if the level is set greater than `9`.

GIOP synchronous invocations will report exceptions raised during invocation if the debug level is greater than `5`.

See Also 16.4, 17.13.8

Example The following example shows how to use the `-ORBDebugLevel` command line option to control the level of debugging information printed from a server.

```
myserver -ORBDebugLevel 6
```

17.13.10 ORBDefaultInitRef *URL_prefix*

Description The `-ORBDefaultInitRef` option supplies the ORB with a default URL prefix to be used in resolving object IDs that have not been mapped to a URL by the `-ORBInitRef` option.

When an application calls `resolve_initial_references("ObjectID")`, the ORB processes the call as follows:

If `ObjectID` is found in the `-ORBInitRef` mapping table, the associated URL is immediately resolved into an object reference.

If `ObjectID` is not found in the `-ORBInitRef` mapping table, then:

- If the ORB was supplied with a default URL prefix through the `-ORBDefaultInitRef` option,
    - A URL is constructed by appending a protocol-specific object key delimiter (e.g., '`/`') to *`URL_prefix`*, then appending `ObjectID` to the resulting string (e.g. URL_prefix/ObjectID).
    - The ObjectURL is then resolved into an object reference.
- If the ORB was not supplied with a default URL prefix, a built-in mechanism (such as IP multicast query) may be used to locate the `ObjectID` object.

Note *TAO does not support the OMG-recommended* `http` *and* `ftp` *formats.*

Usage Both server and client applications may use this option. Although it is not an error to repeat this option, each successive use of it will overwrite the previous one, so there is no advantage to be gained from repeating it.

When many initial services share a common URL prefix, it can be convenient to provide a -ORBDefaultInitRef as an alternative to a separate -ORBInitRef for each service.

The -ORBDefaultInitRef option should be used with caution, because it overrides the built-in resolution mechanism for every initial service, except those specified by -ORBInitRef. For example, suppose that an application uses both the Naming Service and the Trading Service, and that these services do not share the same address. If the Trading Service is specified through the -ORBInitRef option, then resolve_initial_references() will use the built-in mechanism to resolve the Naming Service.

However, if the Trading Service address is specified through -ORBDefaultInitRef, and the Naming Service is not specified by -ORBInitRef, then resolve_initial_references() will use the Trading Service address to construct an ObjectURL for both the Naming Service and the Trading Service. Any attempt by the application to connect to the Naming Service will fail.

The format for *URL_prefix* is:

```
<URL_prefix> = <corbaloc> | <protocol>
<corbaloc> = "corbaloc:" [<protocol_id>] ":" <address_list>
<protocol> = [<protocol_id_loc> "://"] <address_list>
<protocol_id> = "iiop" | "uiop" | "shmiop" | <ppiop>
<protocol_id_loc> = <iiop> | <uiop> | <shmiop> | "mcast" | <ppiop>
<iiop> = "iiop" | "iiopl oc"
<uiop> = "uiop" | "uiopl oc"
<shmiop> = "shmiop" | "shmiopl oc"
<ppiop> = pluggable protocol identifier added to TAO
<address_list> = [<address> ","]* <address>
<address> = <iiop_addr> | <uiop_addr> | <shmiop_addr> | <mcast_addr> |
<ppiop_addr>

<iiop_address> = [<version> <host> [":" <port>]]
<uiop_addr> = [<version> <path> ["/" <filename>]]
<shmiop_addr> = [[<version>] [<host> ":"] [<port>]]
<mcast_addr> = [<mcast_group>] ":" [<mcast_port>] ":" [<nic_addr>] ":" [<tll>]

<ppiop_addr> = defined by the pluggable protocol
<host> = DNS-style_Host_Name | ip_address | "[" ipv6_address "]"
<port> = number
<version> = <major> "." <minor> "@" | empty_string
<major> = <minor> = number
<path> = ["/" <directory_name>]*
<filename> = <directory_name> = string
```

Use of the new `corbaloc` syntax is preferred. The use of all other URLs, such as `iioploc`, is deprecated.

For `corbaloc`, the default protocol is “`iiop`”, and the default port is 2809. `corbaloc` also supports the “`rir`” (resolve initial reference) protocol, but this is not applicable to `-ORBDefaultInitRef` or `-ORBInitRef` because we are defining the initial references. See 22.4.1 for more details on `corbaloc`.

IPv6 style addresses are supported only in IIOP version 1.2, therefore you must explicitly specify version 1.2 when supplying an IPv6 address. For example `corbaloc::1.2@[::1]:12345/service` to define a service running on the localhost at port 12345. IPv6 style addresses are only available when TAO is built with IPv6 support enabled, through either the MPC `ipv6` feature or `ACE_HAS_IPV6`.

Multicast allows clients to discover the address of a service. The service listens on the multicast address for requests from clients and the service responds with the service’s address. The default initial reference is `mcast://:::`.

For <`mcast_addr`>, the default multicast group is 224.9.9.2. This multicast group must be a class D address in the range 224.0.0.0 to 239.255.255.255. The default multicast port is 10013[1]. The default NIC is `eth0`. The default TTL value is 1. TTL is the Time To Live—the number of hops outgoing packets will travel. A value of 1 means outgoing packets will only travel as far as the local subnet.

Note *If* `-ORBInitRef` *is used to define a particular ObjectID, that definition will take precedence over* `-ORBDefaultInitRef`.

See Also 17.13.43, 17.13.36
For more information on using TAO’s pluggable protocols, see Chapter 14.

Examples ***Using multicast for service discovery***
The following examples show how to use the `mcast:` protocol to find a service.

1. If a <object_key> is appended, as allowed in `-ORBInitRef`, the default is different for the different well-defined services. 10013 is the default if an <object_key> is not provided or “NameService” is the <object_key>. See `$TAO_ROOT/tao/default_ports.h` for the other defaults.

Start the Naming Service listening to multicast requests by specifying the `-m 1` option:

```
tao_cosnaming -m 1
```

Start an application server without specifying an initial reference for the Naming Service, thereby relying on the default multicast discovery behavior:

```
myserver
```

The invocation of

```
orb->resolve_initial_references("NameService");
```

from within `myserver` causes the ORB to retrieve the object reference using multicast.

If you want to specify a different multicast group instead of the default, you can use the following syntax:

```
tao_cosnaming -ORBMulticastDiscoveryEndpoint 234.5.6.7:8910 -m 1
myserver -ORBDefaultInitRef mcast://234.5.6.7:8910
```

Using both `-ORBInitRef` *and* `-OBDefaultInitRef`

The following example shows how to use both `-ORBDefaultInitRef` and `-ORBInitRef` to specify the locations of the Naming Service, Trading Service, and ImplRepo Service.

Start a Naming Service on `tango`:

```
tao_cosnaming -ORBListenEndpoints iiop://tango:6666
```

Start another Naming Service on `salsa`:

```
tao_cosnaming -ORBListenEndpoints iiop://salsa:7777
```

Start the Trading Service on `polka`:

```
tao_costrading -ORBListenEndpoints iiop://polka:8888
```

Start the ImplRepo Service on `waltz`:

```
tao_imr_locator -ORBListenEndpoints iiop://waltz:2809
```

Start an application server:

```
myserver -ORBInitRef TradingService=corbaloc::polka:8888 \
  -ORBDefaultInitRef corbaloc::tango:6666,salsa:7777,waltz
```

The invocation of

```
orb->resolve_initial_references("TradingService");
```

from within `myserver` causes the ORB to attempt `string_to_object()` with the following URL:

```
corbaloc:iiop:polka:8888/TradingService
```

When `myserver` attempts to connect to the Trading Service, the ORB will attempt to connect on `polka:8888`.

The invocation of

```
orb->resolve_initial_references("NameService");
```

from within `myserver` causes the ORB to attempt `string_to_object()` with the following URLs:

```
corbaloc:iiop:tango:6666/NameService
corbaloc:iiop:salsa:7777/NameService
corbaloc:iiop:waltz:2809/NameService
```

or equivalently:

```
corbaloc::tango:6666,:salsa:7777,:waltz:2809/NameService
```

When `myserver` attempts to connect to the Naming Service, the ORB will attempt to connect first on `tango:6666`, then on `salsa:7777`. If both of these attempts fail, it will make an attempt to connect on `waltz:2809`, but this attempt will fail because the Naming Service is not running on `waltz`.

The invocation of

```
orb->resolve_initial_references("ImplRepoService");
```

from within `myserver` causes the ORB to attempt `string_to_object()` with the following URLs:

```
corbaloc::tango:6666,:salsa:7777,:waltz:2809/ImplRepoService
```

When `myserver` attempts to connect to the Implementation Repository, the ORB will attempt to connect first on `tango:6666`, then on `salsa:7777`, but these attempts will fail because the ImplRepo Service is not running on either `tango` or `salsa`. The ORB will then attempt to connect on `waltz:2809`, where the ImplRepo Service was started.

Multiple services share the same address

The following example shows how to specify a default URL prefix for a Naming Service and Trading Service that share the same address.

Assume that a custom driver called `Finder` has been written for a Naming Service and Trading Service that share the same ORB. (For an example of a custom Naming Service driver, see 22.6.2.)

Start both the Naming Service and the Trading Service on `tango:9999`

```
Finder -ORBListenEndpoints iiop://tango:9999
```

The application server may be started using either the `-ORBInitRef` option

```
myserver -ORBInitRef NameService=corbaloc::tango:9999/NameService -ORBInitRef
TradingService=corbaloc::tango:9999/TradingService
```

or the `-ORBDefaultInitRef` option (much less typing involved)

```
myserver -ORBDefaultInitRef corbaloc::tango:9999
```

In either case, the invocation of

```
orb->resolve_initial_references("NameService");
```

from within `myserver` causes the ORB to use the following URL:

```
corbaloc:iiop:tango:9999/NameService
```

The invocation of

```
orb->resolve_initial_references("TradingService");
```

causes the ORB to use the following URL:

```
corbaloc:iiop:tango:9999/TradingService
```

Thus, when `myserver` attempts to connect to either the Naming Service or the Trading Service, the ORB will attempt to connect on `tango:9999`.

Some Valid Examples

```
-ORBDefaultInitRef corbaloc::tango//uses IIOP and port 2809
-ORBDefaultInitRef corbaloc:iiop:tango:9999
-ORBDefaultInitRef corbaloc::tango,:salsa:8888,iiop:tango:9999
```

Some Invalid Examples

```
-ORBDefaultInitRef host:port           // not corbaloc
-ORBDefaultInitRef corbaloc:iiop::port // no host
```

17.13.11 ORBDisableRTCollocation *boolean*

Values for *boolean*

`0 (default)`	The default value leaves real-time collocation resolution decisions to the real-time collocation resolver used by the RT CORBA ORB
`1`	Do not use the real-time collocation resolver; instead, rely on the ORB's default collocation resolution method

Description This option controls how collocation optimization decisions are made in RT CORBA applications. As described in 17.13.4, TAO normally optimizes collocated invocations (where the client and the target object are in the same address space). The effect of the ORB's default collocation optimization is such that the client thread is used to carry out the request. As described in 15.4.5, this effect may be undesirable in real-time applications. Therefore, TAO's implementation of RT CORBA employs a special "real-time collocation resolver" (`RT_Collocation_Resolver`) to determine whether an invocation should be subject to collocation optimization. The `RT_Collocation_Resolver` considers the following factors when making collocation decisions:

- The ORB and POA of the target object.
- The thread pool policy of the target's POA.

- The thread pool id and thread pool lanes of the target's POA.
- The priority model of the target's POA.
- The invoking thread.

These factors are considered in making collocation decisions to ensure the request is carried out at the appropriate priority.

However, not all applications need such a precise definition of collocation. For these applications, the `-ORBDisableRTCollocation` option can be used to bypass the real-time collocation resolver and use the ORB's default collocation resolution method, as described in 17.13.4 and 17.13.5. A value of `1` (true) disables real-time collocation resolution decisions and falls back on the default collocation decisions implemented in the default ORB. This behavior may result in better performance, but the invocation may not be subject to the appropriate RT CORBA thread and priority constraints.

The default value of this option is `0`.

Usage Both server and client applications may use this option. It only affects client invocations on objects within the same address space.

See Also 17.13.4, 17.13.5, Chapter 8

Example The following example shows how to use the `-ORBDisableRTCollocation` option to specify that the default ORB's collocation resolution method should be used instead of the real-time collocation resolver.

```
myserver -ORBDisableRTCollocation 1
```

17.13.12 ORBDontRoute *enabled*

Values for *enabled*

`0` (default)	Do not set the `SO_DONTROUTE` socket option on the TCP sockets used by IIOP.
`1`	Set the `SO_DONTROUTE` socket option on the TCP sockets used by IIOP.

Description This option controls whether the `SO_DONTROUTE` socket option is set on the TCP sockets used. By default it is not set.

This option only affects IIOP.

Usage Setting this option bypasses the routing table and selects a local interface based on the destination address.

See Also W. Richard Stevens, *UNIX Network Programming*, Volume 1, page 199.

Example The following example shows how to use the `-ORBDontRoute` command line option to set the `SO_DONTROUTE` option on any sockets used:

```
myserver -ORBDontRoute 1
```

17.13.13 ORBDottedDecimalAddresses *enabled*

Values for *enabled*

`0` (Non-MS Windows default)	Use host/domain names in IORs.
`1` (MS Windows default)	Use IP addresses in IORs.

Description An IOR exported by an ORB contains information about the communication endpoint of the server. When TCP/IP is used for communication between ORBs, the server endpoint is identified by a host-and-port tuple of the form `host:port`. The host name is typically specified as a domain name (e.g., `tango.acme.com`) rather than an IP address (e.g., `128.252.165.61`). This allows Domain Name Servers (DNS) and routers to make intelligent choices about what network routes clients will use in communicating with the server. It also allows network managers to facilitate rudimentary load balancing by reconfiguring a DNS to return different IP addresses for a particular domain name. However, there are occasions when the routers and DNS's of a network do not know about a particular domain name. In this case, the only way for the client to connect to the server is to use the IP address, bypassing the domain-name-to-IP-address conversion.

TAO will encode any endpoints specified through the `-ORBEndpoint` or `-ORBListenEndpoints` options or the `TAO_ORBENDPOINT` environment variable exactly as they are given. The `-ORBDottedDecimalAddresses` option will override this behavior.

When using an IP address, there is a slight decrease in latency during the initial connection between client and server ORBs since no DNS look-up is performed.

Usage Both server and client applications may use this option. Provide a value of `1` to instruct the ORB to encode IP addresses, instead of domain names, in IORs.

Note *Since dotted decimal address specifications apply only to IP networks, and not all ORB protocols are implemented on IP, this option is currently treated as a suggestion for each loaded protocol to use a character representation for the numeric address (if* `enabled=1`*), otherwise to use a logical name.*

See Also 17.13.51

Examples The following examples show how to use the `-ORBDottedDecimalAddresses` option to specify that object references generated by the ORB should have IP addresses instead of host/domain names encoded within them:

```
myserver -ORBDottedDecimalAddresses 1
myserver -ORBDottedDecimalAddresses 1 -ORBListenEndpoints iiop://host.domain.com
```

17.13.14 ORBDynamicThreadPoolName *name*

Description This option works in conjunction with the dynamic thread pool library and the DTP_Config service object. In order to supply a dynamic thread pool to an ORB, the application must use this option to supply the name of the dynamic thread pool definition to use.

See Also 15.3.8

17.13.15 ORBEndpoint *endpoint(s)*

Note *The TAO-specific* `-ORBEndpoint` *option has been deprecated in favor of the OMG-standard* `-ORBListenEndpoints` *option. See 17.13.43. These options have identical forms and you can simply replace the option name.*

Description Starting with the CORBA 3.0 specification there is a standard ORB initialization option for specifying the endpoints on which the ORB will listen for requests. The format of the endpoints argument is left up to the ORB implementation. In TAO, the `-ORBListenEndpoints` and `-ORBEndpoint` options accept the same endpoint specifiers and have exactly the same effect. In existing applications, you can simply replace "`-ORBEndpoint`" with "`-ORBListenEndpoints`".

See Also 17.13.43

17.13.16 ORBEnforcePreferredInterfaces *enforce*

Values for *enforce*

`1`	Enforce use of preferred interfaces only, thereby causing the ORB to raise an exception if preferred interfaces cannot be used to connect to a target host/interface.
`0 (default)`	Do not enforce use of preferred interfaces only.

Description This option specifies whether the `-ORBPreferredInterfaces` option is enforced. This option can help in cases where the preferred local network may not have a route to the target host/interface. The ORB can choose a default local interface to send the message, or it can raise an exception to the application that preferred interfaces are not usable. This option determines the ORB's behavior in this case. If the value of the `-ORBEnforcePreferredInterfaces` option is set to *1*, unusable preferred interfaces will make the ORB raise an exception to the application. The default value for this option is *0*.

Note *Not all transport protocols are based on TCP/IP. Thus, the arguments provided to such options may be interpreted in a protocol-specific way.*

Usage This option is only significant for ORBs in a client role and when the `-ORBPreferredInterfaces` option is also used.

See Also 17.13.55

Example The following example shows how to use the `-ORBEnforcePreferredInterfaces` command line option:

```
myclient -ORBPreferredInterfaces "*.sometargethost.com:malory.ociweb.com"
-ORBEnforcePreferredInterfaces 1
```

17.13.17 ORBForwardDelay *msec*

Values for *msec*

`100 (default)`	default delay is 100 msec
`0 or more`	wait this number of milliseconds

Description When object references are managed by the implementation repository there can be a slight delay in the initialization of the service depending on how much work is involved starting up, loading resources, etc. A TAO client that forwards based on an exception, as described by the `ORBForwardOn`... options listed below, can have its performance fine tuned with this value.

Usage The default value of 100msec may be increased by suppling a higher delay value. This would typically be useful only when the client wants to use servers that take a particularly long time to start.

Example The following example shows how to use the `-ORBForwardDelay` command line option to change the ORB's default behavior:

```
myclient -ORBForwardDelay 200
```

17.13.18 ORBForwardInvocationOnObjectNotExist *enabled*

Values for *enabled*

`0` `(default)`	The `OBJECT_NOT_EXIST` exception is always reported to the client process. This is the behavior defined by the CORBA specification.
`1`	The `OBJECT_NOT_EXIST` exception causes the client process to try the next profile in the IOR.

Description This option controls whether `OBJECT_NOT_EXIST` exceptions are reported to the client application or whether the client ORB attempts to retry the invocation with a different profile.

Usage This option can be useful in certain situations where location forwards are occurring and servers are being restarted. For example, if the Implementation Repository is being used it will forward invocations to the appropriate servers. Unfortunately, if the server fails and restarts on a different endpoint, subsequent invocations can result in an `OBJECT_NOT_EXIST` exception.

Enabling this option causes the ORB to handle this exception and retry it via the Implementation Repository and locate the new server instance.

Example The following example shows how to use the `-ORBForwardInvocationOnObjectNotExist` command line option to change the ORB's default behavior:

```
myclient -ORBForwardInvocationOnObjectNotExist 1
```

17.13.19 ORBForwardOnceOnObjectNotExist *enabled*

Values for *enabled*

`0 (default)`	Forwarding of requests is not retained, an exception will always cause a retry, potentially leading to a loop.
`1`	A previous forward attempt is remembered, preventing the possibility of a forward retry loop.

Description The forward once on exception options all help refine behavior when working with invocations that are libel to induce request forwarding. These can be any sort of indirect reference, including those defined using interoperable naming service style endpoints, such as `corbaloc` strings, objects that are managed by the implementation repository and others.

Historically, a TAO client could get stuck in a loop if the target of the forward throws an exception, as the default behavior on an exception during a forward is to try the invocation again. When the exceptional condition is itself short-lived, such as start up of an IMR-ified server that does not use proper notification semantics, this forwarding retry behavior can shield the client application from unnecessary exception handling.

Unfortunately this can behavior can hang a client if the exception is legitimate. For this reason, the -ORBForwardOnceOnObjectNotExist option is provided to limit the stub to a single retry attempt after a given exception. If the exception persists, it is passed back to the client code to be handled by the application.

Usage Use -`ORBForwardOnceOnObjectNotExist` when the client application is likely to use forwarding invocations and the target may possibly raise an `OBJECT_NOT_EXIST` exception.

Example The following example shows how to use the `-ORBForwardOnceOnObjectNotExist` option to change the ORB's default behavior:

```
myclient -ORBForwardOnceOnObjectNotExist 1
```

17.13.20 ORBForwardOnceOnCommFailure *enabled*

Values for *enabled*

`0 (default)`	Forwarding of requests is not retained, an exception will always cause a retry, potentially leading to a loop.
`1`	A previous forward attempt is remembered, preventing the possibility of a forward retry loop.

Description The forward once on exception options all help refine behavior when working with invocations that are libel to induce request forwarding. These can be any sort of indirect reference, including those defined using interoperable naming service style endpoints, such as `corbaloc` strings, objects that are managed by the implementation repository and others.

Historically, a TAO client could get stuck in a loop if the target of the forward throws an exception, as the default behavior on an exception during a forward is to try the invocation again. When the exceptional condition is itself short-lived, such as start up of an IMR-ified server that does not use proper notification semantics, this forwarding retry behavior can shield the client application from unnecessary exception handling.

Unfortunately this can behavior can hang a client if the exception is legitimate. For this reason, the -ORBForwardOnceOnCommFailure option is provided to limit the stub to a single retry attempt after a given exception. If the exception persists, it is passed back to the client code to be handled by the application.

Usage Use -`ORBForwardOnceOnCommFailure` when the client application is likely to use forwarding invocations and the target may possibly raise an `COMM_FAILURE` exception.

Example The following example shows how to use the `-ORBForwardOnceOnCommFailure` option to change the ORB's default behavior:

```
myclient -ORBForwardOnceOnCommFailure 1
```

17.13.21 ORBForwardOnceOnTransient *enabled*

Values for *enabled*

`0` (default)	Forwarding of requests is not retained, an exception will always cause a retry, potentially leading to a loop.
`1`	A previous forward attempt is remembered, preventing the possibility of a forward retry loop.

Description The forward once on exception options all help refine behavior when working with invocations that are libel to induce request forwarding. These can be any sort of indirect reference, including those defined using interoperable naming service style endpoints, such as `corbaloc` strings, objects that are managed by the implementation repository and others.

Historically, a TAO client could get stuck in a loop if the target of the forward throws an exception, as the default behavior on an exception during a forward is to try the invocation again. When the exceptional condition is itself short-lived, such as start up of an IMR-ified server that does not use proper notification semantics, this forwarding retry behavior can shield the client application from unnecessary exception handling.

Unfortunately this can behavior can hang a client if the exception is legitimate. For this reason, the -ORBForwardOnceOnTransient option is provided to limit the stub to a single retry attempt after a given exception. If the exception persists, it is passed back to the client code to be handled by the application.

Usage Use -`ORBForwardOnceOnTransient` when the client application is likely to use forwarding invocations and the target may possibly raise an `TRANSIENT` exception.

Example The following example shows how to use the `-ORBForwardOnceOnTransient` option to change the ORB's default behavior:

```
myclient -ORBForwardOnceOnTransient 1
```

17.13.22 ORBForwardOnceOnInvObjref *enabled*

Values for *enabled*

`0 (default)`	Forwarding of requests is not retained, an exception will always cause a retry, potentially leading to a loop.
`1`	A previous forward attempt is remembered, preventing the possibility of a forward retry loop.

Description The forward once on exception options all help refine behavior when working with invocations that are libel to induce request forwarding. These can be any sort of indirect reference, including those defined using interoperable naming service style endpoints, such as `corbaloc` strings, objects that are managed by the implementation repository and others.

Historically, a TAO client could get stuck in a loop if the target of the forward throws an exception, as the default behavior on an exception during a forward is to try the invocation again. When the exceptional condition is itself short-lived, such as start up of an IMR-ified server that does not use proper notification semantics, this forwarding retry behavior can shield the client application from unnecessary exception handling.

Unfortunately this can behavior can hang a client if the exception is legitimate. For this reason, the -ORBForwardOnceOnInvObjref option is provided to limit the stub to a single retry attempt after a given exception. If the exception persists, it is passed back to the client code to be handled by the application.

Usage Use -`ORBForwardOnceOnInvObjref` when the client application is likely to use forwarding invocations and the target may possibly raise an `Inv_OBJREF` exception.

Example The following example shows how to use the `-ORBForwardOnceOnInvObjref` option to change the ORB’s default behavior:

```
myclient -ORBForwardOnceOnInvObjref 1
```

17.13.23 ORBForwardOnCommFailureLimit *limit*

Values for *limit*

`0 (default)`	No profile cycling will be done for COMM_FAILURE exceptions.
`> 0`	Cycle through base and forward profiles at most *`limit`* times until a reply is successfully received from the server.

Description When the client is waiting for a reply from the server, it may get a `CORBA::COMM_FAILURE` exception with a completion status of `COMPLETED_NO`. In this case, re-connection attempts will be made cycling through the base and forward profiles at most *`limit`* times.

If the first base profile is encountered during the cycling, a delay will be made if `-ORBForwardDelay` is used.

Note *This option is also available as an option for the client strategy factory discussed in chapter Client Strategy Factory.*

Usage Use this option for example if an IMR-ified server does not use proper notification semantics and takes a long time to initialize.

See Also 5.2, 17.13.17, 17.13.20

Example The follow example shows how to attempt at most 10 times to get a reply from the server waiting 0.05 seconds between profile cycles:

```
myserver -ORBForwardOnCommFailureLimit 10 -ORBForwardDelay 50
```

17.13.24 ORBForwardOnInvObjrefLimit *limit*

Values for *limit*

`0 (default)`	No profile cycling will be done for `OBJ_INVREF` exceptions.
`> 0`	Cycle through base and forward profiles at most *`limit`* times until a reply is successfully received from the server.

Description When the client attempts to send a request to the server, it may get a `CORBA::INV_OBJREF` exception. In this case, invocation attempts will be made cycling through the base and forward profiles at most *`limit`* times.

If the first base profile is encountered during the cycling, a delay will be made if `-ORBForwardDelay` is used.

Note *This option is also available as an option for the client strategy factory discussed in chapter Client Strategy Factory.*

Usage Use this option for example if there are multiple profiles available in the IOR to try as alternatives and you are not certain that the servers being referenced the IORs are immediately available.

See Also 5.2, 17.13.22, 17.13.17

Example The follow example shows how to attempt at most 10 times when getting `INV_OBJREF` exceptions to send a request to the server waiting 0.05 seconds between profile cycles:

```
myserver -ORBForwardOnObjrefLimit 10 -ORBForwardDelay 50
```

17.13.25 ORBForwardOnObjectNotExistLimit *limit*

Values for *limit*

`0 (default)`	No profile cycling will be done for `OBJECT_NOT_EXIST` exceptions.
`> 0`	Cycle through base and forward profiles at most *`limit`* times until a reply is successfully received from the server.

Description When the client attempts to send a request to the server, it may get a `CORBA::OBJECT_NOT_EXIST` exception. In this case, invocation attempts will be made cycling through the base and forward profiles at most *`limit`* times.

If the first base profile is encountered during the cycling, a delay will be made if `-ORBForwardDelay` is used.

Note *This option is also available as an option for the client strategy factory discussed in chapter Client Strategy Factory.*

Usage Use this option for example if there are multiple profiles available in the IOR to try as alternatives and you are not certain that the servers being referenced the IORs are immediately available.

See Also 5.2, 17.13.19, 17.13.17

Example The follow example shows how to attempt at most 10 times when getting `OBJECT_NOT_EXIST` exceptions to send a request to the server waiting 0.05 seconds between profile cycles:

```
myserver -ORBForwardOnObjectNotExistLimit 10 -ORBForwardDelay 50
```

17.13.26 ORBForwardOnReplyClosedLimit *limit*

Values for *limit*

`0 (default)`	No profile cycling will be done if the server returns zero bytes as a reply.
`> 0`	Cycle through base and forward profiles at most `limit` times until a reply is successfully received from the server.

Description When waiting for a reply from the server using the IIOP protocol, on many platforms if the server's reply consists of zero bytes, then this indicates that the server performed a disorderly shutdown or the connection to the server was abruptly terminated. In this case, invocation attempts will be made cycling through the base and forward profiles at most `limit` times.

Note *Care should be used with this option as this could result in the request being processed more than once by the server. Depending on the nature of the request, this could lead to undesirable effects.*

The `-ORBForwardOnReplyClosedLimit` option should be used if this option is used to avoid a `TRANSIENT` exception from being thrown.

If the first base profile is encountered during the cycling, a delay will be made if `-ORBForwardDelay` is used.

Note *This option is also available as an option for the client strategy factory discussed in chapter Client Strategy Factory.*

Usage Consider using this option for example if the IOR is an IMR-ified IOR using a activator configured to restart the server if it crashes. If the server then crashes in the middle of a preparing for a reply, the activator will detect this and restart the server. Attempt will be made during this time get a complete reply from the server.

See Also 5.2, 17.13.23, 17.13.17

Example The follow example shows how to attempt at most 10 times when getting zero bytes from the server and waiting 0.05 seconds between profile cycles:

```
myserver -ORBForwardOnReplyClosedLimit 10 -ORBForwardDelay 50
```

17.13.27 ORBForwardOnTransientLimit *limit*

Values for *limit*

`0 (default)`	No profile cycling will be done for `TRANSIENT` exceptions.
`> 0`	Cycle through base and forward profiles at most *`limit`* times until a reply is successfully received from the server.

Description When the client attempts to make an initial connection with the server, it may get a `CORBA::TRANSIENT` exception. In this case, invocation attempts will be made cycling through the base and forward profiles at most *`limit`* times.

If the first base profile is encountered during the cycling, a delay will be made if `-ORBForwardDelay` is used.

Note *This option is also available as an option for the client strategy factory discussed in chapter Client Strategy Factory.*

Usage Use this option for example if the IOR is an IMR-ified IOR using a activator configured to start the server on demand. If the server takes a long time to initialize or

See Also 5.2, 17.13.21, 17.13.17

Example The follow example shows how to attempt at most 10 times when getting `OBJECT_NOT_EXIST` exceptions to send a request to the server waiting 0.05 seconds between profile cycles:

```
myserver -ORBForwardOnObjectNotExistLimit 10 -ORBForwardDelay 50
```

17.13.28 ORBFTSendFullGroupTC *enabled*

Values for *enabled*

`0 (default)`	Transmit only the group version as part of the `IOP::FT_GROUP_VERSION` service context.
`1`	Transmit the full Fault Tolerance Group information as part of the `IOP::FT_GROUP_VERSION` service context.

Description This option only has an effect when Fault Tolerant CORBA features such as Interoperable Object Group References (IOGR) are being used. It affects what

is contained in the `IOP::FT_GROUP_VERSION` service context that is included in each request sent to an IOGR. By default, TAO only includes the group version as specified by CORBA specification. When this option is enabled, the full identification information from the IOGR's tagged component is sent in the service context.

Usage This option should developers using TAO's fault-tolerance features to more easily write servers that can support multiple fault-tolerant CORBA objects.

Example The following example shows how to use the `-ORBFTSendFullGroupTC` command line option to send the full, TAO-specific, service context:

```
myserver -ORBFTSendFullGroupTC 1
```

17.13.29 ORBGestalt *context_name*

Values for *context_name*

`GLOBAL` `(default)`	Use the global service gestalt for this ORB.
`LOCAL`	Define a local service gestalt for this ORB.
`CURRENT`	Use the current service gestalt for this ORB.
`ORB:`*`orbid`*	Share the service gestalt defined by another ORB with the *orbid* specified.

Description ORBs store configuration information supplied by service directives in a *context*. There is a *global* context, which contains values visible to all ORBs. By default, the first ORB initialized uses the global context for its configuration and services defined outside of ORB_init are also placed into the global context.

As subsequent ORBs are initialized, the application may use `-ORBGestalt` to differentiate or share service configuration contexts as necessary. When no option is given, the new ORBs will continue to use the global context.

Supplying a context name of `LOCAL` causes the new ORB to create a new *local* context. Any service loaded into this local context will override any in the global context.

When the context name is `GLOBAL` this explicitly forces the ORB to use the global context. In this case, service directive behavior depends on settings for the specific service object. Some service objects allow multiple processing

replacing an earlier instance while others may ignore subsequent reloads, keeping settings from the first initialization.

When the context name is `ORB` then the new ORB shares the local context of a previously loaded ORB, which is identified by its ORB ID. An error occurs if the identified ORB is not available.

Usage Use this option to configure multiple ORBs within the single process with different configurations. The most common use case is to give each ORB its own local context.

Example The following example shows how to use the `-ORBGestalt` option to create two ORBs with each having their own context and corresponding configuration. This is done directly via C++, as the command line is insufficient for initializing two ORBs. ORB1 uses the configuration from `orb1.conf` and ORB2 uses the configuration from `orb2.conf`.

```
char* argv1[] = {
  "dummy",                          // argv[0] is skipped
  "-ORBGestalt", "local",           // specify a local gestalt
  "-ORBSvcConf", "orb1.conf",       // load the config file for ORB1
  0                                 // argv[] should end with a null value
};
int argc1 = (sizeof(argv1)/sizeof(char*)) - 1;

char* argv2[] = {
  "dummy",                          // argv[0] is skipped
  "-ORBGestalt", "local",           // specify a local gestalt
  "-ORBSvcConf", "orb2.conf",       // load the config file for ORB1
  0                                 // argv[] should end with a null value
};
int argc2 = (sizeof(argv2)/sizeof(char*)) - 1;

CORBA::ORB_var orb1 = CORBA::ORB_init(argc1, argv1, "ORB1");
CORBA::ORB_var orb2 = CORBA::ORB_init(argc2, argv2, "ORB2");
```

17.13.30 ORBId *name*

Description This is a CORBA specified option used to provide an explicit identifier for an ORB. As shown in the interface specification in section 17.2, using `-ORBId` is interchangeable with supplying a third argument to `CORBA::ORB_init()`. In applications that make use of multiple ORBs, supplying an Id is required to

initialize a new ORB instance. An ORB Id supplied on the command line will override the third argument to `CORBA::ORB_init()`.

Usage Use this option to provide an identifier to the ORB.

See Also 17.2

Example The following example shows how to use the `-ORBId` command line option to supply an identifier.

```
myserver -ORBId myorb1
```

17.13.31 ORBIgnoreDefaultSvcConfFile *enabled*

Values for *enabled*

`0 (default)`	Allow the `svc.conf` file to be processed if present.
`1`	Don't process the `svc.conf` file, if present.

Description The default service configuration is ordinarily processed if present, and no other configuration file is specified through other means. Use this option to force the ORB to ignore the default service configuration file.

Usage Use this option to ensure that an application's configuration is not altered by a `svc.conf` file placed in the start up directory.

See Also 17.13.63

Example The following example shows how to use `-ORBIgnoreDefaultSvcConfFile` to avoid `svc.conf`.

```
myserver -ORBIgnoreDefaultSvcConfFile 1
```

17.13.32 ORBIIOPClientPortBase *base*

Values for *base*

`0 (default)`	Client is unconstrained.
`>0`	Client will attempt to bind locally to the base port, or any available port in a range defined by a port span, if set.

Description Occasionally a distributed system needs to control the range of ports used by client applications when making connections. For example, clients deployed

on a host which defines blocks of ports to be shared by a number of server applications need to ensure ports in those blocks are not inadvertently taken.

Usage Use this `ORB_Init` argument along with `ORBIIOPClientPortSpan` to constrain the ORB to using local ports between `base` and `base + span`. If no ports are available, a TRANSIENT exception is thrown. In cases where an IOR contains multiple profiles, or the client is running on a multi homed host, the client may use the same port number with different IP addresses.

Clients are unconstrained by default.

Only a single client port span is kept per ORB. Using `ORBIIOPClientPortBase` repeatedly will simply reset the base value.

See Also 17.13.33

Example The following example shows how to set up a range of ports from 10000 to 10100.

```
./client -ORBIIOPClientPortBase 10000 -ORBIIOPClientPortSpan 100
```

17.13.33 ORBIIOPClientPortSpan *span*

Values for *span*

`0` (default)	Client is limited to only the supplied base port.
`>0`	Client is limited to a range of ports from base to base + span. Ignored if base is 0.

Description Occasionally a distributed system needs to control the range of ports used by client applications when making connections. For example, clients deployed on a host which defines blocks of ports to be shared by a number of server applications need to ensure ports in those blocks are not inadvertently taken.

Usage Use this `ORB_Init` argument along with `ORBIIOPClientPortBase` to constrain the ORB to using local ports between `base` and `base + span`. If no ports are available, a TRANSIENT exception is thrown. In cases where an IOR contains multiple profiles, or the client is running on a multi homed host, the client may use the same port number with different IP addresses.

Clients are unconstrained by default. If a base value is supplied but no span, the client will only be allow to bind to that one port locally. If no base is supplied, or a base value of 0 is used, any supplied span is ignored.

Only a single client port span is kept per ORB. Using `ORBIIOPClientPortSpan` repeatedly will simply reset the span value.

Impact Caution should be taken when defining a client port span. TAO clients are prone to open multiple connections as well as keeping those connections open a long time.

See Also 17.13.32

Example The following example shows how to set up a range of ports from 10000 to 10100.

```
./client -ORBIIOPClientPortBase 10000 -ORBIIOPClientPortSpan 100
```

17.13.34 ORBIMREndpointsInIOR *enabled*

Values for *enabled*

1 (default)	Use the Implementation Repository's endpoints when `-ORBUseIMR` is set and the objects are in persistent POAs.
0	Don't use the Implementation Repository's endpoints.

Description This option controls which endpoints get used in object references produced by persistent POAs when the `-ORBUseIMR` option is enabled. By default, when `-ORBUseIMR` is enabled, the server uses the Implementation Repository's endpoints in each persistent object's IOR. When this option is disabled, the server uses its own endpoints. When `-ORBUseIMR` is not enabled, the server always uses its own endpoints.

Usage Use this option when you want the other effects of `-ORBUseIMR` without the use of the Implementation Repository's endpoints in the IOR.

See Also Chapter 28, 17.13.66

Example The following example shows how to use the `-ORBIMREndpointsInIOR` command line option to avoid using IMR supplied endpoints:

```
myserver -ORBUseIMR 1 -ORBIMREndpointsInIOR 0
```

17.13.35 ORBImplRepoServicePort *port*

Description When `CORBA::ORB::resolve_initial_references`(“`ImplRepoService`”) is called, TAO by default uses IP multicast to find the TAO Implementation Repository (ImplRepo) Service. Often, more than one ImplRepo service is

running on a local network during development or in a deployed application where a large number of objects are logically partitioned in some way. In such cases, it is undesirable for all the ImplRepo service servers to be listening for multicast requests on the same port.

At startup, the TAO ImplRepo service is assigned a port on which to listen for multicast requests. The `-ORBImplRepoServicePort` option allows the user to specify this port. If this option is not used, the `ImplRepoServicePort` environment variable (if set) is used. If this option is not used and the environment variable is not set, then the port number is set to the value of `TAO_DEFAULT_IMPLREPO_SERVER_REQUEST_PORT`, defined in `$TAO_ROOT/tao/default_ports.h` as `10018`.

Usage Use `-ORBImplRepoServicePort` when starting the TAO ImplRepo Service to specify on which port it is to listen for multicast requests. Use it when starting clients of the ImplRepo Service to tell them the multicast port to use when multicasting a TAO ImplRepo service discovery request. This option is only available on platforms that support IP multicast. Both server and client applications may use this option.

This option has the following format:

```
-ORBImplRepoServicePort port
```

where *`port`* is the ImplRepo service's multicast request port number. This port number must be valid for your operating system and user permissions.

Example The following example shows how to use the `-ORBImplRepoServicePort` command line option to specify the multicast request port to use.

Start ImplRepo service and tell it to listen on port `1234` for multicast requests:

```
$TAO_ROOT/orbsvcs/ImplRepo_Service/tao_imr_locator -ORBImplRepoServicePort 1234
```

Start an application and tell it to use port `1234` for sending ImplRepo service multicast requests:

```
myserver -ORBImplRepoServicePort 1234
```

17.13.36 ORBInitRef *ObjectID=ObjectURL*

Description Using the `-ORBInitRef` option causes the ORB to create a mapping between a specified object identifier (*`ObjectID`*) and a specified object reference

(*ObjectURL*). The ORB maintains a table of these mappings for use in the processing of calls to `CORBA::ORB::resolve_initial_references()`.

When an application calls `resolve_initial_references("Obj_ID")`, the ORB processes the call as follows:

- If `Obj_ID` is found in the mapping table, the associated *ObjectURL* is immediately resolved into an object reference.
- If `Obj_ID` is not found in the mapping table, then
  - If the ORB was supplied with a default URL prefix through the `-ORBDefaultInitRef` option, an object URL is constructed from this prefix, then resolved into an object reference (see 17.13.10).
  - If a default URL prefix was not supplied, a built-in mechanism (such as IP multicast query) may be used to locate the `Obj_ID` object.

ObjectID represents an object identifier that may be defined either by TAO or by the application developer. TAO reserves the following *ObjectID* names: `NameService`, `TradingService`, `ImplRepoService`.[2]

ObjectURL is a "stringified" object reference that satisfies one of the following URL formats (see 17.13.43):

- `IOR:` format, e.g.,

```
IOR:000000000000001749444c3a4d44432f436f6e74726f6c6c65723a...
```

- *`protocol_id`*`:` format, e.g.,

```
corbaloc::tango:9999/TradingService
```

- `file:` format, e.g.,

```
file:///home/testing/file1.ior
```

Each of these formats directly specifies an address list and an object key.

Usage Both server and client applications may use this option. It can be used either to configure the ORB at run time with new initial service *ObjectID*s that were not defined when the ORB was installed or to override default initial service resolution mechanisms established for standard services (e.g., the Naming Service or Trading Service) at installation. It takes precedence over the use of the prefix passed with `-ORBDefaultInitRef`. This option may be repeated.

The *ObjectURL* formats are described below:

2. The names `IORManipulation`, `ORBPolicyManager`, `POACurrent`, `PolicyCurrent`, `RootPOA`, and `TypeCodeFactory` are also reserved, but are never used with `-ORBInitRef`.

- The `IOR:` format is simply a “stringified” IOR that consists of the string “`IOR:`” plus a sequence of hexadecimal digits that encode the address list, object key, and possibly other information, e.g.,

```
-ORBInitRef NameService=IOR:00000000000000001749444c3a4d44432f436f6e7...
```

  The stringified IOR is normally exported by the object to be referenced, then captured somehow by the client. As such, it is most often used with the `file://`*`path`*`/`*`filename`* format described below.

- The *`protocol_id`*`:` format is defined as follows:

```
<objectURL> = (<corbaloc> | <protocol> ) [<delimiter> <object_key>]
<corbaloc> = "corbaloc:" [<protocol_id>] ":" <address_list>
<protocol> = [<protocol_id_loc> "://"] <address_list>
<protocol_id> = "iiop" | "uiop" | "shmiop" | <ppiop>
<protocol_id_loc> = <iiop> | <uiop> | <shmiop> | "mcast" | <ppiop>
<iiop> = "iiop" | "iioploc"
<uiop> = "uiop" | "uioploc"
<shmiop> = "shmiop" | "shmioploc"
<ppiop> = pluggable protocol identifier added to TAO
<address_list> = [<address> ","]* <address>
<address> = <iiop_prot_addr> | <uiop_addr> | <shmiop_addr> | <mcast_addr>
| <ppiop_addr>

<iiop_prot_addr> = <iiop_id><iiop_addr>
<iiop_id> = ":" | <iiop_prot_token>":"
<iiop_prot_token> = "iiop"
<iiop_addr> = [<version> <host> [":" <port>]]
<uiop_addr> = [<version> <path> ["/" <filename>]]
<shmiop_addr> = [[<version>] [<host> ":"] [<port>]]
<mcast_addr> = [<mcast_group>] ":" [<mcast_port>] ":" [<nic_addr>] ":"
[<tll>]

<ppiop_addr> = defined by the pluggable protocol
<host> = DNS-style_Host_Name | ip_address | "[" ipv6_address "]"
<port> = number
<version> = <major> "." <minor> "@" | empty_string
<major> = <minor> = number
<path> = ["/" <directory_name>]*
<filename> = <directory_name> = string
<delimiter> = <iiop_del> | <uiop_del> | <shmiop_del> | <mcast_del> |
<ppiop_del>
<iiop_del> = <shmiop_del> = <mcast_del> = "/"
<uiop_del> = "|"
<ppiop_del> = defined by the pluggable protocol
<object_key> = ObjectID
```

  Use of the new `corbaloc` syntax is preferred. The use of all other object URLs, such as `iioploc`, is deprecated.

An IPv6 address syntax is only supported for IIOP version 1.2. The default IIOP version for corbaloc specifiers is 1.0, therefore you must explicitly specify version 1.2 when supplying an IPv6 address.

See 17.13.10 for a description of the `mcast:` and `corbaloc:` protocols.

A `uiop` *`ObjectURL`* must be enclosed by single quotes so that the shell does not interpret the vertical bar (‘`|`’) delimiter as a command (i.e., the “pipe” symbol).

The *`protocol_id`*`:` format is preferred over the IOR and file notation because it provides a stringified object reference that is easily manipulated in TCP/IP- and DNS-centric environments such as the Internet.

- The `dll://objectname` format is used to obtain a reference to an object loaded by the service configuration framework. In order to use this syntax, an object loader must be registered with the global object repository using the object name supplied in the URL. To use this format:
  - Create an object loader service object by specializing the `TAO_Object_Loader` base class.
  - Implement the `TAO_Object_Loader::create_object()` method. This method is called by the `TAO_DLL_Parser` object which is a helper to the ORB.

  Note that while create_object() takes an ORB reference, argc, and argv parameters, the DLL parser does not pass argc and argv. The loader must be able to configure and initialize an object reference without these arguments.

  An example of using the DLL format is available in `TAO/tests/Object_Loader`.
- The `file://`*`path`*`/`*`filename`* format is used as follows:
  - Create a URL using either the `IOR`: or *`protocol_id`*: format.
  - Store this URL in a file.
  - Type `-ORBInitRef` *`ObjectID`*`=file://`*`path`*`/`*`filename`*

  where *`filename`* is the name of the file in which the URL is stored and *`path`* is the full path to the file. The contents of the file are interpreted by `-ORBInitRef` as a single object reference.

- The `http://endpoint/filename` format is used to load an IOR file from an HTTP server rather than the local file system. In this case, `endpoint` is any web address in the form `host[:port]` with port defaulted to 80. The identifier following the endpoint is interpreted by the server as an entity containing some IOR, be it a file or not.

Note *TAO does not currently support the OMG-recommended* `ftp` *format.*

See Also 17.13.10, 17.13.43

For more information on using TAO's pluggable protocols, see Chapter 14.

Examples The following examples show various ways to use `-ORBInitRef`.

`protocol_id`: format - preferred `corbaloc` style

```
myserver -ORBInitRef NameService=corbaloc::1.1@tango:9999/NameService
client1 -ORBInitRef NameService=corbaloc:shmiop:2020/NameService
```

`protocol_id`: format - with multiple addresses

```
myclient -ORBInitRef \ NameService=corbaloc::tango:9999,:waltz:2809/NameService
```

When `myclient` first attempts to connect to the Naming Service, the ORB will attempt to connect on `tango:9999`. If that fails, it will attempt to connect on `waltz:2809`.

`IOR`: *format*

```
myserver -ORBInitRef NameService=IOR:000000000000001749444c3a4d44432f43...
```

`protocol_id`: format - `mcast` style

```
myserver -ORBInitRef \
 TradingService=mcast://234.1.2.3:12345:eth1:2/TradingService
myclient -ORBInitRef NameService=mcast://:12345::/NameService
```

`protocol_id`: format - deprecated `iiop | shmiop | uiop` style

```
myserver -ORBInitRef TradingService=iiop://traderhost:16001/TradingService
myclient -ORBInitRef NameService=iiop://1.2@tango:9999/NameService
```

`protocol_id`: format - deprecated `iiopLoc | shmiopLoc | uiopLoc` style

```
myserver -ORBInitRef NameService=iioploc://1.1@tango:9999/NameService
client1 -ORBInitRef NameService=shmioploc://2020/NameService
```

Note that the `uioploc` URL in the next example is enclosed in quotes:

```
client2 -ORBInitRef 'NameService=uioploc://1.2@/tmp/foo1|NameService'
```

Pluggable-protocol-style `iop:format` *example*

```
myserver -ORBInitRef NameService=myioploc://1.2@tango:9999/NameService
```

`file:` *format*

```
client1 -ORBInitRef TestService=file:///usr/local/testing/TestService.ior
```

17.13.37 ORBIPHopLimit *hops*

Description This option allows users to specify the TTL (IPv4) or hop limit (IPv6) value used when datagrams are sent over a socket. If this option is not specified, the Operating System default is used. The *hops* value must be at least zero and may be at most 255.

This option affects IIOP, DIOP, SCIOP, and MIOP.

Usage Use this option when you want to restrict the number of hops your CORBA traffic can take in the network.

See Also W. Richard Stevens, *UNIX Network Programming*, Volume 1.

Example The following example shows how to use the `-ORBIPHopLimit` command line option to specify a hop limit of 16:

```
myserver -ORBIPHopLimit 16
```

17.13.38 ORBIPMulticastLoop *enabled*

Values for *enabled*

1 (default)	Specifies that the `IP_MULTICAST_LOOP` (IPv4) or `IPV6_MULTICAST_LOOP` option is set on multicast sockets.
0	Specifies that the IP_MULTICAST_LOOP (IPv4) or IPV6_MULTICAST_LOOP option is not set on multicast sockets.

Description When using MIOP, the default behavior is to set the `IP_MULTICAST_LOOP` or `IPV6_MULTICAST_LOOP` option on each multicast socket. This means that datagram sent on that socket is looped back to that host and received by all

processes listening on that host. To avoid this behavior, disable this option. A side effect of disabling this option is that processes on the same host will not receive the datagrams.

This option only affects MIOP.

Usage This option is only useful if you are using MIOP.

See Also W. Richard Stevens, *UNIX Network Programming*, Volume 1.

Example The following example shows how to use the `-ORBIPMulticastLoop` command line option to disable the multicast loop behavior:

```
myserver -ORBIPMulticastLoop 0
```

17.13.39 ORBKeepAlive *enabled*

Values for *enabled*

`0` (default)	Do not set the `SO_KEEPALIVE` socket option on the TCP sockets used by IIOP.
`1`	Set the `SO_KEEPALIVE` socket option on the TCP sockets used by IIOP.

Description This option controls whether the `SO_KEEPALIVE` socket option is set on the TCP sockets used. By default it is not set.

This option only affects IIOP.

Usage Setting this option causes the Operating System to send *keep-alive probes* to the peer.

See Also W. Richard Stevens, *UNIX Network Programming*, Volume 1, pages 200-202.

Example The following example shows how to use the `-ORBKeepAlive` command line option to set the `SO_KEEPALIVE` option on any sockets used:

```
myserver -ORBKeepAlive 1
```

17.13.40 ORBLaneEndpoint *lane endpoint(s)*

Description Identical to the `-ORBLaneListenEndpoints` option. It is presented as an alias to maintain consistency with the TAO specific legacy option, `-ORBEndpoint` and its new standard alias, `-ORBListenEndpoints`.

See Also 17.13.43, 17.13.41, Section 8.5.1.2 of the CORBA 3.1 specification (OMG Document formal/08-01-04)

17.13.41 ORBLaneListenEndpoints *lane endpoint(s)*

Description This option is used for supplying endpoint definitions for specific RTCORBA thread pool lanes. A thread pool is a collection of threads that allow the ORB to process many requests simultaneously. Within a thread pool, groups of threads may be partitioned by runtime priority. These thread partitions are known as lanes. For instance, you might create a process that has separate high priority and low priority thread lanes. Each of these thread lanes must have its own endpoint to ensure that requests of a certain priority are handed by the appropriate threads and there is no chance for priority inversion.

For this endpoint, a “lane” is identified by a string consisting of a thread pool identifier and a lane identifier within that pool. Symbolically, this is represented as `<pool_id>:<lane_id>`, where `pool_id` and `lane_id` are both integers or a wildcard character, ’*’. Unfortunately, the RT CORBA specification contains no information regarding identifiers for pools and lanes, so this identification is dependent upon consistent initialization of the RT ORB and consistent initialization of thread pools. When creating a thread pool with lanes, each lane is defined by a structure, and many lane definitions are combined in a sequence. For a given pool, the lane identifier is the index of the lane’s position in the sequence. The pool identifier is defined by a counter that is an ORB internal resource. This counter starts at `1`, and is incremented for each thread pool that is created, whether or not that pool contains thread lanes.

This presents a discontinuity; if you wish to serve persistent objects from a server that uses thread pools with lanes, you must ensure that the thread pools are defined consistently, along with the lane endpoint definitions. The most reliable way to manage this is to generate the `-ORBLaneListenEndpoints` arguments programatically, rather than relying on an external script or user to enter them on the command line. An example of this technique is shown in 17.2.

The wildcard character of ’*’ matches all pool IDs or lane IDs, depending on its placement. A *lane* specified as “*:*” would match all lane and pool IDs. A *lane* specified as *“*1:**”* would match all lane IDs in pool 1. A *lane* specified as “*:1” would match lane 1 of all pool IDs.

Each lane endpoint definition may include many endpoints, in the same manner as the `-ORBEndpoint` and `-ORBListenEndpoints` options.

See Also 17.13.43, 17.13.40, Section 8.5.1.2 of the CORBA 3.1 specification (OMG Document formal/08-01-04)

Example In this example, we configure a single thread pool with two lanes:

```
-ORBLaneListenEndpoints 1:0 iiop://:1234 \
-ORBLaneListenEndpoints 1:1 iiop://:1235
```

Here is another example where we are specifying that all lanes in pool 1 should use IIOP and the network interface with the hostname bart:

```
-ORBLaneListenEndpoints 1:* iiop://bart
```

17.13.42 ORBLingerTimeout *timeout*

Description This option causes the ORB to set the `SO_LINGER` option on TCP sockets, with a specified timeout value in seconds, before closing the sockets. This option is only useful when using IIOP. The timeout value can be in the range of zero to the maximum signed integer value for the particular platform on which TAO is running.

Usage When a TAO client, configured to use IIOP, encounters an error while writing to a socket, the ORB attempts to close the socket and open a new one. TAO does not use the `SO_LINGER` socket option when sockets are opened. However, on some platforms, the ORB can have problems closing a socket after an error, for example if the other end is not responding, some data has already been written to the socket, and the socket buffers are not empty. If this occurs, the socket may be left in a `FIN_WAIT_1` state. Meanwhile, the ORB may keep opening new sockets to try to continue sending. Eventually, the client may run out of resources (e.g., file descriptors or mbufs).

If the `-ORBLingerTimeout` option is specified, the ORB will set the `SO_LINGER` socket option with the specified timeout value before closing the socket. By specifying a very short timeout value, the ORB can successfully close the socket without waiting for its buffers to become empty.

Since this option uses TCP socket options, it is only useful when using IIOP.

The most common use case is to set the timeout value to zero, so sockets are closed immediately when there is an error, thereby helping to avoid resource exhaustion on some platforms.

Example The following example shows how to use the `-ORBLingerTimeout` option to set the `SO_LINGER` timeout value to zero.

```
myclient -ORBLingerTimeout 0
```

17.13.43 ORBListenEndpoints *endpoint(s)*

Description Starting with the CORBA 3.0 specification, there is a standard ORB initialization option for specifying the endpoints on which the ORB will listen for requests. The format of the endpoints argument is left up to the ORB implementation. In TAO, the `-ORBListenEndpoints` and `-ORBEndpoint` options accept the same endpoint specifiers and have exactly the same effect. In existing applications, you can simply replace "`-ORBEndpoint`" with "`-ORBListenEndpoints`".

This option tells the ORB to listen for requests on the interfaces specified by the listed *`endpoint(s)`*. Endpoints are specified using a URL. An endpoint has the form:

```
protocol://V.v@addr1,...,W.w@addrN
```

where `V.v` and `W.w` are optional protocol versions for each address. An example of an IIOP endpoint is:

```
iiop://hostname:port
```

Sets of endpoints may be specified using multiple `-ORBListenEndpoints` options or by delimiting endpoints with a semi-colon ('`;`'). For example,

```
-ORBListenEndpoints iiop://localhost:9999 -ORBListenEndpoints
uiop:///tmp/mylocalsock -ORBListenEndpoints shmiop://10002
```

is equivalent to

```
-ORBListenEndpoints
'iiop://localhost:9999;uiop:///tmp/mylocalsock;shmiop://10002'
```

Note the single quotes (`'`) in the latter option specification. Single quotes are needed to prevent the shell from interpreting text after the semi-colon as another command.

If an endpoint is specified without an address, such as

```
-ORBListenEndpoints uiop:// -ORBListenEndpoints shmiop://
```

then a default endpoint will be created for the specified protocol.

Valid endpoint protocols include IIOP, UIOP (on platforms that support local IPC), SHMIOP, DIOP, and any additional protocols whose factories were loaded via the resource factory's `-ORBProtocolFactory` option. See 18.2.6 for more information on protocol factories. See Chapter 14 for information on using TAO's pluggable protocols.

If the `-ORBListenEndpoints` option is not used, then one endpoint is created by each protocol factory that is registered with the resource factory. Typically, this means an IIOP endpoint is provided, using the host name of the local machine and a randomly-selected port number, as if `-ORBListenEndpoints iiop://` had been specified. The value of the port number may be different each time the server is run.

Each endpoint that appears as an argument to the `-ORBListenEndpoints` option can itself accept endpoint-specific options. Such options will only apply to the endpoint for which they were specified.

Currently, TAO supports the `portspan`, `ssl_port` and `hostname_in_ior` options. The `portspan` option allows you to specify a range of ports for the ORB to use for endpoints. The ORB will pick the first port in the range that is not already in use. The `portspan` option is useful when you want to start several instances of a server on a range of ports or if you want to restrict the range of ports that your servers can use. Use the syntax `portspan=value` where `value` is the number of ports in the range. The `ssl_port` option is used with the TAO SSLIOP pluggable protocol. See 14.10 for more information on using SSLIOP. The `hostname_in_ior` option is used to explicitly specify the host name used in IORs. The server ORB does not validate the specified host name.

Note *You can only use the* `-ORBListenEndpoints` *option with servers. It has no effect on pure clients.*

Note *You can also use the* `TAO_ORBENDPOINT` *environment variable to specify the ORB's listen endpoints. The value of this environment variable has exactly the same syntax as the* `-ORBListenEndpoints` *option. If the* `TAO_ORBENDPOINT` *environment variable is used in addition to the* `-ORBEndpoint` *or* `-ORBListenEndpoints` *option, the endpoint(s) specified by the environment variable are added to the list of the endpoints specified by the above options.*

IIOP Endpoints

TAO's IIOP pluggable protocol utilizes TCP/IP as its underlying transport mechanism. IIOP endpoints in TAO have the form:

```
-ORBListenEndpoints iiop://V.v@hostname1:port1,...,W.w@hostname2:port2
```

where "`V.v`" and "`W.w`" are the IIOP protocol versions associated with the given address (`hostname:port`). Currently supported versions are 1.0, 1.1, and 1.2.

Options are separated from addresses by a forward slash ('`/`'). For instance, if an IIOP endpoint is to occupy a port in the range from `5000-5009`, the following endpoint specification could be used:

```
myserver -ORBListenEndpoints iiop://host:5000/portspan=10
```

The preceding example will start a server on each of the ports 5000-5009 if you issue the command 10 times. The 11th time will fail, since all ports in the range are already in use.

IIOP addresses are comprised of a hostname (or an IP address) and a TCP port on which the server listens. The hostname is used to select the network interface on which to set up the endpoint and is used to generate the IOR. Suppose a host has the following network interfaces:

```
eth0: foo1.bar.baz (DEFAULT)
eth1: foo2.bar.baz
```

To set up an endpoint on the second network interface, "`eth1`," either of the following endpoint specifications could be used:

```
-ORBListenEndpoints iiop://foo2
```

or

```
-ORBListenEndpoints iiop://foo2.bar.baz
```

An available port will be chosen by TAO (actually the operating system kernel) and placed into the IOR.

To set up an endpoint on a specific port, simply use an endpoint of the form:

```
-ORBListenEndpoints iiop://foo2:1234
```

where `1234` is the TCP port on which the endpoint will be opened. In this case, an endpoint will be opened on the network interface associated with the hostname `foo2` on port `1234`.

Port *`names`* are also accepted. For example, if a UNIX installation has a service called "`my_protocol`" associated with port `1234` in the service database in `/etc/services`, then the following would cause an endpoint to be opened on the port associated with that service:

```
-ORBListenEndpoints iiop://foo2:my_protocol
```

Port numbers range from `0` (port is chosen by the operating system) to `65335`. Port numbers less than `1024` on UNIX systems are considered privileged and require super-user privileges to access them. Also be aware that some ports may already be in use by other applications.

If no address is specified (e.g., `-ORBListenEndpoints iiop://`), then an endpoint with an automatically-chosen port number will be set up on each network interface detected by TAO. Each endpoint will use the same port number and will be represented in the generated IOR as a separate profile or as an alternate address within a single IOR profile. Note that network interface detection will only work on platforms that support this feature. If network interface detection is not supported, then the default network interface will be chosen.

Using a specification of the form `-ORBListenEndpoints iiop://:1234` will create an endpoint with TCP port `1234` on each detected network interface. Note that there is a colon (‘`:`’) preceding the port number `1234`. That colon is necessary for TAO to interpret `1234` as a port. Without the colon, TAO would interpret `1234` as a hostname associated with a given network interface.

Note *Testing under Windows XP/2000 has shown that the* `gethostname()` *function, used internally, may not return the fully-qualified domain name (i.e., returning "*`host`*" instead of "*`host.domain.com`*"). You may want to use the* `-ORBListenEndpoints` *option on Windows XP/2000 to ensure that your servers generate IORs that are usable from outside the local network.*

When TAO is built with IPv6 support (through either the `ipv6` MPC feature or `ACE_HAS_IPV6`) you may also use an IPv6 endpoint definition along with or instead of an IPv4 endpoint. IPv6 addresses 128 bits long and are typically expressed as groups of hexadecimal quartets separated by a colon, such as fe80:0134::1024:a401. Note that the syntax for IPv6 addresses contains a shortcut for a single long span of zeros. As shown in the example address, two consecutive colons indicates that the spanned hextets are all 0000.

To disambiguate IPv6 addresses with optional port specifiers. The address portion is framed in square brackets, []. Thus an explicit IPv6 endpoint looks like `iiop://[fe80:0134::1024:a401]:12345`.

Defaulted addresses may be constrained to either IPv4 or IPv6 by using the explicit INADDR_ANY or IN6ADDR_ANY specifier, 0.0.0.0, or [::] respectively. When using a defaulted address, TAO will enumerate all non-local interface addresses, unless there are no non-local interfaces available. Note that if you use the ifconfig or ipconfig system command to view your available interval addresses, you will see the loopback designated as ::1, and some other interface specifying an address starting with fe80:. The fe80 address is a “link local” address that is not routable and thus TAO does not use it in defaulted endpoints. Link local addresses may be used explicitly.

SHMIOP Endpoints

TAO's SHMIOP pluggable protocol utilizes shared memory as its underlying transport mechanism. SHMIOP endpoints in TAO have the form

```
-ORBListenEndpoints shmiop://V.v@port1,...,W.w@port2
```

where “`V.v`” and “`W.w`” are the SHMIOP protocol versions associated with the given address (port). Currently supported versions are 1.0, 1.1, and 1.2.

SHMIOP addresses consist of a port number on which the server listens. Port numbers range from `0` (port is chosen by operating system) to `65335`. Port numbers less than `1024` on UNIX systems are considered privileged and require super-user privileges to access them. Also be aware that some ports may already be in use by other applications.

TAO will automatically choose an address for a SHMIOP endpoint if the address is omitted from the specification (i.e., `-ORBListenEndpoints shmiop://`).

UIOP Endpoints

TAO’s UIOP pluggable protocol utilizes local IPC (i.e., UNIX domain sockets) as its underlying transport mechanism.

UIOP endpoints in TAO have the form:

```
-ORBListenEndpoints uiop://V.v@rendezvous_point1,...,W.w@rendezvous_point2
```

where “`V.v`” and “`W.w`” are the UIOP protocol versions associated with the given rendezvous point. Currently supported versions are 1.0, 1.1, and 1.2.

A UIOP address is the rendezvous point on which the server listens. This rendezvous point is generally the full path to the desired UNIX domain socket filename. Though relative paths can be used, their use is discouraged. The maximum length of the rendezvous point is 108 characters, as dictated by the POSIX.1g specification for local IPC rendezvous points. TAO will truncate any rendezvous point name longer than 108 characters.

A UIOP endpoint with the *absolute* path rendezvous point `/tmp/foobar` is created by specifying `-ORBListenEndpoints uiop:///tmp/foobar`, where the optional protocol version and endpoint-specific options have been omitted.

A UIOP endpoint with the *relative* path rendezvous point `foobar` is created in the current directory by specifying `-ORBListenEndpoints uiop://foobar`, but rendezvous points with relative paths are discouraged because it is possible that other rendezvous points with the same base name exist on a given system, giving rise to potential ambiguities.

Omitting the rendezvous point (i.e., specifying `-ORBListenEndpoints uiop://`) will cause TAO to automatically create an absolute-path rendezvous point. The rendezvous point will be located in a system temporary directory and its name will begin with “`TAO`.”

SSLIOP Endpoints

SSLIOP stands for the Secure Sockets Layer (SSL) Inter-ORB Protocol. This protocol is defined by the OMG as part of the CORBA Security Service specification. SSLIOP uses GIOP as a messaging protocol and SSL as the transport protocol. It is a drop-in replacement for IIOP, providing secure communication between hosts.

SSLIOP endpoints are specified similarly to IIOP endpoints. An SSLIOP endpoint is specified just like an IIOP endpoint with the addition of the `ssl_port` endpoint-specific option:

```
-ORBListenEndpoints iiop://hostname:iiop_port/ssl_port=secure_port
```

where the hostname and IIOP port are defined just like in IIOP endpoints and the `ssl_port` option specifies the port that will be used to establish a secure connection for secure communications.

SSLIOP is described in more detail in 14.10 and 27.10.

DIOP Endpoints

DIOP stands for Datagram Inter-ORB Protocol and is a UDP-based transport protocol. This protocol is only partially implemented; as such, there are restrictions on its use. The DIOP implementation uses connectionless UDP sockets, and therefore is intended for use as a low-overhead protocol for certain classes of applications. The original motivation for this protocol was applications that use only `oneway` operations.

Endpoints for DIOP are composed of the prefix “`diop://`”, followed by a host and port combination, similar to IIOP endpoints. An example of a DIOP endpoint is:

```
-ORBListenEndpoints diop://example.ociweb.com:12345
```

DIOP is described in more detail in 14.9.

Endpoint-specific Options
An endpoint-specific option is used as follows:

```
-ORBListenEndpoints iiop://foo:1234/option=value
```

Additional options can be specified by separating each option with an ampersand (‘&’) as follows:

```
-ORBListenEndpoints 'iiop://foo:1234/option1=value1&option2=value2'
```

Note that the address and the endpoint-specific options are separated by a forward slash (‘/’) in this case, i.e., for IIOP endpoints. This character is normally a slash (‘/’), but may differ for other types of pluggable protocol endpoints. For example, UIOP endpoint-specific options are separated from the address by a vertical bar (‘|’). Also note that when using more than one option, quotes should be used to prevent the shell from interpreting the ampersand (‘&’) as indicating that the process should be run in the background.

See Also 17.13.40, 17.13.52, 18.6.7, and Section 8.5.1.2 of the CORBA 3.1 specification (OMG Document formal/08-01-04).

Examples ***IIOP Endpoint Examples***
In the following example, we use the `-ORBListenEndpoints` command line option to restrict IIOP connections to clients running on the same host as the server by specifying `localhost` as the host name of the endpoint. No port number is specified, so it will be selected randomly. The command line is:

```
myserver -ORBListenEndpoints iiop://localhost:
```

Here are some additional examples of IIOP endpoints:

```
-ORBListenEndpoints iiop://1.1@foo1:0
-ORBListenEndpoints iiop://1.1@foo:0,1.2@bar,baz:3456
-ORBListenEndpoints iiop://1.1@foo:0,1.2@bar,baz:3456/portspan=10
-ORBListenEndpoints iiop:///portspan=5 (note three slashes "///")
-ORBListenEndpoints iiop://:2020/portspan=20
```

SHMIOP Endpoint Examples
Here are some additional examples of SHMIOP endpoints:

```
-ORBListenEndpoints shmiop://1.1@0
```

```
-ORBListenEndpoints shmiop://1.1@0,3456
```

UIOP Endpoint Examples

Here are some additional examples of UIOP endpoints:

```
-ORBListenEndpoints uiop://1.1@/tmp/foo1
-ORBListenEndpoints uiop://1.1@/tmp/foo,1.2@/home/bar/baz
```

Multiple Protocol Endpoint Examples

The following example shows how to use the `-ORBListenEndpoints` command line option to specify that the ORB is to listen on an IIOP endpoint at port number `12345` on host `tango` and at the same time on a default UIOP endpoint.

```
myserver -ORBListenEndpoints iiop://tango:12345 -ORBListenEndpoints uiop://
```

Here are some additional examples of multiple protocol endpoint specification:

```
-ORBListenEndpoints
'iiop://1.1@foo1:0;shmiop://1.1@0,3456;uiop://1.1@/tmp/foo1'
```

17.13.44 ORBLogFile *file*

Description The `-ORBLogFile` option causes all `ACE_DEBUG` and `ACE_ERROR` output to be redirected to *`file`*.

Usage The default destination for output from `ACE_DEBUG` and `ACE_ERROR` is `stderr`. Use this option to redirect this output to *`file`*.

This option may be used with the `-ORBDaemon` option (17.13.7) to capture the `ACE_DEBUG` and `ACE_ERROR` output that would otherwise be lost because the `-ORBDaemon` option causes both `stdout` and `stderr` to be closed.

See Also 17.13.7, 17.13.8, 17.13.9

Example The following example shows how to redirect `ACE_DEBUG` and `ACE_ERROR` output to the file `myLogFile`:

```
myserver -ORBLogFile /tmp/myLogFile
```

The following example shows how to capture the `ACE_DEBUG` and `ACE_ERROR` output from a daemon process:

```
myclient -ORBDaemon -ORBLogFile /tmp/myLogFile
```

17.13.45 ORBMaxMessageSize *maxsize*

Values for *maxsize*

`0 (default)`	Allow GIOP messages of unlimited size. No GIOP fragmentation occurs
`> 0`	Specifies the maximum number of bytes for a GIOP message. Messages larger than this are fragmented and sent in their individual fragments.

Description This option controls whether and when TAO fragments GIOP messages. By default, TAO does not fragment any messages and sends messages of any size as a single GIOP message. When this option is set to a non-zero value, TAO uses that as a maximum message size and will fragment messages as necessary to bring them within that limit. This option only affects TAO's sending behavior for that particular ORB and does not affect its ability to process GIOP fragments that it receives.

Usage Applications that send large GIOP request or reply messages may want to set this option to avoid the processing overhead inherent in long GIOP messages.

Example The following example shows how to use the `-ORBMaxMessageSize` command line option to limit GIOP messages sent to 10KB:

```
myserver -ORBMaxMessageSize 10240
```

17.13.46 ORBMulticastDiscoveryEndpoint *endpoint*

Description When `CORBA::ORB::resolve_initial_references("NameService")` is called, TAO, by default, uses IP multicast to find the TAO Naming Service. It is often the case that more than one Naming Service is running on a local network during development, or in a deployed application where a large number of objects are logically partitioned in some way. In such a case, it is undesirable for all of the Naming Service servers to be listening for multicast requests on the same endpoint.

At startup, the TAO Naming Service joins a multicast group and is assigned a multicast request port. The multicast group IP address and the request port together form the endpoint on which the Naming Service listens for multicast requests. The TAO Naming Service default multicast group IP address is defined in `$ACE_ROOT/ace/OS.h` as `224.9.9.2` and the default multicast request port number is defined in `$TAO_ROOT/tao/default_ports.h` as `10013`.

The `-ORBMulticastDiscoveryEndpoint` option allows the user to specify the multicast group IP address and the multicast request port for the TAO Naming Service. The option `-ORBNameServicePort` allows the user to set the multicast request port, but does not allow the user to set the multicast group. In fact, `-ORBMulticastDiscoveryEndpoint 224.9.9.2:`*`port`* is functionally equivalent to `-ORBNameServicePort` *`port`*. Both options can be used to run more than one Naming Service on the same subnet.

Usage Use `-ORBMulticastDiscoveryEndpoint` when starting the TAO Naming Service server to specify the multicast group it is to join and on which port it is to listen for multicast requests. Use it when starting clients of the TAO Naming Service to specify the multicast group and port to use when multicasting a TAO Naming Service discovery request. This option is only available on platforms that support IP multicast. Both server and client applications may use this option.

In development situations where there is a limitation as to the port(s) a process is allowed to use, `-ORBMulticastDiscoveryEndpoint` may be useful by allowing multiple Naming Services to listen on the same port, but have different multicast IP addresses.

This option has the following format:

```
-ORBMulticastDiscoveryEndpoint address:port
```

where *`address`* is the Naming Service's multicast group IP address. This must be a class D address in the range `224.0.0.0` to `239.255.255.255`. The Naming Service will fail if it is passed an address outside of this range.

`port` is the Naming Service's multicast request port number. This port number must be valid for your operating system and user permissions.

See Also 17.13.47, 22.2

Example This example shows how to use the -`ORBMulticastDiscoveryEndpoint` command line option to specify the multicast request group and port to use:

```
tao_cosnaming -m 1 -ORBMulticastDiscoveryEndpoint 224.1.1.1:9999
myserver -ORBMulticastDiscoveryEndpoint 224.1.1.1:9999
myclient -ORBMulticastDiscoveryEndpoint 224.1.1.1:9999
```

Alternatively the clients can use `-ORBInitRef` or `-ORBDefaultInitRef` to specify the multicast endpoint:

```
myserver -ORBInitRef NameService=mcast://224.1.1.1:9999/NameService
myclient -ORBDefaultInitRef mcast://224.1.1.1:9999
```

17.13.47 ORBNameServicePort *port*

Description When `CORBA::ORB::resolve_initial_references("NameService")` is called, TAO, by default, uses IP multicast to find the TAO Naming Service. It is often the case that more than one Naming Service is running on a local network during development, or in a deployed application where a large number of objects are logically partitioned in some way. In such a case, it is undesirable for all of the Naming Service servers to be listening for multicast requests on the same port.

At startup, the TAO Naming Service is assigned a port on which to listen for multicast requests. The `-ORBNameServicePort` option allows the user to specify this port. If this option is not used, then the `NameServicePort` environment variable (if it is set) is used. Otherwise, if this option is not used and the environment variable is not set, then the port number is set to the value of `TAO_DEFAULT_NAME_SERVER_REQUEST_PORT`, defined in `$TAO_ROOT/tao/default_ports.h` as `10013`.

Usage Use `-ORBNameServicePort` when starting the TAO Naming Service to specify on which port it is to listen for multicast requests. Use it when starting clients of the Naming Service to tell them the multicast port to use when multicasting a TAO Naming Service discovery request. This option is only available on platforms that support IP multicast. Both server and client applications may use this option.

This option has the following format:

```
-ORBNameServicePort port
```

where *port* is the Name Server's multicast request port number. This port number must be valid for your operating system and user permissions.

See Also 17.13.46, 22.2

Example The following example shows how to use the `-ORBNameServicePort` command line option to specify the multicast request port to use:

```
tao_cosnaming -ORBNameServicePort 12345
myserver -ORBNameServicePort 12345
myclient -ORBNameServicePort 12345
```

which is equivalent to:

```
tao_cosnaming -ORBMulticastDiscoveryEndpoint 224.9.9.2:12345
myserver -ORBMulticastDiscoveryEndpoint 224.9.9.2:12345
myclient -ORBInitRef NameService=mcast://:12345/NameService
```

17.13.48 ORBNegotiateCodesets *enabled*

Values for *enabled*

`0`	Disables the use of the codeset negotiation feature for character and wide character data
`1` (default)	Enables codeset negotiation and possibly character and wide character data translation

Description Codeset negotiation is a tool used by CORBA applications to determine the numerical codes used to represent character data. Codeset negotiation enables applications running on systems that use different character representations, such as UTF-8 or Latin1, to accurately exchange character data, by having one side or the other translate the character codes so that the same character is represented.

There are may instances where this functionality is simply unused, for instance when the peer applications are all using TAO and running on identical or similar hardware, with similarly configured operating systems. In fact, applications that use TAO often operate on embedded systems that would benefit from not even loading the code to support codeset negotiation and character translation into memory.

The default value of this option for dynamically linked applications is 1. Statically linked applications that do not explicitly link to and initialize the codeset library will behave as though `-ORBNegotiateCodesets 0` was set.

Usage All of the functional code for supporting codeset negotiation resides in a separately loadable library called `TAO_Codeset`. When the `-ORBNegotiateCodesets` option is enabled in a dynamically linked application, this library is loaded automatically.

Statically linked applications that want to allow configuration of codeset negotiation must explicitly link to the `TAO_Codeset` library and must also include the initializer code somewhere in the application with:

```
#include "tao/Codeset/Codeset.h"
```

You must enable the MPC feature `negotiate_codesets` to have static applications link to the `TAO_Codeset` library. For example, you may set the following option in `$ACE_ROOT/bin/MakeProjectCreator/config/default.features`:

```
negotiate_codesets=1
```

TAO will report an error, "`Unable to load TAO_Codeset`", if the `-ORBNegotiateCodesets` option is set and the `TAO_Codeset` library is not available. If this occurs, the application will continue to run, but without codeset support.

See Also 18.2.8, 4.3.2.3

Example The following example shows how to disable the use of codeset negotiation:

```
myserver -ORBNegotiateCodesets 0
```

17.13.49 ORBNodelay *enabled*

Values for *enabled*

`0`	Disables `TCP_NODELAY`, thus enabling the Nagle algorithm and introducing a delay into TCP packet sends.
`1` (default)	Eliminates the delay in packet sends introduced by the Nagle algorithm.

Description TAO enables the `TCP_NODELAY` socket option by default, resulting in the disabling of the Nagle algorithm. The Nagle algorithm is used to reduce the number of small packets on a WAN. Using `-ORBNodelay` with an argument of `0` disables the `TCP_NODELAY` socket option and thus enables the Nagle algorithm.

Usage Applications with many small request/reply messages and stringent timing requirements can increase performance and improve predictability by eliminating the delay in TCP packet sends introduced by the Nagle algorithm.

See Also W. Richard Stevens, *UNIX Network Programming*, Volume 1, pages 202-204.

Example The following example shows how to disable `TCP_NODELAY`, thereby enabling the Nagle algorithm:

```
myserver -ORBNodelay 0
```

17.13.50 ORBNoProprietaryActivation

Description This is a new CORBA 3 ORB initialization option. It indicates that a server should avoid the use of any proprietary activation framework upon start up. Registration with an Implementation Repository (IMR) is an example of such proprietary activation framework behavior that could be performed. Future implementations may provide additional behaviors, based on CORBA 3's object reference template (ORT) specification.

Note *This option is not currently supported in TAO. However, if this option is present in the argument list (*`argv[]`*),* `CORBA::ORB_init()` *will accept the option, but raises the* `CORBA::NO_IMPLEMENT` *system exception. Also, TAO currently requires that this option, if present, must be followed by an argument (e.g.,* `-ORBNoProprietaryActivation 1`*), otherwise* `CORBA::ORB_init()` *will not correctly parse the remaining options.*

Usage This option should not be used in the current version of TAO. It is described here only because some ORB initialization code exists to parse this option.

See Also 17.13.66, Section 8.5.1.3 of the CORBA 3.1 specification (OMG Document formal/08-01-04)

17.13.51 ORBNoServerSideNameLookups *enabled*

Values for *enabled*

`0` (default)	Look up host names on the server side.
`1`	Do not look up host names on the server side.

Description By default, TAO looks up the host name when opening endpoints on the server side. This option allows developers to avoid this look-up. This behavior is also controlled by `-ORBDottedDecimalAddresses`, but that option also affects whether names or numbers are placed in the IORs produced. This option only controls the lookup.

Usage Use this option when you want to avoid the name look-up for servers but you still want host names to appear in the IORs produced.

See Also 17.13.13

Example The following example shows how to use the `-ORBNoServerSideNameLookups` command line option to avoid name lookups on the server side:

```
myserver -ORBNoServerSideNameLookups 1
```

17.13.52 ORBObjRefStyle *style*

Values for *style*

`IOR` (default)	Use the standard OMG IOR format.
`URL`	Use a more readable format, similar to the form used in Universal Resource Locators (URLs).

Description This option specifies the format used for printing Interoperable Object References (IORs) as strings. Stringified IORs can be printed using the standard OMG IOR format (“`IOR:`” followed by a long string of hexadecimal

digits) or the more readable Universal Resource Locators (URL)-like format. OMG IOR format is the default.

Usage During development and testing, you will often want to print IORs using the URL-like format for easier debugging. However, the OMG standard IOR format is slightly more efficient when used with operations such as `CORBA::ORB::string_to_object()`.

URL-style object references are provided in a format that is compatible with the OMG-specified `corbaloc` format. See the description of `-ORBInitRef` in 17.13.36 for more information on `corbaloc` object references.

In TAO, UIOP (local IPC) URL-style object references have a similar syntax, as follows:

```
<uiopLoc> = "uiopLoc://"[<addr_list>]["|"<key_string>]
<addr_list> = [<address> ","]* <address>
<address> = [<version> <rendezvous point>]
<rendezvous point> = Valid Filesystem Path
<version> = <major> "." <minor> "@" | empty_string
<major> = number
<minor> = number
<key_string> = <string> | empty_string
```

IORs printed using the URL-like format will have the following form:

```
protocol://V.v@endpoint[/poa_name/...]/ObjectId
```

where `V.v` represents the major and minor protocol version.

For example:

```
corbaloc:iiop:1.2@host.domain.com:46432/NameService

uiopLoc://1.2@/tmp/TAOBZMY5Y|NameService
```

Note *TAO uses a vertical bar '`|`' in UIOP URL-style object references to separate the object key from the rendezvous point. The forward slash '`/`' could not be used because it is a valid character in both the object key and the rendezvous point.*

For transient object references, a time stamp is encoded within the IOR as well.

Both server and client applications may use this option.

Example The following example shows how to use the `-ORBObjRefStyle` command line option to specify that IORs should be displayed using a URL-like format:

```
myserver -ORBObjRefStyle URL
```

17.13.53 ORBParallelConnectDelay *msec*

Description This option, when used with `-ORBUseParallelConnects` option, specifies a delay for use between parallel connects to different endpoints in a profile. The default value for this option is zero, which means that there is no delay between connection attempts. The specified delay value is in milliseconds.

Parallel connection attempts do not have to be truly parallel. In applications that connect to a server with a lot of endpoints, perhaps frequently, the time waiting for a connection to succeed may cause the application to run out of socket descriptors. To avoid this problem, use `-ORBParallelConnectDelay` to introduce a brief delay between starting successive connection attempts. An early successful connection can then avoid many failed ones. If the delay is small, a late success can still be reached quickly.

Usage Attempting parallel connects to a busy server with no connect delay specified can cause unnecessary additional load for the server. In these cases it is better to introduce a delay via this option.

See Also 17.13.69

Example The following example shows how to use the `-ORBParallelConnectDelay` option to set a parallel connection delay of 100 milliseconds.

```
myclient -ORBUseParallelConnects 1 -ORBParallelConnectDelay 100
```

17.13.54 ORBPreferIPV6Interfaces *enabled*

Values for *enabled*

`0` (default)	Clients prefer neither IPv4 nor IPv6 interfaces.s by default, prefer IPv4 when 0 is supplied
`1`	When connecting to a server, TAO should prefer IPv6 interfaces.

Description This option is only available when TAO is built with IPv6 support enabled, through either the MPC `ipv6` feature or `ACE_HAS_IPV6`.

When TAO is connecting to a server, this option specifies that all IPv6 interfaces should be tried before any IPv4 interfaces. Explicitly supplying a 0 argument causes TAO to prefer IPv4 first.
When `ORBPreferIPV6Interfaces` is not provided, interfaces are tried in the order discovered.

This option affects IIOP and SSLIOP.

See Also 17.13.6, 17.13.67

Example The following example shows how to use the `-ORBPreferIPV6Interfaces` command line option to force TAO to try using all IPv6 interfaces first:

```
myserver -ORBPreferIPV6Interfaces 1
```

17.13.55 ORBPreferredInterfaces *list*

Description This option allows a client ORB to be configured during the initialization with the capability to choose preferred interface(s) during the invocation phase. This capability is very useful for clients on multi-homed hosts because it allows the client to choose specific interfaces/networks with which to communicate with specific remote targets, or all remote targets.

Note *Not all transport protocols use sockets. Thus, the arguments provided to these options may be interpreted in a protocol-specific way.*

Usage This option is only significant for ORBs in a client role. The *list* parameter is defined as a comma separated list of target destinations and the interface addresse(s) used to connect to it. The target and interface identifiers are

separated with either an equal sign('=') or a colon (':') although the colon is not supported for IPv6 enabled builds. This option may be repeated to build up a list of associations. The format of the list is:

```
target_network:interface,target_network:interface:...
target_network=interface,target_network=interface:...
```

Both the `target_network` and the `interface` parameters may be specified as host names, IP addresses, or the wild-card character ('*').

See Also 17.13.16

Examples Suppose a client is running on a host with two network interface cards with the following host names:

```
malory.ociweb.com
arthur.ociweb.com
```

Now, suppose this client needs to communicate with a (target) server running on a host named `www.sometargethost.com`. The application wishes to constrain all connections to the server on that target host through the interface `malory.ociweb.com`.

The following example shows how to use the `-ORBPreferredInterfaces` command line option to achieve the desired result:

```
myclient -ORBPreferredInterfaces "www.sometargethost.com=malory.ociweb.com"
```

If *all* connections with remote servers at `sometargethost.com` are to be constrained to the `malory.ociweb.com` interface, the wild-card character `*` could be used, as in the following example:

```
myclient -ORBPreferredInterfaces "*.sometargethost.com=malory.ociweb.com"
```

When your interfaces are not named, you can use IP addresses for the target network or local interface identifiers. In this example, all CORBA traffic is routed through a single network interface regardless of destination:

```
myclient -ORBPreferredInterfaces "*=10.100.201.35"
```

17.13.56 ORBRcvSock *buffersize*

Description You may find it useful, when tuning an application's performance, to control the sizes of the protocol's receive buffer in an attempt to maximize throughput. This can be especially helpful if you have *a priori* knowledge of the application's needs and the dynamics of the network in which the application is operating. The `-ORBRcvSock` option allows you to control the buffer size of the socket on the receiving side.

Note *Not all transport protocols use sockets. Thus, the arguments provided to these options can be thought of as general I/O buffer sizes that may be interpreted in a protocol-specific way.*

Usage Both server and client applications may use this option. The default value for this option, `ACE_DEFAULT_MAX_SOCKET_BUFSIZ`, is defined as 65536 in `$ACE_ROOT/ace/OS.h`. Specify a value in bytes that is less than 65536. Larger values generally improve throughput.

See Also 17.13.61

Example The following example shows how to use the `-ORBRcvSock` command line option to specify receiving socket buffer size:

```
myserver -ORBRcvSock 8192
```

17.13.57 ORBServerId *server_id*

Description The CORBA specification (starting with CORBA 3.0) defines an ORB initialization option to uniquely identify a server to an Implementation Repository (IMR). The `-ORBServerId` option accepts a string for the *server_id* argument. All object reference templates created in this ORB will return the specified server id in the `server_id` attribute. All ORBs created in a server must share the same server id. The default server id in TAO is an empty string.

Usage This option currently has little effect on TAO in terms of how servers interact with the IMR. The ORB's server id is, however, used in the creation of an

object reference template by each new POA that is created in a server. Future revisions may assign more behavior to the server id specified by this option.

See Also 17.13.66, Section 8.5.1.1 of the CORBA 3.1 specification (OMG Document formal/08-01-04)

Example The following example shows how to use the `-ORBServerId` command line option to control the server id of the ORB.

```
myserver -ORBServerId MyServer
```

17.13.58 ORBServiceConfigLoggerKey *logger_key*

Description This option allows you to specify where to write ORB logging output. This option is equivalent to passing the key to the `ACE_Service_Config::open()` function.

Usage Both server and client applications may use this option. The default value is `ACE_DEFAULT_LOGGER_KEY`, which depends on the platform type. Platforms that support stream pipes use `/tmp/server_daemon`. Others use `localhost:20012`.

Example The following will cause logging output to be written to the specified port on the local machine.

```
myserver -ORBServiceConfigLoggerKey localhost:9999
```

17.13.59 ORBSingleReadOptimization *enabled*

Values for *enabled*

`0`	Disables the single read optimization, thereby requiring two read operations to read each GIOP message (one to read the GIOP message header, one to read the message body).
`1` (default)	Each time the ORB reads a GIOP message from the transport, it will read `TAO_MAXBUFSIZE` bytes, hoping to read an entire message. If it reads more than one message (including a partial message), it will queue them for later processing.

Description A GIOP message includes a fixed-size message header and a message body. A field in the message header indicates the size of the message in bytes. An ORB can always read exactly one GIOP message by using the following algorithm:

- Read the fixed-size message header.
- Get the message size from the message header.
- Read the rest of the message as indicated by the size field in the header.

However, performing two read operations for each message may not be the most efficient use of resources. By attempting to always read a larger chunk of bytes and queuing any extra messages (or partial message) for later processing, performance can be greatly improved.

The *single read optimization* in TAO enables the ORB to attempt to read each message in a single read operation rather than in two read operations.

Usage Both server and client applications may use this option.

Normally, the default value of 1 (true) for this option is appropriate because it results in better performance since fewer read operations are performed during request and reply processing.

In real-time applications, this option should be set to 0 (false) to disable the single read optimization, otherwise it may lead to priority inversions. For example, suppose two or more requests arrive at a socket and are held in the socket's receive buffer for reading. If multiple requests are read from the socket in a single read, the first request will be processed and the rest of the requests will be queued. A reactor notification will then be used to wake up a follower thread in the ORB's leader-followers model. Meanwhile, new higher priority requests may arrive on other sockets. But, since the TP reactor (the default reactor type in TAO) dispatches notifications before normal I/O, the

lower-priority queued messages will be processed *before* newly-arrived higher-priority requests, thereby leading to a priority inversion.

See Also Chapter 8

Example The following example shows how to disable TAO's single read optimization:

```
myserver -ORBSingleReadOptimization 0
```

17.13.60 ORBSkipServiceConfigOpen

Description This option skips processing of any service configuration options for this ORB.

Usage Use this option to ensure that the default `svc.conf` is not loaded.

See Also 17.13.63

Example Here is an example of applying the `-ORBSkipServiceConfigOpen` option to prevent loading of a service configuration file:

```
server -ORBSkipServiceConfigOpen
```

17.13.61 ORBSndSock *buffersize*

Description You may find it useful, when tuning an application's performance, to control the sizes of the protocol's send buffer in an attempt to maximize throughput. This can be especially helpful if you have *a priori* knowledge of the application's needs and the dynamics of the network in which the application is operating. The `-ORBSndSock` option allows you to control the buffer size of the socket on the sending side.

Note *Not all transport protocols will use sockets. Thus, the arguments provided to these options can be thought of as general I/O buffer sizes that may be interpreted in a protocol-specific way.*

Usage Both server and client applications may use this option. The default value for this option, `ACE_DEFAULT_MAX_SOCKET_BUFSIZ`, is defined as 65536 in

`$ACE_ROOT/ace/OS.h`. Specify a value in bytes that is less than 65536. Larger values generally improve throughput.

See Also 17.13.56

Example The following example shows how to use the `-ORBSndSock` command line option to specify the sending socket buffer size.

```
myserver -ORBSndSock 4096
```

17.13.62 ORBStdProfileComponents *enabled*

Values for *enabled*

`1` (default)	The ORB generates the OMG standard profile components.
`0`	The ORB does not generate the OMG standard profile components.

Description Agents that accept object requests or that provide locations for objects (i.e., servers) publish object locations using opaque, protocol-specific *profiles.* The standard IORs encapsulate these profiles. A single IOR may contain multiple profiles for a single CORBA object. For example, an IIOP profile includes the host name and port at which the server ORB listens for requests.

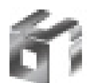
A profile may also contain, in its so-called *tagged components*, additional information, such as the character set understood by the server, security tokens, and priority information. These tagged components are optional, but if present, must be retained as part of the IOR, even if the IOR is passed between ORBs. The default is to generate the optional standard profile components.

This option controls generation of the following optional tagged components:

- `TAG_ORB_TYPE`
- `TAG_CODE_SETS`

Usage Use the `-ORBStdProfileComponents` option to control whether the ORB will generate optional tagged components in IORs. This option is applicable only to server applications.

Specify a value of `0` for this option to suppress generation of the optional tagged components, if clients do not need them, are operating under severe memory constraints, or anticipate a very large number of IORs.



Impact Suppressing the generation of these optional tagged components saves 28 bytes per IOR. Since these components are optional, the use of this option to suppress them does not impact interoperability with other GIOP 1.1- or 1.2-compliant ORBs.

See Also The definition of `TAO_STD_PROFILE_COMPONENTS` in `$TAO_ROOT/tao/orbconf.h`.

Example The following example shows how to use the `-ORBStdProfileComponents` command line option to instruct the ORB not to generate optional standard profile components:

```
myserver -ORBStdProfileComponents 0
```

17.13.63 ORBSvcConf *config_file_name*

Description By default, a service-configurator-based application looks in the current directory for a file named `svc.conf`, which supplies directives to the service configurator. You can use the `-ORBSvcConf` option to specify a different file name. The *config_file_name* argument can be any valid path. The target file may contain directives for non-TAO-related service objects and for TAO components. You can supply multiple files to the service configurator by supplying multiple `-ORBSvcConf` options on the command line. The directives in each file are supplied to the service configurator in an additive fashion. Using `-ORBSvcConf` on the command line is equivalent to passing the `-f` option to the `open()` function of the `ACE_Service_Config` class.

Usage Both server and client applications may use this option. It is only applicable on platforms with file systems.

A common usage for this option is when a directory contains several service-configurator-based applications. Each needs its own service configuration file, so it is not possible to use the default file name `svc.conf`. Use this option to supply a different configuration file to each application. You may also use it to supply different configurations to separate instances of the same application.

See Also 16.4

Example The following examples show how to use the `-ORBSvcConf` command line option to specify alternate service configuration files to applications:

```
myserver -ORBSvcConf tsscommon.conf
```

```
myserver -ORBSvcConf $TESTROOT/rt_test/svc.conf -ORBSvcConf $HOME/svc.conf
```

17.13.64 ORBSvcConfDirective *directive*

Description Use `-ORBSvcConfDirective` to supply a single directive to the service configurator from the command line. This option may be repeated to pass multiple directives. The *directive* argument can be any valid service configurator directive string. Using `-ORBSvcConfDirective` on the command line is equivalent to passing the `-S` option to the `open()` function of the `ACE_Service_Config` class.

Usage Both server and client applications may use this option. When it is not possible to use a service configuration file with an application, applications may use this option. A good example is an environment where there is no file system. Developers may also use this option during development and testing to quickly try alternate configurations, without modifying the configuration file.

See Also 16.4, 17.13.63

Example The following example shows how to use the `-ORBSvcConfDirective` command line option to pass directives to the ORB's service configurator:

```
myserver -ORBSvcConfDirective "static Resource_Factory '-ORBReactorType tp'"
```

17.13.65 ORBTradingServicePort *port*

Description When `CORBA::ORB::resolve_initial_references("TradingService")` is called, TAO, by default, uses IP multicast to find the TAO Trading Service. It is often the case that more than one Trading Service is running on a local network during development, or in a deployed application where a large number of objects are logically partitioned in some way. In such a case, it is undesirable for all of the Trading Service servers to be listening for multicast requests on the same port.

At startup, the TAO Trading Service is assigned a port on which to listen for multicast requests. The `-ORBTradingServicePort` option allows the user to specify this port. If this option is not used, then the `TradingServicePort` environment variable (if it is set) is used. Otherwise, if this option is not used

and the environment variable is not set, then the port number is set to the value of `TAO_DEFAULT_TRADING_SERVER_REQUEST_PORT`, defined in `$TAO_ROOT/tao/default_ports.h` as `10016`.

Usage Use `-ORBTradingServicePort` when starting the TAO Trading Service to specify on which port it is to listen for multicast requests. Use it when starting clients of the Trading Service to tell them the multicast port to use when multicasting a TAO Trading Service discovery request. This option has the following format:

```
-ORBTradingServicePort port
```

where *`port`* is the Trading Service's multicast request port number. This port number must be valid for your operating system and user permissions.

Both server and client applications may use this option. This option is only available on platforms that support IP multicast.

Example The following example shows how to use the `-ORBTradingServicePort` command line option to specify the multicast request port to use:

```
$TAO_ROOT/orbsvcs/Trading_Service/tao_costrading -ORBTradingServicePort 12345
myserver -ORBTradingServicePort 12345
```

17.13.66 ORBUseIMR *enabled*

Values for *enabled*

`0` (default)	Do not use the Implementation Repository (ImplRepo).
`1`	Use the ImplRepo.

Description The use of the `-ORBUseIMR` option has the following effects:

- In the ImplRepo, each of the server's persistent POAs has its address set to the server's address and its current execution status set to `running`.
- The server embeds the ImplRepo's address, rather than its own address, in the IORs of its persistent objects.

Usage In order for a server to use the ImplRepo,

- The name of each of its persistent POAs must be added to the ImplRepo.
- It must obtain an object reference to the ImplRepo.

- It must be started using the `-ORBUseIMR` option with a value of `1`.

The environment variable `TAO_USE_IMR` can be used instead of passing `-ORBUseIMR` option on a server's command line. Also, if the `tao_imr_activator` is used to start a server, the `-ORBUseIMR` option is automatically added to the server's command line options by the Activator.

See Also Chapter 28

Example If the command

```
tao_imr_locator -o implrepo.ior
```

is used to start the ImplRepo, the command:

```
tao_imr -ORBInitRef ImplRepoService=file://implrepo.ior add MyPoa
```

is used (see 28.3) to add the name `MyPoa` to the repository, and if `MyServer` is a server with a single persistent POA named `MyPoa`, the command

```
MyServer -ORBUseIMR 1 -ORBInitRef ImplRepoService=file://implrepo.ior
```

causes the following to occur:

- `MyServer` is started.
- In the ImplRepo, `MyPoa`'s address is set to `MyServer`'s address.
- In the ImplRepo, `MyPoa`'s current execution status is set to `running`.
- The address of the ImplRepo that was obtained from `implrepo.ior` is used to construct IOR's for `MyPoa`'s objects.

17.13.67 ORBUseIPV6LinkLocal *enabled*

Values for *enabled*

`0 (default)`	Disallow use of the IPv6 link local address.
`1`	Allow use of the IPv6 link local address.

Description Link local IPv6 addresses are non-routable, and thus are not suitable for use in object references that are going to be distributed across a network. The are disabled by default and can be enable using this option.

This option affects IIOP and DIOP.

Usage This option is useful when performing validation tests between hosts that are connected to the same link.

See Also 17.13.6, 17.13.54

17.13.68 ORBUseLocalMemoryPool *enabled*

Values for *enabled*

`0 (default)`	TAO should use the platform's default memory allocator.
`1`	TAO should use a local memory pool in place of the platform's default memory allocator.

Description This option determines which memory allocator TAO uses for a variety of internal purposes (such as message buffers). You can override the default value at build time via the `TAO_USES_LOCAL_MEMORY_POOL` macro or set it at run-time via this option.

You can directly specify the allocator to use for output CDR buffers via the `-ORBOutputCDRAllocator` resource factory option. For those buffers, that option can override the allocator specified here.

Usage The trade-off between the platform's default memory allocator and TAO's memory pool is dependent on a number of factors including the platform you are on, the nature of your application, and the nature of your usage patterns. TAO's memory pool can often be more efficient than the platform's allocator. One disadvantage of TAO's memory pool is that once it allocates memory it never returns it to the system.

See Also 18.6.12

Example The following example shows how to use the `-ORBUseLocalMemoryPool` command line option to specify use of the memory pool allocator:

```
myserver -ORBUseLocalMemoryPool 1
```

17.13.69 ORBUseParallelConnects *enabled*

Values for *enabled*

`0 (default)`	Endpoints in a profile are tried one at a time. If a connect to one succeeds it is used. If it fails, then the next endpoint in the profile is tried.
`1`	Parallel connect feature is enabled. Parallel connections are attempted to all endpoints in the profile. The first to succeed is used.

Description A server on a multi-homed host can produce profiles for object references that contain many endpoints. For specific clients, some of the endpoints in the profiles will be unreachable. By default, TAO clients try connecting to endpoints sequentially, perhaps encountering a very long delay while failing to reach unreachable endpoints.

When this option is enabled, TAO will attempt to connect to all the endpoints in the profile in parallel. The first to either be found in the cache or successfully connect is used and the other connection attempts are terminated.

The `-ORBParallelConnectDelay` option can be used to introduce delays between the start of each endpoint connect.

Usage Use this option when you are using servers that publish IORs with multiple endpoints and you want to optimize your connection times. For example, hosts that belong to multiple networks could have multiple endpoints, only one of which is reachable by a given client. Clients of such a server should use this option to avoid lengthy delays when connecting.

Note To take full advantage of this option, you must use enable the `-ORBUseSharedProfile` option on the server to allow shared profiles. Failure to do so would result in separate profiles for each endpoint and would not result in any parallelism for the clients.

Impact Use of this option will cause increased resource usage in terms of memory, processing, and the network.

See Also 17.13.53, 17.13.70

Example

```
myserver -ORBUseSharedProfile 1
myclient -ORBUseParallelConnects 1
```

17.13.70 ORBUseSharedProfile *enabled*

Values for *enabled*

`0`	Specifies that the ORB should create a new profile for each endpoint when building profiles for IORs.
`1` (default)	Specifies that the ORB will combine multiple endpoints (with the same protocol and protocol version) into a single profile.

Description When an IOR is created, the endpoints on which the ORB is listening for requests are encoded into the IOR. The endpoints are contained within one or more *profiles* in the IOR. An IOR may have multiple profiles and each profile may have multiple endpoints. This option controls whether or not TAO will combine multiple endpoints having the same protocol and protocol version (e.g., IIOP 1.2) into a single profile.

Usage Use the `-ORBUseSharedProfile` option to control whether the ORB will combine multiple endpoints into a single profile in IORs. This option is applicable only to server applications.

Impact Combining multiple endpoints into a single profile results in slightly smaller IORs. There is no appreciable performance impact associated with the use of this option. Some legacy client ORBs may not be able to access more than one endpoint from a single profile.

See Also 17.13.43

Example

```
myserver -ORBUseSharedProfile 0
```

17.13.71 ORBVerboseLogging level

Values for *level*

`0(default)`	The same as not using the option.
`1`	Adds a timestamp to prefix each line of code.
`2`	Add even more verbosity

Description This option controls the amount of status data printed on each line of the debug log. Higher numbers generate more output.

Usage Use the `-ORBVerboseLogging` option to obtain additional timing information about each debug log entry.

Impact The additional information printed out for each log entry may decrease your application performance.

See Also 17.13.9, 17.13.44

Example

```
myserver -ORBVerboseLogging 1 -ORBDebugLevel 10
```

CHAPTER 18

Resource Factory

18.1 Introduction

The resource factory is responsible for constructing and providing access to various resources used by ORBs within both clients and servers. ORB resources include a reactor, protocol factories, and allocators for buffers used during encoding and decoding of data transmitted between ORBs. Protocol factories are responsible for supplying acceptors, connectors, and connection handlers to enable interprocess communication.

TAO provides several different types of resource factories, summarized in Table 18-1. The choice of resource factory to use depends upon such considerations as the features your application requires and the environment in which it will run. Many applications will function just fine with the default resource factory and will not require any of the special features found in the alternate resource factory implementations.

Table 18-1 Resource Factories provided with TAO

Resource Factory	Section	Description
Default Resource Factory	18.2	Unless configured otherwise, this is the resource factory used.
Qt Resource Factory	18.3	This is a specialized resource factory providing the means for integrating with the Qt GUI toolkit from Trolltech.
Xt Resource Factory	18.4	This is a specialized resource factory providing the means for integrating with the X Window System's Xt Intrinsics toolkit.
Fox Resource Factory	N/A	This is a specialized resource factory providing the means for integrating with the Fox toolkit.
Fl Resource Factory	N/A	This is a specialized resource factory providing the means for integrating with the Fast Light toolkit (fltk).
Tk Resource Factory	N/A	This is a specialized resource factory providing the means for integrating with the Tk framework.
Advanced Resource Factory	18.5	This factory provides more advanced configuration options in addition to all the features of the default resource factory.

With the exception of the advanced resource factory, the resource factory used by the ORB is registered with the service configurator as `Resource_Factory`. TAO's default resource factory is statically registered with the service configurator, so the `static` directive is used to supply initialization options to it. To change the behavior of the default resource factory, add a line similar to the line below to your service configuration file:

```
static Resource_Factory "-ORBoption value -ORBoption value ..."
```

The service configurator is discussed in greater detail in 16.3. The options supported by the default resource factory are discussed in the following pages.

18.2 Interface Definition

Within a TAO application, the resource factory is accessed via an interface defined by the base class `TAO_Resource_Factory`. Here we show all of the operations of the resource factory and describe their use in relationship to the ORB, as well as the behavior of the default resource factory implementation provided with TAO.

Whereas a reference to the resource factory may be obtained through the service configurator, it is not intended for use external to the ORB core. TAO's ORB core object supplies the interface for retrieving objects supplied by the resource factory.

Synopsis

```
class TAO_Export TAO_Resource_Factory : public ACE_Service_Object {
   enum Purging_Strategy {
      LRU,
      LFU,
      FIFO,
      NOOP
   };

   enum Resource_Usage{
      TAO_EAGER,
      TAO_LAZY
   };

   TAO_Resource_Factory (void);
   virtual ~TAO_Resource_Factory (void);
   virtual void disable_factory (void) = 0;
   virtual ACE_Reactor* get_reactor (void);
   virtual void reclaim_reactor (ACE_Reactor* reactor);
   virtual TAO_LF_Strategy* create_lf_strategy (void) = 0;
   virtual TAO_Acceptor_Registry* get_acceptor_registry (void);
   virtual TAO_Connector_Registry* get_connector_registry (void);
   virtual ACE_Allocator* input_cdr_dblock_allocator (void);
   virtual ACE_Allocator* input_cdr_buffer_allocator (void);
   virtual ACE_Allocator* input_cdr_msgblock_allocator (void);
   virtual int input_cdr_allocator_type_locked (void);
   virtual ACE_Allocator* output_cdr_dblock_allocator (void);
   virtual ACE_Allocator* output_cdr_buffer_allocator (void);
   virtual ACE_Allocator* output_cdr_msgblock_allocator (void);
   virtual int use_locked_data_blocks (void) const;
   virtual TAO_Flushing_Strategy* create_flushing_strategy (void) = 0;
   virtual TAO_Connection_Purging_Strategy* create_purging_strategy (void) = 0;
   virtual int purge_percentage (void) const;
   virtual int cache_maximum (void) const;
   virtual int max_muxed_connections (void) const;
```

```
virtual int locked_transport_cache(void);
virtual ACE_Lock* create_cached_connection_lock (void);
virtual ACE_Lock* create_object_key_table_lock (void);
virtual int init_protocol_factories (void);
virtual TAO_ProtocolFactorySet* get_protocol_factories (void);
virtual int get_parser_names (char**& names, int& number_of_names);
virtual ACE_Allocator* amh_response_handler_allocator(void);
virtual ACE_Allocator* ami_response_handler_allocator(void);
virtual ACE_Allocator* create_corba_object_lock(void);
virtual TAO_Resource_Factory::Resource_Usage resource_usage_strategy(void)
const = 0;
virtual TAO_Codeset_Manager* codeset_manager (void);
virtual TAO_Configurable_Refcount create_corba_object_refcount (void);
virtual auto_ptr<TAO_GIOP_Fragmentation_Strategy>
    create_fragmentation_strategy (TAO_Transport * transport,
                                   CORBA::ULong max_message_size) const = 0;
virtual bool drop_replies_during_shutdown () const = 0;
virtual void use_local_memory_pool (bool);
};
```

18.2.1 Reactor Management

The resource factory interface defines an operation that returns a pointer to a newly-created reactor. Each ORB instance in an application will have a reactor, and may even have multiple reactors, but no reactor will be shared among ORBs.

```
virtual ACE_Reactor* get_reactor (void);
```

TAO will use a reactor when dealing with connection management involving the predefined protocols, for example IIOP, UIOP, and SHMIOP. At the very least, TAO uses the reactor to supply non-blocking event-handling behavior in the ORB. The interface also defines an operation that reclaims or deallocates any resources used by the reactor.

```
virtual void reclaim_reactor (ACE_Reactor* reactor);
```

In addition to the default reactor, TAO provides various other reactors for specialized needs. Specialized reactors are used with the Qt, Fl, Fox, Tk, and Xt resource factories. Also, the advanced resource factory allows the selection of other TAO-provided reactors if the default does not suit your needs. See 18.5 for information on using the advanced resource factory and selecting other reactors.

Another operation that relates to the reactor is `create_lf_strategy()`:

```
virtual TAO_LF_Strategy* create_lf_strategy (void) = 0;
```

The `TAO_LF_Strategy` returned by this operation tells the ORB the method by which threads vie for access to the reactor. The default resource factory always returns a valid strategy, so the leader-follower model will be used to synchronize access to the reactor. The advanced resource factory will return a valid leader-follower strategy unless the reactor type chosen is `select_st`, in which case it will return a null strategy.

An advanced resource factory option, shown in table Table 18-2, is used to specify the type of reactor to be used by the ORB.

Table 18-2 Options for Configuring the ORB's Reactor Type

Option	Section	Description
`-ORBReactorType` *`reactor_type`*	18.7.6	Specify the type of reactor the ORB uses to handle incoming connections, requests, and other events.

18.2.2 Allocators for CDR Conversion

Before data is transferred between ORBs, either as parameters to an operation or as the result from an operation, the data must be put into a form that is suitable for transmission. TAO is responsible for encoding the data on the sending side and decoding the data on the receiving side. Encoding the data involves *marshaling*, which is gathering up the bits of information to be transmitted and placing them in a predictable order. The predictable order specified by GIOP is called the Common Data Representation, or CDR. The CDR is discussed in detail in *Advanced CORBA Programming with C++,* 13.3.

When TAO encodes data for sending or decodes data received, space is required to hold the intermediate form of the data. This interim space is managed using `ACE_Message_Block`s. Message blocks contain data in units of `ACE_Data_Block`s, which provide a reference-counted wrapper around the actual data buffer. The space used by these buffers is owned by the ORB that manipulates them, therefore the allocation strategy used to manage the space is provided by the resource factory. Different allocators can be used for `ACE_Message_Blocks`. In addition, the `ACE_Message_Block` provides the ability to use separate allocators for the `ACE_Data_Blocks` and the buffers.

Therefore, the resource factory interface provides operations used by the ORB to access allocators used to create `ACE_Message_Block` and `ACE_Data_Block` objects and data buffers for input and output. The input CDR allocator accessors are:

```
virtual ACE_Allocator* input_cdr_msgblock_allocator (void);
virtual ACE_Allocator* input_cdr_dblock_allocator (void);
virtual ACE_Allocator* input_cdr_buffer_allocator (void);
```

The buffers used to receive and decode incoming data may be passed up the processing chain from the actual communication endpoint, through the ORB, to the application code responsible for dealing with the data. Locking may be required if multiple threads will access the data. The locked input CDR allocator is:

```
virtual int input_cdr_allocator_type_locked (void);
```

The advanced resource factory option `-ORBInputCDRAllocator` (see 18.7.4) is provided to specify the type of lock used to protect access to the input CDR allocators and the data they allocate. See 18.5 for information on using the advanced resource factory. The default resource factory always returns a non-zero value, which means access to the allocator for creating and destroying data buffers is synchronized and therefore safe to be used by many threads.

The following operations are used by the ORB to access allocators used for encoding outgoing data:

```
virtual ACE_Allocator* output_cdr_msgblock_allocator (void);
virtual ACE_Allocator* output_cdr_dblock_allocator (void);
virtual ACE_Allocator* output_cdr_buffer_allocator (void);
```

Access to the `ACE_Data_Blocks` used to encode data being transmitted does not normally need to be synchronized since the data blocks are not used for any application-level processing. The ORB core provides data blocks used during the CDR conversion process and uses the following resource factory operation to determine the type of data block it is expected to produce:

```
virtual int use_locked_data_blocks (void) const;
```

The default resource factory's implementation returns a non-zero value from `use_locked_data_blocks()` when `-ORBConnectionCacheLock` is set to `thread` and returns zero when `-ORBConnectionCacheLock` is set to `null`. The `-ORBConnectionCacheLock` option also affects connection caching. (See 18.2.5 for a discussion of connection caching in TAO.)

In addition, the advanced resource factory's implementation returns a non-zero value from `use_locked_data_blocks()` when `-ORBInputCDRAllocator` is set to `thread`, and returns zero when `-ORBInputCDRAllocator` is set to `null`.

Since `-ORBConnectionCacheLock` and `-ORBInputCDRAllocator` are separate options, both of which affect data-block locking, it is possible to provide conflicting values when using the advanced resource factory. If conflicting values are given, the value provided by `-ORBConnectionCacheLock` will be used for data-block locking. If you use both of these options, they should have the same setting to prevent unexpected results.

The `-ORBUseLocalMemoryPool` `ORB_init()` option can be used to select the type of allocator to use for the allocators discussed in this section. See 17.13.68 for details.

An advanced resource factory option, shown in table Table 18-3, is used to control the type of locking used for accessing the input CDR conversion allocator. Also shown is the corresponding option for controlling the output CDR allocator and the relate zero-copy write optimization option.

Table 18-3 Options for Configuring Input CDR Conversion Allocator

Option	Section	Description
`-ORBInputCDRAllocator {`**`thread`**` | null}`	18.7.4	Specify the type of locking to be used for accessing the input CDR conversion allocator.
`-ORBOutputCDRAllocator {`**`default`**` | mmap | local_memory_pool}`	18.6.12	Specify the allocator type to use for output CDR buffers.
`-ORBZeroCopyWrite`	18.6.18	Enable the ORB to use a zero-copy write optimization (implicitly specifies the output CDR buffer allocator).

18.2.3 Message Flushing Strategies

Data sent from one ORB to another must be converted into a common data representation (CDR). After CDR conversion, the data is stored in `ACE_Message_Block`s. The message blocks are queued until they can be flushed to the outgoing transport layer. The strategy by which messages are flushed can vary, and affects the timing of the data being sent. The resource factory interface defines the following operation, which returns the strategy to be used for flushing messages:

```
virtual TAO_Flushing_Strategy* create_flushing_strategy (void) = 0;
```

The default flushing strategy returned by the default resource factory uses the leader-follower pattern. Table 18-4 shows the option that affects which message flushing strategy is used by the ORB.

Table 18-4 Message Flushing Strategy Options

Option	Section	Description
`-ORBFlushingStrategy {`**`leader_follower`**` | reactive | blocking}`	18.6.7	Selects the strategy for how messages are flushed from the queue.

18.2.4 Shutdown Strategies

When `shutdown` is called on an ORB, TAO can either let waiting client threads continue to wait for a reply keep waiting or throw an exception. Table 18-5 shows the option that affects this behavior of the ORB.

Table 18-5 Shutdown Strategy Options

Option	Section	Description
`-ORBDropRepliesDuringShutdown (0 | 1)`	18.6.6	Specifies whether pending replies are dropped when `shutdown` is called on the ORB.

18.2.5 Connection Cache Management Strategies

A client ORB establishes at least one connection to every server with which it communicates. Under certain circumstances, more than one connection might be established between a given client and a given server. For example, you can configure the client-side ORB such that it will not multiplex requests over a single connection (see 20.3.6), or you can establish multiple connections to a

server where each connection is used to transmit requests of a certain priority or range of priorities (see 8.3.10).

The ORB maintains connections in a cache. When a request is about to be sent, the ORB looks for an existing connection in the cache to use to transmit the message. If a suitable connection is not available, the ORB may establish a new connection for the message, then add the new connection to the cache. TAO relies on these cached connections in both the server and the client for optimization. As connections accumulate in the cache, the cache may need to be purged to allow new connections to be added.

The `TAO_Acceptor` and `TAO_Connector` both use cache management (purging) strategies supplied by the resource factory. The following operations of the resource factory interface deal with cache management or purging:

```
enum Purging_Strategy {
   LRU,
   LFU,
   FIFO,
   NOOP
};

virtual int locked_transport_cache (void);
virtual TAO_Connection_Purging_Strategy* create_purging_strategy (void) = 0;
virtual int purge_percentage (void) const;
virtual int cache_maximum (void) const;
virtual int max_muxed_connections (void) const;
virtual ACE_Lock* create_cached_connection_lock (void);
```

There are four defined purging strategies used by these operations:

- *least recently used* (`LRU`)—this strategy removes connections that have been idle the longest.
- *least frequently used* (`LFU`)—this strategy removes connections that are used least often.
- *first in first out* (`FIFO`)—this strategy removes the oldest connections, regardless of how often they are used.
- *no operation* (`NOOP`)—this strategy does not purge connections at all.

The `create_purging_strategy()` operation creates and returns a `TAO_Connection_Purging_Strategy` object that will match one of the above-defined strategies. The `TAO_Connection_Purging_Strategy`

object is passed to the TAO_Transport_Cache_Manager, thereby instructing it on how to purge the cache. The default resource factory uses the LRU strategy exclusively. The advanced resource factory can return any one of these strategies as instructed by the -ORBConnectionPurgingStrategy option. See 18.5 for information on using the advanced resource factory.

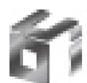
The purge_percentage() operation indicates the quantity of the cache to remove if conditions require purging. The value returned must be between 0 and 100. The default resource factory returns 20 as the purge percentage by default.

The cache_maximum() operation returns the maximum number of connections that may be cached in the ORB. The default value is system dependent and is based on the maximum number of file descriptors available to the process.

The max_muxed_connections() operation returns the maximum number of multiplexed connections that are allowed for a single remote endpoint. The default value is zero, meaning there is no theoretical limit on the number of connections that can be established (and cached) for a given remote endpoint. If a value greater than zero is returned, and that limit is reached (i.e., the maximum number of multiplexed connections have already been established to a given endpoint), the thread needing a connection to transmit a request will wait on a condition variable to be awaken when an existing connection becomes idle.

The create_cached_connection_lock() operation returns the type of lock that should be used when threads indirectly access the connection cache through the ORB. Concrete, protocol-specific, derived classes of the TAO_Transport abstract base class use this lock to ensure thread-safe access to the connection cache when sending a request. The possible lock types returned are a null lock (for no locking, as in single-threaded applications) and a thread-safe lock (for use in multithreaded applications). The default resource factory returns the thread-safe lock by default.

The locked_transport_cache() operation returns a boolean value to indicate whether the connection cache needs to have a lock or not. The TAO_Transport_Cache_Manager uses the value returned by this operation to determine whether a lock should be used when threads access the connection cache directly through its interface. The default resource factory's implementation of this operation returns false when the -ORBConnectionCacheLock option is set to null and true when this



option is set to `thread`. By default, the default resource factory returns `true`. Therefore, unless configured otherwise, access to the connection cache is synchronized.

Table 18-6 shows the options that control the cache management behavior of the default resource factory.

Table 18-6 Connection Cache Management Options

Option	Section	Description
`-ORBConnectionCachePurgePercentage` *percent*	18.6.4	Supply the amount of the cache to purge. The default is 20 percent.
`-ORBConnectionCacheMax` *limit*	18.6.3	Supply the maximum number of connections that may be cached in the ORB. The default is system dependent.
`-ORBConnectionCacheLock {`**`thread`**` | null}`	18.6.2	Select the type of lock the ORB will use to access the connection cache. The default is `thread`.
`-ORBMuxedConnectionMax` *limit*	18.6.9	Supply the maximum number of multiplexed connections allowed per remote endpoint.

18.2.6 Protocol Factories

TAO is intended to support more than just the common TCP/IP-based IIOP for communication between ORBs. There are many situations where alternative protocols provide a performance advantage or where TCP/IP is not available. You can enable one or more of the other transport protocols supplied with TAO or you can use a custom protocol.

Since multiple protocols can be enabled and made available to the ORB, the resource factory interface defines operations that initialize and return the protocols the ORB should support. These operations are:

```
virtual int init_protocol_factories (void);
virtual TAO_ProtocolFactorySet* get_protocol_factories (void);
```

Enabling the use of multiple protocols is a two-step process:

1. Define the protocol by supplying a protocol factory (see 14.18.3).
2. Make the protocol factory available to the ORB.

Table 18-7 shows the option to make a protocol factory available to the ORB. This option can be repeated to make multiple protocol factories available.

Table 18-7 Protocol Factory Registration Options

Option	Section	Description
`-ORBProtocolFactory` *`pfactory`*	18.6.13	Declare that a protocol factory named *`pfactory`* is available for supplying connections.

The protocols supplied with TAO are described in Chapter 14.

18.2.7 Custom IOR Parsers

CORBA defines a generic format, called Interoperable Object Reference (IOR), to identify CORBA objects and define one or more paths through which an object can be accessed. Each path references a server location that implements the object and an opaque object key that is valid relative to that particular server's location.

TAO supports several IOR formats, including `corbaloc:`, `corbaname:`, `IOR:`, and `file:`. However, some applications may benefit from other formats. For example, an `http:` IOR format might allow applications to download an object reference from a web server.

However, for a particular IOR format can be used, the ORB must be able to parse it to make it usable to the application. TAO's resource factory allows application developers to implement custom IOR parsers and dynamically register them with the ORB.

The resource factory interface defines the following operation, which the ORB uses to retrieve the names of the IOR parsers it should use:

```
virtual int get_parser_names (char**& names, int& number_of_names);
```

The ORB's `string_to_object()` operation queries each available IOR parser to see if it is can parse the given stringified IOR format. When the ORB finds a match, it uses that parser to convert the string into an object reference. Using an IOR parser is more convenient than adding configuration code in the application's `main()` function. It also allows for easier integration with other TAO components, such as the `-ORBInitRef` option (see 17.13.36).

To implement an IOR parser, you must implement a class derived from `TAO_IOR_Parser`. The example below shows what an HTTP IOR parser might look like:

```
class HTTP_Parser : public TAO_IOR_Parser
{
public:
  virtual int match_prefix (const char* ior_string) const;
  virtual CORBA::Object_ptr parse_string (const char* ior,
                                          CORBA::ORB_ptr orb);
};
```

The `match_prefix()` function must recognize all the IOR prefixes that this parser supports. Normally, each IOR parser understands only one prefix. A typical implementation of the `match_prefix()` function might look like this:

```
int HTTP_Parser::match_prefix (const char* ior_string) const
{
  static const char http_prefix[] = "http:";
  int cmp = ACE_OS::strncmp (ior_string, http_prefix, sizeof(http_prefix));
  return (cmp == 0);
}
```

The `parse_string()` function implements the actual string parsing. In your implementation of `parse_string()`, you can safely assume that the string has been validated by the `match_prefix()` function. Typically, `parse_string()` will obtain or construct a string matching one of the predefined IOR formats, such as a `corbaloc:` Object URL. In this example, the more "normal" IOR is obtained by downloading a document from a web server. It then uses `string_to_object()` on the downloaded IOR string to return the object reference:

```
CORBA::Object_ptr HTTP_Parser::parse_string (
    const char* ior,
    CORBA::ORB_ptr orb)
{
  // Parse IOR as if it was an http: URL
  ACE_URL_Addr* url_addr = ACE_URL_Addr::create_addr (ior);

  ACE_HTTP_Addr* http_addr = dynamic_cast<ACE_HTTP_Addr*>(url_addr);

  // Connect to the remote host and download the web page, store the
  // contents in:
```

```
  char* document_contents = ...;

  return orb->string_to_object (document_contents);
}
```

TAO uses the ACE Service Configurator framework to find the IOR parsers. See 16.5 for more information on the ACE Service Configurator. The next part of the example shows how to integrate the IOR parser with the service configurator.

First you must declare, in the header file, a factory function and a description of the service, this is easily accomplished via the following ACE macros:

```
ACE_STATIC_SVC_DECLARE_EXPORT (Export_Prefix, HTTP_Parser)
ACE_FACTORY_DECLARE (Export_Prefix, HTTP_Parser)
```

If you are only going to use Unix-like compilers and linkers, then you can simply use `TAO` in place of `Export_Prefix`. However, under Microsoft Windows variants, this string must be the prefix of the DLL export/import macros used by your library. If you are going to statically link your IOR Parser into the application you will also need to add the `ACE_STATIC_SVC_REQUIRE` macro, as follows:

```
ACE_STATIC_SVC_DECLARE_EXPORT (Export_Prefix, HTTP_Parser)
ACE_FACTORY_DECLARE (Export_Prefix, HTTP_Parser)
ACE_STATIC_SVC_REQUIRE (HTTP_Parser)
```

Next, you must implement the services defined above. Using another group of helper macros, you should add the following in your source file:

```
ACE_STATIC_SVC_DEFINE (HTTP_Parser,
                       ACE_TEXT ("HTTP_Parser"),
                       ACE_SVC_OBJ_T,
                       &ACE_SVC_NAME (HTTP_Parser),
                       ACE_Service_Type::DELETE_THIS |
                       ACE_Service_Type::DELETE_OBJ,
                       0)
ACE_FACTORY_DEFINE (Export_Prefix, HTTP_Parser)
```

The second argument to `ACE_STATIC_SVC_DEFINE` is the name of the service in the ACE Service Configurator. It is customary, but not required, to use the name of the class.

The IOR parsers in the ORB can serve as more complete examples. You might want to look in `$TAO_ROOT/tao` at `FILE_Parser.h`, `CORBALOC_Parser.h`, `CORBANAME_Parser.h`, `DLL_Parser.h`, or `MCAST_Parser.h`.

Finally, you can dynamically add your IOR parser using the `-ORBIORParser` resource factory option. For example, to add our HTTP IOR parser, we would add the following service configurator directive:

```
static Resource_Factory "-ORBIORParser HTTP_Parser"
```

Table 18-8 shows the option that allows adding custom parsers to the resource factory.

Table 18-8 Options for Adding IOR Parsers

Option	Section	Description
`-ORBIORParser` *`parser_name`*	18.6.8	Specify the service name of an IOR parser that should be added to the resource factory.

18.2.8 Code Set Identifiers and Translators

The CORBA specification defines a *character set* as a finite set of different characters used for the representation, organization, or control of data. Examples of character sets include the English alphabet, Kanji or sets of ideographic characters, corporate character sets (commonly used in Japan), and the characters needed to write certain European languages. A *coded character set* (or *code set*) is defined as a set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation or numeric value. Examples include ASCII, ISO 8859-1, JIS X0208 (which includes Roman characters, Japanese hiragana, Greek characters, Japanese kanji, etc.), and Unicode.

If a TAO client or server must operate in a non-US English host environment, it may be necessary to declare the particular code set used to render text data, either with `char` or `wchar` type codes. The resource factory allows you to declare a specific *code set identifier* used natively for either `char` or `wchar` data. The *native* code set is what is used when interacting with users, files, printers, etc.

Occasionally, it is necessary to interoperate with other processes using different native code sets. For each peer to render the data correctly, the character data must be transformed from one code set to another. This

transformation is the responsibility of a *code set translator*. The resource factory is responsible for managing code set translators for `char` and `wchar` data. The translators themselves are created by factory objects, which are separately-loaded service objects. It is possible to configure many translator factories, but only the translators that correspond to the configured native `char` or `wchar` code set will participate in inter-ORB communication.

Four resource factory options, shown in table Table 18-9, control the configuration of code set information.

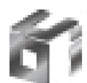
Table 18-9 Options for Configuring Code Sets

Option	Section	Description
`-ORBNativeCharCodeset` *`id`*	18.6.10	Specify the native code set identifier for `char` data. The identifier may be expressed by either a number or a locale name.
`-ORBNativeWCharCodeset` *`id`*	18.6.11	Specify the native code set identifier for `wchar` data. The identifier may be expressed by either a number or a locale name.
`-ORBCharCodesetTranslator` *`factory`*	18.6.1	Name a translator available to convert between the native `char` code set and some other code set.
`-ORBWCharCodesetTranslator` *`factory`*	18.6.17	Name a translator available to convert between the native `wchar` code set and some other code set.

18.2.9 Allocators for AMH and AMI Response Handling

When processing long duration requests, such as a database access or media I/O, a service can take advantage of *asynchronous method handling* (AMH) to decouple the receipt of a request from the processing of that request. AMH allows a request to be received by one thread, processed by another, and possibly have the reply issued by a third. The server's ORB creates an AMH response handler when the request is received. When request processing is completed, this response handler is used to generate the reply sent back to the client. For a detailed discussion of AMH, see Chapter 7.

Similarly, clients can use *asynchronous method invocation* (AMI) to avoid blocking while waiting for a reply. The client's ORB creates an AMI response handler whenever an operation is invoked asynchronously on a remote proxy.



The response handler asynchronously receives the reply from the server and delivers it to the proxy. For a detailed discussion of AMI, see Chapter 6.

The resource factory can be configured to use different types of allocators to create AMH and AMI response handlers. The resource factory interface provides the following operations that are used by the ORB to access the allocators used to create AMH and AMI response handlers:

```
virtual ACE_Allocator* amh_response_handler_allocator (void);
virtual ACE_Allocator* ami_response_handler_allocator (void);
```

In a multithreaded environment, a locked allocator may be required. In a resource constrained environment, such as an embedded system where memory usage is a concern, the creation of locks may be undesirable. In such cases, it is possible to configure the resource factory to create lock-free allocators. By default, the advanced resource factory will create locked allocators.

The advanced resource factory provides options to configure the type of allocators used for AMH and AMI response handlers. These options are shown in Table 18-10.

Table 18-10 Options for Configuring AMH and AMI Response Handler Allocators

Option	Section	Description
`-ORBAMHResponseHandlerAllocator {`**`thread`**` | null}`	18.7.1	Specify the type of locking used when accessing the AMH response handler allocator.
`-ORBAMIResponseHandlerAllocator {`**`thread`**` | null}`	18.7.2	Specify the type of locking used when accessing the AMI response handler allocator.

18.2.10 CORBA Object Synchronization

Locking is needed to synchronize access to the internal state of proxies to remote CORBA objects when those proxies are shared by multiple threads. An example of such internal state is the reference counts used to manage the memory associated with the proxies. Single-threaded clients do not need locking to protect the internal state and can therefore reduce memory usage and locking overhead by using a null locking strategy.

The resource factory interface defines the following operation, which returns the type of lock that should be used when threads access proxies to remote CORBA objects.

```
virtual ACE_Lock* create_corba_object_lock (void);
```

The possible lock types returned from this operation are a null lock and a thread-safe lock. The `null` strategy provides no locking to synchronize access to the internal state of proxies to remote CORBA objects. The `thread` strategy provides thread-safe access to this state. The default resource factory returns a thread-safe lock by default.

Table 18-11 shows the option to configure the locking strategy used by the ORB to synchronize access to the internal state of proxies to remote CORBA objects.

Table 18-11 CORBA Object Synchronization Strategy Options

Option	Section	Description
`-ORBCorbaObjectLock {`**`thread`**` | null}`	18.6.5	Selects the type of lock threads use to access the internal state of proxies to remote CORBA objects. The default behavior is to use a thread-safe locking strategy.

18.2.11 Resource Usage Strategy

The ORB is responsible for creating various resources dynamically. For example, when an object reference is received as part of a request or is returned in a reply, the IOR is encoded in a input CDR stream. The ORB must extract the IOR from the input CDR stream before the application can use it. The ORB can either create a stub immediately upon extracting the IOR, which is computationally expensive and resource intensive, or it can create a simple encapsulation of the IOR and defer stub creation until a stub is actually needed by the application (for example, to invoke a request via the object reference).

Some applications, such as those that deal with hundreds or even thousands of object references (e.g., the Naming Service), can benefit by deferring stub creation if a stub is not needed immediately. The benefits are realized in terms of run time memory footprint and performance when large numbers of object references are transmitted.

The ORB uses a *resource usage strategy* to determine whether it should create such resources immediately or defer creating them until needed by the application. The following operation of the resource factory interface allows the ORB to access the resource usage strategy:

```
  enum Resource_Usage{
   TAO_EAGER,
    TAO_LAZY
  };

virtual TAO_Resouce_Factory::Resource_Usage resource_usage_strategy (void)
  const = 0;
```

The `resource_usage_strategy()` operation returns the type of resource usage strategy to be used by the ORB. There are two defined resource usage strategies that may be returned from this operation:

- *eager* (`TAO_EAGER`)—this strategy instructs the ORB to create certain resources immediately.
- *lazy* (`TAO_LAZY`)—this strategy instructs the ORB to defer creation of certain resources.

Note *Currently, the resource usage strategy only affects the creation of stubs when object references are extracted from input CDR streams. If the* `eager` *strategy is used, a stub is created for each object reference as it is extracted. If the* `lazy` *strategy is used, a simple encapsulation of the object reference is created and a stub is only created if the application later uses the reference. The behavior of the resource usage strategy may be extended in the future to affect the creation of other types of resources, as well.*

By default, the default resource factory provides an eager resource usage strategy. This default behavior can be changed at compile time by defining the following preprocessor macro in your `$ACE_ROOT/ace/config.h` file:

```
#define TAO_USE_LAZY_RESOURCE_USAGE_STRATEGY 1
```

The type of resource usage strategy can also be changed at run time by using the `-ORBResourceUsage` resource factory option as shown in Table 18-12.

Table 18-12 Resource Usage Strategy Configuration

Option	Section	Description
`-ORBResourceUsage {eager|lazy}`	18.6.16	Determines how the ORB uses resources when creating object references. The default is `eager`.

18.2.12 Refresh IOR Table

A server using defaulted listen endpoints along with the IORTable on a host with late-enabled or transient network interfaces can enable the IOR refresh feature to ensure forwarded object references contain up to date profile and endpoint lists. This may be enabled programmatically as shown in Section 13.4, “IOR Refresh.” Alternatively use the `-ORBRefreshIORTable` resource factory option as shown in Table 18-13.

Table 18-13 Refresh IOR Table Configuration

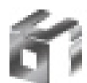
Option	Section	Description
`-ORBRefreshIORTable {0|1}`	18.6.15	Sets the initial enabled status of the IOR Table’s IOR refresh feature.

18.3 Resource Factory for Qt GUI Toolkit

If you are building a TAO application that is integrated with the Qt GUI toolkit from Trolltech `<http://www.trolltech.com>`, you must use the `ACE_QtReactor`. For example, a TAO CORBA server that also has a graphical user interface, developed with the Qt toolkit, needs to be responsive to both CORBA requests and Qt events. Such an application must use the `ACE_QtReactor`.

Note *To use the* `ACE_QtReactor`*, you must place* `#define ACE_HAS_QT` *in your* `$ACE_ROOT/ace/config.h` *file when you build ACE and TAO.*

Initializing the `ACE_QtReactor` requires supplying a valid `QApplication` pointer to the reactor constructor. A specialized resource factory is supplied with TAO, providing the means for integrating with the Qt toolkit. This factory, called the `TAO_QtResource_Factory`, is derived from the default resource factory and may be used in its place.



The TAO_QtResource_Factory accepts the same options as the default resource factory. To use the TAO_QtResource_Factory, you must add a directive to your service configuration file similar to the directive shown here:

```
dynamic Resource_Factory Service_Object* TAO:_make_TAO_QtResource_Factory()""
```

In addition to loading the new resource factory, a valid QApplication pointer must be supplied to the factory before TAO attempts to initialize the reactor. The file $TAO_ROOT/tests/QtTests/client.cpp contains the following example code:

```
#include "testC.h"
#include <ace/Get_Opt.h>
#include "client.h"

int main (int argc, char* argv[])
{
  QApplication app (argc, argv);
  TAO_QtResource_Factory::set_context (&app);
  TAO_ENV_DECLARE_NEW_ENV;
  ACE_TRY
    {
      CORBA::ORB_var orb =
        CORBA::ORB_init (argc, argv, "" TAO_ENV_ARG_PARAMETER);
      ACE_TRY_CHECK;
      //...
    }
  //...
  ACE_ENDTRY;
  return 0;
}
```

In this example, we create a QApplication object and supply its address to the TAO_QtResource_Factory via the static set_context() function. Once the factory is initialized, the application proceeds to initialize and use the ORB as usual.

18.4 Resource Factory for X Windowing Toolkit

If you are building a TAO application that is integrated with the X Window System's Xt Intrinsics toolkit, you must use the ACE_XtReactor. For example, a TAO CORBA server that also has a graphical user interface,

developed with an Xt-based toolkit, needs to be responsive to both CORBA requests and Xt events. Such an application must use the `ACE_XtReactor`.

Note *To use the* `ACE_XtReactor`*, you must place* `#define ACE_HAS_XT` *in your* `$ACE_ROOT/ace/config.h` *file when you build ACE and TAO.*

Initializing the `ACE_XtReactor` requires supplying an `XtAppContext` object to the reactor constructor. A specialized resource factory is supplied with TAO, providing the means for integrating with the Xt toolkit. This factory, called the `TAO_XT_Resource_Factory`, is derived from the default resource factory and may be used in its place.

The `TAO_XT_Resource_Factory` accepts the same options as the default resource factory. To use the `TAO_XT_Resource_Factory`, you must add a directive to your service configuration file similar to the directive shown here:

```
dynamic Resource_Factory Service_Object* TAO:_make_TAO_XT_Resource_Factory()""
```

In addition to loading the new resource factory, an `XtAppContext` must be supplied to the factory before TAO attempts to initialize the reactor. The file `$TAO_ROOT/tests/Xt_Stopwatch/client.cpp` contains the following example code:

```
#include "testC.h"
#include <ace/Get_Opt.h>
#include "Control.h"
#include "Client.h"

int main (int argc, char* argv[])
{
  XtAppContext  app;
  Widget        toplevel = XtAppInitialize (&app, "Start & Stop", NULL, 0,
                                            &argc, argv, NULL, NULL, 0 );
  TAO_XT_Resource_Factory::set_context (app);
  // ...
  ACE_DECLARE_NEW_CORBA_ENV;
  ACE_TRY
  {
   CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "", TAO_ENV_ARG_PARAMETER);
    ACE_TRY_CHECK;
    //...
  }
  //...
```

```
  ACE_ENDTRY;
  return 0;
}
```

In this example, we create an `XtAppContext` object, initialize the top level widget, and supply the context object to the `TAO_XT_Resource_Factory` via the static `set_context()` function. Once the factory is initialized, the application proceeds to initialize and use the ORB as usual.

18.5 Advanced Resource Factory

The advanced resource factory gives more control than the default resource factory over the types of resources used and how those resources are accessed. In addition to the options provided by the default resource factory, the advanced resource factory provides options that allow selecting different reactors, changing the locking method on input CDR allocators, and adjusting the connection caching strategy. The advanced resource factory was created to allow more advanced options, while keeping the footprint of the default resource factory small.

The advanced resource factory inherits from the default resource factory and accepts all of that factory's options, in addition to its own. It can be loaded dynamically using a service configurator directive of the form shown below, all of which should be on one line:

```
dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies:_make_TAO_Advanced_Resource_Factory () "-ORBoption value
-ORBoption value ..."
```

It can also be loaded statically as follows:

- Add the following preprocessor `#include` directive to the file containing `main()`:

  ```
  #include "tao/Strategies/advanced_resource.h"
  ```

  You can omit this header file if you always use dynamic libraries.

- Link the `TAO_Strategies` library into the executable, or just inherit your MPC project from the `strategies` base project.
- Specify a service configurator directive of the form:

```
static Advanced_Resource_Factory "-ORBoption value -ORBoption value ..."
```

See 16.3 for more information on using the service configurator. Once you have loaded the advanced resource factory, directives for the default resource factory (i.e., `Resource_Factory`) have no effect and will generate warnings.

One of the key features of the advanced resource factory is the ability to select a different reactor for use with the ORB. This variability enables the use of TAO in an ACE-based application that needs a specialized reactor for different types of events. For instance, the `msgWFMO_Reactor` is used with Win32-based applications that support the COM messaging model as well.

The Fl, Fox, Tk, Qt and Xt reactors are not selected using the advanced resource factory. These reactors require specialized resource factories because the correct application contexts must be supplied to the reactor prior to ORB initialization. The Qt resource factory (see 18.3) and Xt resource factory (see 18.4) are specializations of the default resource factory, so they can provide this capability.

For additional information on the different types of reactors and their uses, please refer to the appropriate ACE documentation via `<http://www.theaceorb.com/references/>`.

The additional options provided by the advanced resource factory are summarized in Table 18-14.

Table 18-14 Additional Options provided by Advanced Resource Factory

Option	Section	Description
`-ORBAMHResponseHandlerAllocator {`**`thread`**` | null}`	18.7.1	Specify the type of locking used when accessing the AMH response handler allocator.
`-ORBAMIResponseHandlerAllocator {`**`thread`**` | null}`	18.7.2	Specify the type of locking used when accessing the AMI response handler allocator.
`-ORBConnectionPurgingStrategy {`**`lru`**` | lfu | fifo | null}`	18.7.3	Select the strategy for determining the connections to purge from the cache.
`-ORBInputCDRAllocator {`**`thread`**` | null}`	18.7.4	Specify the type of locking used when accessing the input CDR allocator.

Table 18-14 Additional Options provided by Advanced Resource Factory

Option	Section	Description
`-ORBReactorThreadQueue {`**`lifo`**` | fifo}`	18.7.5	Choose the strategy for determining the order in which waiting threads are selected to run the reactor's event loop when using the thread-pool reactor type.
`-ORBReactorType {fl | msg_wfmo | select_mt | select_st | tk_reactor | `**`tp`**` | wfmo}`	18.7.6	Use this option to select an alternative type of reactor for use with the ORB.

18.6 Resource Factory Options

This section describes the individual options interpreted by the default resource factory and any of its specializations, such as the Qt, Fl, Tk, Fox, and Xt resource factories. These options are supplied to the resource factory by the service configurator through the use of a static initialization directive (see 16.3).

This section does not describe the additional options supported by the advanced resource factory. See 18.7 for those options.

18.6.1 ORBCharCodesetTranslator *factory*

Description Character data may require conversion from one code set to another in order to interoperate with applications using different native code sets. While the translator factory objects are loaded by their own service configuration directive, this option causes the resource factory to evaluate the specified factory to see if its translator is compatible with this application's native code set.

Usage There is no default code-set translator defined. Translators are declared by the name given to the factory object that loads it. TAO may be used successfully without loading any translators. In this case, the process is simply limited to communicating in its configured native `char` code set. Many translators may be configured, by repeating this option for each translator.

Impact Each `char` code set translator works by converting text data sent to a remote process from the native `char` code set to a mutually agreed upon transmission code set. The translator similarly converts text data received from a remote process into the native `char` code set from the transmission code set.

Having many translators present increases the ORB's ability to communicate with a wide variety of remote processes. The code set used in transmission may be the native code set for either side, or may be a third code set. If it is the native code set for both sides, no translation is required. Otherwise, the non-native side will have to translate the data. It is possible to use a third code set that is translated by both the sides of the connection.

See Also 18.2.8, 18.6.10

Example

```
dynamic AsciiToEbcdic Service_Object * mycodesetlib:_make_AsciiToEbcdic ()
static Resource_Factory "-ORBCharCodesetTranslator AsciiToEbcdic"
```

18.6.2 ORBConnectionCacheLock *lock_type*

Values for *lock_type*

thread (default)	Specifies inter-thread mutex to guarantee exclusive access.
null	No locking.

Description TAO uses a cached-connection strategy to improve efficiency in establishing connections. This strategy allows for the reuse of connection handlers, if possible, rather than creating new ones.

In a multithreaded environment, access to the cache must be synchronized. This option specifies the type of lock used to protect the connection cache.

Usage The default is acceptable for use in all applications. In applications that are single-threaded, or those for which all interaction with a connection is through a single thread in a multithreaded application, a `null` lock may be used.

A `null` lock must *not* be used when multiple threads are obtaining connections from the cache.

Impact Use of a `null` lock will reduce the latency involved with obtaining a connection. The `null` mutex class, supplied by ACE to minimize code changes when switching from multi- to single-threaded applications, reduces all locking operations to no-ops.

Note *When using the advanced resource factory, it is possible for this option and the* `-ORBInputCDRAllocator` *option to have conflicting values. Both options affect data-block locking. If conflicting values are given, the value provided by* `-ORBConnectionCacheLock` *will be used. If you use both of these options, they should have the same setting to prevent unexpected results.*

See Also 18.2.2, 18.7.4

Example

```
static Resource_Factory "-ORBConnectionCacheLock null"
```

18.6.3 ORBConnectionCacheMax *limit*

Description Opened connections are added to the cache to facilitate their reuse. If a process continues to run, and these connections are not reused appropriately, the cache will continue to grow. Therefore, before each new connection, the cache is checked and purged if it has reached the limit specified by this option.

The default is system dependent, but can be overridden at compile time by defining the preprocessor macro `TAO_CONNECTION_CACHE_MAXIMUM`.

Usage Determining an appropriate limit for the cache may be completely dependent upon available resources.

Impact The larger the cache limit, the more connection resources that may be used.

See Also 18.6.4

Example

```
static Resource_Factory "-ORBConnectionCacheMax 1024"
```

18.6.4 ORBConnectionCachePurgePercentage *percent*

Description Opened connections are added to the cache to facilitate their reuse. If a process continues to run, and these connections are not reused appropriately, the cache will continue to grow. Therefore, before each new connection, the cache is checked and purged if it has reached the limit specified by the `-ORBConnectionCacheMax` option or the system default, if that option was not used. This option is used to set the amount of the connection cache actually purged when it is time to do so. The default amount is twenty (20) percent. If zero is supplied as a percent, then no cache purging will occur, regardless of the strategy selected. A value less than zero will be interpreted as zero and a value 100 or greater will result in removal of all cache entries.

Usage Determining an appropriate percentage of the cache to remove at one time requires balancing the time required to actually perform the purge versus how often purging will be necessary.

Impact The act of purging connections takes time. Depending on the frequency of connection creation, a large percentage may be necessary to lower the time between purges.

See Also 18.6.3

Example

```
static Resource_Factory "-ORBConnectionCachePurgePercentage 50"
```

18.6.5 ORBCorbaObjectLock *lock_type*

Values for *lock_type*

thread (default)	The internal state (e.g., reference count) of CORBA object proxies is protected using a thread-safe locking strategy.
null	No locking is used to protect the internal state of CORBA object proxies.

Description When proxies to remote CORBA objects are shared among multiple threads, access to their internal state (such as their reference counts) must be synchronized. The `-ORBCorbaObjectLock` resource factory option allows the user to specify the locking strategy to be used when accessing and manipulating the internal state of remote CORBA object proxies.

Usage The default value of `thread` is appropriate for most applications. A thread-safe locking strategy allows multiple threads to safely use and manipulate proxies to remote CORBA objects.

Use the `null` strategy to reduce the memory and performance overhead associated with locks. The `null` strategy should only be used in single-threaded applications.

See Also 18.2.10

Example

```
static Resource_Factory "-ORBCorbaObjectLock null"
```

18.6.6 ORBDropRepliesDuringShutdown *enabled*

Values for enabled

0 (default)	When `shutdown(true)` is called on an ORB, any pending replies continue to be processed until `destroy()` is called.
1	When `shutdown(true)` is called on an ORB, any pending replies from servers are dropped and not processed.

Description When `shutdown()` is called on an ORB, this option will allow TAO to continue processing any replies that are currently pending. When this option is enabled, any pending replies are dropped and if they are subsequently received they are not processed.

Disabling this option allows for the ability to process these pending replies after a `shutdown()` call. Any pending replies will be dropped when `destroy()` is called on that ORB.

This option is only effective when `shutdown()` is called with the `wait_for_completion` parameter set to `true`. When `wait_for_completion` is `false`, the ORB terminates right away regardless of the drop replies status.

Usage Applications that want to ignore replies after a shutdown should enable this option.

Impact Some ill-behaved servers may never send their replies and applications waiting for them may become hung.

Example

```
static Resource_Factory "-ORBDropRepliesDuringShutdown 1"
```

18.6.7 ORBFlushingStrategy *strategy*

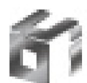
Values for *strategy*

`leader_follower` (default)	Use the reactor and non-blocking I/O to send the outgoing messages. This strategy participates in the leader-follower pattern to synchronize access to the reactor.
`reactive`	Use the reactor, but do not take part in the leader-follower pattern. This strategy is better used only in single-threaded applications.
`blocking`	Use the blocking strategy, which flushes the queue as soon as it becomes “full,” and blocks the thread until all the data is sent.

Description When one ORB must communicate with another, it looks at the request, creates a message that will be understood by the other ORB, then sends the message. The parts of the message are stored as `ACE_Message_Block`s. The `ACE_Message_Block`s build up in a queue before they are flushed and the data in them is sent. You can adjust the strategy by which the messages are flushed using this option. For example, you can use this option to alter the timing of message sending in your application.

Usage The default value is appropriate for most applications. A slight performance gain can be achieved in a single-threaded application when the `reactive` strategy is used because it does not perform any locking to synchronize access to the reactor. Use the `blocking` strategy to avoid allowing the thread that is waiting to flush messages to be made available to the ORB for potentially handling incoming requests.

Impact The only optimization benefit is gained in a single-threaded application. In this configuration, the `reactive` strategy will remove synchronization that



controls access to the reactor when sending messages. The `blocking` strategy ensures that the thread that is waiting to write to the transport will block until it can write and cannot be used by the ORB to dispatch incoming requests. The `blocking` strategy can lead to poor responsiveness and potential deadlocks if all the threads used by the ORB are blocked waiting to write and none is available to respond to incoming requests.

See Also 15.4.3

Example

```
static Resource_Factory "-ORBFlushingStrategy reactive"
```

18.6.8 ORBIORParser *parser_name*

Description TAO supports several IOR formats, including `corbaloc:`, `corbaname:`, `IOR:`, and `file:`. However, some applications may benefit from other formats. For example, `http:` could allow applications to download an object reference from a web server. This option allows application developers to implement their own IOR parsers and dynamically add them to the ORB.

Using an IOR parser is more convenient than adding configuration code in the application's `main()` function. It also allows for easier integration with other TAO components, such as the `-ORBInitRef` options.

Usage First create an IOR parser class that inherits from `TAO_IOR_Parser`. The *`parser_name`* should be the ACE Service Configurator service name you defined using the `ACE_STATIC_SVC_DEFINE` macro in your IOR Parser class. Your custom parser should not attempt to match one of the predefined IOR prefixes. Redefining or matching one of the predefined IOR prefixes may cause unexpected behavior.

You can repeat this option to add multiple parsers.

Impact Additional IOR parsers can impact performance when calling the `string_to_object()` operation. Each IOR parser added is queried to see if it is a match. If it is a match, the parser is used to return the object reference. The additional match checking and any overhead within the custom parser's code to look up the object reference can impact performance.

See Also 18.2.7

Example

```
static Resource_Factory "-ORBIORParser HTTP_Parser"
```

18.6.9 ORBMuxedConnectionMax *limit*

Description Opened connections are added to the cache to facilitate their reuse. Multiple connections may be established for a given remote endpoint and stored in the cache. This option allows an application to limit the number of connections that can be established for a given remote endpoint.

Usage Determining an appropriate limit for the number of connections that can be established for a given remote endpoint may be completely dependent upon available resource and the number of client threads that are likely to be concurrently invoking requests on objects in a given remote server.

Impact The larger the limit, the more per-endpoint connections that may be established.

See Also 18.2.5, 20.3.6

Example

```
static Resource_Factory "-ORBMuxedConnectionMax 1"
```

18.6.10 ORBNativeCharCodeset *id*

Values for *id*

0c00010001 (default)	The Code Set Registry id for ISO 8859-1, the CORBA defined default for 8-bit character codes.
`numeric code or locale name`	When not using the default code set, use some other Code Set Registry Id value or locale name.

Description The native code set for characters is described by an entry in the Open Software Foundation's (OSF) Code and Character Set Registry, currently version 1.2g. This value is embedded in object-reference profiles as a declaration of the code set used to render text data written to files, consoles, printers, etc. By default, TAO uses ISO 8859-1 as its character code set.

Usage Use this option to configure a different code set for character data. This may be useful if you are deploying an application internationally, where different installations may render text differently.

In order to use locale names in place of numeric values, you must separately configure the code-set registry database in ACE to assign locale names to the individual entries. The OSF does not define locale names for entries. The code set registry database shipped with TAO defines entries for "ASCII," and "EBCDIC" as locale names for two common 8-bit code sets.

It is also possible to compile with an alternative code set id for `char` data by changing the value of `TAO_DEFAULT_CHAR_CODESET_ID` in the source file `$TAO_ROOT/tao/corbafwd.h`. Of course, this change requires the TAO library be rebuilt, and affects every application that links to it. A default code set id changed this way may still be overridden by the resource factory.

Impact Altering the native code set impacts information used to define the capabilities of an ORB process. When then native code set of a client differs from that of a server, the client must chose an alternate code set based on any translators in either the client or server. If no suitable translator is available, then a client will not be able to exchange text data with a server.

See Also 18.2.8, 18.6.1

Example
```
static Resource_Factory "-ORBNativeCharCodeset 0x10020417"
```

18.6.11 ORBNativeWCharCodeset *id*

Values for *id*

0c00010109 (default)	The Code Set Registry id for UTF-16, also known as Unicode.
`numeric code or locale name`	When not using the default code set, use some other Code Set Registry Id value or locale name

Description The native code set for wide characters is described by an entry in the Open Software Foundation’s (OSF) Code and Character Set Registry, currently version 1.2g. This value is embedded in object reference profiles as a declaration of the code set used to render text data written to files, consoles, printers, etc. By default, TAO uses UTF-16 as its wide character code set. This default is a 2-octet code set commonly used by MS Windows applications. On platforms such as Solaris, UCS-4, a 4-octet code set, is a better choice.

Usage Use this option to configure a different code set for wide character data. This may be useful if you are deploying an application internationally, where different installations may render text differently.

In order to use locale names in place of numeric values, you must separately configure the code set registry database in ACE to assign locale names to the individual entries. The OSF does not define locale names for entries. The code set registry database shipped with TAO defines entries for “UCS-4,” and “Unicode” as locale names for two common wide character code sets.

It is also possible to compile with an alternative code set id for `wchar` data by changing the value of `TAO_DEFAULT_WCHAR_CODESET_ID` in the source file `$TAO_ROOT/tao/corbafwd.h`. Of course, this change requires the TAO library be rebuilt, and affects every application that links to it. A default code set id changed this way may still be overridden by the resource factory.

Impact Altering the native code set impacts information used to define the capabilities of an ORB process. When then native code set of a client differs from that of a server, the client must chose an alternate code set based on any translators in either the client or server. If no suitable translator is available, then a client will not be able to exchange text data with a server.

See Also 18.2.8, 18.6.17

Example

```
static Resource_Factory "-ORBNativeWCharCodeset 0x00010104"
```

18.6.12 ORBOutputCDRAllocator *allocator*

Values for *allocator*

`default (default)`	Specifies the use of the platform’s default allocator for output CDR buffers.
`mmap`	Specifies the use of a memory-mapped file allocator for output CDR buffers.
local_memory_pool	Specifies use of a local memory pool allocator for output CDR buffers.

Description This option specifies the type of memory allocator to use for output CDR buffers. The three choices are the default platform allocator, the memory mapped file allocator, and the local memory pool allocator.

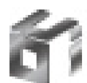
Setting the memory-mapped or local memory allocators for this option overrides the `-ORBUseLocalMemoryPool` option.

Usage The memory mapped file allocator must be used in order for the sendfile optimizations to be used. See 18.6.18 for more details. The most efficient allocator for other applications is dependent on the platform and application details.

See Also 17.13.68, 18.6.18

Example

```
static Resource_Factory "-ORBOutputCDRAllocator mmap"
```

18.6.13 ORBProtocolFactory *factory_name*

Values for *factory_name*

`IIOP_Factory` (default)	The factory providing protocol elements to support Internet Inter-ORB Protocol.
`DIOP_Factory`	Support for GIOP over UDP/IP. This is not interoperable with other ORBs.
`SHMIOP_Factory`	Support for Shared Memory Inter-ORB Protocol. This is not interoperable with other ORBs.
`SSLIOP_Factory`	Support for Secure Socket Layer Inter-ORB Protocol.
`UIOP_Factory`	Support for Unix domain (local IPC) Inter-ORB Protocol. This is not interoperable with other ORBs.
`custom factory`	Application developers may produce specialized protocol factories that satisfy the requirements of the interface and provide behavior not already supplied with TAO.

Description This option is used to “plug” pluggable protocols into TAO. Pluggable protocols offer application designers the ability to deploy TAO-based objects in an environment other than one using TCP/IP. The protocols specified here are the transport layer, not the messaging layer. The inter-ORB messaging protocol is GIOP. Pluggable protocols are discussed in depth in Chapter 14.

A GIOP message consists of a message header and, for request messages, a sub-header containing additional information. The GIOP version 1.2 message header contains the following fields: GIOP magic number; GIOP version; GIOP flags; message type; and message length. For request messages there is an additional request header that contains the following fields: service context; request identifier; response flags; target address; and operation. Applications operating in homogeneous (e.g., embedded, real-time) environments do not require all of these fields.

See 14.10 and 27.10 for more information on using the `SSLIOP_Factory` and using SSL with TAO.

Usage Multiple protocol factories may be added to the resource factory. Every time the option is repeated, the specified protocol factory is added to a list of available protocol factories.

The default, `IIOP_Factory`, must be specified if that protocol is desired, along with any other protocol. If UIOP is available but IIOP is desired exclusively, then `IIOP_Factory` must be specified as the only protocol factory.

Additional protocols impact performance at the initialization of servers. For each protocol initialized, a profile is added to an IOR, increasing its size.

See Also 17.13.43

Example

```
dynamic DIOP_Factory Service_Object * TAO:_make_TAO_DIOP_Protocol_Factory() ""

dynamic My_Protocol_Factory Service_Object * mylib:_make_my_protocol_factory()
""

static Resource_Factory "-ORBProtocolFactory DIOP_Factory \
                         -ORBProtocolFactory UIOP_Factory \
                         -ORBProtocolFactory My_Protocol_Factory"
```

18.6.14 ORBReactorMaskSignals *state*

Values for *state*

`0`	Do not mask signals.
`1` (default)	Mask signals.

Description When using a select-based reactor, this option provides an additional level of control. The `ACE_Select_Reactor`, in any of its threaded variations, ordinarily masks signals during its operation. This provides a level of signal-safety during these operations. Signal masking involves calls to the kernel to mask the signals, then later to unmask them. The constructor for `ACE_Select_Reactor` accepts a value to control this behavior. The default for this value is 1, meaning “yes, mask signals,” but may also be 0 meaning “no, do not mask signals.” The option `-ORBReactorMaskSignals` provides the TAO developer the means for controlling this behavior.

Usage If your application registers signal handlers with the reactor, the default value for this option is appropriate. Applications that do not use signal handlers and wish to have the greatest performance possible may see a slight performance gain by not masking signals.

Impact This option controls the behavior of the `ACE_Select_Reactor`, and only on non-win32 platforms. Turning off signal masking in applications that also register signal handlers with the reactor may cause unexpected behavior.

See Also 18.7.6

Example

```
static Resource_Factory "-ORBReactorMaskSignals 0"
```

18.6.15 ORBRefreshIORTable *enable*

Values for *enable*

`0` (default)	The IORTable's refresh feature is not enabled initially.
`1`	The IORTable's refresh feature is enabled initially.

Description This option allows you to enable the refreshing of endpoint lists in IORTable supplied object references without application involvement. Applications are still able to use the IORTable interface to override this setting.

Usage The refresh feature of the IORTable is only useful on hosts that have transient network interfaces. Use this service configurator option to select this behavior at deployment time.

See Also 18.2.12, 13.4

Example
```
static Resource_Factory "-ORBRefreshIORTable 1"
```

18.6.16 ORBResourceUsage *usage_type*

Values for *usage_type*

`eager` (default)	The `eager` strategy instructs the ORB to create certain types of resources (e.g., stubs) immediately.
`lazy`	The `lazy` strategy instructs the ORB to defer creation of certain types of resources until actually needed by the application.

Description This option allows you to control the type of resource usage strategy used by the ORB. The ORB uses the resource usage strategy to determine whether it should create certain types of resources immediately or defer creating them until needed by the application.

By default, an `eager` resource usage strategy is used. This strategy assumes the application will immediately want to use resources, such as object references. The `lazy` resource usage strategy allows the ORB to defer creation of such resources until the application attempts to use them, if at all.

Usage Currently, this option only affects the creation of stubs when object references are extracted from input CDR streams. With the `eager` resource usage strategy, a stub is immediately created as each object reference is extracted. With the `lazy` resource usage strategy, a stub is not created immediately. Instead, a simple encapsulation of the object reference is created upon

extraction and actual stub creation is deferred until the application actually attempts to use the object reference (e.g., to invoke a request).

Some applications, such as the Naming Service, deal with hundreds or even thousands of object references, but may never use them to invoke requests. Such applications can benefit from the lazy resource usage strategy to avoid creating unnecessary stubs. The benefits are realized in terms of reduced memory footprint and better performance when object references are transmitted.

The default resource usage strategy can be changed by adding the following preprocessor macro definition to your `$ACE_ROOT/ace/config.h` file and recompiling TAO:

```
#define TAO_USE_LAZY_RESOURCE_USAGE_STRATEGY 1
```

See Also 18.2.11

Example `static Resource_Factory "-ORBResourceUsage lazy"`

18.6.17 ORBWCharCodesetTranslator *factory*

Description Wide character data may require conversion from one code set to another in order to interoperate with applications using different native code sets. While the translator factory objects are loaded by their own service configuration directive, this option causes the resource factory to evaluate the specified factory to see if its translator is compatible with this application's native code set.

Usage There is no default code set translator defined. Translators are declared by the name given to the factory object that loads it. TAO may be used successfully without loading any translators. In this case the process is simply limited to communicating in its configured native `char` code set. Many translators may be configured, by repeating this option for each translator.

Impact Each `wchar` code set translator works by converting text data sent to a remote process from the native `wchar` code set to a mutually-agreed-upon transmission code set. The translator similarly converts text data received from a remote process into the native `wchar` code set from the transmission code set.

The number of octets transmitted per character depends on the maximum character size as defined by the code set specification, not the width of a

wchar in the native environment. For instance, if a host platform defines wide characters as being 4 octets, but the selected code set has a maximum width of 2 octets, then only 2 octets will be transmitted.

Having many translators present increases the ORB's ability to communicate with a wide variety of remote processes. The code set used in transmission may be the native code set for either side, or may be a third code set. If it is the native code set for both sides, no translation is required. Otherwise, the non-native side will have to translate the data. It is possible to use a third code set that is translated by both sides of the connection.

See Also 18.2.8, 18.6.11

Example The following example shows how to load a hypothetical wchar code set translator for converting between the UCS-4 (32-bit) code set and the UTF-16 (16-bit) code set. The name given to our translator is UCS4ToUTF16, and it is loaded from the MyTranslators library.

```
dynamic UCS4ToUTF16 Service_Object* MyTranslators:_make_UCS4ToUTF16()
static Resource_Factory "-ORBWCharCodesetTranslator UCS4ToUTF16"
```

18.6.18 ORBZeroCopyWrite

Description Specifies that the ORB should use a zero-copy write method (currently this means sendfile) to send messages. Because this requires the use of a memory mapped allocator, this option implicitly sets the output CDR allocator to memory mapped (see the -ORBOutputCDRAllocator description, 18.6.12).

Usage If used on a platform or build that does not support sendfile, this option will give a warning.

Impact Use of this option should increase the performance of TAO applications that are sending many large messages.

See Also 18.6.12

Example This example shows how this option can be enabled:

```
static Resource_Factory "-ORBZeroCopyWrite"
```

18.7 Advanced Resource Factory Options

This section describes the individual options interpreted by the advanced resource factory. The advanced resource factory supports these options in addition to those supported by the default resource factory described in 18.6. See 18.5 for more information on using the advanced resource factory.

18.7.1 ORBAMHResponseHandlerAllocator *type*

Values for *type*

thread (default)	Uses a thread-safe allocator.
null	Uses non thread-safe allocator.

Description Specify the type of locking used when accessing the AMH response handler allocator.

Usage The default value of `thread` is appropriate for most applications. A thread-safe AMH response handler allocator allows multiple threads to safely access the allocator.

Use the `null` strategy to reduce the overhead associated with locking for single-threaded applications.

This option only affects servers that use AMH.

See Also .18.2.9, Chapter 7

Example

```
dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies:_make_TAO_Advanced_Resource_Factory ()
"-ORBAMHResponseHandlerAllocator null"
```

18.7.2 ORBAMIResponseHandlerAllocator *type*

Values for *which*

thread (default)	Uses a thread-safe allocator.
null	Uses non thread-safe allocator.

Description Specify the type of locking used when accessing the AMI response handler allocator.

Usage The default value of `thread` is appropriate for most applications. A thread-safe AMI response handler allocator allows multiple threads to safely access the allocator.

Use the `null` strategy to reduce the overhead associated with locking for single-threaded applications.

This option only affects clients that use AMI.

See Also .18.2.9, Chapter 6

Example

```
dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies:_make_TAO_Advanced_Resource_Factory ()
"-ORBAMIResponseHandlerAllocator thread"
```

18.7.3 ORBConnectionPurgingStrategy *strategy*

Values for *strategy*

`lru` (default)	Purge least recently used connections.
`lfu`	Purge least frequently used connections.
`fifo`	Purge oldest connections (first in, first out).
`null`	Do not purge connections.

Description Opened connections are added to the connection cache so they can be reused. However, if a process continues to run and these connections are not reused, the cache will continue to grow. Therefore, before each new connection, the cache is checked and purged if it has reached the limit specified by the `-ORBConnectionCacheMax` option or the system default if that option was not used. This option is used to select a cache management strategy that ensures connections are available as needed.

Usage The correct connection cache management strategy depends upon the behavior of the server's clients. Those that use connections only once will benefit from the FIFO strategy. When pools of connections are used for a while, then not used, the LRU strategy will provide a better means of removing unused connections. The LFU strategy will work best if the connections are used regularly, some more than others. Finally, if the number of connections used by a process can be determined *a priori* and does not exceed the maximum number of simultaneous connections allowed on your

system, the `null` strategy is appropriate to avoid the instantiation of cache management objects.

Impact The use of connection cache management may close connections in use, depending on the algorithm chosen. If the FIFO strategy is selected, for instance, and some old connections are not used for a long time, they may get closed. Subsequent attempts to use these connections will suffer the penalty of reconnection costs.

Purging the cache occurs infrequently, but will occur when a new connection is required and none is available.

See Also 18.6.3, 18.6.4

Example

```
dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies:_make_TAO_Advanced_Resource_Factory ()
"-ORBConnectionPurgingStrategy fifo
-ORBConnectionCachePurgePercentage 50"
```

18.7.4 ORBInputCDRAllocator *lock_type*

Values for *lock_type*

`thread` (default)	Access to the allocator for creating and destroying data buffers is synchronized and therefore safe to be used by many threads.
`null`	The allocators are not thread safe and may only be used within a single-thread context.

Description The input CDR allocator is used internally by the ORB to facilitate the decoding and unmarshaling of data received from another ORB. To avoid copying data that is received as arbitrarily large octet sequences, the receiving buffer is passed to the application code that processes it. This transfer of data may or may not involve passing ownership of the data between threads.

Usage If the data buffer containing the octet sequence is to be shared between threads, access to it and to the allocator responsible for it must be synchronized by using the `thread` strategy.

If the allocation and the processing of data in the buffer will be achieved within the same thread, greater efficiency and predictability are possible by avoiding the use of mutex locks. This is done by specifying the `null` strategy.

Note *When using the advanced resource factory, it is possible for this option and the* `-ORBConnectionCacheLock` *option to have conflicting values. Both*

options affect data block locking. If conflicting values are given, the value provided by `-ORBConnectionCacheLock` *will be used. If you use both of these options, they should have the same setting to prevent unexpected results.*

See Also 18.2.2, 18.6.2

Example

```
dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies:_make_TAO_Advanced_Resource_Factory () "-ORBInputCDRAllocator
null"
```

18.7.5 ORBReactorThreadQueue *type*

Values for *type*

`lifo` (default)	Threads that are waiting in the thread-pool reactor will be selected to run in a last-in-first-out order (LIFO) order.
`fifo`	Threads that are waiting in the thread-pool reactor will be selected to run in a first-in-first-out order (FIFO) order.

Description This option allows an application using the thread-pool reactor to specify the order in which threads should be selected to run by the `ACE_Select_Reactor_Token`. By default, threads are selected in FIFO order by the `ACE_Select_Reactor_Token`.

The thread-pool reactor implements the *Leader/Followers* architectural pattern described in *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* (POSA2). Briefly, if two or more threads are waiting to run the reactor's event loop, one thread becomes the *leader* and gets into the event loop; the remaining threads are *followers*. When the reactor dispatches an event onto the leader thread, a new leader is selected from among the group of followers, then the leader continues to process the event.

This option controls the order in which a follower thread is selected to become the new leader.

Usage This option is only usable when the reactor type is the thread-pool (tp) reactor. Either setting should be suitable for any application.

Impact By using the LIFO reactor thread queue type, multithreaded applications may experience a performance gain by maximizing CPU "cache affinity" by ensuring that the thread waiting the shortest amount of time is selected as the new leader first. However, the LIFO strategy requires an additional data structure in the reactor to hold a list of waiting threads, rather than just using a

native operating system synchronization object, such as a semaphore or condition variable.

Results of using one strategy over the other will vary widely across different platforms depending upon the number of CPUs, the amount of CPU cache, and the efficiency of the operating system's cache management strategy. In addition, if the amount of time each thread spends in an upcall exceeds the time between events, threads will not have to wait for events (i.e., they will usually all be busy processing events), so the selection will have little effect.

See Also 18.7.6

Example

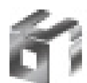
```
dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies:_make_TAO_Advanced_Resource_Factory () "-ORBReactorThreadQueue
fifo"
```

18.7.6 ORBReactorType *reactor_type*

Values for *reactor_type*

`fl`	The FL ("fast, light") toolkit reactor. This reactor is available only when ACE is built with support for the FL toolkit.
`msg_wfmo`	The *Message Wait For Multiple Objects* reactor, available on Win32 systems only.
`select_mt`	Multithreaded select reactor. This reactor uses the `select()` system call to monitor sockets for input and output availability, and employs locks to enforce synchronization between threads.
`select_st`	The single-threaded select reactor. This reactor is also based on the `select()` system call, but assumes it is running in a single thread and therefore does not perform any locking.
`tk_reactor`	The Tk toolkit reactor. This reactor is available only when ACE is built with support for the Tk toolkit.
`tp` (default)	The thread-pool reactor. This reactor allows for many threads to be used to handle events. If a thread is available to handle an event, the event is dispatched right away. Otherwise it is queued until a thread becomes ready.
`wfmo`	The *Wait For Multiple Objects* reactor, available on Win32 only.

Description ACE supplies a variety of reactors that may be used to support specialized types of event-driven applications. The reactor is the component of ACE that separates the detection of events from the handling of those events. TAO uses a reactor to accept new connections in a socket context, as well as responding



to incoming and outgoing data. The default reactor is perfectly suitable to handle the needs of the ORB in any context.

Usage Only one reactor type may be specified. If this option is repeated, the last value specified will be used.

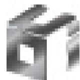
Use this option for applications that require a specialized reactor. For example, a Win32 GUI-based application must use the reactor built on the `WaitForMultipleObjects()` system call to properly integrate with the environment.

Impact The choice of reactor has very little impact on the application. The only optimization benefit is gained in a single-threaded select-based application. In this configuration, the single-threaded select reactor will remove a single lock from the event-handling loop. Otherwise, the choice of reactor is entirely dependent on the configuration of the application.

See Also 15.3.11, 18.6.7

Example

```
dynamic Advanced_Resource_Factory Service_Object*
TAO_Strategies:_make_TAO_Advanced_Resource_Factory () "-ORBReactorType msg_wfmo"
```



CHAPTER 19

Server Strategy Factory

19.1 Introduction

Certain elements of the ORB relate only to server-side behavior. In this context, the server is any application that passively accepts connections from other processes and receives requests from those other connections. The server strategy factory is responsible for supporting features of TAO that are specific to servers, including demultiplexing and concurrency-related strategies. The demultiplexing strategies are used to locate POAs and active objects responsible for handling requests. The concurrency strategies control the thread-creation flags and other concurrency-related behaviors.

The server strategy factory is registered with the service configurator using the name `Server_Strategy_Factory`. TAO's default server strategy factory is statically registered with the service configurator and is initialized using the `static` directive. To supply options to the default server strategy factory, add a line, similar to the line shown below, to your service configuration file:

```
static Server_Strategy_Factory "-ORBoption value -ORBoption value ..."
```

The service configurator is discussed in greater detail in Chapter 16.

19.2 Interface Definition

Within a TAO application, the server strategy factory is accessed via an interface defined by the base class `TAO_Server_Strategy_Factory`. Here we show all the public and protected functions of the server strategy factory and describe their use in relationship to the ORB, and the behavior of the default server strategy factory implementation.

The implementation behind this interface is intended to supply parameters to a number of TAO internal objects, such as the ORB core, POA, POA manager, and the various acceptor implementations supporting different pluggable protocols.

Synopsis

```
class TAO_Server_Strategy_Factory : public ACE_Service_Object
{
public:
   struct Active_Object_Map_Creation_Parameters
   {
      Active_Object_Map_Creation_Parameters (void);
      CORBA::ULong active_object_map_size_;
      TAO_Demux_Strategy object_lookup_strategy_for_user_id_policy_;
      TAO_Demux_Strategy object_lookup_strategy_for_system_id_policy_;
      TAO_Demux_Strategy reverse_object_lookup_strategy_for_unique_id_policy_;
      int use_active_hint_in_ids_;
      int allow_reactivation_of_system_ids_;

      CORBA::ULong poa_map_size_;
      TAO_Demux_Strategy poa_lookup_strategy_for_transient_id_policy_;
      TAO_Demux_Strategy poa_lookup_strategy_for_persistent_id_policy_;
      int use_active_hint_in_poa_names_;
   };
   TAO_Server_Strategy_Factory (void)
   virtual ~TAO_Server_Strategy_Factory (void)
   virtual int open (TAO_ORB_Core* orb_core);
   virtual int enable_poa_locking (void);
   virtual int activate_server_connections (void);
   virtual int thread_per_connection_timeout (ACE_Time_Value &timeout);
   virtual int server_connection_thread_flags (void);
   virtual int server_connection_thread_count (void);
   virtual const Active_Object_Map_Creation_Parameters&
           active_object_map_creation_parameters (void) const
protected:
   Active_Object_Map_Creation_Parameters active_object_map_creation_parameters_
};
```

19.2.1 Factory Initialization

The constructor and destructor are defaults and otherwise not interesting. Since the class `TAO_Server_Strategy_Factory` is derived from `ACE_Service_Object`, the initialization of the factory is expected to be performed by an implementation of the virtual function `ACE_Service_Object::init(int argc, ASYS_CHAR *argv[])`. The default server strategy factory implements this function by parsing options provided to it by the service configurator. See below for a complete listing of the options parsed by the default server strategy factory.

To complete the initialization of the server strategy factory, the `open()` operation is provided:

```
virtual int open (TAO_ORB_Core* orb_core);
```

The ORB core calls `TAO_Server_Strategy_Factory::open()` during its initialization, after it has processed its configuration options. This allows all of the service objects that may be dynamically loaded to be initialized before there is any chance that the server strategy factory may need them. The default server factory does nothing in `open()`.

19.2.2 Server Concurrency

An important configuration issue is controlling concurrency-related behavior. The use of threads within an application can increase the capacity of the application, particularly on hosts with multiple processors, but at a cost of increased complexity within the application. The most difficult issue to deal with in a multithreaded application is the synchronization of access to different parts of the application and to data. TAO has been designed with the goal of creating servers that may support hundreds or thousands of clients on very large hosts, as well as creating extremely small, special purpose servers that may have few clients, and must be extremely efficient. To support these different goals, the server strategy factory provides functions to control the concurrency behavior of the server, as well as thread-creation parameters.

```
virtual int activate_server_connections (void);
```

This function is used by other parts of the ORB to determine how to deal with multithreaded servers. If the concurrency strategy is to use a thread per connection, then the function `activate_server_connections()` must

return non-zero. The default server factory returns a non-zero value for `activate_server_connections()` if `-ORBConcurrency` is set to *thread-per-connection*, or zero if *reactive* is specified.

Table 19-1 Server Concurrency Options

Option	Section	Description
`-ORBConcurrency {`**`reactive`**`|thread-per-connection}`	19.3.5	The default value, `reactive`, specifies that requests will be handled using an event handler model. The `thread-per-connection` value specifies that a thread will be dedicated to handling requests for each connection established by clients.

When the server is configured to use a thread per connection, the service handlers associated with each new connection are made active. That is, a new thread is spawned, which runs code for the handler. This is possible because the connection handlers used in TAO are based on the ACE Active Object pattern, and use the service handler's `activate()` function to start running in a separate thread. The server strategy factory is responsible for supplying the appropriate parameters to be used in the call to `activate()` through the following functions:

```
virtual int server_connection_thread_flags (void);
virtual int server_connection_thread_count (void);
```

The default server strategy factory supplies both `THR_BOUND` and `THR_DETACHED` as the values of the server connection thread flags. The default server factory always returns `1` for the thread count. There is no mechanism provided for using more than one thread per connection.

Table 19-2 Thread Flags Option

Option	Section	Description
`-ORBThreadFlags {`**`THR_BOUND`**`	THR_DAEMON	`**`THR_DETACHED`**`	THR_NEW_LWP	THR_SUSPENDED}`	19.3.9	Multiple thread flags may be specified as a single value by combining the desired flags using a vertical bar ('|') between the flags. When `-ORBThreadFlags` is not specified, the default flags `THR_BOUND` and `THR_DETACHED` apply. The `THR_DAEMON` flag is not allowed on Win32 platforms.

In addition, for the thread-per-connection concurrency model, each thread that is spawned to handle requests arriving on a given connection must be periodically awaken to check for possible ORB shutdown. The server strategy factory provides the following function for this purpose:

```
virtual int thread_per_connection_timeout (ACE_Time_Value &timeout);
```

This function returns `-1` if the compile-time default (specified as the preprocessor macro `TAO_DEFAULT_THREAD_PER_CONNECTION_TIMEOUT` in `$TAO_ROOT/tao/orbconf.h`) should be used; zero if threads should block without checking for ORB shutdown; or a positive non-zero value and sets the `timeout` parameter with the value specified by the following option:

Table 19-3 Thread-per-Connection Timeout Option

Option	Section	Description
`-ORBThreadPerConnectionTimeout {infinite | milliseconds}`	19.3.10	Specifies how frequently threads should check for possible ORB shutdown in servers using the thread-per-connection concurrency model.

19.2.3 Demultiplexing Strategies

When resolving a request, the ORB must perform three lookups. The word *demultiplexing* or *demux* is used to describe the act of looking up an object based on an object identifier. The first lookup is to locate the POA identified by the request. The second is a lookup on the POA to find the servant, or active object, that will handle the request. The final lookup is performed within the active object to resolve the function signature requested. The function signature lookup is controlled by the IDL compiler via the `-H` command line option (see 4.10), and will generally use the *perfect hashing* algorithm as generated by the GNU `gperf` application.

TAO provides three configurable strategies to servers for demultiplexing POAs and servants: the Linear Search strategy, the Dynamic Hash strategy, and the Active Demux strategy.

The server strategy factory is not directly responsible for providing the demultiplexing operations; rather it supplies the configuration information to the parts of the ORB that maintain the tables used to demultiplex components of requests. These parameters are supplied by the following function:

```
virtual const Active_Object_Map_Creation_Parameters&
  active_object_map_creation_parameters (void) const
```

The value returned from this function is a structure that encapsulates all of the state information needed to construct the desired demultiplexing strategies. The default strategy factory populates the active object map creation parameters by interpreting two sets of options, the POA Map options and the Active Object Map options. The strategy used within an active object to resolve a particular function signature is determined by the IDL complier, and therefore is not included as part of the active object map creation parameters structure.

19.2.3.1 Linear Search

The linear search strategy is the simplest demultiplexing strategy. Entities are stored in an unsorted array. As elements are removed from the list, the position of the hole is retained in a free list. New elements added to the list will fill these holes, or be appended to the list. Locating elements of the list requires examining the key value of each element in the list sequentially, until the matching element is found.

The cost of adding a new element to the linear search map is small, as no calculations are performed to convert the key value from one form to another.

Another point to consider is the cost of growing the list. As described below, it is possible to specify an initial size for the active object maps and for the POA maps. If the number of elements in the map extends beyond this size, the map will grow. This growth involves allocating twice as much space as the previous map contained, duplicating each existing element, using its copy constructor, finally releasing the old space by calling each element's destructor.

When comparing relative performance of the linear search strategy versus other strategies, the lookup time is O(n/2) on average, and O(n) worst case. The linear search strategy does not scale well to large numbers of objects.

19.2.3.2 Dynamic Hash

The dynamic hash strategy is substantially more efficient than the linear search for handling larger numbers of elements. This strategy uses a hash table to achieve a nearly constant lookup time regardless of the size of the table. Occasionally two or more keys will result in the same hash table index. Using

a sufficiently large table will mitigate this, but when such “collisions” occur, the elements sharing the table index are searched linearly.

Compared to the linear search strategy, there is a greater constant cost for each access of the map. The key value must be hashed before it may be used to insert or locate an element. The hashing algorithm used for object identifiers and POA names is encoded in `ACE::hash_pjw()`.

Unlike the linear search strategy, or the active demux strategy, the map size of the dynamic hash remains fixed throughout the life of the process. If the selected map size is smaller than the actual number of elements stored, hashing collisions will occur, resulting in degraded lookup performance.

19.2.3.3 Active Demux

For the greatest performance in demultiplexing, TAO uses the active demultiplexing strategy. Active demultiplexing takes advantage of the ACE Active Map, that uses array indices and object “generations” as the search key. The effect of this is that searching the active map involves only a single array dereference. Active maps are optimized for storage, reusing slots made available by the removal of earlier entries. To guard against incorrect references, entries in an active map have a generation, which is the count of the number of times a particular slot has had a value assigned to it. For this reason, active demultiplexing is available only when an object reference is transient and guaranteed not to be reused. The size of the active map will increase over time in the same manner as with the linear search strategy.

In case of persistent objects, or objects with references that may be reused, TAO provides an extension to the standard object reference demultiplexing. An active “hint” may be added to the POA name or the object key that is used to attempt to locate an object in a secondary active map. It is not guaranteed that the hint is valid, and if it is not, then the dynamic hash table or linear search will resolve the reference. The use of active hints in object identifiers or POA names will increase the size of each map accordingly.

19.2.4 POA Map Options

All POAs within a server must register themselves with the root POA. The root POA submits these registrations to the object adapter, which adds them to the appropriate map. The TAO object adapter maintains two separate maps, one for POAs with persistent IDs and one for POAs with transient IDs. To

construct its maps, the object adapter obtains a reference to the `struct Active_Object_Map_Creation_Parameters` and refers to the following values:

```
CORBA::ULong       poa_map_size_;
TAO_Demux_Strategy poa_lookup_strategy_for_transient_id_policy_;
TAO_Demux_Strategy poa_lookup_strategy_for_persistent_id_policy_;
int                use_active_hint_in_poa_names_;
```

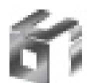
The default server strategy factory options that set these values are shown in this table.

Table 19-4 POA Map Creation Options

Option	Section	Description
`-ORBActiveHintInPOANames {`**`1`**` | 0}`	19.3.2	Allows the use of active hints for POAs when the policy otherwise does not allow active demux. This sets the value of the parameter `use_active_hint_in_poa_names_`
`-ORBPersistentIDPolicyDemuxStrategy {`**`dynamic`**` | linear}`	19.3.6	Select strategy for demultiplexing POAs defined with the Persistent ID policy. This sets the value of `poa_lookup_strategy_for_persistent_id_policy_`.
`-ORBPOAMapSize` *`size`*	19.3.7	Sets the initial number of entries in the POA lookup tables. This sets the value of the parameter `poa_map_size_`. Default is 24.
`-ORBTransientIDPolicyDemuxStrategy {`**`active`**` | dynamic | linear}`	19.3.11	Selects the strategy for demultiplexing POAs with the `TRANSIENT_ID` policy. This sets the value of the parameter `poa_lookup_strategy_for_transient_id_policy_`.

19.2.5 Active Object Map Parameters

Once a POA is resolved, it must locate an active object that will be used to handle the request. Each POA contains maps of active objects, one for objects created using the system ID policy, and another one for objects created using the user ID policy. When the unique ID policy is used, a map is created for performing reverse lookups of ID, based on the object adapter.



To construct its maps, the POA obtains a reference to the struct Active_Object_Map_Creation_Parameters and refers to the following values:

```
CORBA::ULong       active_object_map_size_;
TAO_Demux_Strategy object_lookup_strategy_for_user_id_policy_;
TAO_Demux_Strategy object_lookup_strategy_for_system_id_policy_;
TAO_Demux_Strategy reverse_object_lookup_strategy_for_unique_id_policy_;
int                use_active_hint_in_ids_;
int                allow_reactivation_of_system_ids_;
```

The default server strategy factory options that set these values are shown in Table 19-5.

Table 19-5 Active Object Map Creation Options

Option	Section	Description
-ORBActiveHintInIDs {**1** | 0}	19.3.1	Allows the use of active hints for active objects when the policy otherwise does not allow active demux. Sets the value of use_active_hint_in_ids_.
-ORBActiveObjectMapSize *size*	19.3.3	Sets the initial number of entries in the active object lookup tables. Default is 64. Sets the value of active_object_map_size_.
-ORBAllowReactivationOfSystemIDs {**1** | 0}	19.3.4	Allows the reactivation of system generated IDs. Sets the value of allow_reactivation_of_ system_ids_.
-ORBSystemIDPolicyDemuxStrategy {**active** | dynamic | linear}	19.3.8	Selects strategy for demultiplexing active objects defined with the System ID policy. Sets the value of object_lookup_strategy_for _system_id_policy_.
-ORBUniqueIDPolicyReverseDemuxStrategy {**dynamic** | linear}	19.3.12	Selects the strategy for looking up object IDs using the active object reference, when the object was defined using the Unique ID policy. This sets the value of reverse_object_lookup_stra tegy_for_unique_id_policy_.

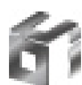


Table 19-5 Active Object Map Creation Options

Option	Section	Description
`-ORBUserIDPolicyDemuxStrategy {`**`dynamic`**` | linear}`	19.3.13	Selects the strategy for demultiplexing active objects defined with the User ID policy. This sets the value of `object_lookup_strategy_for_user_id_policy_`.

19.3 Default Server Strategy Factory Options

The remainder of this chapter describes the individual options interpreted by the default server strategy factory supplied with TAO. These options are supplied to the default server strategy factory by the service configurator, through the use of the `static` initialization directive (see 16.3).

19.3.1 ORBActiveHintInIDs *enabled*

Values for *enabled*

`1` (default)	Hints are to be used in object IDs.
`0`	Hints are not to be used in object IDs.

Description This option controls the embedding of additional information in the object reference for use as a hint in resolving object references. These hints may allow for faster resolution of object references. When the POA policy `USER_ID` is used in creating new object IDs, TAO is not able to use its active demux strategy based solely on the ID. This can have an impact on performance, as the number of active objects bound to a POA increases. By allowing TAO to concatenate some additional information to the object ID, a secondary active map may be used to look up the object reference. As long as the hint is valid, the associated object will be located, using the fast lookup of the active demux strategy. If the hint is no longer valid, perhaps from the active object having been destroyed, then reconstructed, the lookup strategy otherwise defined will be used to locate the object.

Usage Add *active hint* information to the object ID and use this information to more efficiently look up the servants in the POA.

Impact The use of active hints as an additional component of the object ID portion of the IOR will generally reduce object lookup time, in some cases substantially so. The cost of doing so is in the form of space. The hint requires an extra eight bytes in the IOR. An active map is used to facilitate the lookup, based on the hint. This is in addition to the secondary lookup method, either a dynamic hash or linear search table.

Active lookup tables are reallocated when space is required to store more entries than is available.

When used in conjunction with -`ORBAllowReactivationOfSystemIDs` with a value of 1, it provides predictable latency regardless of the number of object IDs in a POA.

See Also 19.3.3

Example
```
static Server_Strategy_Factory "-ORBActiveHintInIDs 0"
```

19.3.2 ORBActiveHintInPOANames *enabled*

Values for *enabled*

`1` (default)	Hints are to be used in POA names.
`0`	Hints are not to be used in POA names.

Description Adding active hints to the POA names embedded in the object key may allow for faster lookup of POAs by the ORB core. The use of hints in POA names is turned on by default.

Usage Turn ON this option when predictability/performance is important, and you are using multiple POAs.

Turn OFF this option to minimize memory usage.

Impact The active hints passed in the object key facilitate quick look up of POA names, and make the lookup time predictable for any level of nested POAs.

Active hints result in larger IORs and extra server-side state. They also increase the request size (typically by eight bytes) since additional hint information is added to the POA name in the object key.

When used in conjunction with active demuxing strategy for POA lookups, active hints provide predictable latency regardless of the number and nesting of POAs.

See Also 19.3.7

Example
```
static Server_Strategy_Factory "-ORBActiveHintInPOANames 0"
```

19.3.3 ORBActiveObjectMapSize *map_size*

Description This option specifies the initial number of entries in the active object map. By default, the active object map is initialized with sixty-four (64) entries if no map size is specified. The default active object map size is specified as the preprocessor macro `TAO_DEFAULT_SERVER_ACTIVE_OBJECT_MAP_SIZE` in `$TAO_ROOT/tao/orbconf.h`. This value affects all of the active object maps and hint tables. When the dynamic hash strategy is used for `SYSTEM_ID` or `USER_ID` active objects, the value specified here determines the total number of hash table buckets. When using either the linear search or active demux strategies, this value specifies the initial number of entries in the map.

Usage If any POA in a server is going to manage more than the default number of active objects, then a larger number should be specified. On the other hand, if memory conservation is important, and the maximum number of active objects on any POA is going to be less than the default, then specify a smaller number.

Impact Setting the initial map size too small degrades the predictability of the application. This degradation takes the form of map resizing when linear or active demux strategies are used, and inefficient lookups when the dynamic strategy is used.

This option controls the initial size of the maps for user-specified object IDs, system-generated object IDs, and the reverse lookup map. Setting this value larger than the number of active objects in the system wastes memory for linear and active demux strategies. When using the dynamic demux strategy, increasing the size of the map may improve performance.

See Also 19.3.1, 19.3.8

Example
```
static Server_Strategy_Factory "-ORBActiveObjectMapSize 100"
```

19.3.4 ORBAllowReactivationOfSystemIDs *enabled*

Values for *enabled*

`1` (default)	Enables the reuse of previous system-generated object IDs.
`0`	Guarantees the uniqueness of system-generated object IDs.

Description This option controls whether system-generated object IDs can be reactivated. When enabled, it allows an ID that was generated by the system to be reused after it has been deactivated. This would typically occur when clients hold references to objects whose servants are deactivated. Further use of the reference would force the server to activate a servant with the old ID.

Usage Servers that use system-generated IDs in an environment where the lifespan of a server (or servant) may be less than the lifespan of its clients should enable this option.

Override the default value to disable the reuse of system ids when it is known that no client will want to reactivate a previously-deactivated servant with a system-generated id.

Impact When disabled, the IORs can be shortened, an extra comparison in the critical upcall path removed, and some memory on the server side can be saved. This allows the use of active demultiplexing as the primary lookup strategy for resolving object references.

The reuse of object IDs presents a problem in that it limits TAO's ability to use the active demultiplexing strategy with system-generated IDs. This situation requires the use of hints to obtain the greatest performance.

See Also 19.3.1, 19.3.8

Example

```
static Server_Strategy_Factory "-ORBAllowReactivationOfSystemIDs 0"
```

19.3.5 ORBConcurrency *strategy*

Values for *strategy*

`reactive` (default)	The ORB's reactor is used to reactively process incoming requests from all connections.
`thread-per-connection`	A new thread is spawned to service each connection.

Description This option specifies the concurrency strategy used by the ORB to control server behavior. When multiple clients attempt to simultaneously connect to a

server, the response time for each client may suffer. The problem is made even worse when each request takes a long time to execute. Response time may be improved by using multiple threads in the server. However, when dealing with legacy code that is not thread safe, multithreading may not be possible.

Usage The default value, `reactive`, is appropriate when requests take a fixed, relatively uniform amount of time and are largely I/O bound. The reactor type used by the ORB is controlled by the `-ORBReactorType` option. Each thread that calls `ORB::run()` or `ORB::perform_work()` can be used by the reactor to process requests. The thread pool reactor must be specified to effectively use multiple threads in the `reactive` mode.

The use of `thread-per-connection` is appropriate when there are multiple connections active simultaneously and the clients are set up to use only one request per connection. It should not be used in situations where the time required to service a request is small compared to the time required to create a thread. The use of `thread-per-connection` requires that data shared by servants be locked and common functions be thread safe.

Impact When using `thread-per-connection`, a new thread is created for each connection made. This allows for more prompt servicing of new connections generally, but adds the cost of thread creation for every new connection.

See Also 15.3.6, 19.3.10, 18.7.6

Example

```
static Server_Strategy_Factory "-ORBConcurrency thread-per-connection"
```

19.3.6 ORBPersistentIDPolicyDemuxStrategy *strategy*

Values for *strategy*

`dynamic` (default)	Use a dynamic hashing strategy.
`linear`	Use a linear search strategy.

Description This option selects the algorithm the ORB core uses for locating a persistent POA, based on an object request. The active demultiplexing strategy is not available for POAs identified with the `PERSISTENT` policy. Persistent POAs are intended to outlive the server process, and active demultiplex index values are valid only during the life of the process that generates them. An active *hint* may be used along with the object id to benefit from the performance boost of active demultiplexing.

Usage If the number of persistent POAs is small, and a design goal is to minimize the memory footprint of a server, the linear search strategy will provide reasonable performance with the smallest memory footprint.

Impact The dynamic hashing strategy incurs the additional lookup cost of hashing the POA name.

The linear search strategy quickly degrades lookup performance as the number of persistent POAs increases and results in additional performance degradation as the addition of new persistent POAs forces the map to resize.

See Also 19.3.2, 19.3.7, 19.3.11

Example

```
static Server_Strategy_Factory "-ORBPersistentIDPolicyDemuxStrategy linear"
```

19.3.7 ORBPOAMapSize *map_size*

Description This option specifies the initial number of entries in the POA map. The default value is twenty-four (24), as specified by the preprocessor macro `TAO_DEFAULT_SERVER_POA_MAP_SIZE` in `$TAO_ROOT/tao/orbconf.h`. This value affects all of the POA maps and hint tables. When the dynamic hash strategy is used for `PERSISTENT` or `TRANSIENT` POAs, the value specified here determines the total number of hash table buckets. When using either the linear search or active demux strategies, this value specifies the initial number of entries in the map.

Usage If any ORB in a server is going to have more than the default number of POAs, then a larger number should be specified. On the other hand, if memory conservation is important, and the maximum number of POAs is going to be less than the default number, it is possible to use a smaller number.

Impact Setting the initial map size too small degrades the predictability of the application. This degradation takes the form of map resizing when linear or active demux strategies are used, and inefficient lookups when the dynamic strategy is used.

This option controls the initial size of the maps for both transient and persistent POAs. Setting this value larger than the number of POAs in the system wastes memory for linear and active demux strategies. When using the dynamic demux strategy, increasing the size of the map may improve performance.

See Also 19.3.2, 19.3.11

Example `static Server_Strategy_Factory "-ORBPOAMapSize 10"`

19.3.8 ORBSystemIDPolicyDemuxStrategy *strategy*

Values for *strategy*

`active` (default)	Use the active demultiplexing strategy.
`dynamic`	Use the dynamic hashing strategy.
`linear`	Use the linear search strategy.

Description This option defines the demultiplexing strategy used by the POA for managing active objects with system-generated object identifiers. The active demultiplexing strategy provides the smallest lookup time per request, but at the cost of an eight-byte identifier. Dynamic hashing provides performance that is nearly as good as the active demux, but using a four-byte identifier. The linear search strategy provides reasonable performance for small maps, using a four-byte identifier, but does not scale very well as the number of active objects increases.

Usage Selecting the appropriate strategy involves determining the number objects a POA must service and the cost associated with an eight-byte identifier versus a four-byte identifier.

Impact The cost associated with the active demultiplexing strategy is a larger object key and the occasional resizing of the map if the initial size is too small. The cost of the dynamic hashing strategy is degraded performance as the number of objects increases. The cost of the linear search is quickly-degraded performance as the number of objects increases and the possible resizing of the map as additional active objects are added.

See Also 19.3.3

Example `static Server_Strategy_Factory "-ORBSystemIDPolicyDemuxStrategy dynamic"`

19.3.9 ORBThreadFlags *flags*

Values for *flags*

THR_BOUND (default)	Each new thread is bound to a LWP, allowing for each to be scheduled separately.
THR_DAEMON	*Not available for Win32 based platforms.* The wait() function cannot be invoked on daemon threads.
THR_DETACHED (default)	Threads do not return status when they terminate.
THR_NEW_LWP	Creates a new light weight process for thread processing.
THR_SUSPENDED	Creates new threads in the “suspended” state.

Description This option specifies the flags to use when creating new threads in a server. These thread creation flags are used when the concurrency strategy is specified as thread-per-connection. In that case, the service handlers activated when a new connection is established are supplied these flags by the connection acceptor. By default, new threads are created with both the THR_BOUND and THR_DETACHED flags.

Usage Multiple thread flags may be specified as a single value by combining the desired flags using a vertical bar (‘|’) between the flags. No spaces may appear between the values and the ‘|’. For example:

```
-ORBThreadFlags THR_BOUND|THR_NEW_LWP
```

Impact This option only has an effect when the concurrency strategy is set to thread-per-connection. The specific impact to your system depends on your operating system’s threading capability.

If there is a limit to the number of lightweight processes that may be created, the use of THR_NEW_LWP can fail if your process exceeds that value.

An application may require that threads be invoked by a scheduler. To achieve this goal in an environment where requests are received asynchronously, the THR_SUSPENDED flag allows newly-created threads to wait to be started by a scheduler. This, of course, requires an application configured to resume all suspended threads appropriately.

See Also 19.3.5

Example

```
static Server_Strategy_Factory "-ORBThreadFlags THR_BOUND|THR_SUSPENDED"
```

19.3.10 ORBThreadPerConnectionTimeout *time*

Values for *time*

infinite	Threads spawned to handle requests for each client connection will never check for ORB shutdown.
n > 0	Threads spawned to handle requests for each client connection will check for ORB shutdown every `n` milliseconds (default is 5000).

Description When using the `thread-per-connection` concurrency strategy for a server, the ORB will spawn a new thread for each new connection, and that thread will be dedicated to handling incoming requests over its connection. These threads will remain active as long as the connection remains established. However, since these threads do not normally participate in the ORB's reactor, they need a way to determine if the ORB has been shut down. A timer can be used to wake up each thread on a specific time interval to check to see if the ORB has shutdown. The default time interval is indicated by the preprocessor macro `TAO_THREAD_PER_CONNECTION_TIMEOUT`, defined in `$TAO_ROOT/tao/orbconf.h` as 5000 milliseconds (5 seconds). You can use this server strategy factory option to either disable the timeout or change its value.

Usage Timeout after 1 second.

```
-ORBThreadPerConnectionTimeout 1000
```

Do not timeout.

```
-ORBThreadPerConnectionTimeout infinite
```

Impact This option only has an effect when the concurrency strategy is set to `thread-per-connection`. The specific impact to your system depends on your operating system's threading capability.

See Also 15.3.6.2, 19.3.5

Example

```
static Server_Strategy_Factory "-ORBThreadPerConnectionTimeout infinite"
```

19.3.11 ORBTransientIDPolicyDemuxStrategy *strategy*

Values for *strategy*

`active` (default)	Use the active demultiplexing strategy.
`dynamic`	Use the dynamic hashing strategy.
`linear`	Use the linear search strategy.

Description This option defines the demultiplexing strategy used by the ORB core for managing transient POAs. The active demultiplexing strategy provides the smallest lookup time per request, but at the cost of an eight-byte identifier. Dynamic hashing provides performance that is nearly as good as the active demux, but using a four-byte identifier. The linear search strategy provides reasonable performance for small maps, using a four-byte identifier, but does not scale very well as the number of active objects increases.

Usage Selecting the appropriate strategy involves determining the number of transient POAs that will be mapped and the cost associated with an eight-byte identifier versus a four-byte identifier.

Impact The active demultiplex strategy results in a larger object key and the occasional resizing of the map if the initial size is too small. The hashing strategy degrades performance as the number of objects increases. The linear search strategy quickly degrades lookup performance as the number of objects increases and results in additional performance degradation when the addition of new active objects forces the map to resize.

See Also 19.3.2, 19.3.7

Example

```
static Server_Strategy_Factory "-ORBTransientIDPolicyDemuxStrategy dynamic"
```

19.3.12 ORBUniqueIDPolicyReverseDemuxStrategy *strategy*

Values for *strategy*

`dynamic` (default)	Use a dynamic hash algorithm.
`linear`	Use a linear search algorithm.

Description This option defines the strategy used by a POA to locate an object identifier, based on a servant reference. This reverse lookup is possible only when the POA's uniqueness policy is set to `UNIQUE_ID`, meaning that each object ID will be associated with a different servant. If `MULITPLE_ID` is selected as the

uniqueness policy, there can be many object identifiers associated with a servant, so a reverse lookup is not possible.

Usage The default strategy is generally better than the linear search strategy. This is because the hashing algorithm for servants is simply to convert a pointer to the servant to an `unsigned long`. In a case of a highly-optimized environment, where the dynamic mapping classes are compiled out of ACE, then the linear search must be used.

Impact Space is allocated for a servant-to-object-ID map only when the POA's uniqueness policy is set to `UNIQUE_ID`. The amount of space initially allocated for the reverse lookup table is the same as the active object map size.

See Also 19.3.3

Example

```
static Server_Strategy_Factory "-ORBUniqueIDPolicyReverseDemuxStrategy linear"
```

19.3.13 ORBUserIDPolicyDemuxStrategy *strategy*

Values for *strategy*

`dynamic` (default)	Use a dynamic hash algorithm.
`linear`	Use a linear search algorithm.

Description This option defines the demultiplexing strategy used by the POA for managing active objects with user-specified object identifiers. Dynamic hashing provides very good lookup performance at the expense of hashing the object identifier. The linear search strategy provides reasonable performance for small maps, but does not scale very well as the number of active objects increases. Active demultiplexing is not an option with user-specified object identifiers, but an active *hint* may be used along with the native strategy for increased performance.

Usage Selecting the appropriate strategy involves determining the number of objects a POA must service and the cost associated with hashing the object identifiers versus iterating through a list comparing object identifiers.

Impact The dynamic hashing strategy incurs the additional lookup cost of hashing the object identifier.

The linear search strategy quickly degrades lookup performance as the number of objects increases, and results in additional performance degradation as the addition of new active objects forces the map to resize.

See Also 19.3.1, 19.3.3

Example

```
static Server_Strategy_Factory "-ORBUserIDPolicyDemuxStrategy linear"
```

CHAPTER 20

Client Strategy Factory

20.1 Introduction

The client strategy factory supports those elements of TAO that are specific to the behavior of clients. A client, for this discussion, is any application that actively establishes connections to other processes, sends requests, and perhaps receives replies. The client strategy factory provides control over several resources used by clients. For example, the factory supplies the mechanism employed when waiting for a response from a server. The client strategy factory also supplies a transport multiplexor that enables multiple asynchronous requests across a single connection. All of these objects are used by TAO internally and are not intended to be used by application code.

The client strategy factory is registered with the service configurator using the name `Client_Strategy_Factory`. TAO's default client strategy factory is statically registered with the service configurator and is initialized using the `static` directive. To supply options to the default client strategy factory, add a line, similar to the line shown below, to your service configuration file:

```
static Client_Strategy_Factory "-ORBoption value -ORBoption value ..."
```

The service configurator is discussed in greater detail in Chapter 16.

20.2 Interface Definition

Within a TAO application, the client strategy factory is accessed via an interface defined by the class `TAO_Client_Strategy_Factory`. Here we show all of the public functions of the client strategy factory, describe their use in relationship to the ORB, and the behavior of the default client strategy factory implementation.

Synopsis

```
class TAO_Export TAO_Client_Strategy_Factory : public ACE_Service_Object
{
public:
   enum Connect_Strategy
   {
     TAO_BLOCKED_CONNECT,
     TAO_REACTIVE_CONNECT,
     TAO_LEADER_FOLLOWER_CONNECT
   };

   TAO_Client_Strategy_Factory (void);
   virtual ~TAO_Client_Strategy_Factory (void);

   virtual ACE_Lock* create_profile_lock (void);
   virtual TAO_Transport_Mux_Strategy* create_transport_mux_strategy
   (TAO_Transport* transport);
   virtual TAO_Wait_Strategy* create_wait_strategy (TAO_Transport* transport);
   virtual int allow_callback (void);
   virtual TAO_Connect_Strategy *create_connect_strategy (TAO_ORB_Core *);

   virtual Connect_Strategy connect_strategy (void) const;
   virtual bool use_cleanup_options (void) const;
   virtual TAO_Configurable_Refcount create_profile_refcount (void);
   virtual ACE_Lock *create_transport_mux_strategy_lock (void);
   virtual int reply_dispatcher_table_size (void) const;
};
```

20.2.1 Profile Locking

CORBA defines a generic format called the Interoperable Object Reference (IOR) to identify objects. An object reference identifies a CORBA object and one or more paths through which the object can be accessed. Each path references one server location that implements the object, and an opaque identifier valid on that particular server. References to server locations are

called *profiles*. An object may have multiple server locations to support load balancing, fault tolerance, or other quality-of-service-specific optimizations.

Usually, an IOR's state is accessed in a read-only fashion by the client-side ORB. However, if the client receives a `LOCATION_FORWARD` reply message from a server, the client-side ORB can modify the IOR's state to update the affected profiles. Without appropriate synchronization, multiple threads can modify the IOR's internal profile information, potentially leading to corruption of the IOR's internal state or to inconsistent views of that state by other threads.

The profile lock interface is used internally by the stub to protect access to profile objects. A profile is an abstraction of the endpoint of a connection. For instance, the profile of an IIOP connection is defined by an open socket. Occasionally a profile in use may have to change, particularly if the location of a remote object changes. A lock is needed to protect against multiple threads attempting to modify the stub's profile. The client strategy factory supplies this lock through the following function:

```
virtual ACE_Lock* create_profile_lock (void);
```

The default client strategy factory creates a lock of the type specified by the `ORBProfileLock` option, shown in Table 20-1.

Table 20-1 Profile Locking Option

Option	Section	Description
`-ORBProfileLock {`**`thread`**` | null}`	20.3.4	Determines the type of profile lock created by the default client strategy factory.

The default behavior is to create a thread mutex. This enables a stub to be used by many threads in a client. If a client is single threaded, or all processing related to a single stub is performed in a single thread, use of a `null` mutex is acceptable and may provide better performance.

20.2.2 Transport Multiplexing Strategies

Transport multiplexing can optimize the use of a single connection between a client and a server for many concurrent requests. There are currently two kinds of transport multiplexing strategies defined: exclusive and multiplexed.

An exclusive transport uses a connection to service a single request and receive a reply before making it available for another request. A multithreaded client using the exclusive transport strategy will open a new connection for each concurrent request.

The multiplexed strategy allows multiple concurrent requests to share a connection, using asynchronous callbacks to handle the distribution of the replies. The multiplexed strategy uses a hash table to store the pending replies. This table's size is configured via the `-ORBReplyDispatcherTableSize` option. Locking for this table as well as other data used by the multiplexed strategy is determined by `-ORBTransportMuxStrategyLock`.

The transport multiplexing strategy is used internally by the transport portion of the communications protocol. The transport portion handles open connections and is defined as part of the pluggable protocols framework:

```
virtual TAO_Transport_Mux_Strategy *create_transport_mux_strategy
(TAO_Transport *transport);
```

The default client strategy factory supplies the appropriate type of strategy object, based on the option supplied to it by the service configurator. The options for selecting and configuring the multiplexing strategy are listed in Table 20-2.

Table 20-2 Transport Multiplexing Strategy Options

Option	Section	Description
`-ORBTransportMuxStrategy {exclusive | **muxed**}`	20.3.6	Determines the multiplexing strategy object created by the default client strategy factory.
`-ORBReplyDispatcherTableSize` *size*	20.3.5	Specifies the size of the dispatching table used by the muxed multiplexing strategy.
`-ORBTransportMuxStrategyLock {null | **thread**}`	20.3.7	Determines the lock used by the muxed multiplexing strategy object.

20.2.3 Wait Strategies

Wait strategy objects are used by the transport layer in the client to control the client's behavior while the client is anticipating a response from a server, after invoking a synchronous request. The wait strategy is also known as the client connection handler. The client strategy factory interface is required to return a wait strategy object for use by the protocol transport object:

```
virtual TAO_Wait_Strategy *create_wait_strategy (TAO_Transport *transport);
```

The default client strategy factory will return a wait strategy object that is one of the following four specializations: *wait-on-read*, *wait-on-reactor*, *wait-on-leader-follower, or wait-on-leader-follower-no-upcall*.

Wait-on-read is the simplest implementation. There is no locking and no reactor is used for the reply. When a request is sent to the server, this wait strategy simply blocks until it has read the reply. This strategy does not support handling new requests, and hence nested upcalls, while waiting for the reply from the server. This strategy provides the lowest possible overhead for a multithreaded environment. In a single-threaded client this strategy has an adverse affect on performance, as the process would block on every call. Wait-on-read corresponds to the `rw` option.

Wait-on-reactor is a very effective strategy for single-threaded clients. With this model, the transport registers with the reactor and uses an event handler internally to wait for a response. This strategy should only be used for single-threaded clients, as the threads must share access to the reactor. Wait-on-reactor corresponds to the `st` option.

Wait-on-leader-follower is the strategy intended for use in reactor-based multithreaded clients. This strategy enables multiple threads to register with a reactor and use an event handler for processing the reply when it is available. Wait-on-leader-follower corresponds to the `mt` option, and is the default.

The wait-on-leader-follower-no-upcall strategy combines the features of the wait-on-read and wait-on-leader-follower strategies. Like the wait-on-leader-follower strategy, it allows multiple threads to register with a reactor and participate in the *leader-follower* interaction described in 15.4.4.3. However, like the wait-on-read strategy, it does not allow these threads to handle nested upcalls. Wait-on-leader-follower-no-upcall corresponds to the `mt_noupcall` option.

Note *The wait-on-leader-follower-no-upcall wait strategy is an* ***experimental*** *wait strategy. It was designed specifically for and has been thoroughly tested in the use cases presented in 15.4.4.4. It has not been exercised in a wide variety of other use cases. Please use caution and due diligence in testing your application's behavior with this option if you decide to use it.*

More information on the effect of wait strategy on threading behavior is provided in 15.3.6 and 15.4.4.

The client strategy factory interface defines another function that indicates whether callbacks should be allowed:

```
virtual int allow_callback (void);
```

This function returns zero when the wait-on-read wait strategy is used. It returns a non-zero value otherwise. The wait-on-read strategy will still allow callbacks as long as the application is multithreaded. This function is currently only used to enable special optimizations when using SHMIOP in combination with the wait-on-read wait strategy.

The default client strategy factory supplies the appropriate type of strategy object, based on the option supplied to it by the service configurator. By using the `ORBWaitStrategy` option, shown in Table 20-3, the choice of a wait strategy is deferred until run time.

Table 20-3 Wait Strategy Options

Option	Section	Description
`-ORBWaitStrategy {`**`mt`**` | st | rw | mt_noupcall}`	20.3.8	Determines the type of wait strategy object created by the default client strategy factory.
`-ORBConnectionHandlerCleanup {`**`0 | 1`**`}`	20.3.1	Specifies connection handler cleanup for use with the receive-wait wait strategy.
`-ORBDefaultSyncScope {none | `**`transport`**` | server | target}`	20.3.3	Specify the default synch scope for one-way calls

Note *The -ORBWaitStrategy option is the same as the -ORBClientConnectionHandler option. The use of the -ORBClientConnectionHandler option has been deprecated.*

Note *If you use the wait-on-read (`rw`) wait strategy option, you should also use the `exclusive` transport multiplexing option. Otherwise, unexpected results may occur. See 20.2.2.*

20.2.4 Connect Strategies

Connect strategy objects are used by the transport layer in the client to control the client's behavior while initiating a connection to a server. The client strategy factory interface is required to return a connect strategy object for use by the transport-protocol-specific connector object:

```
virtual TAO_Connect_Strategy *create_connect_strategy (TAO_ORB_Core
*orb_core);
```

The default client strategy factory will return a connect strategy object that is one of the following three specializations: blocking, reactive, or leader-follower.

The blocking connect strategy is the simplest implementation. There is no locking and no reactor is used to wait for the connection to succeed or fail. When a connection is attempted to a server, this connect strategy simply blocks until the connection attempt completes successfully or fails with an error. This strategy provides the lowest possible overhead for a multithreaded environment. In a single-threaded client this strategy may have an adverse affect on performance, as the process would block each time a new connection is established. The blocking connect strategy corresponds to the `blocked` option.

The reactive connect strategy is a very effective strategy for single-threaded clients. With this model, the connector registers with the reactor and uses an event handler internally to wait for the connection attempt to complete. The thread is not blocked while the connection is being attempted. This strategy should only be used for single-threaded clients, as the threads must share access to the reactor. The reactive connection strategy corresponds to the `reactive` option.

The leader-follower connect strategy is intended for use in reactor-based multithreaded clients. This strategy enables multiple threads to register with a reactor and use an event handler for dealing with completed connection attempts. The leader-follower connect strategy corresponds to the `lf` option, and is the default.

More information on the effect of connect strategy on threading behavior is provided in 15.4.1.

The default client strategy factory supplies the appropriate type of strategy object, based on the option supplied to it by the service configurator. By using

the `ORBConnectStrategy` option, shown in Table 20-4, the choice of a connect strategy is deferred until run time.

Table 20-4 Connect Strategy Option

Option	Section	Description
`-ORBConnectStrategy {lf | reactive | blocked}`	20.3.2	Determines the type of connect strategy object created by the default client strategy factory.

20.3 Client Strategy Factory Options

The remainder of this chapter describes the individual options interpreted by the default client strategy factory. These options are supplied to the default client strategy factory by the service configurator through the use of a `static` initialization directive (see 16.3).

20.3.1 ORBConnectionHandlerCleanup *enabled*

Values for *enabled*

`0` (default)	No connection handler cleanup happens when errors occur while using the receive-wait wait strategy.
`1`	Enables connection handler cleanup when errors occur while using the receive-wait wait strategy.

Description When using the receive-wait wait strategy, the default behavior (for performance reasons) will not consistently clean up connection handlers when errors occur. This option specifies that the receive-wait strategy should ensure that these resources are properly cleaned up.

Usage Use this option when using the receive-wait wait strategy and you need to ensure that resources are cleaned up properly after errors.

This option requires that the ORB's event processing loop is active. You must have a thread calling `orb::run()` or `orb::perform_work()` in order for the connection handlers to be cleaned up as appropriate.

Impact There is a small performance impact to enabling this option as it requires registering and unregistering connection handler with the reactor for each invocation.

See Also 20.3.8

Example

```
static Client_Strategy_Factory "-ORBConnectionHandlerCleanup 1"
```

20.3.2 ORBConnectStrategy *connect_type*

Values for *handler_type*

`lf` (default)	Use the multithreaded connect strategy, which is based on the leader-follower model.
`reactive`	Use the reactive connect strategy, which allows non-blocking connects in a single-threaded client.
`blocked`	Use the blocking connect strategy. Client threads block while initiating a connection with a server.

Description This option specifies the way clients wait while initiating connections to servers. The choice of connect strategy depends on the architecture of the client.

The default multithreaded `lf` strategy uses the reactor and exhibits non-blocking behavior when initiating a connection to a server. This strategy participates in the leader-follower protocol to synchronize access to the reactor.

The `reactive` strategy is based on a traditional reactor that waits for events, then loops through event handlers sequentially within a single thread of control.

Finally, in the `blocked` strategy, the transport-level connector blocks for the duration of the connection attempt with the server. This option should only be used in multithreaded environments.

Usage In a single-threaded application, either the leader-follower or reactive options are acceptable. In a multithreaded application, either the leader-follower or the blocking options are acceptable.

Impact The reactive option provides a performance improvement in single threaded applications by avoiding the overhead of thread management and locking.

The blocking option provides the lightest weight code, but pushes the burden of thread management to the application developer.

See Also 15.4.1

Example

```
static Client_Strategy_Factory "-ORBConnectStrategy reactive"
```

20.3.3 ORBDefaultSyncStrategy *type*

Values for *type*

none	equivalent to SYNC_NONE
transport (default)	equivalent to SYNC_WITH_TRANSPORT
server	equivalent to SYNC_WITH_SERVER
target	equivalent to SYNC_WITH_TARGET

Description This option is intended to allow an application to be deployed with different one-way synchronization than it was originally written. The default sync scope as defined in the CORBA Messaging specification is `SYNC_WITH_TRANSPORT` which means that a one-way call will block only as long as it takes to establish a connection if necessary then send the request. It does not wait for any sort of a reply, which could be an exception. If you need to be able to handle exceptions, at least `SYNC_WITH_SERVER` is required. Ordinarily this is done explicitly by the application developer applying a CORBA::SyncScope policy to an object reference or the ORB.

There may be cases where the developer is not aware of this need, such as when a server it uses is deployed behind an Implementation repository, load balanced, or made fault tolerant. Or even if a `corbaloc` style IOR is used. All of these situations make use of location forwarding to have the client stub code update its connection information and retransmit a request. This cannot happen if the request is a one-way.

Usage Set this option to `server` to ensure proper request forwarding.

Example

```
static Client_Strategy_Factory "-ORBDefaultSyncStrategy server"
```

20.3.4 ORBProfileLock *lock_type*

Values for *lock_type*

thread (default)	Access to the object profiles is guarded with a mutual exclusion lock.
null	Object profiles are not locked.

Description This option controls the locking strategy for the internal state of an object reference. In particular, the profile portion is guarded to avoid modification as the result of receiving `LOCATION_FORWARD` messages.

By default, a `thread` mutex lock is used to synchronize access to the profile during forwarding events.

Usage A client application may safely avoid the use of locks if it is single threaded, if object references are not shared among threads, or if `LOCATION_FORWARD` messages are guaranteed not to be received.

Impact Using a `null` mutex for object references, when it is safe to do so, enhances performance.

The extra lock introduces potential for priority inversion.

Example

```
static Client_Strategy_Factory "-ORBProfileLock null"
```

20.3.5 ORBReplyDispatcherTableSize *size*

Values for *size*

`16` (default)	Allocate space for 16 elements in the reply dispatcher table that is used for multiplexed connections.
`> 0`	Specify the number of elements to allocate in the reply dispatcher table used for multiplexed connections.

Description When `-ORBTransportMuxStrategy` is set to `muxed`, this option defines the size of the hash table used for demultiplexing reply messages that are received on that connection.

Usage If your client applications can have many pending replies on a single connection, setting this option to a higher value should allow for more efficient dispatching of replies.

Impact Each increment of this value causes the ORB to consume at least 16 additional bytes of memory per connection.

See Also 20.3.6

Example

```
static Client_Strategy_Factory "-ORBReplyDispatcherTableSize 64"
```

20.3.6 ORBTransportMuxStrategy *strategy*

Values for *strategy*

`exclusive`	Only one request may be pending on a connection.
`muxed` `(default)`	Pending requests are multiplexed on a connection.

Description This option controls the request multiplexing strategy of the transport. The `exclusive` strategy means that no more than one request may be pending on a single connection. If another request is issued, a new connection must be established. The `muxed` strategy enables more than one pending request on a single connection. With this strategy, pending requests and the related callbacks are stored in a table so that responses may be routed appropriately. The size of this table is specified by the `-ORBReplyDispatcherTableSize` option.

Note *Use of this option does not guarantee that multiple requests to the same server will use the same connection, it just allows the ORB to do so. A number of factors may cause TAO to open multiple connections to the same server, even when the muxed strategy is used.*

The transport multiplexing strategy is used by the pluggable transport. See 14.18.3 for details of the pluggable transport.

Usage The `exclusive` strategy is intended for use with ordinary synchronous request/reply behavior. This strategy provides a smaller memory footprint as no table is needed to store pending requests.

Impact The `muxed` strategy is required for asynchronous messaging. The `muxed` strategy may also be desirable in a multithreaded client where many client threads are simultaneously invoking synchronous requests through a single ORB to the same target server. If the `exclusive` strategy is used in this case, multiple connections may be created between the client and server ORBs to transmit multiple simultaneous requests, which can be a problem in environments where resources, such as file descriptors, are scarce.

See Also 15.4.2, 20.3.5, 20.3.7

Example

```
static Client_Strategy_Factory "-ORBTransportMuxStrategy exclusive"
```

20.3.7 ORBTransportMuxStrategyLock *lock_type*

Values for *lock_type*

thread (default)	When the transport multiplexing strategy is muxed, its data is guarded with a mutual exclusion lock.
null	When the transport multiplexing strategy is muxed, its data is not locked.

Description This option controls the locking strategy for the muxed strategy of -ORBTransportMuxStrategy.

By default, a thread mutex lock is used to synchronize access to the multiplexed strategy's table during the processing of requests and replies.

Usage A client application may safely avoid the use of these locks if it is single threaded or if it can guarantee only one thread is engaged in TAO-related activities.

Impact Using a null mutex for this strategy, when it is safe to do so, enhances performance.

See Also 20.3.6

Example

```
static Client_Strategy_Factory "-ORBTransportMuxStrategyLock null"
```

20.3.8 ORBWaitStrategy *handler_type*

Values for *handler_type*

mt (default)	Use the multithreaded wait strategy, which is based on the leader-follower model.
mt_noupcall	Use the multithreaded wait strategy, which is based on the leader-follower model, but which does not allow client threads to handle nested upcalls while they are waiting for replies from a server.
st	Use the single-threaded wait strategy, which allows multiple connection handlers to share a reactor.
rw	Use the receive-wait wait strategy. Client threads block while waiting for replies from servers.

Description This option specifies the way client threads wait for replies during two-way invocations. In two-way invocations, the client thread is expected to behave synchronously with respect to the server. The client submits a request, then

immediately waits for a reply. The choice of wait strategy depends on the architecture of the client application.

The multithreaded strategy implements the *leader-follower* pattern, in which many connection-handler threads are used to asynchronously handle replies from the server.

The single-threaded reactive strategy is based on a traditional reactor that waits for events, then loops through event handlers sequentially within a single thread of control.

In the receive-wait strategy, the connection handler blocks in the `recv()` system call waiting for a reply from the server. This option should only be used in multithreaded environments.

Note *This option has no effect for AMI requests as they do not ever wait for a reply.*

Usage In a single-threaded application, either the multithreaded or single-threaded reactor-based options are acceptable. In a multithreaded application, either the multithreaded or the receive-wait options are acceptable.

Impact The single-threaded reactor option provides a performance improvement over the multithreaded strategy by avoiding the overhead of thread management and locking.

The receive-wait option provides the lightest weight code, but pushes the burden of thread management to the application developer.

The multithreaded-no-upcall option allows multiple connection-handler threads to share the same reactor, but avoids nested upcalls. It is considered and ***experimental*** feature.

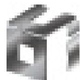
See Also 15.3.6, 18.6.7, 20.3.2, 20.3.6

Example

```
static Client_Strategy_Factory "-ORBWaitStrategy st"
```

Note *If you use the wait-on-read (`rw`) wait strategy option, you should also use the `exclusive` transport multiplexing option (20.3.6) to avoid sharing a single connection among threads making concurrent outgoing requests to the same server.*

Note *You must also use the* `blocked` *connect strategy (20.3.2) and the* `blocking` *flushing strategy (18.6.7) when using the wait-on-read (*`rw`*) wait strategy to keep the client thread from entering the ORB's leader-follower during connection establishment or during output flushing and potentially being used by the ORB to handle incoming requests.*



Part 4

TAO Services

CHAPTER 21

TAO Services Overview

21.1 Introduction

The OMG's *CORBAservices* specifications define basic services—such as Naming, Events, and Notification—that can be used in a wide variety of applications. Each CORBAservice specification includes standard IDL interfaces that help insulate application developers from differences in implementation. Applications benefit by reusing existing components; CORBA vendors benefit by being able to focus on quality of implementation rather than on interface design.

A governing design principle for the OMG services is that each service does only one thing, but it does it very well. For example, the Naming Service and Object Trading Service provide similar functionality. The Naming Service allows applications to look up objects by name; it is like the telephone *white pages*. By contrast, the Object Trading Service looks up objects based on the properties of the services they provide; it is like the *yellow pages*. Yet, in spite of the commonality between the object lookup strategies for *white pages* and *yellow pages*, there is a separate service defined for each. This minimizes the interfaces and keeps them as simple as possible. It also makes it possible to

use the basic OMG services as building blocks for more sophisticated custom services.

Another governing design principle is to use CORBA features, such as IDL, for specifying the interface separately from the implementation. Consequently, the OMG services from any CORBA compliant ORB can be used by servers and clients from another CORBA compliant ORB. You could, for example, use TAO's Naming Service for distributed objects that are using a CORBA ORB based on Java, such as JacORB.

As mentioned in Chapter 1, TAO implements a number of the OMG's CORBAservices, as well as certain TAO-specific services. The OMG-defined CORBAservices are easily distinguished from non-OMG-defined services by the names of the modules; all the OMG services include the prefix "Cos" in the module name. For example, the Naming Service interfaces are all contained within the module `CosNaming`. Table 21-1 lists the services that are provided with TAO and the libraries with which applications that use each service must link.

21.2 Customizing Access to the Services

The CORBAservices specifications do not deal with implementation issues such as executable file names or allowable command-line arguments. In fact, you can view the driver programs (the code that initializes and registers the servants implementing the CORBAservices) used for the TAO services as *sample* drivers. If you examine the source code of the drivers you will find that they are quite small and simple; most of the work is done by the servants in the libraries that implement the IDL interfaces of the services.

Note *To avoid confusion, when discussing CORBA and TAO services, the term* ***service*** *refers to the library code that implements the IDL interfaces, whereas the term* ***server*** *refers to the executable that instantiates and initializes the servant(s) running this code.*

The servants that implement the CORBA and TAO services have been designed so that they can be easily created and initialized within your own application programs. By accessing these services programmatically, you can guarantee that the service you need will be available during the lifetime of

your application, take advantage of collocation optimizations, and reduce the number of processes in your application.

21.3 TAO's ORB Services Libraries

The code that implements the IDL interfaces for each of the services included with TAO is located within one or more libraries. Services with a single library put all their code in that single library which must be linked by all clients and servers participating in that service. Many other services break the service functionality into multiple libraries. A common pattern, shared by several services, is to have individual stub, skeleton, and server libraries. For example, the Naming Service has the following libraries:

- `TAO_CosNaming` - Naming service stub and client code
- `TAO_CosNaming_Skel` - Naming service skeleton code
- `TAO_CosNaming_Serv` - Naming service server code

The `TAO_CosNaming` library contains the code needed by Naming Service clients that only need to interact with this service as CORBA clients. The `TAO_CosNaming_Skel` library contains the generated skeletons and is required if your Naming Service client implements any service-defined IDL interfaces. The `TAO_CosNaming_Serv` library contains all the code necessary in building servers for the Naming Service. The server library usually includes the vast majority of a service's code and means that clients no longer need to link this code with their application.

Some services place some of their functionality in additional libraries and require application developers to link those libraries when those features are desired. These feature-oriented libraries are discussed in the chapter documenting that service.

The source code for these libraries is contained in the `$TAO_ROOT/orbsvcs/orbsvcs` directory and its subdirectories. The TAO services and their individual libraries are listed in Table 21-1.

Table 21-1 TAO ORB Services Libraries

Service	Library Name
Audio/Video Streaming	`TAO_AV`

Table 21-1 TAO ORB Services Libraries

Service	Library Name
Concurrency Control	TAO_CosConcurrency TAO_CosConcurrency_Skel TAO_CosConcurrency_Serv
Event	TAO_CosEvent TAO_CosEvent_Skel TAO_CosEvent_Serv
Fault Tolerance	TAO_FaultTolerance TAO_FT_ClientORB TAO_FTORB_Utils TAO_FTRT_ClientORB TAO_FtRtEvent TAO_FTRTEventChannel TAO_FT_ServerORB
Interface Repository	TAO_IFRService
Life Cycle	TAO_CosLifeCycle TAO_CosLifeCycle_Skel
Load Balancing	TAO_CosLoadBalancing
Log (Admin)	TAO_DsLogAdmin TAO_DsLogAdmin_Skel TAO_DsLogAdmin_Serv
Log (Event-based)	TAO_DsEventLogAdmin TAO_DsEventLogAdmin_Skel TAO_DsEventLogAdmin_Serv
Log (Notify-based)	TAO_DsNotifyLogAdmin TAO_DsNotifyLogAdmin_Skel TAO_DsNotifyLogAdmin_Serv
Log (RTEvent-based)	TAO_RTEventLogAdmin TAO_RTEventLogAdmin_Skel TAO_RTEventLogAdmin_Serv
Naming	TAO_CosNaming TAO_CosNaming_Skel TAO_CosNaming_Serv
Notification	TAO_CosNotification TAO_CosNotification_Skel TAO_CosNotification_Serv TAO_CosNotification_Persist TAO_CosNotification_MC TAO_CosNotification_MC_Ext TAO_RTNotification
Object Trading	TAO_CosTrading TAO_CosTrading_Skel TAO_CosTrading_Serv
Property	TAO_CosProperty TAO_CosProperty_Skel TAO_CosProperty_Serv

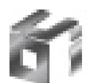


Table 21-1 TAO ORB Services Libraries

Service	Library Name
Real-Time Event	TAO_RTEvent TAO_RTEvent_Skel TAO_RTEvent_Serv TAO_RTCORBAEvent TAO_RTKokyuEvent TAO_RTSchedEvent
Real-Time Scheduling	TAO_RTSched TAO_RTCosScheduling
Security	TAO_Security
Time	TAO_CosTime TAO_CosTime_Skel TAO_CosTime_Serv

In addition, some common functionality shared by multiple services is located in utility libraries such as `TAO_Svc_Utils` and `TAO_PortableGroup`.

The combination of multi-library services, dependencies between services, and the utility libraries make generating the full set of service libraries a challenge. The easiest way to resolve these dependencies is to use MPC and TAO's service base projects to generate your link commands. For example, to build an executable that uses TAO's Naming Service as a pure CORBA client, create an MPC project that inherits from the `namingexe` base project. The following sample MPC project file is taken from `$TAO_ROOT/orbsvcs/tests/Simple_Naming/Simple_Naming.mpc`:

```
project(*Client) : namingexe, utils, portable_server {
  Source_Files {
    client.cpp
  }
}
```

The `namingexe` project simply inherits from the `naming` and `taoexe` projects. The naming project ensures that we get the proper include paths, environment variables, and link to the `TAO_CosNaming` library. You can also derive your projects directly from `naming`, `naming_skel`, and `naming_serv` base projects that TAO defines.

Using any of these projects simplifies your build process and may make it more portable if there are further library changes in later versions of TAO. Look in `$ACE_ROOT/bin/MakeProjectCreator/config` for all of TAO's base projects. See Chapter 3 and the MPC documentation for more information on using MPC.

To access the header files for the services, use the form:

```
#include <orbsvcs/service-nameC.h>
```

For example, to include the header file for the Naming Service, add the following line to your code:

```
#include <orbsvcs/CosNamingC.h>
```

21.4 Locating Service Objects

Many of TAO's services can be located by passing a pre-defined object identifier to the ORB's `resolve_initial_references()` operation. For example:

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Resolve the Naming Service.
CORBA::Object_var naming_obj = orb->resolve_initial_references("NameService");
```

When `resolve_initial_references()` is passed an object identifier (such as "`NameService`"), it attempts to find the corresponding service object. The `resolve_initial_references()` implementation performs the following steps, in order, until the operation succeeds, an exception occurs, or the operation times out:

Note *The general algorithm here is compliant with section 8.5.3, "Configuring Initial Service References," of the OMG CORBA 3.1 specification. The details of the environment variables (step 3) and multicast discovery (step 4) are specific to TAO.*

1. If an initial reference to the service object was previously specified using `CORBA::ORB::register_initial_reference()`, that object reference is returned.
2. If the initial reference to the service object was specified using the `ORBInitRef` option, e.g.,

```
-ORBInitRef NameService=corbaloc::tango:2809/NameService
```

then the stringified object reference provided as an argument to `-ORBInitRef` is passed to `CORBA::ORB::string_to_object()` to obtain the object reference to the service object. See 17.13.36 for more information about the `ORBInitRef` option.

3. An environment variable name is constructed by appending “IOR” to the object identifier. If this environment variable is set, its value is passed to `CORBA::ORB::string_to_object()` as above. For example, when the object identifier is `NameService`, the corresponding environment variable is `NameServiceIOR`.
4. If the service object identifier is one of `NameService`, `TradingService`, `InterfaceRepository`, or `ImplRepoService` then TAO attempts multicast discovery of the service. See 21.4.1 for details of multicast discovery.
5. If the `ORBDefaultInitRef` option was used, e.g.,

```
-ORBDefaultInitRef corbaloc::tango:2809
```

   then a slash character (“`/`”) plus the service object identifier are appended to the URL prefix provided as an argument to `-ORBDefaultInitRef`. The resulting stringified object reference is then passed to `CORBA::ORB::string_to_object()` as above. See 17.13.10 for more information about the `ORBDefaultInitRef` option.
6. Return a null object reference.

21.4.1 Multicast Service Discovery

TAO implements a multicast discovery mechanism for locating the following services:

- Naming
- Trading
- Implementation Repository
- Interface Repository

Whenever the CORBA-standard and environment variable mechanisms are not available for locating one of these services, TAO tries to use its multicast mechanism to locate the service. The basic process is that a datagram is sent to a multicast address and port requesting the location of the service. Any servers

listening on this address reply with their object reference. If multiple services reply, the client uses the first response it receives. If no server replies within the timeout specified by the preprocessor macro `TAO_DEFAULT_SERVICE_RESOLUTION_TIMEOUT` defined in `$TAO_ROOT/tao/orbconf.h`, an exception of type `CORBA::ORB::InvalidName` is raised.

By default, multicast discovery is enabled for clients, meaning they attempt to use it when other mechanisms are not used. Most servers disable it by default and supply command line options to enable it (typically `-m 1`).

Although multicast discovery works well for some applications, its use is problematic for others. The most common issue encountered is when multiple servers are configured to use the same multicast address. Different client applications then randomly connect to one of the available servers based on which reply is received. Typically, this is not what the application requires or expects.

The following steps describe how the multicast address and port are determined for a given service. Once the address and port are defined the remaining steps are skipped.

1. If the object indentifier is `NameService` and the `ORBMulticastDiscoveryEndpoint` option was specified, the value of its argument is used as the IP address and port number.
2. Except for the Interface Repository, each of the other services that use multicast discovery defines an ORB initialization option that allows for run-time specification of the multicast discovery port.
3. The multicast discovery port can also be specified via an environment variable for each of the services.
4. If all of the previous methods fail, the default multicast discovery port number for that service is used. The default port numbers are defined in `$TAO_ROOT/tao/default_ports.h` using preprocessor macros. For example, the Naming Service's default port number is defined by `TAO_DEFAULT_NAME_SERVER_REQUEST_PORT`.

If the address is not specified in step 1 above, then the default multicast address of 224.9.9.2 is used. The following table summarizes the Object ID,

ORB initialization option, environment variable, and default port for each of the services that use multicast.

Table 21-2 Multicast Discovery Services

Object ID	ORB Initialization Option	Environment Variable	Default Port
NameService	-ORBNameServicePort	NameServicePort	10013
TradingService	-ORBTradingServicePort	TradingServicePort	10016
ImplRepoService	-ORBImplRepoServicePort	ImplRepoServicePort	10018
InterfaceRepository	N/A	InterfaceRepoServicePort	10020

In addition, multicast discovery can also be specified via the -ORBInitRef and -ORBDefaultInitRef options. See 17.13.36 for more details.

CHAPTER 22

Naming Service

22.1 Introduction

The OMG Naming Service version 1.3 (OMG Document formal/04-10-03) defines a means for mapping names to object references. Names are composed of sequences of simple structures that are, in turn, constructed from text strings. The Naming Service allows applications in a distributed system to locate CORBA objects by name rather than by caching object references or passing around stringified IORs. It is similar to the *white pages* in a phone book that maps telephone numbers to names. Chapter 18 of *Advanced CORBA Programming with C++* presents a detailed discussion of the Naming Service.

TAO provides a complete implementation of the Naming Service specification.

Note *TAO does not support the Lightweight Naming Service defined in section 3 of the Version 1.3 Naming Service Specification. The Lightweight Naming Service is a subset of the full naming service of resource-constrained systems.*

The Naming Service may be run as a stand-alone executable called `tao_cosnaming`, or it can be built into an application by linking in the appropriate libraries. The Naming Service interfaces and data types from the OMG's `CosNaming` module are defined in the file `$TAO_ROOT/orbsvcs/orbsvcs/CosNaming.idl`. Applications that use TAO's Naming Service implementation should include the header file `$TAO_ROOT/orbsvcs/orbsvcs/CosNamingC.h` and link with the `TAO_CosNaming` library. Applications that wish to implement the Naming Service internally will need to link with the `TAO_CosNaming_Skel` and `TAO_CosNaming_Serv` libraries.

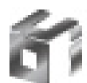
A variant of the Naming Service, discussed in 22.9, is available that provides a fault tolerant implementation which supports replication of state between dual redundant servers, transparent and seamless failover, and load balancing functionality.

22.1.1 Road Map

The Naming Service is perhaps the most widely used of all the OMG CORBA services. It is well documented in sources such as *Advanced CORBA Programming with C++* and *Pure CORBA*. Therefore, this chapter does not attempt to completely explain how to use the standard Naming Service from your application. Instead, this chapter focuses on the TAO-specific features of the Naming Service.

If you want to learn more about...

- *how to find the Naming Service* from within your application, see 22.2, "Resolving the Naming Service."
- *code that uses the basic Naming Service features* from an application's clients and servers, see 22.3, "Naming Service Example."
- *using the "corbaloc" and "corbaname" Object URL features* of the Interoperable Naming Service specification, see 22.4, "Object URLs."
- *code that uses the extended Naming Service interfaces (NamingContextExt)*, see 22.5, "The NamingContextExt Interface."
- *TAO's classes that implement Naming Service features*, for both clients and servers, see 22.6, "TAO-Specific Naming Service Classes."
- *how to use TAO's Naming Service utility programs*, see 22.7, "Naming Service Utilities."



- *command-line options affecting TAO's tao_cosnaming executable*, including the *Naming Service persistence features*, see 22.8, "Naming Service Command Line Options."
- *using the Fault Tolerant Naming Service,* see 22.9, "Fault Tolerant Naming Service."

To fully understand and take advantage of the Naming Service's features, be sure to read Chapter 18 of *Advanced CORBA Programming with C++* or Chapter 6 of *Pure CORBA* in addition to this chapter.

Full source code for all the examples presented in this chapter is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NamingService`.

22.2 Resolving the Naming Service

To find the Naming Service, your application can use the `CORBA::ORB::resolve_initial_references()` operation, passing it the string "`NameService`". For example:

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Resolve the Naming Service.
CORBA::Object_var naming_obj = orb->resolve_initial_references("NameService");
```

`CORBA::ORB::string_to_object()` may also be used to resolve the Naming Service. Examples using this technique are provided in 23.4.

When `resolve_initial_references()` is passed the string "`NameService`", it attempts to find the root Naming Context of the Naming Service. TAO's `resolve_initial_references()` implementation is discussed in more detail in 21.4.

If `resolve_initial_references()` succeeds, it returns the CORBA object reference to the Naming Service's root Naming Context. To use this object reference as a `CosNaming::NamingContext`, you must first narrow it, as follows:

```
CosNaming::NamingContext_var root =
  CosNaming::NamingContext::_narrow(naming_obj.in());
```

TAO's Naming Service implements the `CosNaming::NamingContextExt` interface from OMG Document formal/02-09-02, so you can also narrow the object reference returned from `resolve_initial_references()` as follows:

```
CosNaming::NamingContextExt_var root =
  CosNaming::NamingContextExt::_narrow(naming_obj.in());
```

22.3 Naming Service Example

To help you become familiar with the TAO Naming Service, we now present a simple example based on the `Messenger` example which was introduced in Chapter 3.

Recall that the original `Messenger` server writes an IOR as a string to a file. The client then reads the IOR string from the file and converts it back into an object reference. Of course, to use this mechanism, the client and server must be running on the same machine or at least on machines that share a file system, or you must transmit the stringified IOR to the client through some other means. To overcome this limitation, we will use the Naming Service and assume that both the server and client have access to the Naming Service.

22.3.1 Source Code Listings for the Example

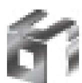
We have modified the `Messenger` server and client source code to use the Naming Service rather than working with a stringified IOR. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NamingService/Messenger`.

22.3.1.1 Server C++ Source Code File

The server now uses the Naming Service rather than writing its IOR as a string to a file. The implementation of the server is in `MessengerServer.cpp`.

```
#include "Messenger_i.h"
#include <orbsvcs/CosNamingC.h>
#include <iostream>

int main(int argc, char* argv[])
{
  try {
```

```
// Initialize orb
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

//Get reference to Root POA
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

// Activate POA Manager
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();

// Create an object
PortableServer::Servant_var<Messenger_i> messenger_servant =
  new Messenger_i();

// Find the Naming Service
CORBA::Object_var naming_obj =
  orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var root =
  CosNaming::NamingContext::_narrow(naming_obj.in());
if(CORBA::is_nil(root.in())) {
  std::cerr << "Nil Naming Context reference" << std::endl;
  return 1;
}

// Bind the example Naming Context, if necessary
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("example");
try {
  CORBA::Object_var dummy = root->resolve(name);
}
catch (const CosNaming::NamingContext::NotFound &) {
  CosNaming::NamingContext_var dummy = root->bind_new_context(name);
}

// Bind the Messenger object
name.length(2);
name[1].id = CORBA::string_dup("Messenger");

PortableServer::ObjectId_var oid =
  poa->activate_object(messenger_servant.in());
CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
root->rebind(name, messenger_obj.in());

std::cout << "Messenger object bound in Naming Service" << std::endl;

// Accept requests
orb->run();
orb->destroy();
```



```
  }
  catch(CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
  }
  return 0;
}
```

22.3.1.2 Client C++ Source Code File

The client now uses the Naming Service rather than reading the server object's IOR as a string from a file. The implementation of the client is in the file `MessengerClient.cpp`.

```
#include "MessengerC.h"
#include <orbsvcs/CosNamingC.h>
#include <iostream>

int main(int argc, char* argv[])
{
  try {
    // Initialize orb
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Find the Naming Service
    CORBA::Object_var naming_obj =
      orb->resolve_initial_references("NameService");
    CosNaming::NamingContext_var root =
      CosNaming::NamingContext::_narrow(naming_obj.in());
    if(CORBA::is_nil(root.in())) {
      std::cerr << "Nil Naming Context reference" << std::endl;
      return 1;
    }

    // Resolve the Messenger object
    CosNaming::Name name;
    name.length(2);
    name[0].id = CORBA::string_dup("example");
    name[1].id = CORBA::string_dup("Messenger");
    CORBA::Object_var obj = root->resolve(name);

    // Narrow the Messenger object reference
    Messenger_var messenger = Messenger::_narrow(obj.in());
    if (CORBA::is_nil(messenger.in())) {
      std::cerr << "Not a Messenger reference" << std::endl;
      return 1;
    }
```

```
    // Send a message
    CORBA::String_var message = CORBA::string_dup("Hello!");
    messenger->send_message("TAO User", "TAO Test", message.inout());
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
  }
  std::cout << "Message was sent" << std::endl;
  return 0;
}
```

22.3.2 Building the Example

The example includes an MPC file `Messenger.mpc` that defines projects for building `MessengerServer` and `MessengerClient`. These projects can be built from the workspace generated from `$TAO_ROOT/orbsvcs/DevGuideExamples/DevGuideExamples.mwc`.

22.3.3 Running the Example

Now that you have built the server and client, you are ready to run the example using the Naming Service. The first step is to start the Naming Service, which can be done with or without support for multicast discovery.

22.3.3.1 Starting the Naming Service with Multicasting

The TAO Naming Service server executable is typically found in the directory `$TAO_ROOT/orbsvcs/Naming_Service`.

TAO clients will try to locate the Naming Service as described in 22.2. If they cannot locate the Naming Service by another method, they will attempt to locate it using multicast discovery. When invoked with the `-m 1` option, the Naming Service will listen for these multicast service discovery requests and respond with the IOR of its root naming context.

Multicasting is connectionless, so a process sending a request need not wait for a recipient to respond. However, datagrams do not guarantee message delivery, nor are messages necessarily received in the same order as they were transmitted.

Note *For more details see the discussions of multicast service discovery in 21.4.1 and the* `mcast:` *address in 17.13.10.*

If your network supports multicast, and you want your applications to find the Naming Service using multicast service discovery, start the TAO Naming Service server as follows (assuming that `tao_cosnaming` is in your `PATH`):

```
$ tao_cosnaming -m 1
```

Next, run the `MessengerServer` in its own terminal window:

```
$ ./MessengerServer
```

and run the `MessengerClient` from a different terminal window:

```
$ ./MessengerClient
```

You should see the following messages in the `MessengerServer`'s window:

```
Message from: TAO User
Subject:      TAO Test
Message:      Hello!
```

and in the `MessengerClient`'s window:

```
message was sent
```

The client terminates, but the `MessengerServer` will continue to run until you kill it. After you kill the `MessengerServer`, the Naming Service will have a binding to an object reference that is no longer valid (persistent object references, notwithstanding). You could kill the Naming Service or send a request to the Naming Service to unbind the reference.

Compare this with the commands needed to run the server and client without the Naming Service as given in 3.3.8. You no longer need to share files between the server and the client. Using the Naming Service makes it possible to manage a large number of objects without relying on the existence of a file system shared among the distributed objects in the system.

Discriminating between Multiple Naming Services

Suppose you want to access a particular Naming Service server when another Naming Service server is running and both are using multicast for service discovery. For example, there may be a *deployment* Naming Service running on your subnetwork, but you want to test your code using a *development* Naming Service without impacting users of the more stable deployment service.

You could use the `-ORBListenEndpoints` and `-ORBInitRef` options as described in 21.4, thus avoiding multicast discovery of the Naming Service. Or, you could specify a multicast port other than the default Naming Service multicast discovery port, as follows:

- Set the multicast port that the Naming Service server should use. For our example, we will use port 10999:

  ```
  $ tao_cosnaming -m 1 -ORBNameServicePort 10999
  ```

- Provide clients of the Naming Service the port to use for locating the Naming Service. Do this by setting the `NameServicePort` environment variable to 10999 before running applications that access the Naming Service (shown here using UNIX shell syntax):

  ```
  $ NameServicePort=10999; export NameServicePort
  ```

  or pass the port as a command line option:

  ```
  $ ./MessengerServer -ORBNameServicePort 10999 &
  ```

You may want to specify a multicast address and a port other than the default Naming Service values. Do this as follows:

- Set the multicast address and port for the Naming Service server to use. For our example, we will use address `224.0.0.3` and port `10999`:

  ```
  $ tao_cosnaming -m 1 -ORBMulticastDiscoveryEndpoint 224.0.0.3:10999
  ```

- Provide clients of the Naming Service the address and port to use for locating the Naming Service. Do this by passing the address and port as a command line option:

  ```
  $ ./MessengerServer -ORBMulticastDiscoveryEndpoint 224.0.0.3:10999
  ```

or

```
$ ./ MessengerServer -ORBInitRef \
                NameService=mcast://224.0.0.3:10999::/NameService
```

Another technique is to have the Naming Service server write the IOR of the root Naming Context to a file:

```
$ $TAO_ROOT/orbsvcs/Naming_Service/tao_cosnaming -o /tmp/ns.ior&

$ ./MessengerServer -ORBInitRef NameService=file:///tmp/ns.ior
```

22.3.3.2 Starting the Naming Service without Multicasting

Suppose you do not have the ability to use multicasting for your particular network configuration or you choose not to use multicasting for discovery of the Naming Service. The approach we use in this case is to specify an endpoint on which the Naming Service server's ORB will listen for CORBA requests, then specify the Naming Service's object reference to the applications using the `-ORBInitRef` option.

Run the Naming Service server in a separate session:

```
$ tao_cosnaming -ORBListenEndpoints iiop://tango:2809
```

The `-ORBListenEndpoints` option is passed to `CORBA::ORB_init()` and causes the Naming Service server's ORB to listen for requests on the specified endpoint. See 17.13.43 for more information on the `-ORBListenEndpoints` option.

Run `MessengerServer` and `MessengerClient` in another session. We specify the object reference of the Naming Service's root Naming Context by passing the `-ORBInitRef` option to each one:

```
$ ./MessengerServer \
  -ORBInitRef NameService=corbaloc:iiop:tango:2809/NameService &

$ ./MessengerClient \
  -ORBInitRef NameService=corbaloc:iiop:tango:2809/NameService
```

The -ORBInitRef option is passed to CORBA::ORB_init() and initializes the application's ORB with the association between the string NameService and the corbaloc Object URL provided as an argument. When the application invokes resolve_initial_references("NameService"), the Object URL is passed to CORBA::ORB::string_to_object() as described in 22.2. See 17.13.36 for more information on the -ORBInitRef option. See 22.4.1 for more information on using corbaloc Object URLs.

You can also specify the Naming Service IOR using the NameServiceIOR environment variable, as shown here using UNIX shell syntax:

```
$ NameServiceIOR=corbaloc:iiop:tango:2809/NameService; export NameServiceIOR
```

22.4 Object URLs

22.4.1 corbaloc URL

An application can find the Naming Service by passing a URL to CORBA::ORB::string_to_object() instead of using CORBA::ORB::resolve_initial_references().

The corbaloc URL syntax provides an easy way for users to manipulate a string form of an object reference. It has the syntax:

```
"corbaloc:"[<protocol id>]":"<protocol addr>
```

There are two protocols defined in the specification:

- The "resolve initial references" form:

  ```
  "corbaloc:rir:"[/<keystring>].
  ```

  For example: corbaloc:rir:/NameService.

- The IIOP form:

  ```
  "corbaloc:"["iiop"]":"<host>[":"<port>][/<keystring>].
  ```

  For example: corbaloc:iiop:rome.here.com:2204.

The default corbaloc protocol is IIOP with a default port of 2809, as defined in the CORBA Core specification. Therefore corbaloc::rome.here.com is equivalent to corbaloc:iiop:rome.here.com:2809.

The corbaloc syntax allows for future/proprietary protocols. TAO supports the proprietary protocols uiop and shmiop.

The corbaloc syntax may be used for the address specified in the options -ORBDefaultInitRef and -ORBInitRef, but not for -ORBListenEndpoints. The corbaloc:rir protocol can be used for neither -ORBDefaultInitRef nor -ORBInitRef because circular references would result.

The description given above is simplified, but covers the most frequently utilized parts of the corbaloc syntax. See 17.13.10 and 17.13.36 for more information on the protocols supported by TAO. See Part 2 of the CORBA 3.1 specification, 7.6.10.1, for the full description of the corbaloc URL syntax and semantics.

22.4.1.1 corbaloc Example Code

We have modified the NamingService/Messenger client source code to use string_to_object(). Full source code for this example is in the TAO source code distribution in the directory $TAO_ROOT/orbsvcs/DevGuideExamples/NamingService/corbaloc_Messenger. Only MessengerClient.cpp has changed. The MessengerClient now accepts a command line parameter that is a URL to be passed to string_to_object(). The only change is that the call to resolve_initial_references() has been replaced with:

```
char* url = "corbaloc:rir:/NameService"; // default URL to InitRef
if (argc < 2) {
  std::cout << "Defaulting URL to " << url << std::endl;
  std::cout << "Usage: " << argv[0]
            << " [-ORB options] [corbaloc URL for the name service]"
            << std::endl;
}
else {
  url = argv[1];
}

// Find the Naming Service
CORBA::Object_var naming_obj = orb->string_to_object(url);
```

The new `MessengerClient` will use a URL provided as a command line parameter to locate the Naming Service.

22.4.1.2 Running the corbaloc example

Here are some examples of using the `corbaloc` object URL with the updated `MessengerClient` example.

Start the Naming Service server using IIOP on the local machine:

```
tao_cosnaming -ORBListenEndpoints iiop://localhost:2809
```

Start the `MessageServer` using `-ORBDefaultInitRef` with a `corbaloc` URL (recall that the default port for `corbaloc` is `2809` and the default transport is IIOP):

```
MessengerServer -ORBDefaultInitRef corbaloc::localhost
```

Or start the `MessengerServer` using `-ORBInitRef` with a `corbaloc` URL:

```
MessengerServer -ORBInitRef NameService=corbaloc::localhost/NameService
```

Or specify a particular port:

```
MessengerServer -ORBInitRef NameService=corbaloc::localhost:2809/NameService
```

Now, start the `MessengerClient` using the URL parameter (default to port 2809):

```
MessengerClient corbaloc::localhost/NameService
```

Or start the `MessengerClient` using a `corbaloc:rir` form URL (you must specify the initial reference):

```
MessengerClient -ORBInitRef NameService=corbaloc::localhost:2809/NameService
corbaloc:rir:/NameService
```

22.4.2 corbaname

An application can find both the Naming Service and a name in the Naming Service by passing a `corbaname` URL to `string_to_object()`.

The `corbaname` URL syntax is

```
"corbaname:"<corbaloc>["#"<string_name>]
```

where `<corbaloc>` is the address of the Naming Service and `<string_name>` is the stringified name of the object to be found in the Naming Service. A stringified name is a string representation of the `CosNaming::Name` sequence with the syntax:

```
<string_name>    =  <name_component>["/"<name_component]*
<name_component> = <name>["."<type>]
```

Here are some examples:

```
corbaname:rir:#root/middle.my_type/leaf
corbaname::ns_node:9999#usa/arizona/tempe
```

The backslash ‘\’ character escapes the reserved meaning of ‘/’, ‘.’, and ‘\’ in a stringified name.

`corbaname` has an additional escaping requirement for the ‘\’ backslash character and other special characters[1]. If a character that requires escaping is present in a name component it is encoded as two hexadecimal digits following a ‘%’ character to represent the octet.

A `CosNaming::Name` “leaf/esc_slash” and type “leaf_type” under the name “root.esc_dot” would have the stringified name of:

```
root\.esc_dot/leaf\/esc_slash.leaf_type
```

And escaped for corbaname[2] thus:

```
root%5c.esc_dot/leaf%5c/esc_slash.leaf_type
```

See the `to_string()` and `to_url()` method descriptions in 22.5 for another example of escaping.

---

1. US-ASCII alphanumeric characters and the following: ";" | "/" | ":" | "?" | "@" | "&" | "=" | "+" | "$" | "," | "-" | "_" | "." | "!" | "~" | "*" | "’" | "("| ")" are not escaped. All others must be escaped.

2. common escapes: ’\’ => “%5c”, ’<’ => “%3c”, ’>’ => “%3e”, ’ ’ => “%20”, ’%’ => “%25”

22.4.2.1 corbaname Example Code

We have modified the `NamingService/Messenger` client source code to pass a `corbaname` URL to `string_to_object()`. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NamingService/corbaname_Messenger`. Only `MessengerClient.cpp` has changed. The `MessengerClient` now accepts a command line parameter that is a URL to be passed to `string_to_object()`. The only change is that the following code:

```
// Find the Naming Service
CORBA::Object_var naming_obj = orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var root =
  CosNaming::NamingContext::_narrow(naming_obj.in());
if (CORBA::is_nil(root.in())) {
  std::cerr << "Nil Naming Context reference" << std::endl;
  return 1;
}

// Resolve the Messenger object
CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("example");
name[1].id = CORBA::string_dup("Messenger");
CORBA::Object_var obj = root->resolve(name);
```

has been replaced with:

```
  char* url = "corbaname:rir:#example/Messenger"; // default URL to InitRef
  if (argc < 2) {
    std::cout << "Defaulting URL to " << url << std::endl;
    std::cout << "Usage: " << argv[0]
              << " [-ORB options] [corbaname URL for message server]"
              << std::endl;
  }
  else {
    url = argv[1];
  }

  // Resolve the Naming Service and the Messenger.
  CORBA::Object_var obj = orb->string_to_object(url);
```

The last statement first finds the Naming Service, then resolves the object reference of the Messenger using the name "`example/Messenger`" (relative to the root Naming Context).

22.4.2.2 Running the corbaname example

Start the Naming Service server and `MessengerServer` as described in 22.4.1.2.

Start the client with a `corbaname` that does not require the `-ORBDefaultInitRef` option:

```
MessengerClient corbaname:iiop:localhost:2809#example/Messenger
```

Or defaults to use the IIOP protocol:

```
MessengerClient corbaname::localhost:2809#example/Messenger
```

Or default protocol (IIOP) and default port (2809):

```
MessengerClient corbaname::localhost#example/Messenger
```

Or use the Resolve Initial Reference form (you must tell the ORB where the Naming Service is located with the `-ORBInitRef` option):

```
MessengerClient -ORBInitRef NameService=corbaloc::localhost:2809/NameService
corbaname:rir:#example/Messenger
```

22.5 The NamingContextExt Interface

The `NamingContextExt` interface is derived from `NamingContext` and defines operations to convert between `CosNaming::Name`s and stringified names. The interface is defined in `$TAO_ROOT/orbsvcs/orbsvcs/CosNaming.idl` as follows:

```
module CosNaming
{
// ... other interfaces ...

  interface NamingContextExt : NamingContext
  {
    typedef string StringName; // Stringified form of a Name.
    typedef string Address;  // URL<address> such as myhost.xyz.com.
    typedef string URLString; // Stringified form of a URL<address>.
    StringName to_string (in Name n) raises (InvalidName);
    Name to_name (in StringName sn)  raises (InvalidName);
    exception InvalidAddress {};
    URLString to_url (in Address addr,
```

```
                  in StringName sn)
      raises (InvalidAddress, InvalidName);
    Object resolve_str (in StringName n)
      raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
  };
};
```

22.5.1 NamingContextExt Operations

The operations of the `NamingContextExt` interface are:

- `to_string()`: Converts a `CosNaming::Name` to a stringified name. If the `Name` is invalid, an exception of type `CosNaming::NamingContext::InvalidName` is raised.
- `to_name()`: Converts a stringified name to a `CosNaming::Name`. If the stringified name is syntactically malformed or violates an implementation limit, an exception of type `CosNaming::NamingContext::InvalidName` is raised.
- `resolve_str()`: A convenience operation that resolves a name in the same manner as `resolve()`, but it accepts a stringified name as a parameter instead of a `CosNaming::Name`.
- `to_url()`: Converts a `corbaloc` URL `<address>/<key_string>` component and a stringified name to a fully formed `corbaloc` Object URL. Examples of the `<address>/<keystring>` parameter are:
  - `iiop:rome.phx.ociweb.com`
  - `:rome.phx.ociweb.com/a/b/c`
  - `shmiop:12345/a/b/c`

  `to_url()` performs any escapes necessary on the parameters and returns a fully formed URL string. An exception is raised if either the `corbaloc` address and key parameter or name parameter are malformed. It is illegal for the stringified name to be empty. If the address is empty, an exception of type `CosNaming::NamingContextExt::InvalidAddress` is raised.

22.5.2 Name Conversion Examples

Here are some examples using the `NamingContextExt` interface:

1. The following code shows the initialization of a `CosNaming::NamingContextExt` object and a call to the `to_string()` method:

```
// Find the Naming Service
CORBA::Object_var naming_obj =
  orb->resolve_initial_references("NameService");
CosNaming::NamingContextExt_var naming_context_ext =
  CosNaming::NamingContextExt::_narrow(naming_obj.in());
if (CORBA::is_nil(naming_context_ext.in())) {
  std::cerr << "Nil Naming Context reference" << std::endl;
  return 1;
}

CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("root.esc-dot");
name[0].kind = CORBA::string_dup("kind1");
name[1].id = CORBA::string_dup("leaf/esc-slash");
name[1].kind = CORBA::string_dup("kind2");

// Convert Name to String Name.
CORBA::String_var str_name = naming_context_ext->to_string(name);
std::cout << "str_name:  " << str_name.in() << std::endl;
```

   It produces the following output:

```
str_name root\.esc-dot.kind1/leaf\/esc-slash.kind2
```

   Notice that the “.” and “/” characters in the `Name` have been escaped.

2. The following code converts the `str_name()` above back into a `CosNaming::Name` object:

```
std::cout << "str_name:  " << str_name.in() << std::endl;

// Convert String Name to Name.
CosNaming::Name * tname = naming_context_ext->to_name(str_name);

std::cout << "converted back to a CosNaming::Name: " << std::endl;
std::cout << "   name[0] = " << (* tname)[0].id.in() << " , "
          << (* tname)[0].kind.in() << std::endl;
std::cout << "   name[1] = " << (* tname)[1].id.in() << " , "
          << (* tname)[1].kind.in() << std::endl;
```

   With the following output:

```
str_name:  root\.esc-dot.kind1/leaf\/esc-slash.kind2
```

```
converted back to a CosNaming::Name:
name[0] = root.esc-dot , kind1
name[1] = leaf/esc-slash , kind2resolve_str
```

3. The following code shows the `to_url()` method being used:

```
// Create a URL string for the application object.
CORBA::String_var address =
  CORBA::string_dup(":berlin.phx.ociweb.com:2809/key/str");

std::cout << "call to_url(\"" <<address.in() << "\"" << std::endl;
std::cout << "           ,\"" << str_name.in() << "\")"<< std::endl;

CORBA::String_var url_string =
  naming_context_ext->to_url (address.in(), str_name.in());

std::cout << "to_url result: " << url_string.in() << std::endl;
```

With these results:

```
call to_url(":berlin.phx.ociweb.com:2809/key/str"
           ,"root\.esc-dot.kind1/leaf\/esc-slash.kind2")
to_url result:
corbaname::berlin.phx.ociweb.com:2809/key/str#root%5c.esc-dot.kind1/leaf%5
c/esc-slash.kind2
```

22.6 TAO-Specific Naming Service Classes

22.6.1 Using the TAO_Naming_Client Class

TAO defines a `TAO_Naming_Client` class that simplifies the interface for accessing and using the Naming Service. The resulting code is now dependent on this TAO specific class, but the class would be trivial to port for use with other ORBs if portability is an issue.

The remainder of this section shows the changes necessary to convert the previous example to use the `TAO_Naming_Client` class. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NamingService/Naming_Client`.

To use the `TAO_Naming_Client`, you must add the following include directive:

```
#include <orbsvcs/Naming/Naming_Client.h>
```

Now instead of calling `resolve_initial_references()` and narrowing the resulting object reference, simply create a `TAO_Naming_Client` object and call `init()` on it.

```
// Find the Naming Service
TAO_Naming_Client naming_client;

if(naming_client.init (orb.in ()) != 0){
  std::cerr << "Could not initialize naming client." << std::endl;
  return 1;
}
```

The `init()` member function will return a non-zero value when it fails to locate the Naming Service and initialize properly. Now, the naming client can be used as a smart pointer to access the root Naming Context via `operator->()`. For example, in the `MessengerClient`:

```
CORBA::Object_var obj = naming_client->resolve(name);
```

Similarly, in the `MessengerServer`:

```
naming_client->rebind(name, messenger_obj.in());
```

22.6.2 Using the TAO_Naming_Server Class

When used with the `TAO_Naming_Server` class and threads, the Naming Service can easily be collocated within any application. This may be desirable for many reasons, for example

- When an embedded target system (e.g., VxWorks) does not support processes, requiring the Naming Service to run in a separate thread.
- For performance reasons you may want the Naming Service collocated to avoid network traffic.
- You may want the Naming Service collocated with other code that handles graceful shutdown.

The remainder of this section shows the changes necessary to convert the previous example to use the `TAO_Naming_Server` class. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NamingService/Naming_Server`.

22.6.3 Example using TAO_Naming Classes

The `ACE_Task_Base` base class, defined in `$ACE_ROOT/ace/Task.h`, is used in this example to create individual threads for running the Naming Service and a `MessengerServer` (as in the previous example).

22.6.3.1 Source Code

Each class overrides the base class function `svc()`, which is the entry point for both threads as they are activated. Both tasks create and initialize their own ORB, because calling `run()` on the same ORB has undesirable effects. The declaration of the Naming Service task is in the file `NamingTask.h`.

```
#include <ace/Task.h>

class NamingTask : public ACE_Task_Base
{
public:
  NamingTask (int argc, char* argv[]);
  virtual int svc();

private:
  int argc_;
  char** argv_;
};
```

The implementation of the Naming Service task is in the file `NamingTask.cpp`.

```
#include "NamingTask.h"
#include <orbsvcs/Naming/Naming_Server.h>

NamingTask::NamingTask (int argc, char* argv[])
: argc_ (argc),
  argv_ (argv)
{
}

int NamingTask::svc()
{
  int status = -1;

  try {
    // Initialize orb
    CORBA::ORB_var orb = CORBA::ORB_init(argc_, argv_, "NamingORB");

    // Get reference to Root POA
```

```
    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

    // Activate POA Manager
    PortableServer::POAManager_var poaManager = poa->the_POAManager();
    poaManager->activate();

    // Initialize the Naming Service
    // We are not going to look for other naming servers
    TAO_Naming_Server naming;
    if (naming.init(orb.in(),
                    poa.in(),
                    ACE_DEFAULT_MAP_SIZE,
                    0,
                    0) == 0) {
      std::cout << "The Naming Service Task is ready." << std::endl;

      // Accept requests
      orb->run();
      status = 0;
    }
    else {
      std::cerr << "Unable to initialize the Naming Service." << std::endl;
    }
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "CORBA exception: " << ex << std::endl;
  }

  return status;
}
```

The implementation of the Messenger server task, located in the file `MessengerTask.cpp`, looks very similar to the `main()` function of the Messenger Server in the previous example, so to avoid repetition, we will not list it here.

The `main()` function for this example, which uses the `NamingTask` and `MessengerTask`, is listed in the file `NamingMessenger.cpp`.

```
#include "NamingTask.h"
#include "MessengerTask.h"
#include <ace/OS.h>

int main(int argc, char* argv[])
{
  // Start the Naming Service task
  NamingTask namingService(argc, argv);
```

```
  namingService.activate();

  // Wait for it to initialize
  ACE_OS::sleep(5);

  // Start the Messenger task
  MessengerTask messenger;
  messenger.activate();

  // Wait for all tasks to complete
  namingService.thr_mgr()->wait();

  return 0;
}
```

22.6.3.2 Running the Example

Assuming that the `MessengerClient` (from the previous example) already has been built, you now are ready to run this example. Begin by starting the `NamingMessenger` binary. This will create the Naming Service thread and then the Messenger Server thread. After the `ready` statements for both are printed, run the `MessengerClient` to see the interaction between the Naming Service and the `MessengerServer`.

22.7 Naming Service Utilities

Several utilities for managing and debugging TAO applications and the Naming Service come with TAO. Source code for these utilities is located in `$TAO_ROOT/utils`. Executables for these utilities are placed in `$ACE_ROOT/bin`.

`tao_nslist` is useful for listing the current Naming Service bindings. `tao_nsadd` and `tao_nsdel` are limited to the root naming context (and are therefore of limited utility).

22.7.1 tao_nslist

By default, the `tao_nslist` utility produces a listing of all the elements in a Naming Service. For example, running it on one of the examples from this section results in the following:

```
$ tao_nslist
Naming Service:
```

```
---------------
+ example: Naming context
| + Messenger: Object Reference
| |   Protocol: IIOP
| |   Endpoint: 192.168.1.101:1027
```

This indicates that the root naming context contains a single node, which is a naming context bound to the name “example”. This naming context contains a single node with the name “Messenger”, which is bound to an object reference that supports IIOP and uses the listed endpoint. Table 22-1 lists the full set of command line options that `tao_nslist` supports.

Table 22-1 tao_nslist Command Line Options

Option	Description	Default
`--nsior`	Simply prints the IOR of the root naming context of the Naming Service and then exits. No list of the contents is printed.	The root naming context’s IOR is not printed.
`--ns ior`	IOR for locating the Naming Service.	Use the standard name service location algorithm.
`--ior`	Show the full IOR for objects that are not naming contexts.	A summary of the object (protocol & endpoint) is shown.
`--ctxior`	Show the full IOR for Naming Contexts	Only the context name is displayed.
`--tree "xx"`	Defines the character string used for indicating the scope of a node in the listing.	“|”
`--node "xx"`	Defines the character string used for indicating nodes in the listing.	“+”
`--noloops`	Inhibits drawing of naming context loops	
`--name name`	Produce the listing starting with the node corresponding to the supplied name.	Start with the root context.
`--ctxsep char`	Separation character between contexts. Used when parsing --name.	“/”
`--kindsep char`	Separation character between id and kind. Used when parsing --name.	“.”
`--max`	Maximum levels of the tree to list.	All levels are listed.
`--rtt seconds`	Round-trip timeout to specify for CORBA calls.	No time-outs are used.

22.7.2 tao_nsadd

The `tao_nsadd` utility is used to add a single binding relative to the root Naming Context. Here are some sample calls:

```
tao_nsadd --name example --ctx
tao_nsadd --name example/level2 --ctx
tao_nsadd --name example/Messenger --ior corbaloc:iiop:tango:1234/MyServer
```

Table 22-2 lists the full set of command line options that `tao_nsadd` supports.

Table 22-2 tao_nsadd Command Line Options

Option	Description	Default
`--name name`	Name (relative to the root context) that is to be bound.	Required.
`--ior`	IOR to bound to the specified name.	Either this option or --ctx are required.
`--ctx`	Bind a naming context to the specified name. If --ior is also specified, use it as the context. If --ior is not specified, create a new context.	Either this option or --ior are required.
`--ns ior`	IOR for locating the Naming Service.	Use the standard name service location algorithm.
`--rebind`	Bind the name regardless of whether it is already bound.	Binding an already bound name results in an exception.
`--ctxsep char`	Separation character between contexts. Used when parsing --name.	"/"
`--kindsep char`	Separation character between id and kind. Used when parsing --name.	"."
`--quiet`	Suppress any progress and status messages.	

22.7.3 tao_nsdel

The `tao_nsdel` utility is used to delete a single binding relative to the root Naming Context. Here are some sample calls that clean up the listing we saw in the `tao_nslist` section:

```
tao_nsdel --name example/Messenger
tao_nsdel --name example --destroy
```

Table 22-3 tao_nsdel Command Line Options

Option	Description	Default
`--name name`	Name (relative to the root context) that is to be deleted.	Required.
`--ns ior`	IOR for locating the Naming Service.	Use the standard name service location algorithm.
`--ctxsep char`	Separation character between contexts. Used when parsing --name.	“/”
`--kindsep char`	Separation character between id and kind. Used when parsing --name.	“.”
`--quiet`	Suppress any progress and status messages.	
`--destroy`	Destroy any name contexts that have their binding deleted.	Naming contexts are “orphaned”.
`--rtt seconds`	Round-trip timeout to specify for CORBA calls.	No time-outs are used.

Table 22-3 lists the full set of command line options that `tao_nsdel` supports.

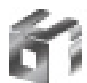
22.7.4 NamingViewer utility

`NamingViewer` is an MFC (Microsoft Foundation Classes) application for viewing and manipulating the bindings in the Naming Service. `wxNamingViewer` is similar to `NamingViewer`, but uses `wxWindows` (a cross-platform GUI toolkit) instead of MFC.

To use `NamingViewer`, you must select or add a Naming Service to display its tree as shown in Figure 22-1. Naming Services are added to your Windows registry so they are available the next time you start the `NamingViewer`. The Naming Service IOR should include the ending stringified object key `"/NameService"`, for example:

```
corbaloc::localhost:2809/NameService
```

Figure 22-2 shows the naming tree and Messenger binding after running the `NamingService/Messenger` example. Double clicking on an object/leaf in the tree will open a **`View IOR`** dialog (as shown). Clicking using the right-most mouse button on an object/leaf or context node will display a menu of



operations for that entry. Objects/leaves may be unbound or viewed.

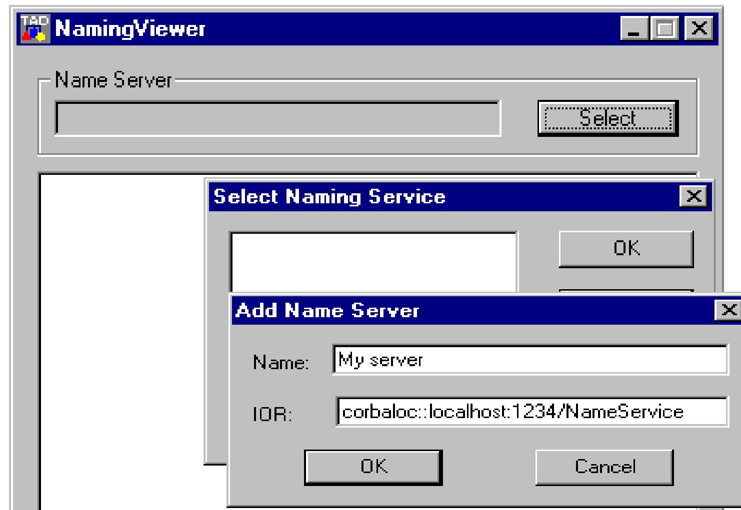


Figure 22-1 Select a Naming Service

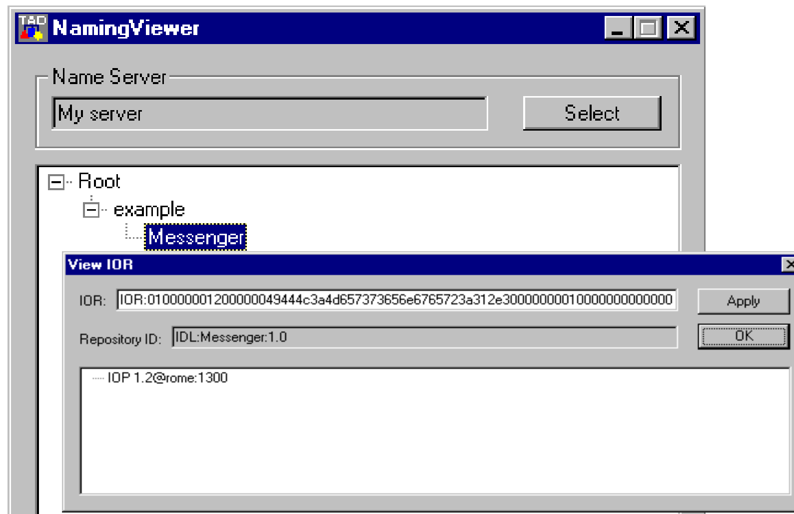


Figure 22-2 Display Name Service Entry

Operations associated with a context include: `bind`, `bind new context`, `unbind object`, `unbind`, `destroy`, `view reference` and `refresh`.

22.8 Naming Service Command Line Options

As stated previously, the TAO Naming Service server may be run as a stand-alone executable called `tao_cosnaming`, found in the path `$TAO_ROOT/orbsvcs/Naming_Service`. Table 22-4 lists command-line options for controlling the behavior of the TAO Naming Service server.

Table 22-4 tao_cosnaming Command Line Options

Option	Description	Default
`-b`	The address used for memory mapping the Naming Service state file supplied with the `-f` option. The value supplied with this option is only used when the FT Naming Service runs in persistent mode, i.e., `-f` option is present.	Platform dependent.
`-d`	Provides Naming Service specific debug information.	No diagnostics given.
`-f` *`persistence_file_name`*	Specify the name of a file in which to store the Naming Contexts and bindings so they can be read if the Naming Service needs to be restarted.	Do not store Naming Contexts and bindings.
`-m (`**`0`**`|1)`	Specify whether the Naming Service should listen for multicast requests. If `-m 0` is used, the Naming Service does not listen for multicast requests.	Naming Service does not listen for multicast requests.
`-n` *`count`*	Specify the number of threads to supply to ORB::run to make a multi threaded naming server	1, the naming service will be single threaded
`-o` *`ior-file-name`*	Specify the name of the output file for writing the IOR of the Naming Service as a string.	Do not write the IOR.
`-p` *`pid-file-name`*	Specify the name of the output file for writing the process ID as a string.	Do not write the process ID.
`-r` *`directory`*	Use *redundant* flat-file persistence; same as the `-u` option, except that more than one instance of the TAO Naming Service server can be run, each using the same set of disk files, to achieve a degree of fault tolerance (as long as *directory* is accessible to both servers).	No redundant Naming Service server persistence.
`-s` *`context-size`*	Specify the size of the hash table to allocate when creating Naming Contexts.	`1024`
`-t` *`listen-time`*	Specify how many seconds the server should listen for requests before exiting.	Listen indefinitely.

Table 22-4 tao_cosnaming Command Line Options

Option	Description	Default
`-u` *`directory`*	Use a flat-file persistence implementation that stores object reference information in a file per context. Each context file is placed in the *directory* specified.	Do not store Naming Contexts and bindings.
`-z` *`time`*	Specify a request/reply round trip timeout value that the Naming Service will use when invoking an operation on a federated naming context. On timeout, a `CosNaming::NamingContext::CannotProceed` exception is raised. The value of *time* is expressed in seconds.	A timeout policy is not used and an exception is not raised.

22.8.1 Using the Naming Service Persistence Options

There are three options that provide persistence of the relationships between names and object references: `-f`, `-u`, and `-r`. The `-f` option causes the Naming Service to use a memory-mapped file as its storage mechanism. This file is a copy of the internal memory used to store the naming graph. Thus, the `-s` option will have some effect on the size of the single file created.

The `-u` and `-r` options both use the same flat-file storage scheme. In this scheme, a separate file is created for each naming context in the naming graph. All of the data for a single naming context is stored in a single file, including both object references of bound objects and references to other naming contexts. These files are plain text and of variable sizes. No special provisions have been made for internationalization at this time.

The argument provided with both the `-u` and `-r` options is the name of the directory in which the persistence files are to be created. *The directory must already exist.* Two text files are initially created in this directory. The first file, called `NameService`, stores the name bindings in the root naming context. The second file, called `NameService_global`, stores a count of children naming contexts created from the root naming context. The initial value of this counter is platform dependent. For every new naming context created, the counter in `NameService_global` is incremented by one and a new text file called `NameService_`*`num`* is created. The value of *`num`* is the integer obtained by decrementing by one the counter stored in `NameService_global`.

The main difference between the `-u` and `-r` persistence options is that the `-r` option uses file locking while the `-u` option does not. The use of file locking

allows the files to be accessed safely by more than one Naming Service server running concurrently, however it has a small impact on performance. The flat-file persistence implementation (-u) does not acquire file locks.

The same base implementation is used for the flat-file (-u) and redundant (-r) persistence options. This implementation makes use of a PortableServer::ServantActivator that activates servants to read the naming context files only when they are used. This implementation was required for the redundant case because a naming context may have been created in one Naming Service server and referenced from another. To make this work, naming context references are stored as textual names rather than stringified IORs. Each textual name corresponds to the name of the file that stores the naming context's data. The object reference for a naming context is created dynamically within each Naming Service server when the naming context is accessed.

.The flat-file storage format consists of a PERSISTENT_HEADER followed by

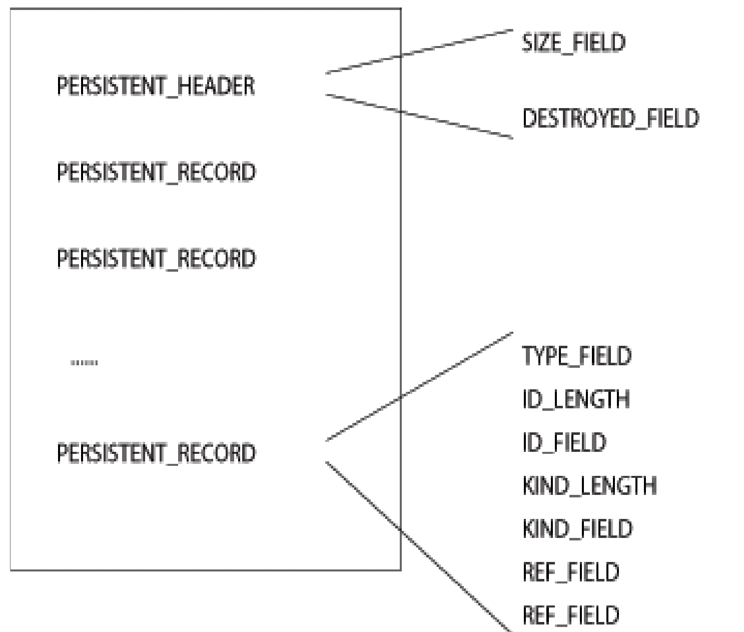


Figure 22-3 Persistent Flat-File Format

zero or more PERSISTENT_RECORDs as shown in Figure 22-3. The

PERSISTENT_HEADER contains information about the context, whereas each PERSISTENT_RECORD represents a name-to-reference binding. Table 22-5 describes the fields that make up a PERSISTENT_HEADER and a PERSISTENT_RECORD

Table 22-5 Flat-File Persistence Fields

Field Name	Description
SIZE_FIELD	An integer that represents the number of PERSISTENT_RECORDs in a naming context.
DESTROYED_FIELD	A flag set to 1 when the context is destroyed and 0 otherwise.
TYPE_FIELD	An integer that specifies what kind of reference is contained in the REF_FIELD. A value of 0 specifies that the field contains a reference to a naming context. A value of 1 specifies that the field contains a reference to a regular object.
ID_LENGTH	An integer that represents the length of the ID_FIELD.
ID_FIELD	A string that represents the id field of a CosNaming::NameComponent in a binding.
KIND_LENGTH	An integer that represents the length of the KIND_FIELD.
KIND_FIELD	A string that represents the kind field of a CosNaming::NameComponent in a binding.
REF_LENGTH	An integer that represents the length of the REF_FIELD.
REF_FIELD	A string that represents the object reference portion of a binding. If the binding is to a naming context, this field will be the name of file that stores that naming context's data. If the binding is to a regular object, this field will be a stringified IOR.

The redundant naming service is only *fully* functional on an HP Tru64 UNIX cluster, which was the target platform for its implementation. In a Tru64 cluster, multiple nodes share a single IP address; additional facilities within the cluster route requests to multiple redundant servers without outside intervention. Since naming context object references contain an IP address, the use of a single IP address for all the nodes of the cluster allows the naming context object references to be valid no matter which machine actually processes the request.

In the redundant naming service, the disk files become the single place where authoritative information can be found. This requires that each node that is running an instance of the Naming Service server have access to the same files

and that a locking mechanism must be available to protect files from simultaneous access. The Tru64 cluster provides these facilities.

Though the redundant Naming Service implementation was targeted for the Tru64 cluster environment, it can also be used on non-clustered platforms as long as certain restrictions are carefully observed:

1. There must be a shared location in which to store the flat files, and a file locking mechanism must be available. The NFS with a locking daemon satisfies this requirement on UNIX and UNIX-like systems. The SMB and built-in locking satisfy this requirement for Windows platforms.
2. The client must explicitly select one of the redundant Naming Service servers to use. Thus, a given client will generally use a single Naming Service server unless it fails. Then, the client must fail-over to a different server. The fail-over mechanism is not inherent to the redundant Naming Service and must be implemented within the client.
3. If a client switches to a different Naming Service server, all Naming Context object references it is holding will no longer be valid and the client must start over with the root Naming Context of the new Naming Service. Clients must be careful to use Naming Context object references only with the Naming Service server from which they were obtained.
4. `CosNaming::BindingIterator` object references, such as those returned from the `CosNaming::NamingContext::list()` operation, are not usable with the redundant Naming Service implementation. The `list()` operation can still be used, but only bindings in the returned `CosNaming::BindingList` are usable.

The performance of the Naming Service is impacted by any use of persistence. Memory-mapped persistence (the `-f` option) has the smallest impact; then non-redundant flat-file persistence (the `-u` option); redundant flat-file persistence (the `-r` option) has the largest impact on performance.

22.8.2 Example using the Naming Service Persistence Options

In this section, we present a simple example of using the Naming Service persistence options. We revisit the Naming Service example presented in 22.3. For this example, we define a new environment variable called `NAMING_DIRECTORY` as follows:

```
NAMING_DIRECTORY=$TAO_ROOT/orbsvcs/DevGuideExamples/NamingService/Messenger
export NAMING_DIRECTORY
```

Start the Naming Service as in 22.3.3.2. Specify either the `-u` or `-r` persistence option, for example:

```
$ tao_cosnaming -ORBListenEndpoints iiop://tango:2809 -u $NAMING_DIRECTORY
```

Next, run the `MessengerServer` and `MessengerClient` as in 22.3.3.2. You should find that the following files are created in the directory specified by `$NAMING_DIRECTORY`:

```
NameService  NameService_0  NameService_global
```

The contents the `NameService` file below indicate that the root context has not been destroyed and that it contains a single binding. The `id` field of the binding is “`example`” and the binding is for a naming context, the contents of which are stored in a file named `NameService_0`.

```
1
0
0
7
example
0

13
NameService_0
```

Similarly, the contents of the `NameService_0` file below indicate that this naming context has not been destroyed and contains a single name binding. The `id` field of the binding is "`Messenger`" and the binding is for a regular object. The stringified object reference is listed as well (it may be different on your system).

```
1
0
1
9
Messenger
0

118
IOR:010000001200049444c3a4d655737374a31312e300000000010000000000000007000000001
010200130000006f6369313332392e6f63697765622e6f6d
```

Since we started the Naming Service using the `-ORBListenEndpoints` option described in 17.13.43, if we later need to restart the Naming Service server, we *must* restart it on the same endpoint(s) as before.

22.9 Fault Tolerant Naming Service

The Fault Tolerant Naming Service provides a dual redundant scheme for supporting fault tolerance for the Naming Service. While it supports all of the same interfaces as the Naming Service, it also supports replication between two Naming Service processes. This is achieved by exporting a multi-profile IOR which can be used by clients to access the Fault Tolerant Naming Service through a single object reference with seamless failover between the Naming Service processes without client intervention.

Another major feature added to the Fault Tolerant Naming Service is support for load balancing through the use of a `NamingManager` interface which extends the OMG's `PortableGroup::ObjectGroupManager` interface. The `NamingManager` interface supports the basic creation of Object Groups and the assignment of a Load Balancing strategy for created Object Groups. The `ObjecGroupManager` interface defines operations for managing the membership of objects in Object Groups, as well as querying object groups. The Load Balancing functionality for the Fault Tolerant Naming Service is provided through a combination of binding the Object Group to a name within the Naming Service and resolving the Object Group from the Naming Service. An example of this is shown in Figure 22-4.

When a client then resolves a Name that is bound to an Object Group, one of the Object Group members is returned according to the scheme supported by the Object Group Load Balancing Strategy.

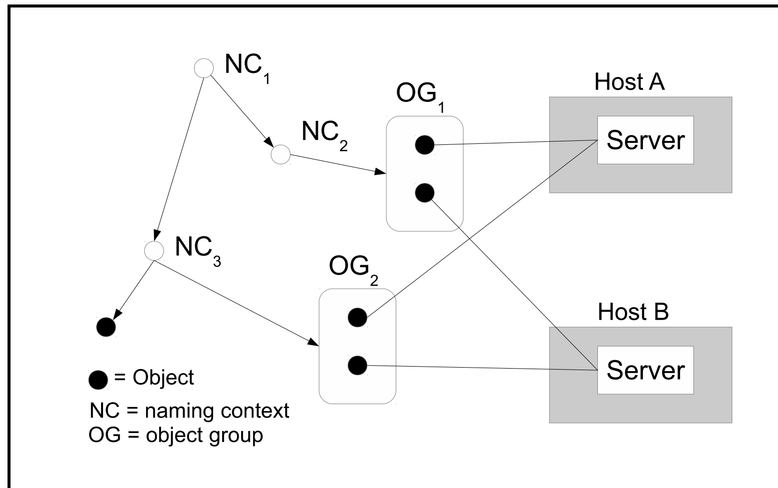


Figure 22-4An example of Naming Contexts referencing Object Groups with member instances on different hosts

22.9.1 Command line options

Many of the command line options used for the Fault Tolerant Naming Service are the same as those for the standard naming service shown in Table 22-4. Options specific to the Fault Tolerant Naming Service are shown in **bold**.

Table 22-6 tao_ft_naming Command Line Options

Option	Description	Default
`--primary`	**The service takes on the role as the primary service**	**Service is neither primary nor backup.**
`--backup`	**The service takes on the role as the backup service**	**Service is neither primary nor backup.**
`-b`	The address used for memory mapping the Naming Service state file supplied with the `-f` option. The value supplied with this option is only used when the FT Naming Service runs in persistent mode, i.e., `-f` option is present.	Platform dependent.
`-d`	Provides Naming Service specific debug information.	No diagnostics given.

Table 22-6 tao_ft_naming Command Line Options

Option	Description	Default
`-f` *`persistence_file_name`*	Specify the name of a file in which to store the Naming Contexts and bindings so they can be read if the Naming Service needs to be restarted. **This option is not compatible with fault tolerant usage.**	Do not store Naming Contexts and bindings.
`-l <Round | Random>`	**Specify a global load balancing policy overriding any per-object group policy.**	**Do not set a global policy.**
`-m (0|1)`	Specify whether the Naming Service should listen for multicast requests. If `-m 0` is used, the Naming Service does not listen for multicast requests.	Naming Service does not listen for multicast requests.
`-n` *`count`*	Specify the number of threads to supply to ORB::run to make a multi threaded naming server	1, the naming service will be single threaded
`-c` ***`ior-file-name`***	**Specify the name of the output file for writing the multi-profile IOR of the Naming Service as a string.**	**Do not write the IOR.**
`-o` *`ior-file-name`*	Specify the name of the output file for writing the IOR of the Naming Service as a string.	Do not write the IOR.
`-g` ***`ior-file-name`***	**Specify the name of the output file for writing the multi-profile IOR of the Naming Manager as a string.**	**Do not write the IOR.**
`-p` *`pid-file-name`*	Specify the name of the output file for writing the process ID as a string.	Do not write the process ID.
`-r` ***`directory`***	**Specify the directory to use for *redundant* flat-file persistence of naming contexts. This is required if server has role of primary or backup server.**	**Naming context persistence will not be used.**
`-v` ***`directory`***	**Specify the directory to use for object group flat-file persistence.**	**This is required.**
`-s` *`context-size`*	Specify the size of the hash table to allocate when creating Naming Contexts.	`1024`
`-t` *`listen-time`*	Specify how many seconds the server should listen for requests before exiting.	Listen indefinitely.

Table 22-6 tao_ft_naming Command Line Options

Option	Description	Default
`-z` *`time`*	Specify a request/reply round trip timeout value that the Naming Service will use when invoking an operation on a federated naming context. On timeout, a `CosNaming::NamingContext::CannotProceed` exception is raised. The value of *time* is expressed in seconds.	A timeout policy is not used and an exception is not raised.

22.9.2 Persistence

When running as a fault tolerant service, the TAO FT Naming Service persists the state of naming contexts and object groups to flat files. In this way, if the process fails it can be restarted with its state restored to what it was before the failure. Furthermore, backup files are created for persistence files that were successfully written. If later a corrupt file is created (this could happen, for example, if the server or host crashed in the middle of writing to the file), this will be detected during the subsequent reading of the file and the backup will be used instead.

If the specified persistence directories for naming context data and object group data do not exist, the server will exit.

If the specified file exists, it is scanned and:

- If a field can not be properly read because the data is corrupt, an attempt will be made to read from the backup of the file. If the backup file can be read it will replace the primary file. Because the state of the naming context or object group reverted to an earlier state, a log message is generated alerting that the backup is being used.
- If the file is recognized and is ok, the state stored in the file becomes the current state of the Naming Service.

22.9.3 Starting the FT Naming Service

In order to achieve the fault tolerance characteristics of the Fault Tolerant Naming Service you must start a primary Naming Server and a backup Naming Server and provide it with a common location in which the two servers can share their replicated state for Naming Contexts and Object Groups. To establish the dual redundant pairing between the primary in the

backup they must be started one after the other. You can then start your application servers and create the object groups. The final step is to supply the multi-profile IOR to the clients so they will use the Fault Tolerant Naming Servers seamlessly.

1) Start the primary Naming Service process. It will use the *name_service_persistence_dir* to store the shared state information for the replicated naming service processes. We assume this directory will be an NFS mounted location that will be shared between the primary and the backup process. The primary will write ns_replica_primary.ior to the directory identified in the *name_service_persistence_dir* which is also where all of the persistent state will be shared by the primary and backup servers. It will also use *object_group_persistence_dir* to persist the state of the object groups. This directory does not need to be the same as the *name_service_persistence_dir*.

```
$ $TAO_ROOT/orbsvcs/Naming_Service/tao_ft_naming \
          --primary                              \
          -r <name_service_persistence_dir>      \
          -v <object_group_persistence_dir>
```

2) Start the backup Naming Service process. It must be started after the primary has started. It reads the primary ior (ns_replica_primary.ior) from the persistence directory to bootstrap communications with the primary. When this file is available, the backup can be started as described below. This process will write the multi-profile ior for the NameService to *naming_ior_filename*. The fault tolerant naming service will be available when the backup has started and has exported the *naming_ior_filename*. This process will also write out the `NamingManager` ior file to *naming_manager_ior_filename* and this object reference provides an interface for creating and managing object groups in support of load balancing.

```
$  $TAO_ROOT/orbsvcs/Naming_Service/tao_ft_naming  \
          --backup                                 \
          -r <name_service_persistence_dir>        \
          -v <object_group_persistence_dir>        \
          -c <naming_ior_filename>                 \
          -g <naming_manager_ior_filename>
```

If either the primary of the backup processes fail separately, they can be restarted using the same arguments they were started with previously. This

will cause them to contact their peer replica and restore the replicated naming service relationship. The multi-profile iors which provide the automatic failover between primary and backup are persistent/user IORs and will not change even if the primary or backup are shutdown and restarted. One of these two servers must be running at all times to ensure access to the bound objects in the namespace.

In a case where both the primary and backup are shut down, then either the backup or primary may be restarted independently, reusing the same naming and object group persistent state and IORs which are set up as part of the initial startup process. If you intend to start up the primary or backup on a different host or port, then you will need to use the restart process as described above (by starting the primary first). If there is no change in the primary or backup, the then the IORs can continue to be used by any of the existing `tao_ft_naming` clients. Additionally, if you wish to clear the bound names and object groups persisted by the FT Naming Service, you must clean out the naming service and object group persistent directories that are passed to `tao_ft_naming` using the `-u` and `-v` options respectively.

Under very heavy loads of 100s to 1000s of resolve requests per second, the time required to locate the load balancing strategy bound to a specific group becomes a substantial part of the request processing. In order to alleviate that overhead, a global load balancing strategy may be specified, overriding the policy supplied for individual object groups. Use the `tao_ft_naming` command line option `-l` with argument `round` to select round-robin load balancing or `random` to select random load balancing.

22.9.4 Start Client/Server

Start a client or server, providing it with the NameService ior written out to the filename provided by the backup naming service process using the argument `-c` *`naming_ior_filename`*. The client or server will use the load balancing feature of the naming service invisibly, with the naming service using the load balancing policy (either round-robin or random) to load balancing between all of the services which are added as members to the group inserted in the object group.

```
$ server -ORBInitRef NameService=file://<naming_ior_filename>  \
        -o <server_a_ior_file>

$ server -ORBInitRef NameService=file://<naming_ior_filename>  \
```

```
-o <server_b_ior_file>
```

22.9.5 Creating and Managing Object Groups

The `tao_nsgroup` command line utility discussed in detail in 22.9.7 can be used to manage the object groups. A few examples are shown below.

Note *You do not have to use the command line to manage object groups. The operations can be performed programmatically as well using the `FT_Naming::NamingManager` interface defined in the `FT_NamingManager.idl` file in `$TAO_ROOT/orbsvcs/orbsvcs`.*

Create an object group for servers to be load balanced:

```
$ $TAO_ROOT/utils/nsgroup/tao_nsgroup
          -ORBInitRef NameService=file://<naming_ior_filename>           \
          -ORBInitRef NamingManager=file://<naming_manager_ior_filename> \
          group_create -group <server_group>                             \
                     -policy round
```

Add members to the group:

```
$ $ACE_ROOT/bin/tao_nsgroup
          -ORBInitRef NameService=file://<naming_ior_filename>           \
          -ORBInitRef NamingManager=file://<naming_manager_ior_filename> \
          member_add -group <server_group>                               \
                               -location <server_a_loc>                  \
                               -ior file://<server_a_ior_file>

$ $ACE_ROOT/bin/tao_nsgroup
          -ORBInitRef NameService=file://<naming_ior_filename>           \
          -ORBInitRef NamingManager=file://<naming_manager_ior_filename> \
          member_add -group <server_group>                               \
                               -location <server_b_loc>                  \
                               -ior file://<server_b_ior_file>
```

Bind the group in the naming service:

```
$ $ACE_ROOT/bin/tao_nsgroup                                              \
         -ORBInitRef NameService=file://<naming_ior_filename>            \
         -ORBInitRef NamingManager=file://<naming_manager_ior_filename>  \
         group_bind -group <server_group>                                \
                            -name <compound_name>
```

Note *It is assumed that the Fault Tolerant Naming Service processes will be monitored and restarted if they fail in order to continue to provide a dual-redundant pair following the intentional or unplanned loss of one of the naming servers.*

22.9.6 Limitations

The Fault Tolerant Naming Service does not provide support for `BindingIterators`. With the use of redundancy between a pair of naming servers, there is no way to guarantee that the context structure being iterated on would remain consistent, so the user must ensure that when invoking the list operation that a `how_many` value must be provided that is sufficiently large to hold all returned bindings in a `BindingList` with no `BindingIterator` being needed. If the `how_many` parameter is insufficiently large, the FT Naming Service will throw a `CORBA::NO_IMPLEMENT` exception.

Similar to the CosNaming Service, this implementation of the Naming Service does not include any form of 'garbage collection' for orphaned naming contexts. It is solely the responsibility of clients to clean up after themselves and not leak server resources. All the resources, including orphaned contexts, are released during the Naming Server shutdown.

22.9.7 Group Management Utility

A separate utility, `tao_nsgroup`, is provided which supports a command line interface for managing the object groups within the Fault Tolerant Naming Service. It accesses the `tao_ft_naming` service through the `NameService` and `NamingManager` interfaces.

Below is a summary of the commands and options. Further details and examples follow.

Table 22-7 tao_ns_group command line options

Command	Option	Description
group_create	-group *group_name*	The name of the object group to add.
	-policy *policy_type*	The load balancing policy to apply for this group. This may be `round` for round-robin or `random` for random.
group_remove	-group *group_name*	The name of the group to remove.
group_bind	-group *group_name*	The object group to bind to a name in the naming service.
	-name *name*	The stringified name in the naming service to bind to a object group.
group_unbind	-name *name*	The stringified name in the naming service to unbind.
group_list		Lists the object groups available.
member_add	-group *group_name*	The object group to add a member to.
	-location *location*	An arbitrary string to identify this member within the group.
	-ior IOR	The unique IOR of the member.
member_remove	-group *group_name*	The object group of the member to removed.
	-location *location*	The identifier for the member.
member_show	-group *group_name*	The object group of the member to show.
	-location *location*	The identifier for the member.
member_list	-group *group_name*	The object group of the members to list.

Table 22-7 tao_ns_group command line options

Command	Option	Description
`help`		List the commands/options available.

Note *The load balancing policy may be round-robin or random. The policy type is required when creating groups, even though it may be overridden by the tao_ft_naming command line selection of a global policy.*

22.9.7.1 group_create

This adds the object group to the load balancing naming manager service with the specified selection policy. On creation, an object group contains no member objects. Returns an error if `group_name` is not unique.

Example:

```
$ $ACE_ROOT/bin/tao_nsgroup group_create -group ieee -policy round \
        -ORBInitRef NameService=file://ns.ior \
        -ORBInitRef NamingManager=file://nm.ior
```

22.9.7.2 group_remove

Removes the specified object group from the load balancing naming manager service.

Example:

```
$ $ACE_ROOT/bin/tao_nsgroup group_remove -group ieee \
            -ORBInitRef NameService=file://ns.ior \
            -ORBInitRefNamingManager=file://nm.ior
```

Note *If the object group is bound with `group_bind`, you must first unbind it with `group_unbind`.*

22.9.7.3 group_bind

Binds the specified object group to the specified stringified name in the naming service. When clients resolve that name, they transparently obtain a member of the specified object group from the load balancing naming manager service.

Example:

```
$ $ACE_ROOT/bin/tao_nsgroup group_bind -group ieee -name iso/ieee \
          -ORBInitRef NameService=file://ns.ior \
          -ORBInitRef NamingManager=file://nm.ior
```

Note *The iso context used in this example must be created before calling* `group_bind`*.*

22.9.7.4 group_unbind

Unbinds the specified stringified name from the naming service, but does not remove the object group.

Example:

```
$ $ACE_ROOT/bin/tao_nsgroup group_unbind -name iso/ieee \
        -ORBInitRef NameService=file://ns.ior \
        -ORBInitRef NamingManager=file://nm.ior
```

22.9.7.5 group_list

Displays all object groups that currently exist in the load balancing naming manager service by load balancing policy.

Example:

```
% $ACE_ROOT/bin/tao_nsgroup group_list \
         -ORBInitRef NameService=file://ns.ior \
         -ORBInitRef NamingManager=file://nm.ior
```

Round Robin Load Balancing Groups:

ieee

22.9.7.6 member_add

Adds an member object to the specified object group. After being added, the member object is available for selection. Returns error if the ior is not unique for the specified object group.

Example:

```
% $ACE_ROOT/bin/tao_nsgroup member_add -group ieee -location local \
            -ior file://mo.ior \
            -ORBInitRef NameService=file://ns.ior \
            -ORBInitRef NamingManager=file://nm.ior
```

22.9.7.7 member_remove

Removes the specified member object location from the specified object group.

Example:

```
$ $ACE_ROOT/bin/tao_nsgroup member_remove -group ieee -location local \
           -ORBInitRef NameService=file://ns.ior \
           -ORBInitRef NamingManager=file://nm.ior
```

22.9.7.8 member_list

Lists the member locations of the specified object group.

Example:

```
$ $ACE_ROOT/bin/tao_nsgroup member_list -group ieee \
         -ORBInitRef NameService=file://ns.ior -ORBInitRef
NamingManager=file://nm.ior

local
```

22.9.7.9 member_show

Displays the object reference that corresponds to the specified member location of an object group.

Example:

```
% $ACE_ROOT/bin/tao_nsgroup member_show -group ieee -location local \
```

```
      -ORBInitRef NameService=file://ns.ior -ORBInitRef
NamingManager=file://nm.ior
```


```
IOR:010000002100000049444c3a6f6d672e6f72672f46542f4e616d696e674d616e61676572
3a312e300000000001000000000000006c000000010102000e00000031302e3230312e323030
2e363400e1841b00000014010f005253541571a65076c60a0000000000010000000100000000
020000000000000008000000010000000004f4154010000001800000001000000010001000100
0000010001050901010000000000
```


22.9.7.10 Troubleshooting

Q1. Why do I get an error Message “Invalid persistence directory” or “Invalid object group persistence directory”?

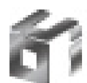
A1. On starting, the error message “Invalid persistence directory” indicates that the supplied value for the -r option does not point to a directory that can be used to store the state of the naming contexts or object groups. Make sure that the provided directories exist and that they are write enabled.

22.10 Using the NT Naming Service

The name is a little archaic, but the TAO CosNaming service has hooks that allow it to be registered with the Windows Service Manager, allowing it to be started automatically on system startup. The NT Naming Service is built using a project that is separate from the CosNaming project although when using MPC, is generally included in the same solution. Building the NT Naming Service yields an executable that serves as a simple command line interface for installing and configuring the service as well as providing the necessary entry point required by the Windows Service Manager.

The executable for the command line interface is `tao_nt_cosnaming` and it takes a variety of options as shown in Table 22-8 to manipulate the service.

Note *The commands that change state, such as inserting the service or setting runtime configuration must be run as Administrator. The easiest way to achive that is to run a command window (cmd.exe) using the “run as Administrator” menu option.*



Continuing on...

Table 22-8 tao_nt_cosnaming command line options

option	description
-c *args*	
-i *n*	
-k	
-r	
-s	
-t *n*	

CHAPTER 23

Event Service

23.1 Introduction

The OMG Event Service version 1.2 specification (OMG Document formal/04-10-02) defines a service for decoupling the *suppliers* of events from *consumers* of those events. This decoupled approach provides a much more appropriate communication model for many applications than the typical request/reply semantics of CORBA object operation invocations. The Event Service defines basic interfaces for suppliers and consumers of events and defines the concept of an *event channel* to provide for decoupling consumers from suppliers and propagating events.

This chapter discusses TAO's support for the Event Service as well as how to use, extend, and embed the Event Service in your applications. TAO also defines two extensions to the Event Service. The Real-Time Event Service (see Chapter 24) extends the OMG's specification for applications with stringent Quality of Service requirements. The OMG Notification Service (see Chapter 25) is a more recent specification than the Event Service, and extends the basic Event Service functionality with features such as event filtering, structured event types, and Quality of Service properties.

23.2 Overview of the Event Service

The Event Service is based on a *publish* and *subscribe* paradigm, where *suppliers* publish events and *consumers* receive events for which they have subscribed. An *event channel* provides a mechanism that decouples suppliers from consumers. A supplier publishes events via the event channel, and a consumer subscribes to them through the event channel. Suppliers are not directly aware of the presence of consumers (nor other suppliers). Similarly, consumers are not directly aware of the presence of suppliers (nor other consumers).

Figure 23-1 shows the relationships among suppliers, consumers, and an event channel, and illustrates typical usage of an event channel to distribute events.

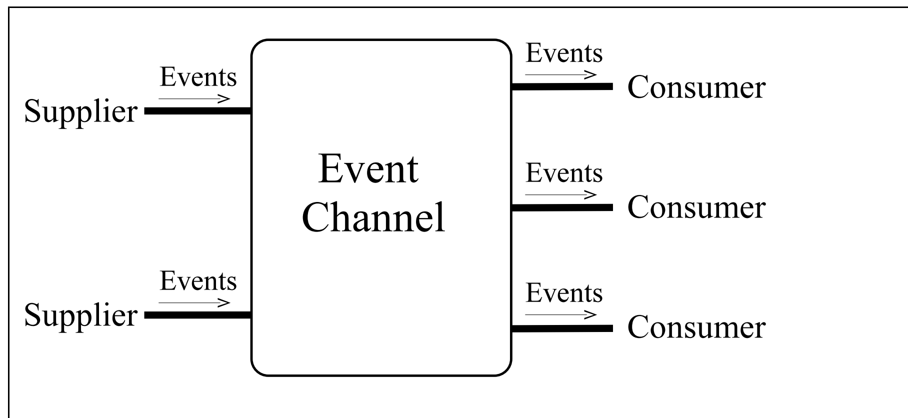


Figure 23-1 Typical Event Channel Usage

The Event Service specification supports *push* and *pull* style event distribution models. In the push model, suppliers push events to the event channel, and the event channel pushes them to all subscribed consumers. In the pull model, consumers pull events from the event channel, and the event channel pulls events from suppliers. The Event Service specification makes it possible to mix the push and pull models in different combinations, even within the same event channel.

The Event Service specification defines two different varieties of event channels, *typed* and *untyped*. Untyped event channels use a `CORBA::Any` type to represent events. This allows suppliers to supply arbitrary event types.

Typed event channels use application-defined IDL to define the event communications and constrain the data types passed.

See Chapter 20 of *Advanced CORBA Programming with C++* for an in-depth discussion of the Event Service and the different distribution models.

23.3 TAO's Event Channel Implementation

TAO provides support for untyped event channels via the `CosEventChannelAdmin::EventChannel` interface. This implementation supports untyped events using `CORBA::Any` parameters with both the push and pull models of event delivery. See 23.4.1 for an example that uses an untyped event channel.

TAO also provides partial support for typed event channels via the `CosTypedEventChannelAdmin::TypedEventChannel` interface. This implementation supports typed event delivery using a push model. The pull model is not currently supported for typed event channels. See 23.4.4 for an example that uses a typed event channel.

TAO does not support the Lightweight Event Service described in Chapter 3 of the Event Service specification.

The `tao_cosevent` server can be used to start both typed and untyped event channels (see 23.5 for details). Developers can directly create or derive their own untyped event channels using the `TAO_CEC_EventChannel` servant class (see 23.4.2).

23.4 How to Use the Event Service

The examples in this chapter restrict themselves to the basic capabilities and interfaces of the Event Service. TAO-specific steps are noted as such. Because these examples use the Naming Service to locate the event channel, ensure that it is running as described in Chapter 22.

23.4.1 A Basic Example

This example shows how to create an event channel, connect suppliers and consumers to it, supply events to it, and consume events from it. It uses push suppliers and consumers. Full source code for this example is in the TAO

source code distribution in the directory
`$TAO_ROOT/orbsvcs/DevGuideExamples/EventServices/OMG_Basic`.

23.4.1.1 Starting the tao_cosevent Server

The `tao_cosevent` server that is delivered with TAO can be used with this example. This server creates a single event channel object and binds it in the root naming context of the Naming Service. By default, the event channel is bound to the name "`CosEventService`." The options that can be passed to this server are described in 23.5 and 23.6. A typical invocation of the server is:

```
$TAO_ROOT/orbsvcs/CosEvent_Service/tao_cosevent
```

This assumes that we are using either multicast discovery or the `NameServiceIOR` environment variable to locate the Naming Service.

23.4.1.2 Creating and Initializing a Supplier and Pushing Events

To implement a supplier, insert the following `#include` directives in your source code (in `EchoEventSupplierMain.cpp`):

```
#include <orbsvcs/CosEventCommC.h>
#include <orbsvcs/CosEventChannelAdminC.h>
#include <orbsvcs/CosNamingC.h>
#include <iostream>
```

The supplier must first initialize the ORB, then connect to the event channel. Since the event channel object is now bound in the Naming Service, we need to get the root naming context and use the `resolve_str()` operation to obtain a proxy for the event channel.

```
int main (int argc, char* argv[])
{
  try
  {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Find the Naming Service.
    CORBA::Object_var obj = orb->resolve_initial_references("NameService");
    CosNaming::NamingContextExt_var root_context =
       CosNaming::NamingContextExt::_narrow(obj.in());

    // Find the EventChannel.
```

```
obj = root_context->resolve_str("CosEventService");

// Narrow the object reference to an EventChannel reference.
CosEventChannelAdmin::EventChannel_var echoEC =
  CosEventChannelAdmin::EventChannel::_narrow(obj.in());
if (CORBA::is_nil(echoEC.in())) {
  std::cerr << "Could not resolve EchoEventChannel." << std::endl;
  return 1;
}
```

Once the event channel is located, it is used to obtain a proxy to a push consumer. The proxy push consumer is then used to connect to the event channel.

```
// Get a SupplierAdmin object from the EventChannel.
CosEventChannelAdmin::SupplierAdmin_var supplierAdmin =
  echoEC->for_suppliers();

// Get a ProxyPushConsumer from the SupplierAdmin.
CosEventChannelAdmin::ProxyPushConsumer_var consumer =
  supplierAdmin->obtain_push_consumer();

// Connect to the ProxyPushConsumer as a PushSupplier
// (passing a nil PushSupplier object reference to it because
// we don't care to be notified about disconnects).
consumer->connect_push_supplier(CosEventComm::PushSupplier::_nil());
```

The `connect_push_supplier()` operation takes a reference to a push supplier as a parameter. The only operation defined on the push supplier interface is `disconnect_push_supplier()`. It is called when the supplier is disconnected from the event channel. Passing a null object reference, as shown above, means the event channel does not need to notify this supplier upon disconnection. For our simple example, we do not need to implement the `PushSupplier` interface. We show how this could be done in 23.4.1.3.

We are now ready to create and publish events using the consumer proxy's `push()` operation.

```
// Create an event (just a string in this case).
CORBA::String_var eventData = CORBA::string_dup("Hello, world.");

// Insert the event data into an any.
CORBA::Any any;
any <<= eventData;

// Now push the event to the consumer
```

```
    consumer->push(any);
  }
  catch ( ... ){
    return 1;
  }
}
```

Our example passes a simple string as the event data. However, since the event is passed as a `CORBA::Any`, virtually *any* IDL type (including user-defined types) can be used as event types.

23.4.1.3 Implementing the Push Supplier Interface

In our simple example, we have chosen to make the supplier a *pure* client of the CosEvent service. However, we could also implement the `CosEventComm::PushSupplier` interface to allow our supplier to receive `disconnect_push_supplier()` callbacks from the CosEvent service. In practice, suppliers are rarely pure clients; they are often middle-tier processes that receive data directly from one or more "raw" data sources and publish the data as events for further processing or display by downstream consumers.

Here is an example `PushSupplier` implementation class definition:

```
#include <orbsvcs/CosEventCommS.h>    // for POA_CosEventComm::PushSupplier

class EchoEventSupplier_i : public virtual POA_CosEventComm::PushSupplier
{
  public:
    // Constructor
    EchoEventSupplier_i(CORBA::ORB_ptr orb);

    // Override operations from PushSupplier interface.
    virtual void disconnect_push_supplier();

  private:
    CORBA::ORB_var orb_;
};
```

Here are the implementations of the constructor and the `disconnect_push_supplier()` operation:

```
// Constructor duplicates the ORB reference.
EchoEventSupplier_i::EchoEventSupplier_i(CORBA::ORB_ptr orb)
  : orb_(CORBA::ORB::_duplicate(orb))
{ }
```

```
// Override the disconnect_push_supplier() operation.
void EchoEventSupplier_i::disconnect_push_supplier()
{
  // Deactivate this object.
  CORBA::Object_var obj = orb_->resolve_initial_references("POACurrent");
  PortableServer::Current_var current =
    PortableServer::Current::_narrow(obj.in());
  PortableServer::POA_var poa = current->get_POA();
  PortableServer::ObjectId_var objectId = current->get_object_id();
  poa->deactivate_object(objectId.in());
}
```

23.4.1.4 Implementing the Push Consumer Interface

To create a consumer, the `CosEventComm::PushConsumer` interface must be implemented. The `PushConsumer` interface defines two operations, `push()` and `disconnect_push_consumer()`, that you must implement. Here is an example `PushConsumer` implementation (from `EchoEventConsumer_i.h`) class definition:

```
#include <orbsvcs/CosEventCommS.h>    // for POA_CosEventComm::PushConsumer

class EchoEventConsumer_i : public virtual POA_CosEventComm::PushConsumer
{
  public:
    // Constructor
    EchoEventConsumer_i(CORBA::ORB_ptr orb);

    // Override operations from PushConsumer interface.
    virtual void push(const CORBA::Any& data);

    virtual void disconnect_push_consumer();

  private:
    CORBA::ORB_var orb_;
};
```

Here are the implementations of the constructor, the `push()` operation and the `disconnect_push_consumer()` operation (from `EchoEventConsumer_i.cpp`):

```
// Constructor duplicates the ORB reference.
EchoEventConsumer_i::EchoEventConsumer_i(CORBA::ORB_ptr orb)
  : orb_(CORBA::ORB::_duplicate(orb))
{ }

// Override the push() operation.
```

```
void EchoEventConsumer_i::push(
  const CORBA::Any& data)
{
  // Extract event data from the Any.
  const char* eventData;
  if (data >>= eventData) {
    std::cout << "EchoEventConsumer_i::push(): Received event: "
              << eventData << std::endl;
  }
}

// Override the disconnect_push_consumer() operation.
void EchoEventConsumer_i::disconnect_push_consumer()
{
  // Deactivate this object.
  CORBA::Object_var obj = orb_->resolve_initial_references("POACurrent");
  PortableServer::Current_var current =
    PortableServer::Current::_narrow(obj.in());
  PortableServer::POA_var poa = current->get_POA();
  PortableServer::ObjectId_var objectId = current->get_object_id();
  poa->deactivate_object(objectId.in());
}
```

23.4.1.5 Creating the Consumer and Connecting to the Channel

To receive events, the application must now create a consumer object, connect to the event channel, and enter the event loop. Locating the event channel is exactly the same as on the supplier side (see `EchoEventSupplierMain.cpp`):

```
int main (int argc, char* argv[])
{
  try
  {
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Find the Naming Service.
    CORBA::Object_var obj = orb->resolve_initial_references("NameService");
    CosNaming::NamingContextExt_var root_context =
       CosNaming::NamingContextExt::_narrow(obj.in());

    // Find the EchoEventChannel.
    obj = root_context->resolve_str("CosEventService");

    // Narrow the object reference to an EventChannel reference.
    CosEventChannelAdmin::EventChannel_var echoEC =
      CosEventChannelAdmin::EventChannel::_narrow(obj.in());
    if (CORBA::is_nil(echoEC.in())) {
```

```
      std::cerr << "Could not narrow EchoEventChannel." << std::endl;
      return 1;
    }
    std::cout << "Found the EchoEventChannel." << std::endl;
```

Next, we create a consumer to receive the events:

```
    // Instantiate an EchoEventConsumer_i servant.
    PortableServer::Servant_var<EchoEventConsumer_i> servant =
      new EchoEventConsumer_i(orb.in());

    // Register it with the RootPOA.
    obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());
    PortableServer::ObjectId_var oid = poa->activate_object(servant.in());
    CORBA::Object_var consumer_obj = poa->id_to_reference(oid.in());
    CosEventComm::PushConsumer_var consumer =
      CosEventComm::PushConsumer::_narrow(consumer_obj.in());
```

Now we obtain a `ProxyPushSupplier` and connect the consumer to the event channel:

```
    // Get a ConsumerAdmin object from the EventChannel.
    CosEventChannelAdmin::ConsumerAdmin_var consumerAdmin =
      echoEC->for_consumers();

    // Get a ProxyPushSupplier from the ConsumerAdmin.
    CosEventChannelAdmin::ProxyPushSupplier_var supplier =
      consumerAdmin->obtain_push_supplier();

    // Connect to the ProxyPushSupplier, passing your PushConsumer object
    // reference to it.
    supplier->connect_push_consumer(consumer.in());
```

After activating the POA and starting the event loop, the consumer is now ready to receive events.

```
    // Activate the POA via its POAManager
    PortableServer::POAManager_var poa_manager = poa->the_POAManager();
    poa_manager->activate();
    std::cout << "Ready to receive events..." << std::endl;

    // Enter the ORB event loop.
    orb->run();
    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
```

```
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
  }
}
```

The `push()` operation of the consumer is now invoked each time a supplier pushes an event onto this event channel.

23.4.2 Creating and Configuring Event Channel Servants

In the OMG's Event Service specification, event channel management is left up to the application. This means that applications must use implementation-specific ways of creating event channels.

Note *The Notification Service specification defines an Event Channel Factory interface for creating and managing event channels. Since the Notification Service event channels are an extension of those in the Event Service, client code based on the Event Service specification can use channels created with TAO's Notification Service. See Chapter 25 for details of TAO's Notification Service.*

The `tao_cosevent` server provides a simple and convenient way to create event channels and a means for suppliers and consumers to find and connect to these channels (via the Naming Service). This approach is sufficient for simple examples like the one above. However, you may want to create and manage your own event channel servants in application processes to:

- Implement multiple event channels in one process.
- Collocate the event channel with a supplier or consumer.
- Provide a different mechanism (other than simple names in the root naming context) for locating event channels.
- Easily and efficiently control the creation and destruction of event channels.
- Federate your event channel with other event channels.
- Customize the behavior of the event channel.

To create and manage an event channel servant that is based on the implementation, use the `TAO_CEC_EventChannel` class. You can either directly instantiate one of these objects or derive your own subclass and

specialize its behavior. The following example shows a supplier process that uses the `TAO_CEC_EventChannel` class to create its own local event channel.

23.4.2.1 Supplier/EC Collocation Example

Most of the code for this example is the same as in the previous example. Here we show only those sections of code that differ. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/EventServices/OMG_SupplierSideEC`.

Any process containing event channel servants must make the following call *before* calling `CORBA::ORB_init()`:

```
// Initialize the CEC Factory so we can customize the CEC
TAO_CEC_Default_Factory::init_svcs ();
```

This code initializes the factory that the servant uses to configure itself (see 23.6 for details). Because this configuration happens via the service configurator, the factory must be initialized before the service configurator is initialized (which occurs during the `ORB_init()` call).

The remaining changes replace the code in the supplier that resolves the event channel from the Naming Service and narrows its object reference.

```
// Get the RootPOA
CORBA::Object_var poa_object = orb->resolve_initial_references("RootPOA");

PortableServer::POA_var poa =
  PortableServer::POA::_narrow (poa_object.in ());
PortableServer::POAManager_var poa_manager = poa->the_POAManager ();
poa_manager->activate ();

// Create and activate the event channel servant
TAO_CEC_EventChannel_Attributes attr(poa.in(), poa.in());
PortableServer::Servant_var<TAO_CEC_EventChannel> ec =
  new TAO_CEC_EventChannel(attr);
ec->activate();
PortableServer::ObjectId_var oid = poa->activate_object(ec.in());
CORBA::Object_var ec_obj = poa->id_to_reference(oid.in());
CosEventChannelAdmin::EventChannel_var echoEC =
  CosEventChannelAdmin::EventChannel::_narrow(ec_obj.in());

// Bind the EventChannel in the Naming Service.
CosNaming::Name_var name = root_context->to_name("CosEventService");
root_context->rebind(name.in(), echoEC.in());
```

The first block of code locates the root POA and activates it via the POA manager. This is required because the supplier process must now become a CORBA server.

Next, the event channel servant is constructed, initialized, and activated. The `TAO_CEC_EventChannel_Attributes` class is used to initialize the event channel. The code in this example simply tells the event channel to use the root POA when activating new CORBA objects. In 23.4.2.2 we describe in more detail the usage of the `TAO_CEC_EventChannel_Attributes` class.

Lastly, the event channel is bound to a name in the Naming Service so that consumers can locate it.

Overall, this example executes in the same way as the previous one, with the exception that it is slightly more efficient because of the collocation of the event channel and supplier. An additional advantage is that an alternative mechanism for advertising the existence and location of the event channel can be used (e.g., writing the IOR as a string to a file, or advertising it via the Trading Service).

23.4.2.2 Setting Attributes of the Event Channel

The event channel has a number of attributes that are set via the `TAO_CEC_EventChannel_Attributes` object that is passed to the constructor of the event channel. Table 23-1 provides a summary of the attributes that can be set:

Table 23-1 Event Channel Attributes

Name	Type	Default	Description
`supplier_poa`	`PortableServer::POA_ptr`	None	POA used by supplier admin and supplier proxies. This is typically the same POA the EC uses.
`consumer_poa`	`PortableServer::POA_ptr`	None	POA used by consumer admin and consumer proxies. This is typically the same POA the EC uses.
`consumer_reconnect`	`int`	0	Enables consumer reconnections when non-zero.
`supplier_reconnect`	`int`	0	Enables supplier reconnections when non-zero.

Table 23-1 Event Channel Attributes

Name	Type	Default	Description
`disconnect_callbacks`	`int`	0	If not zero, the event channel sends disconnect callbacks when a disconnect operation is called on a proxy.

In the previous example, the only attributes set are the supplier and consumer POAs. These are set via the `EventChannel_Attributes` constructor as follows:

```
TAO_CEC_EventChannel_Attributes attributes (poa.in (),   // Supplier POA
                                            poa.in ());  // Consumer POA
TAO_CEC_EventChannel* ec = new TAO_CEC_EventChannel(attributes);
```

The event channel uses the supplier POA to activate `SupplierAdmin`, `ProxyPushSupplier`, and `ProxyPullSupplier` servants, and the consumer POA to activate `ConsumerAdmin`, `ProxyPushConsumer`, and `ProxyPullConsumer` servants. In our case, we pass the same POA object reference for use as both the supplier POA and the consumer POA.

All other attributes of the event channel are set using public data members of the `TAO_CEC_EventChannel_Attributes` class. For example:

```
TAO_CEC_EventChannel_Attributes attributes (poa.in (), poa.in ());
attributes.disconnect_callbacks = 1;
TAO_CEC_EventChannel* ec_impl = new TAO_CEC_EventChannel(attributes);
```

The `disconnect_callbacks` attribute controls whether the consumer and supplier disconnect callbacks are called when the corresponding disconnect operation is called on the proxy object. For example, if this attribute is set to `true`, then when a consumer calls `disconnect_push_supplier()` on its proxy, the event channel invokes `disconnect_push_consumer()` on the consumer. Similar behaviors exist for pull suppliers as well as both types of consumers. It is a matter of debate as to whether the Event Service specification requires these callbacks to be made. These callbacks are always called when the event channel initiates the disconnection.

The `supplier_reconnect` and `consumer_reconnect` attributes allow suppliers and consumers to call the operations `connect_push_supplier()` and `connect_push_consumer()` multiple times without disconnecting.

This allows you to more efficiently replace the supplier or consumer object passed to the event channel.

The default values for these attributes are defined as preprocessor macros in the `$TAO_ROOT/orbsvcs/orbsvcs/CosEvent/CEC_Defaults.h`. You can use your own project-specific defaults by setting these macros in your `config.h` file and recompiling the event service.

23.4.3 Pull Model Support

The Event Service Specification gives a great deal of latitude in the terms of how an implementation supports the pull model. This section describes the details of TAO's implementation of the pull model.

When pull suppliers are connected, the event channel periodically attempts to pull events from each pull supplier. The default period between attempted pulls is 5 seconds. This value can be modified via a service configurator option, see 23.6.1.5 for details.

Pushed and pulled events are immediately delivered to all push consumers and are queued for delivery to pull consumers. The event channel maintains a separate event queue for each pull consumer. When pull consumers attempt to pull events, the oldest event is removed from its queue and returned. If the queue is empty, then the call blocks until an event is available.

Note *TAO's implementation of the CosEvent service's pull model works reliably only when the service is using the* `thread-per-connection` *concurrency model. See 19.3.5 for more information on how to configure this behavior.*

23.4.4 Typed Event Channel Example

This example shows how a typed event channel can be used with push consumers and suppliers. We utilize a modified version of the `Messenger` interface that is used throughout this book. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/EventServices/OMG_TypedEC`.

23.4.4.1 The Messenger Interface

In order to use a typed event channel, your application must define an IDL interface that constrains the event data that is passed through the EC. This is a

normal IDL interface but with certain restrictions on the operations it contains. The operations contained in the interface must not have return values and can only contain `in` parameters. These are effectively the same restrictions as those placed on oneway operations (although operations on the typed EC interface can be oneway or synchronous).

Here is the Messenger interface with the modifications necessary to allow its usage with a typed event channel.

```
interface Messenger {
  void send_message(in string user_name,
                    in string subject,
                    in string message);
};
```

23.4.4.2 Creating the Typed Event Channel

TAO's typed event channel implementation utilizes the Interface Repository to allow it to take a single invocation on our application interface from a particular supplier and propagate it to multiple consumers. Therefore, we need to start the Interface Repository server and load the interface description of our Messenger interface into it before creating and using the typed event channel. We use the following commands to start and populate the Interface Repository:

```
export InterfaceRepositoryIOR=file://ifr.ior
$TAO_ROOT/orbsvcs/IFR_Service/tao_ifr_service -o ifr.ior &
$ACE_ROOT/bin/tao_ifr Messenger.idl
```

For more information about TAO's Interface Repository see Chapter 26. Now, using the `tao_cosevent` server, we can create the typed event channel.

```
$TAO_ROOT/orbsvcs/CosEvent_Service/tao_cosevent -t &
```

Just as in the previous examples, this stores the event channel object reference in the naming service with the only difference being the -t option which means that the event channel now implements the `CosTypedEventChannelAdmin::TypedEventChannel` interface.

23.4.4.3 Implementing the Typed Supplier

Most of the supplier code remains very similar to the untyped supplier we saw previously. The main differences are the use of the corresponding "typed" interfaces in this example and the details about how typed events are published. First, we substitute `#include` directives for the typed event channel IDL. We also include the `Typecode.h` header, so we can later use the `TypeCode` interface to get the repository ID of the Messenger interface.

```
#include <orbsvcs/CosTypedEventCommC.h>
#include <orbsvcs/CosTypedEventChannelAdminC.h>
#include <tao/AnyTypeCode/Typecode.h>
```

The first differences in the body of the program are where the event channel is narrowed as it is now a `TypedEventChannel`.

```
    // Find the EventChannel.
    obj = root_context->resolve_str("CosEventService");

    // Downcast the object reference to a TypedEventChannel reference.
    CosTypedEventChannelAdmin::TypedEventChannel_var ec =
      CosTypedEventChannelAdmin::TypedEventChannel::_narrow(obj.in ());
```

Connecting to the event channel goes through the same steps as the untyped example: get the supplier admin, obtain a proxy consumer, and connect to the proxy consumer.

```
    // Get a SupplierAdmin object from the EventChannel.
    CosTypedEventChannelAdmin::TypedSupplierAdmin_var supplierAdmin =
      ec->for_suppliers();

    // Get a ProxyPushConsumer from the SupplierAdmin.
    CosTypedEventChannelAdmin::TypedProxyPushConsumer_var consumer =
      supplierAdmin->obtain_typed_push_consumer(::_tc_Messenger->id());

    // Connect to the ProxyPushConsumer as a PushSupplier
    // (passing a nil PushSupplier object reference to it because
    // we don't care to be notified about disconnects).
    consumer->connect_push_supplier(CosEventComm::PushSupplier::_nil());
```

One key difference above is that the `obtain_typed_push_consumer()` operation takes a string parameter that specifies the IDL interface type by passing its repository ID. We obtain the `Messenger` interface's repository ID by using the `TypeCode` object's `id()` operation.

In order to publish typed events, we now need to obtain an object reference that implement's the Messenger interface. The `TypedProxyPushConsumer` interface supports a `get_typed_consumer()` operation that returns an object reference that implements the type associated with that proxy.

```
// Obtain the interface from the event channel
CORBA::Object_var messenger_obj = consumer->get_typed_consumer();

// Narrow the interface
Messenger_var messenger = Messenger::_narrow(messenger_obj.in () );
```

We can now publish events to any connected consumers by simply invoking the `send_message()` operation on our Messenger object reference. The typed event channel implementation is responsible for multiplexing this operation so that all eligible consumers receive the same request.

```
// Send one event per second. (approx)
while (1) {
  messenger->send_message("King Lizard",
                          "Proclamations",
                          "Hello, world");

  ACE_Time_Value event_delay(0, 1000 * EVENT_DELAY_MS);
  orb->run(event_delay);
}
```

Note *Because the* `TypedProxyPushConsumer` *interface is derived from the* `ProxyPushConsumer` *interface it also implements the* `push()` *operation. According to the Event Service specification, suppliers to typed event channels can also publish untyped events by utilizing this operation. TAO's typed event channel implementation does not support this feature and any attempt to call* `push()` *on a typed proxy push consumer results in a* `NO_IMPLEMENT` *exception.*

23.4.4.4 Implementing the Typed Consumer

Implementing the typed consumer is slightly more complicated than the untyped consumer. We still need to implement a consumer servant and connect it to the event channel, but now we also have to implement our application-specific interface (Messenger in our case) and associate it with our consumer.

First, let’s look at our messenger servant.

```
class  Messenger_i : public virtual POA_Messenger {
public:
  Messenger_i (CORBA::ORB_ptr orb, int event_limit);
  virtual ~Messenger_i ();

  virtual void send_message (const char * user_name,
                             const char * subject,
                             const char * message);

private:
  CORBA::ORB_var orb_;
  int            event_limit_;

};
```

The constructor takes two arguments, an ORB reference and an event limit. Both values are stored in data members for use by `send_message()`, which prints each message’s data and shuts down the ORB when the event limit is reached.

```
void Messenger_i::send_message (const char * user_name,
                                const char * subject,
                                const char * message)
{
  std::cout << "Message from: " << user_name << std::endl;
  std::cout << "Subject:      " << subject << std::endl;
  std::cout << "Message:      " << message << std::endl;

  if (--event_limit_ <= 0) {
    orb_->shutdown(0);
  }
}
```

Next, let’s look at our typed consumer servant which we’ll connect to the event channel and use to hold our Messenger object reference.

```
#include <orbsvcs/CosTypedEventCommS.h>

class Consumer_i : public virtual POA_CosTypedEventComm::TypedPushConsumer {
  public:
    Consumer_i(CORBA::ORB_ptr orb,
               CORBA::Object_ptr obj);
```

```
    // Override operations from TypedPushConsumer interface.
    virtual CORBA::Object_ptr get_typed_consumer ();

    virtual void push(const CORBA::Any & data);

    virtual void disconnect_push_consumer();

  private:
    CORBA::ORB_var orb_;
    CORBA::Object_var object_;
};
```

The `disconnect_push_consumer()` member function is implemented as before for untyped consumers. The `push()` member function is used by the event channel to deliver any untyped events published to that channel. If we do not wish to receive untyped events we may simply throw a `NO_IMPLEMENT` exception.

```
void Consumer_i::push(const CORBA::Any & data) {
  throw CORBA::NO_IMPLEMENT ();
}
```

Note *Because TAO does not support publication of untyped events via typed event channels, this operation should never be called when using TAO's typed event channel implementation.*

The `get_typed_consumer()` member function should return an object reference to a CORBA object that implements the specific interface we are using for this consumer. We pass the object reference as a constructor parameter, store it in a data member, and return it from `get_typed_consumer()`. By using `CORBA::Object` as the object reference data type, we allow this class to be potentially used with other interface types (besides `Messenger`).

```
Consumer_i::Consumer_i(CORBA::ORB_ptr orb, CORBA::Object_ptr obj)
  : orb_(CORBA::ORB::_duplicate(orb)), object_(CORBA::Object::_duplicate(obj))
{
}

CORBA::Object_ptr
Consumer_i::get_typed_consumer ()
{
```

```
  return CORBA::Object::_duplicate(object_.in());
}
```

23.4.4.5 Connecting the Typed Consumer

Connecting our typed consumer to the event channel is very similar to the untyped example with the important differences being the use of “typed” interfaces and some details of the consumer object construction. First, we see some different headers we’ll need:

```
#include <orbsvcs/CosTypedEventCommC.h>
#include <orbsvcs/CosTypedEventChannelAdminC.h>
#include <tao/AnyTypeCode/Typecode.h>
```

After we get the event channel reference from the naming service, we can narrow it to a `TypedEventChannel`, get the consumer admin reference, and obtain a typed proxy supplier.

```
obj = root_context->resolve_str("CosEventService");

// Downcast the object reference to an TypedEventChannel reference.
CosTypedEventChannelAdmin::TypedEventChannel_var ec =
  CosTypedEventChannelAdmin::TypedEventChannel::_narrow(obj.in());

// Get a ConsumerAdmin object from the EventChannel.
CosTypedEventChannelAdmin::TypedConsumerAdmin_var consumerAdmin =
  ec->for_consumers();

// Get a ProxyPushSupplier from the ConsumerAdmin.
CosEventChannelAdmin::ProxyPushSupplier_var supplier =
  consumerAdmin->obtain_typed_push_supplier(::_tc_Messenger->id());
```

We used the repository ID of the Messenger interface when obtaining the typed proxy supplier which associates that interface type with the consumer when we connect to that proxy. Now we are ready to create our consumer and connect it to the event channel.

```
// Get the RootPOA.
// Activate the POA manager here before we connect our consumer.
obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());
PortableServer::POAManager_var poa_manager = poa->the_POAManager();
poa_manager->activate();

// Create our Messenger_i servant and activate the CORBA object
```

```
PortableServer::Servant_var<Messenger_i> servant =
  new Messenger_i(orb.in(), EVENTS_TILL_SHUTDOWN);
PortableServer::ObjectId_var oid = poa->activate_object(servant.in());
CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());

// Create our Consumer servant and pass it the Messenger
// object reference. Activate the consumer CORBA object.
PortableServer::Servant_var<Consumer_i> consumer_servant =
  new Consumer_i(orb.in(), messenger_obj.in());
PortableServer::ObjectId_var cons_oid =
  poa->activate_object(consumer_servant.in());
CORBA::Object_var consumer_obj = poa->id_to_reference(cons_oid.in());
CosTypedEventComm::TypedPushConsumer_var consumer =
  CosTypedEventComm::TypedPushConsumer::_narrow(consumer_obj.in());

// Connect to the typed proxy push supplier.
supplier->connect_push_consumer(consumer.in());
```

Because the consumer contains a reference to the Messenger object, the event channel can retrieve the Messenger object reference and call operations on it each time a supplier publishes an event.

23.4.4.6 Mixing Types in a Typed Event Channel

The repository ID passed to the connect operations by the consumers and suppliers acts as a key for the event channel to match clients using the same interface. Suppliers using a particular interface only send events to consumers that utilize that same interface. When more than one type is mixed in the event channel, the interface type acts as a filter to ensure that only events of the requested type are delivered to a consumer.

23.5 tao_cosevent Command Line Options

The `tao_cosevent` server supplies the capability to start a single event channel in its own process. It can bind the created event channel to a supplied name in the root naming context of the Naming Service. The Naming Service

must be running to use this server (unless the `-x` option is used). Table 23-2 describes the available command line options.

Table 23-2 tao_cosevent Command Line Options

Option	Description	Default
`-n` *`COS_EC_name`*	Specifies the name with which to bind the event channel (in the root naming context of the Naming Service).	`CosEventService`
`-o` *`filename`*	Specify the name of the output file for writing the Event Channel's IOR as a string.	Do not write the IOR to a file.
`-p` *`filename`*	Specify the name of the output file for the process ID to be written to.	Do not write out the process ID.
`-r`	Use the `rebind()` operation to bind the event channel in the Naming Service. If the name is already bound, and this flag is not passed, then the process exits with an `AlreadyBound` exception.	The `bind()` operation is used.
`-x`	Do not use the Naming Service. This simply creates an event channel.	Bind the EC in the Naming Service.
`-t`	Create a typed event channel	Create an untyped event channel
`-d`	Destroy flag for typed event channels. Determines whether shutdown of a typed EC shuts down the ORB.	Don't shutdown the ORB for typed ECs.
`-b`	Causes the event channel to send disconnect callbacks when a disconnect operation is called on a proxy.	Callbacks are not called when disconnect is called on the proxy.

When -t is specified, the event channel object implements the `CosTypedEventChannelAdmin::TypedEventChannel` interface. When -t is not specified, the event channel object implements the `CosEventChannelAdmin::EventChannel` interface.

When the `destroy()` operation of the event channel is called, the event channel is destroyed and other cleanup functionality may be performed. If -d is passed, a typed event channel also unbinds from the Naming Service and exits the process. Untyped event channels currently do not perform these cleanup tasks.

The `-n` and `-r` options are ignored if `-x` is specified.

Note *If the -o option is specified without the -x option, the Event Channel object reference is stringified and written to the specified file* ***and*** *bound in the Naming Service.*

23.6 Event Channel Resource Factory

The event channel resource factory is responsible for creating many strategy objects that control the behavior of the event channel.

The behavior of the event channel is typically controlled by using the service configurator to select the appropriate behaviors for the default factory implementation. Applications are also free to implement their own resource factories, but this is not commonly done. See Chapter 16 for more information on using the service configurator.

The event channel resource factory is registered with the service configurator using the name `CEC_Factory`. The default event channel resource factory is statically registered with the service configurator, so the `static` directive is used to supply initialization options to it. To change the behavior of the default event channel factory, add a line similar to the line shown below to your service configuration file:

```
static CEC_Factory "-CECDispatching mt -CECDispatchingThreads 5"
```

The option descriptions begin in 23.6.1.1. For these options to be effective, you must make sure that the following function call occurs *before* the ORB is initialized:

```
TAO_CEC_Default_Factory::init_svcs ();
```

This function creates a default event channel resource factory and statically registers it. If this is not done, the service configurator is not able to find and initialize the `CEC_Factory`.

The `-ORBSvcConf` option allows you to use file names other than `svc.conf` for service configurator initialization. See 17.13.63 for more information on this option.

The default values for many of the event channel resource factory options are defined in `$TAO_ROOT/orbsvcs/orbsvcs/CosEvent/CEC_Defaults.h`

as preprocessor macros. You can use your own project-specific defaults by setting these macros in your `config.h` file and recompiling the event service.

23.6.1 CEC_Factory Option Overview

This section provides an overview of the configuration options supported by the default `CEC_Factory`. The following section provides detailed documentation of each of the individual options.

23.6.1.1 Dispatching

When the event channel is pushing events to interested consumers, choosing the thread used to push the event is a decision that has far-reaching affects on the performance and behavior of the application. The event channel resource factory allows for selection of a dispatching strategy that defines how to push events received from suppliers to the interested consumers on the appropriate thread. The default event channel resource factory allows for either reactive or multithreaded dispatching strategies. In addition, when a multithreaded dispatching strategy is selected, the number of threads to be used can be specified. Table 23-3 shows the options related to dispatching strategies.

Table 23-3 Dispatching related options

Option	Section	Description
`-CECDispatching {`**`reactive`**` | mt}`	23.6.2.5	Supply this option to select the dispatching strategy for supplier-produced events.
`-CECDispatchingThreads` *`nthreads`*	23.6.2.6	Specify the number of threads to create and use for the multithreaded dispatching strategy. Defaults to one thread.

The `reactive` dispatching strategy delivers events on the same thread as they were received (or generated). This is usually the reactor's main thread. The `mt` (multithreaded) dispatching strategy creates a pool of threads and dispatches each event on a randomly-selected member of the pool.

23.6.1.2 Locking Options

The locking options allow the event channel resource factory to define the lock type desired for various components in the event channel. The default

factory allows specification of the lock type for consumer and supplier proxies using the options shown in Table 23-4.

Table 23-4 Locking options

Option	Section	Description
`-CECProxyConsumerLock {null | thread | recursive}`	23.6.2.8	Specifies the lock type for the consumer proxy object.
`-CECProxySupplierLock {null | thread | recursive}`	23.6.2.11	Specifies the lock type for the supplier proxy object.

These options can be set to `null` to increase performance if the event channel does not access given components from multiple threads. The default values ensure that the proxy is thread safe, but recursive locks may be required to avoid deadlocks in certain complex systems.

23.6.1.3 Consumer and Supplier Control Options

The following group of options allows the event channel resource factory to define how the event channel handles *dangling* (ill-behaved) suppliers and consumers. Consumers and suppliers that remain connected to the event channel when their CORBA objects are no longer accessible from the event channel process are considered ill-behaved. Such consumers and suppliers result when the consumer or supplier process fails to call disconnect, terminates abnormally, or has its node disconnected from the network. The default factory allows specification and configuration of the control policy for consumer and supplier proxies via the options shown in Table 23-5.

Table 23-5 Consumer and supplier control options

Option	Section	Description
`-CECConsumerControl {null | reactive}`	23.6.2.1	Define the policy for handling ill-behaved consumers.
`-CECSupplierControl {null | reactive}`	23.6.2.13	Define the policy for handling ill-behaved suppliers.
`-CECConsumerControlPeriod` *period*	23.6.2.2	Define the polling period in microseconds of the reactive consumer control policy.
`-CECSupplierControlPeriod` *period*	23.6.2.14	Define the polling period in microseconds of the reactive supplier control policy.

Table 23-5 Consumer and supplier control options

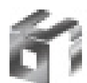
Option	Section	Description
`-CECConsumerControlTimeout` *`timeout`*	23.6.2.3	Round-trip timeout in microseconds for the consumer control ping.
`-CECSupplierControlTimeout` *`timeout`*	23.6.2.15	Round-trip timeout in microseconds for the supplier control ping.
`-CECConsumerOperationTimeout` *`timeout`*	23.6.2.4	Round-trip timeout in microseconds for the consumer operations other than the ping.
`-CECSupplierOperationTimeout` *`timeout`*	23.6.2.16	Round-trip timeout in microseconds for the supplier operations other than the ping.
`-CECProxyDisconnectRetries` *`n`*	23.6.2.9	Number of retries allowed for the reactive control strategy.

The default control policy of `null` leaves consumers and suppliers connected to the event channel even if the event channel is unable to access them. This policy allows consumers and suppliers to continue to be connected even in the face of intermittent communications.

The `reactive` control policy disconnects a consumer or supplier from the event channel when the event channel fails to contact it. It also periodically polls (by default every 5 seconds) each consumer and supplier to ensure its continued connection. Failure to respond to the polling requests before a timeout (by default 10 milliseconds) also results in disconnection. By default, the first failure to contact a consumer or supplier results in disconnection. You can specify a number of retries using the `-CECProxyDisconnectRetries` option. The periods for the periodic polls can be set via the `-CECSupplierControlPeriod` and `-CECConsumerControlPeriod` options. If the polling period is set to 0, polling is completely disabled. The round-trip timeout for the periodic polls can be set via the `-CECSupplierControlTimeout` and `-CECConsumerControlTimeout` options. The round-trip timeout for operations other than the periodic polls can be set via the `-CECSupplierOperationTimeout` and `-CECConsumerOperationTimeout` options.

23.6.1.4 Proxy Collection Options

The proxy collection options define the types of collections used to hold consumer and supplier proxies. The default factory allows specification of the



collection type for consumer and supplier proxies via the options shown in Table 23-6.

Table 23-6 Proxy collection options

Option	Section	Description
`-CECProxyConsumerCollection` *`flags`*	23.6.2.7	Define the characteristics of the collection used to store proxy consumers in the event channel.
`-CECProxySupplierCollection` *`flags`*	23.6.2.10	Define the characteristics of the collection used to store proxy suppliers in the event channel.

The flags passed to these collection options fall into three separate groups, with each group specifying a different characteristic of the collection. A colon is used as a separator between flags (e.g., `mt:rb_tree:immediate`).

First, the lock type used to control access to the collection can be specified. The `st` flag allows specification of a `null` lock. The `mt` flag specifies a thread-safe lock. A thread-safe lock is specified by default.

The second characteristic is the actual collection type used. The `list` flag specifies that an ordered-list collection is used. The `rb_tree` flag specifies that a collection using a red-black tree is used. By default, the event channel uses an ordered-list collection.

The third characteristic specifies how the concurrent use of the collection is controlled, specifically the case where the collection is being iterated over while a client attempts an operation that adds or removes a member of the collection. An example is the distribution of events to push consumers (via iteration over the proxy push suppliers collection) while one of the consumers is attempting to disconnect itself. The default factory provides four different strategies for this characteristic: `immediate`, `copy_on_read`, `copy_on_write`, and `delayed`.

The `immediate` flag causes each operation to block until it receives access to the collection. In the example above, the consumer that is attempting to disconnect blocks until the event distribution iteration completes. Note that it is possible that the disconnect request may be processed in the same thread as the event distribution (via a nested upcall). If this occurs, immediate access is granted (the thread already has the lock for the collection), and the iterator may be invalidated. It is the developer's responsibility to ensure that the iterator is not invalidated. Using the `-CECDispatching` option (see 23.6.2.5)

to establish a separate dispatching thread is the most common way to ensure this validity. (In other words, the `immediate` collection update flag should not be used with `-CECDispatching reactive`.)

The `copy_on_read` flag causes the iterators to copy the collection before proceeding. This allows iterators to release the lock after the copy is made. Subsequent changes to the collection can occur while iteration is ongoing without affecting the iteration. In the above example, this means that the consumer can disconnect without harm to the event dispatching. The main disadvantage of this approach is the extra performance overhead incurred when the copy of the collection is allocated and replicated.

The `copy_on_write` flag causes any modifiers to the collection to make copies of the collection before proceeding. This means changes to the collection can occur while iteration is ongoing without affecting the iteration. In the above example, this means that the consumer can disconnect without harm to the event dispatching. The main disadvantage of this approach is the extra performance overhead incurred when the copy of the collection is allocated and replicated. Note that the `copy_on_write` strategy makes a copy each time the collection is changed (connect, disconnect, reconnect, or shutdown), whereas the `copy_on_read` strategy makes copies each time the collection is iterated.

The `delayed` flag causes changes to the collection to be queued while iterations are ongoing. When all iterations have completed, the queued modifications are made. The event channel attributes of `busy_hwm` and `max_write_delay` allow bounds to be set on how many iterators access the collection at a time and how many iterators may access it before modification occurs. See 23.4.2.2 for details of these attributes and how to set them.

23.6.1.5 Miscellaneous Options

The options shown in Table 23-7 allow for control of the pull model behavior and the ORB that the event channel uses.

Table 23-7 Miscellaneous options

Option	Section	Description
`-CECReactivePullingPeriod` *`period`*	23.6.2.12	Defines the polling period in microseconds that the reactive pulling strategy uses. The default period is 5000000 (5 seconds).

Table 23-7 Miscellaneous options

Option	Section	Description
`-CECUseORBId` *orbid*	23.6.2.17	Specifies the id of the ORB that the default factory uses.

The reactive pulling period is the period of time between attempted *pulls* on pull suppliers. Events pulled are immediately delivered to push consumers and queued for eventual delivery to pull consumers.

The default factory requires an ORB for a variety of operations. It normally uses the default ORB (with a null string for the ORB id). Specify an ORB id using the `-CECUseORBId` option to force the default factory to use a different ORB. Typically, this option is used to ensure that the default factory is using the same ORB as was used to activate the event channel.

23.6.2 Event Channel Resource Factory Options

The remainder of this chapter describes the individual options interpreted by the default event channel factory. These options are applied to the default event channel resource factory by the service configurator as described in 23.6.

23.6.2.1 CECConsumerControl *control_policy*

Values for *control_policy*

`null` (default)	Do not discard dangling consumers.
`reactive`	Use a reactive policy to discard dangling consumers.

Description This option specifies the policy to be used when dealing with dangling consumers. The `null` control policy never disconnects ill-behaved consumers.

The `reactive` policy disconnects consumers after a number of communication failures. A communication failure is either a failure to push an event to a consumer or the failure of the periodic ping performed on each consumer. By default, the first communication failure results in the disconnection of the consumer. The number of retries allowed can be set via the `-CECProxyDisconnectRetries` option.

Usage Use the `reactive` control policy when consumers could possibly be destroyed without disconnecting. Use the default control policy (`null`) when you can guarantee that all consumers disconnect properly (or not at all), and you do not want to incur the overhead of the `reactive` policy.

Impact The `null` consumer control strategy causes degraded throughput when consumers are destroyed without first disconnecting. The `reactive` strategy requires slightly more overhead in normal operation, may result in consumers having to reconnect when the network quality is bad (with potential for missed messages), and requires slightly more memory.

See Also 23.6.2.2, 23.6.2.3, 23.6.2.9, 23.6.2.4, 23.6.2.13

Example

```
static CEC_Factory "-CECConsumerControl reactive"
```

23.6.2.2 CECConsumerControlPeriod *period*

Description Sets the period (in microseconds) that the `reactive` consumer control policy uses to poll the state of the consumers. The default period is 5000000 (5 seconds). A value of zero disables consumer state polling.

Usage For event channels using the `reactive` consumer control policy, use this option to control the time to wait between attempted *pings* on each consumer. The reactive consumer control strategy object pings the consumer by invoking `CORBA::Object::_non_existent()` on the consumer's object reference; this is a synchronous call. The `-CECConsumerControlPeriod` option is ignored when the consumer control policy is not `reactive`.

Impact Shorter periods require more bandwidth and processing to validate the existence of the consumers. Longer periods consume less of these resources. You can disable the ping altogether by setting the period to zero.

See Also 23.6.2.1, 23.6.2.14

Example

```
static CEC_Factory "-CECConsumerControl reactive -CECConsumerControlPeriod 
1000000"
```

23.6.2.3 CECConsumerControlTimeout *timeout*

Description Sets the relative round-trip timeout (in microseconds) that the reactive consumer control policy uses for polling consumers. The default timeout is 10000 (10 milliseconds).

Note *The* `-CECConsumerControlRoundtripTimeout` *is an alias for the* `-CECConsumerControlTimeout` *option.*

Usage For event channels using the `reactive` consumer control policy, use this option to control the time the event channel waits for a consumer to respond to an attempted *ping*. The reactive consumer control strategy object pings the consumer by invoking `CORBA::Object::_non_existent()` on the consumer's object reference; this is a synchronous call. Failure to respond within the specified timeout period results in the event channel classifying that ping as a communication failure for that consumer. The `-CECConsumerControlTimeout` option is ignored when the consumer control policy is not `reactive`.

Impact Smaller timeout values may result in more timeout failures and consumers being disconnected more often. A larger timeout value means it takes longer to detect and remove dead consumers.

See Also 23.6.2.5, 23.6.2.14

Example

```
static CEC_Factory "-CECConsumerControl reactive -CECConsumerControlTimeout
50000"
```

23.6.2.4 CECConsumerOperationTimeout *timeout*

Description Sets the relative round-trip timeout (in microseconds) that the reactive consumer control policy uses for all operations besides the polling operation. The default timeout is zero, which is evaluated as no timeout. The timeout for the polling operation is covered by the `-CECConsumerControlTimeout` option.

Usage For event channels using the `reactive` consumer control policy, use this option to control the time the event channel waits for a consumer to respond to all operations besides the *ping*. This mainly affects the `push()` call for push consumers.

Failure to respond within the specified timeout period results in the event channel classifying that request as a communication failure for that consumer. The `-CECConsumerOperationTimeout` option is ignored when the consumer control policy is not `reactive`.

Impact Smaller timeout values may result in more timeout failures and consumers being disconnected more often. A larger timeout value means it takes longer to detect and remove dead consumers.

See Also 23.6.2.1, 23.6.2.3

Example

```
static CEC_Factory "-CECConsumerControl reactive -CECConsumerOperationTimeout
50000"
```

23.6.2.5 CECDispatching *dispatching_strategy*

Values for *dispatching_strategy*

`reactive` (default)	The event channel delivers events to consumers on the same thread that received them.
`mt`	The event channel uses separate threads to receive and deliver events to consumers. It randomly selects a thread from a pool. The number of threads in the pool is set via the `-CECDispatchingThreads` option (the default is `1`).

Description This option controls the dispatching strategy that the event channel uses. The default strategy is `reactive`.

Usage This strategy determines the order in which the event channel delivers events as well as its overall throughput. The multithreaded (`mt`) strategy should allow for greater performance than the `reactive` model. The `reactive` strategy is appropriate when it is acceptable and/or desired for all events to be delivered in the order they are received. The `mt` strategy is especially effective for decoupling the suppliers from the consumers' execution time, especially in the collocated case. The `mt` strategy also reduces the maximum latency of event deliveries.

Impact The `reactive` model is inappropriate when slow processing by individual consumers can affect other users, greater throughput is desired, or the system requires strict prioritization of events. Note that the `reactive` strategy delivers events on the same thread as they were received. The ORB's configuration determines the receiving thread. The `mt` strategy may increase the time required to dispatch an event in lightly-loaded event channels and also require the allocation of additional resources.

See Also 19.3.5, 23.6.2.6

Example

```
static CEC_Factory "-CECDispatching mt"
```

23.6.2.6 CECDispatchingThreads *nthreads*

Description By default the multithreaded dispatching strategy creates one thread to use for the delivery of supplier-originated events to consumers. Use this option to specify a different number of threads to be created and used.

Usage Using the multithreaded dispatching strategy with the default of one thread provides the user the benefit of separating the dispatching thread from the receiving thread. This allows for a greater decoupling between suppliers and consumers. Use of this option to specify additional dispatching threads results in additional decoupling between consumers as well as potentially increasing throughput.

Impact Specifying additional dispatching threads consumes additional resources.

See Also 23.6.2.5

Example

```
static CEC_Factory "-CECDispatching mt -CECDispatchingThreads 5"
```

23.6.2.7 CECProxyConsumerCollection *flags*

Description This option controls the type of collection the event channel uses to hold consumer proxies. The flags passed describe the characteristics of the desired collection. Colons should separate the flags (e.g., `mt:list`). The allowed flags are described in Table 23-8. Only one flag per type should be specified.

Table 23-8 Collection Type Flags

Flag	Type	Description
`mt` (default)	Synchronization	Use a thread-safe lock for the collection.
`st`		Use a null lock for the collection.
`list` (default)	Collection	Implement the collection using an ordered list.
`rb_tree`		Implement the collection using a red-black tree.
`immediate` (default)	Iterator	Threads block until they can execute a change to the collection.
`copy_on_read`		Before initiating an iteration of the collection, a copy of the complete collection is performed.
`copy_on_write`		Before initiating a modification to the collection, a copy of the complete collection is performed.
`delayed`		Changes that cannot be made immediately are queued for later execution.

For a more detailed discussion of the collection types, see 23.6.1.4.

Usage Applications that guarantee that a consumer proxy collection is only accessed from a single thread can specify the `st` flag to improve performance.

Event channels that connect and disconnect suppliers often, and wish to optimize these operations (at the expense of iteration speed), should specify the `rb_tree` flag.

Applications that use the `immediate` flag must guarantee that the thread iterating over the proxy collection does not attempt to modify the collection, as this invalidates the iterator. One way to insure that such a collection is not modified is to specify a separate dispatching thread. If you wish to minimize priority inversions between publication and supplier connections and disconnections, use the `delayed` flag. Collections using the `copy_on_read` flag are only applicable to systems with small numbers of consumer proxies that require low latencies for proxy collection modifications. Collections using the `copy_on_write` flag are only applicable to systems with small

numbers of consumer proxies that require low latencies for proxy collection iterations.

Impact The `mt` flag incurs additional overhead over the `st` flag during connection/disconnection of suppliers and iteration over the collection.

List-based collections result in slower updates to the collection. Red-black tree collections are slower during iteration over the collection.

Immediate update of consumer proxy collections (during connection or disconnection of suppliers) may cause priority inversions because of the long-lived locks involved. Copy on read collections incur dynamic allocation and copy costs for each iteration of the proxy collection. Copy on write collections incur dynamic allocation and copy costs for each modification to the proxy collection. Delayed updates to collections can result in long intervals between the requested change and its actual occurrence.

See Also 23.6.2.10

Example

```
static CEC_Factory "-CECProxyConsumerCollection mt:delayed"
```

23.6.2.8 CECProxyConsumerLock *lock_strategy*

Values for *lock_strategy*

`null`	Do not use any locking on the proxy consumers.
`thread` (default)	Use a thread-safe lock on the proxy consumers.
`recursive`	Use a recursive thread-safe lock on the proxy consumers.

Description This option defines the type of lock to be used in synchronizing access to the proxy consumer objects.

Usage Single-threaded applications can use the `null` lock to increase the efficiency of the push consumer. Multithreaded applications may need to set the lock to `recursive` in cases where operations on the proxy consumer may cause recursive access to the proxy consumer. In all other situations, the `thread` lock should be used.

Impact The `null` lock causes problems in applications that access the proxy from more than one thread. The `thread` lock causes additional locking overhead that may not be needed in applications that restrict proxy access to a single thread. The `recursive` lock is even more expensive than the `thread` lock, but is required by applications that must recursively access the lock.

See Also 23.6.2.11

Example

```
static CEC_Factory "-CECProxyConsumerLock recursive"
```

23.6.2.9 CECProxyDisconnectRetries *n*

Description Sets the number of retries that the `reactive` consumer and supplier control policies use when determining whether to disconnect clients (consumers or suppliers). The default number of retries is zero, meaning that the first failure results in the client being disconnected from the event channel. Each successful communication with the client resets the retry count.

Usage For event channels using the `reactive` control policy, use this option to be more tolerant of ill-behaved consumers and suppliers. This option is ignored when the consumer and supplier control policies are not `reactive`.

Impact The default value of zero retries means that any failure to contact a consumer or supplier results in their disconnection. This may be too strict for some applications. Larger retry values mean that it takes longer for the event channel to detect and remove ill-behaved clients. This can impact the overall efficiency and performance of the event channel.

See Also 23.6.2.1, 23.6.2.13

Example

```
static CEC_Factory "-CECConsumerControl reactive -CECProxyDisconnectRetries 3"
```

23.6.2.10 CECProxySupplierCollection *flags*

Description This option controls the type of collection the event channel uses to hold supplier proxies. The flags passed describe the characteristics of the desired collection. Colons should separate the flags (e.g., `mt:list`). The flags are described in Table 23-9. Only one flag per type should be specified.

Table 23-9 Collection Type Flags

Flag	Type	Description
`mt` (default)	Synchronization	Use a thread-safe lock for the collection.
`st`		Use a null lock for the collection.
`list` (default)	Collection	Implement the collection using an ordered list.
`rb_tree`		Implement the collection using a red-black tree.
`immediate` (default)	Iterator	Threads block until they can execute a change to the collection.
`copy_on_read`		Before initiating an iteration of the collection, a copy of the complete collection is performed.
`copy_on_write`		Before initiating a modification to the collection, a copy of the complete collection is performed.
`delayed`		Changes that cannot be made immediately are queued for later execution.

For a more detailed discussion of the collection types see 23.6.1.4.

Usage Applications that can guarantee that a supplier proxy collection is only accessed from a single thread can specify the `st` flag to improve performance.

Event channels that connect and disconnect consumers often, and wish to optimize these operations (at the expense of iteration speed), should specify the `rb_tree` flag.

Applications that use the `immediate` flag must guarantee that the thread iterating over the proxy collection does not attempt to modify the collection, as this invalidates the iterator. One way to insure that such a collection is not modified is to specify a separate dispatching thread. If you wish to minimize priority inversions between publication and consumer connections and disconnections, use the `delayed` flag. Collections using the `copy_on_read` flag are only applicable to systems with small numbers of supplier proxies that require low latencies for proxy collection modifications. Collections using the

`copy_on_write` flag are only applicable to systems with small numbers of supplier proxies that require low latencies for proxy collection iterations.

Impact The `mt` flag incurs additional overhead over the `st` flag during connection/disconnection of consumers and iteration over the collection.

List-based collections result in slower updates to the collection. Red-black tree collections are slower during iteration over the collection.

Immediate update of supplier proxy collections (during connection or disconnection of consumers) may cause priority inversions because of the long-lived locks involved. Copy on read collections incur dynamic allocation and copy costs for each iteration of the proxy collection. Delayed updates to collections can result in long intervals between the requested change and its actual occurrence.

See Also 23.6.2.7

Example

```
static CEC_Factory "-CECProxySupplierCollection mt:delayed"
```

23.6.2.11 CECProxySupplierLock *lock_strategy*

Values for *lock_strategy*

`null`	Do not use any locking on the proxy suppliers.
`thread` (default)	Use a thread-safe lock on the proxy suppliers.
`recursive`	Use a recursive thread-safe lock on the proxy suppliers.

Description This option defines the type of lock to use in synchronizing access to the proxy supplier objects.

Usage Single-threaded applications can use the `null` lock to increase the efficiency of the push supplier. Multithreaded applications may need to set the lock to `recursive` in cases where operations on the proxy supplier may cause recursive access to the proxy supplier. In all other situations, the `thread` lock should be used.

Impact The `null` lock causes problems in applications that access the proxy from more than one thread. The `thread` lock causes additional locking overhead that may not be needed in applications that restrict proxy access to a single thread. The `recursive` lock is even more expensive than the `thread` lock, but is required by applications that must recursively access the lock.

See Also 23.6.2.8

Example

```
static CEC_Factory "-CECProxySupplierLock recursive"
```

23.6.2.12 CECReactivePullingPeriod *period*

Description Set the period (in microseconds) the reactive pulling strategy uses to poll all the pull suppliers for events. The default period is 5000000 (5 seconds).

Usage The reactive pulling strategy periodically attempts to pull events from each pull supplier attached to an event channel. This option allows applications to customize the period between attempted pulls. These periodic attempts are made using the reactor associated with the default factory's ORB. These pull requests use a default timeout value of ten milliseconds. Alternate timeout values are specified via the `-CECSupplierControlTimeout` option.

Impact Shorter periods incur greater bandwidth and processing demands. Longer periods cause greater delays between the availability of events and their eventual delivery.

See Also 23.6.2.17, 23.6.2.15

Example

```
static CEC_Factory "-CECReactivePullingPeriod 1000000"
```

23.6.2.13 CECSupplierControl *control_policy*

Values for *control_policy*

`null` (default)	Do not discard dangling suppliers.
`reactive`	Use a reactive policy to discard dangling suppliers.

Description This option specifies the policy used when dealing with *dangling* suppliers. The `null` control policy never disconnects ill-behaved suppliers.

The `reactive` policy disconnects suppliers after a number of communication failures. A communication failure is either the failure of the periodic ping performed on the supplier or any other failure to correctly invoke an operation on the supplier. By default, the first communication failure results in the disconnection of the supplier. The number of retries allowed can be set via the `-CECProxyDisconnectRetries` option.

Usage Use the `reactive` control policy when suppliers could possibly be destroyed without disconnecting. Use the default control policy (`null`) when you can guarantee that all suppliers disconnect properly (or not at all), and you do not want to incur the overhead of the reactive policy.

Impact The `null` supplier control strategy causes degraded throughput when suppliers are destroyed without first disconnecting. The `reactive` strategy requires slightly more overhead in normal operation, may result in suppliers having to reconnect when the network quality is bad, and requires slightly more memory.

See Also 23.6.2.1, 23.6.2.14, 23.6.2.15, 23.6.2.9

Example

```
static CEC_Factory "-CECSupplierControl reactive"
```

23.6.2.14 CECSupplierControlPeriod *period*

Description Sets the period (in microseconds) that the reactive supplier control policy uses to poll the state of the suppliers. The default period is 5000000 (5 seconds). A value of zero disables supplier state polling.

Usage For event channels using the `reactive` supplier control policy, use this option to control the time to wait between attempted *pings* on each supplier. The reactive supplier control strategy object pings the supplier by invoking `CORBA::Object::_non_existent()` on the supplier's object reference; this is a synchronous call. The `-CECSupplierControlPeriod` option is ignored when the consumer control policy is not `reactive`.

Impact Shorter periods require more bandwidth and processing to validate the existence of the suppliers. Longer periods consume less of these resources. You can disable the ping altogether by setting the period to zero.

See Also 23.6.2.2, 23.6.2.13

Example

```
static CEC_Factory "-CECSupplierControl reactive -CECSupplierControlPeriod
1000000"
```

23.6.2.15 CECSupplierControlTimeout *timeout*

Description Sets the relative round-trip timeout (in microseconds) used for polling suppliers. This timeout is used both for polling pull model suppliers and by the `reactive` supplier control strategy. The default timeout is 10000 (10 milliseconds).

Note *The* `-CECSupplierControlRoundtripTimeout` *is an alias for the* `-CECSupplierControlTimeout` *option.*

Usage For event channels using the `reactive` supplier control policy, use this option to control the time the event channel waits for a supplier to respond to an attempted *ping*. For pull suppliers, use this option to control the time the event channel waits for each `pull()` call to a pull supplier. The reactive supplier control strategy object pings the supplier by invoking `CORBA::Object::_non_existent()` on the supplier's object reference; this is a synchronous call. Failure to respond within the specified timeout period results in the event channel classifying that ping as a communication failure for that supplier.

Impact Smaller timeout values may result in more timeout failures and suppliers being disconnected more often. A larger timeout value means it takes longer to detect and remove dead suppliers.

See Also 23.6.2.13, 23.6.2.3

Example

```
static CEC_Factory "-CECSupplierControl reactive -CECSupplierControlTimeout
50000"
```

23.6.2.16 CECSupplierOperationTimeout *timeout*

Description Sets the relative round-trip timeout (in microseconds) that the reactive supplier control policy uses for all operations besides the polling operation. The default timeout is zero, which is evaluated as no timeout. The timeout for the polling operation is covered by the `-CECSupplierControlTimeout` option.

Usage For event channels using the `reactive` supplier control policy, use this option to control the time the event channel waits for a supplier to respond to all operations besides the *ping*. This mainly affects the `pull()` call for pull suppliers.

Failure to respond within the specified timeout period results in the event channel classifying that request as a communication failure for that supplier. The `-CECSupplierOperationTimeout` option is ignored when the supplier control policy is not `reactive`.

Impact Smaller timeout values may result in more timeout failures and suppliers being disconnected more often. A larger timeout value means it takes longer to detect and remove dead suppliers.

See Also 23.6.2.1, 23.6.2.3

Example

```
static CEC_Factory "-CECSupplierControl reactive -CECSonsumerOperationTimeout
50000"
```

23.6.2.17 CECUseORBId *orb-id*

Description Sets the name of the ORB used by the default factory implementation. The default factory creates strategy objects that use this ORB to perform remote invocations and to gain access to the ORB's reactor.

Usage This option is only useful in applications that create multiple ORBs and activate the event channel in one of them. Use it to ensure that the objects created by the default factory use the same ORB as the event channel and related objects.

Impact This option may cause the creation of a new ORB (and associated resources), if the ORB with the given name has not been initialized.

Example

```
static CEC_Factory "-CECUseORBId Orb2"
```

CHAPTER 24

Real-Time Event Service

24.1 Introduction

Though the OMG Event Service defines the basic interfaces necessary for decoupling suppliers and consumers of data, it leaves a number of important details out, particularly in areas affecting real-time systems. The TAO Real-Time Event Service (RTES) addresses a number of these issues by adding support for features such as event filtering, event correlation, real-time event dispatching/scheduling, and periodic event processing. Many non-real-time applications also benefit from using the RTES as a fast and predictable event distribution mechanism with additional features not included in the OMG Event Service.

24.2 Overview of the TAO Real-Time Event Service

The TAO Real-Time Event Service's interfaces for publishing and receiving events remain mostly the same as in the standard OMG Event Service. The programmer must provide additional information when creating an event channel and when registering suppliers and consumers to allow the TAO

Real-Time Event Service to function as desired. The Real-Time Event Service only supports the push model of operation.

Figure 24-1 shows the basic architecture of the TAO Real-Time Event Service:

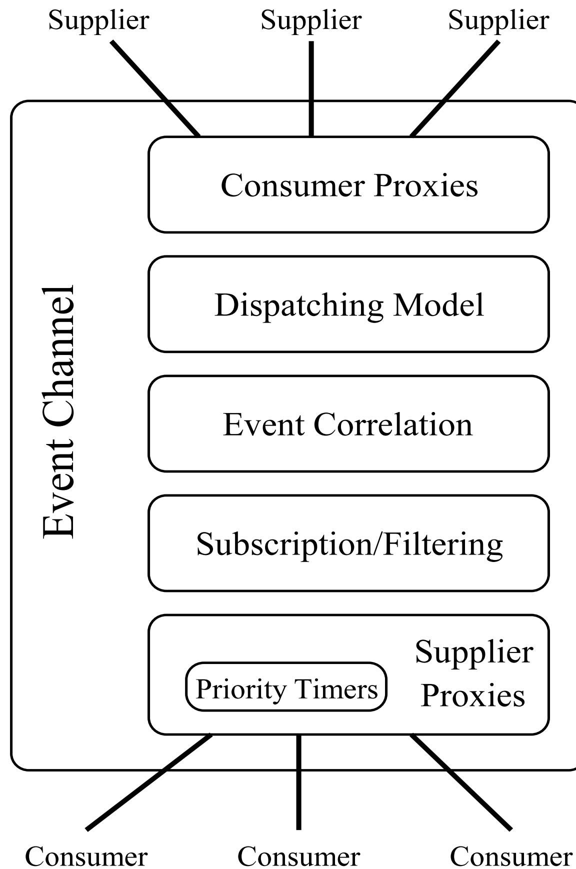
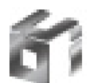


Figure 24-1 Real-Time Event Service Architecture

Each element of the Real-Time Event Service architecture is described below.

24.2.1 Consumer Proxies

This module implements a `SupplierAdmin` interface analogous to that of the OMG Event Service. Suppliers use this interface to create objects that support



the `ProxyPushConsumer` interface. Suppliers use the `ProxyPushConsumer` interface to connect and disconnect from the event channel as well as publish events via the `push()` operation.

24.2.2 Priority Timers

The RTES allows consumers to define timeout events that occur when a desired duration has elapsed. These timeout events can be made dependent upon the occurrence of other events. This module generates and manages these timeout events.

24.2.3 Subscription and Filtering

This module handles distribution of events to the consumers. TAO adds support for event filtering on consumers based on the source and/or type of the event published.

24.2.4 Event Correlation

This module correlates events based on details the consumer provides. The consumer may request to only receive particular events when some other related event has also occurred. In this case the Event Correlation module holds the first event and delivers them both when the second is published.

24.2.5 Dispatching Module

This module determines when to deliver events based on a variety of information the user provides.

24.2.6 Supplier Proxies

This module implements the `ConsumerAdmin` interface analogous to that of the OMG Event Service. Consumers use this interface to create objects that support the `ProxyPushSupplier` interface. The `ProxyPushSupplier` interface is used to connect and disconnect from the event channel. TAO extends this interface to allow the consumer to indicate the dependencies and other details the Real-Time Event Service needs.

24.2.7 Real-Time Event Service Libraries

The RTES functionality is split into the libraries defined in Table 24-1. Applications that are pure clients of the RTES IDL can simply link with the

TAO_RTEvent library. Applications that are building servants based on RTES IDL (such as consumers), need to also link with the TAO_RTEvent_Skel library. All examples and features in this section require this library to be linked with your application. When other libraries are required we will explicitly state the library required. The TAO_RTEvent library contains the code necessary to include event channels in your application.

Table 24-1 Real-Time Event Service Libraries

Library Name	Description
TAO_RTEvent	Basic RT Event Service client functionality (stub and client code).
TAO_RTEvent_Skel	Basic RT Event Service functionality for building servers (skeletons).
TAO_RTEvent_Serv	Basic RT Event Service server functionality (event channels, etc.).
TAO_RTKokyuEvent	Allows integration of RTES with the Kokyu Scheduling Framework.
TAO_RTSchedEvent	Allows integration of RTES with TAO's original scheduling service.
TAO_RTCORBAEvent	Allows use of the Real-Time CORBA thread pools for RTES dispatching.

24.3 Using the TAO Real-Time Event Service

The examples in this chapter explore some of the features that the TAO Real-Time Event Service introduces. The Naming Service server must be started prior to running these examples and the other processes must be configured to use the same naming server (either via multicast, the NameServiceIOR environment variable, or the -ORBInitRef option).

24.3.1 A Basic Example

This example is functionally the same as the basic example in Chapter 23. The main differences are accounted for by the fact that the Real-Time Event Service defines interfaces for the Event Channel, Supplier, Consumer, and Admin concepts that are analogous to the OMG Event Service interfaces, but the RTES operations take slightly different parameters than the standard event service equivalents. Full source code for this example is in the TAO source

code distribution in the directory
`$TAO_ROOT/orbsvcs/DevGuideExamples/EventServices/RTEC_Basic`.

24.3.1.1 Starting the tao_rtevent Process

This example utilizes the `tao_rtevent` server that is delivered with TAO. This server creates a single RTES event channel object and binds it in the root naming context of the Naming Service. By default, it binds to the name "EventService". The options that can be passed to this server are covered in 24.4. A typical invocation of the server is:

```
$TAO_ROOT/orbsvcs/Event_Service/tao_rtevent
```

This starts an Event Channel with a local Scheduler and binds both in the root naming context of the Naming Service. See the Scheduling Service documentation for additional information regarding the scheduler.

24.3.1.2 The Supplier

To use the RTES, your supplier must include the header files that the IDL compiler generated from the Real-Time Event Service's IDL. This IDL file defines interfaces that are very similar to those defined by the standard OMG Event Service. To differentiate the interfaces, the module and file names of TAO's Real-Time Event Service are given the `Rtec` prefix. Listed below are the include directives for `EchoEventSupplierMain.cpp`.

```
#include <orbsvcs/RtecEventCommC.h>
#include <orbsvcs/RtecEventChannelAdminC.h>
```

After initializing the ORB and resolving and narrowing the Naming Service's root context, this context is used to retrieve an object reference to the event channel using the name "EventService". This event channel reference is used to get a supplier administration object reference that is then used to create a consumer proxy. All of this works as in the standard OMG Event Service, except for the `Rtec` prefix on the modules/namespaces in the source code.

```
// Get the Event Channel using the Naming Service
CORBA::Object_var obj = orb->resolve_initial_references("NameService");
CosNaming::NamingContextExt_var root_context =
                   CosNaming::NamingContextExt::_narrow(obj.in());
obj = root_context->resolve_str("EventService");
```

```
    // Narrow the object reference to an EventChannel reference.
    RtecEventChannelAdmin::EventChannel_var ec =
      RtecEventChannelAdmin::EventChannel::_narrow(obj.in());
    if (CORBA::is_nil(ec.in())) {
      std::cerr << "Could not resolve Event Channel." << std::endl;
      return 1;
    }

    // Get a SupplierAdmin object from the EventChannel.
    RtecEventChannelAdmin::SupplierAdmin_var admin = ec->for_suppliers();

    // Get a ProxyPushConsumer from the SupplierAdmin.
    RtecEventChannelAdmin::ProxyPushConsumer_var consumer =
      admin->obtain_push_consumer();
```

Next, create the supplier servant that implements the `RtecEventComm::PushSupplier` interface. Only the `disconnect_push_supplier()` operation is required for this object, although it is often given many other application-specific responsibilities.

```
    // Instantiate an EchoEventSupplier_i servant.
    PortableServer::Servant_var<EchoEventSupplier_i> servant =
      new EchoEventSupplier_i(orb.in());

    // Register it with the RootPOA.
    CORBA::Object_var poa_obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(poa_obj.in());
    PortableServer::ObjectId_var oid = poa->activate_object(servant.in());
    CORBA::Object_var supplier_obj = poa->id_to_reference(oid.in());
    RtecEventComm::PushSupplier_var supplier =
      RtecEventComm::PushSupplier::_narrow(supplier_obj.in());
```

To connect to the event channel using the Real-Time Event Service, the supplier must provide some quality of service (QoS) information. This is created using a QoS factory object. Information about the events that this supplier publishes are inserted into the QoS factory, which builds up an internal data structure that is later passed to `connect_push_supplier()`.

```
   const RtecEventComm::EventSourceID MY_SOURCE_ID = ACE_ES_EVENT_SOURCE_ANY + 1;
   const RtecEventComm::EventType MY_EVENT_TYPE = ACE_ES_EVENT_UNDEFINED + 1;

    // Publish the events the supplier provides.
    ACE_SupplierQOS_Factory qos;
    qos.insert (MY_SOURCE_ID,      // Source ID
                MY_EVENT_TYPE,     // Event type
                0,                 // handle to the rt_info structure
```

```
            1);                // number of calls

    // Connect as a supplier of the published events.
    consumer->connect_push_supplier (supplier.in (),
                                     qos.get_SupplierQOS ());
```

When inserting the QoS data, the source identifier must be a unique non-zero long integer for each supplier in the system. User-defined event types must start at a number higher than the value defined by the preprocessor macro `ACE_ES_EVENT_UNDEFINED`. Passing the preprocessor macro `ACE_ES_EVENT_ANY` as the event type allows a supplier to publish events of any type.

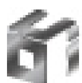
Passing zero as the handle to the real-time information (`rt_info`) data structure tells the event channel to just deliver the events in the order received. The `rt_info` data structure is the main mechanism for controlling the real-time capabilities of the real-time Event Service. See the appropriate Scheduling Service documentation for more details.

Once the supplier is connected, the event data can be created. Note that the RTES-based interfaces publish sets of events and that these events have more fields than the OMG Event Service's events. Here an `EventSet` is created with one event and a payload of a string stored in the `any_value` member:

```
    CORBA::String_var eventData = CORBA::string_dup("Hello, world.");

    // Create an event set for one event
    RtecEventComm::EventSet events (1);
    events.length (1);

    // Initialize event header.
    events[0].header.source = MY_SOURCE_ID;
    events[0].header.type = MY_EVENT_TYPE;

    // Initialize data fields in event.
    events[0].data.any_value <<= eventData;
```

Publishing the events is done using the `push()` operation on the proxy consumer interface:

```
    consumer->push (events);
```



24.3.1.3 Implementing the Push Consumer Interface

To create a consumer, the `RtecEventComm::PushConsumer` IDL interface must be implemented. There are only two operations in this interface, `push()` and `disconnect_push_consumer()` as shown in the file `EchoEventConsumer_i.h` for this example.

```
#include <orbsvcs/RtecEventCommS.h>

class EchoEventConsumer_i : public virtual POA_RtecEventComm::PushConsumer
{
  public:
    // Constructor
    EchoEventConsumer_i(CORBA::ORB_ptr orb);

    // Override operations from PushConsumer interface.
    virtual void push(const RtecEventComm::EventSet& events);

    virtual void disconnect_push_consumer();

  private:
    CORBA::ORB_var orb_;
};
```

The implementations for these operations are contained in the file `EchoEventConsumer_i.cpp`. The `push()` operation implementation is the only code that significantly differs from an OMG Event Service push consumer:

```
// Implement the push() operation.
void EchoEventConsumer_i::push(const RtecEventComm::EventSet& events)
{
  for (u_int i = 0; i < events.length (); ++i) {
    // Extract event data from the any.
    const char* eventData;
    std::cout << "Received event,"
              << "  type: "   << events[i].header.type
              << "  source: " << events[i].header.source;
    if (events[i].data.any_value >>= eventData) {
      std::cout << "  text: "   << eventData;
    }
    std::cout << std::endl;
  }
}
```

The first difference is that the `push()` operation must now process a set of events instead of a single event. Secondly, instead of a simple `CORBA::Any`

parameter, the Real-Time Event Service's `Event` data structure has significantly more fields. In this case, the supplier has used the `any_value` field embedded in the `Event`'s body to pass a string to the consumer. The consumer also extracts the `type` and `source` fields from the header portion of the `Event` data structure.

24.3.1.4 Creating the Consumer and Connecting to the Channel

The consumer application accesses the event channel using the same code as shown previously for the supplier (refer to `EchoEventConsumerMain.cpp` for this example). The event channel is then used to obtain a proxy push supplier:

```
// Obtain a reference to the consumer administration object.
RtecEventChannelAdmin::ConsumerAdmin_var admin = ec->for_consumers();

// Obtain a reference to the push supplier proxy.
RtecEventChannelAdmin::ProxyPushSupplier_var supplier =
  admin->obtain_push_supplier();
```

Now an instance of the consumer servant can be created and connected to the event channel using QoS information similar to that seen on the supplier side. The QoS is created using a consumer QoS factory and the only information required is an event type and a handle to the real-time information data structure. This example uses as the event type `ACE_ES_EVENT_ANY`, which is a wildcard that accepts all event types. It passes `0` for the handle to the real-time information, causing the event channel to deliver the events in the order received.

```
// Get the RootPOA.
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());

// Instantiate an EchoEventConsumer_i servant.
PortableServer::Servant_var<EchoEventConsumer_i> servant =
  new EchoEventConsumer_i(orb.in());

// Register it with the RootPOA.
PortableServer::ObjectId_var oid = poa->activate_object(servant.in());
CORBA::Object_var consumer_obj = poa->id_to_reference(oid.in());
RtecEventComm::PushConsumer_var consumer =
  RtecEventComm::PushConsumer::_narrow(consumer_obj.in());

// Connect as a consumer.
```

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_type (ACE_ES_EVENT_ANY,  // Event Type
                 0);                // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

After activating the POA manager and starting the event loop the application is ready to receive events.

```
// Activate the POA via its POAManager.
PortableServer::POAManager_var poa_manager = poa->the_POAManager();
poa_manager->activate();

std::cout << "EchoEventConsumerMain.cpp: Ready to receive events..."
          << std::endl;

// Enter the ORB event loop.
orb->run();

// If we have reached this, we must be shutting down...
// Disconnect the ProxyPushSupplier.
supplier->disconnect_push_supplier();
supplier = RtecEventChannelAdmin::ProxyPushSupplier::_nil();
admin = RtecEventChannelAdmin::ConsumerAdmin::_nil();

orb->destroy();
```

All events from the event channel are now delivered to the consumer object's `push()` operation.

24.3.2 Managing Connections

The real-time event service interfaces provide operations for consumers and suppliers to manage their connections and control the delivery of events.

24.3.2.1 Connecting and Disconnecting Consumers

The previous example connected its consumer using the `connect_push_consumer()` operation on the `ProxyPushSupplier` interface:

```
supplier_proxy->connect_push_consumer (consumer.in (),
                                       qos.get_ConsumerQOS ());
```

Once this has executed, the consumer remains connected to the event channel until the corresponding disconnect operation is called:

```
supplier_proxy->disconnect_push_supplier ();
```

This disconnects the consumer from the event channel. Failure to disconnect from the event channel before ending the process or destroying the consumer leaves references to the consumer in the event channel. By default, the event channel continues to attempt to push events to the consumer, but fails each time. See 24.5.1.4 for details on how to configure the real-time event channel so as to ensure automatic removal of such “dangling” consumer object references.

24.3.2.2 Connecting and Disconnecting Suppliers

Suppliers are connected and disconnected using operations on the proxy push consumer that are analogous to operations on the proxy push supplier:

```
consumer_proxy->connect_push_supplier (supplier.in (),
                                       qos.get_SupplierQOS ());

// Do all the publication required...

consumer_proxy->disconnect_push_consumer ();
```

Failure to disconnect from the event channel leads to similar problems in the supplier case. Instead of affecting each push though, it generally affects the speed of connection and disconnection by consumers. See 24.5.1.4 for details on how to configure the real-time event channel so as to ensure automatic removal of “dangling” supplier object references.

24.3.2.3 Reconnecting Consumers and Suppliers

To change details of their connection (such as filter information or timeouts), consumers and suppliers can invoke the appropriate connect operation even after a connection is established. This is more efficient than explicitly disconnecting and reconnecting. For example, consider the following:

```
supplier_proxy->connect_push_consumer (consumer.in (),
                                       qos.get_ConsumerQOS ());

// Process events for a while...
```

```
supplier_proxy->connect_push_consumer (consumer.in (),
                                       qos2.get_ConsumerQOS ());
```

If we assume that `qos` and `qos2` refer to different subscription details such as filtering for different event types then this code changes the subscription while remaining connected to the event channel. No events should be lost during this change.

Note *To support this feature the event channel must have the attributes* `consumer_reconnect` *and* `supplier_reconnect` *set to true. See 24.3.6.2 for details on setting these attributes. Attempting to reconnect if these attributes are not set to true results in the* `connect_*()` *operation throwing an* `RtecEventChannelAdmin::AlreadyConnected` *exception.*

24.3.2.4 Suspending and Resuming Consumer Connections

If a consumer wants to stop processing events for a time, it could remain connected to the event channel and simply ignore events as they arrive. However, this wastes network bandwidth and processor resources. A more efficient alternative is to temporarily suspend the connection, then later resume the connection. To suspend the consumer's connection to the event channel, use the `suspend_connection()` operation, which causes the event channel to stop pushing events to the consumer. You can later resume the connection and again receive events by invoking the `resume_connection()` operation.

Another approach is to disconnect from the event channel and reconnect. However, the `suspend_connection()` and `resume_connection()` operations are more efficient because they leave the subscription in place (and thereby avoid locking the proxy collection so it can be changed). This allows the event channel to resume the subscription much faster than establishing a new subscription.

```
supplier_proxy->suspend_connection ();

// Do something else for a while...

supplier_proxy->resume_connection ();
```

The `suspend_connection()` operation sets a flag in the proxy supplier of the event channel that blocks all publications until the

`resume_connection()` operation is called and resets the flag. Any events published while the connection is suspended are lost to that consumer.

There is a slight overhead involved in keeping the suspended subscriptions in the event channel that may make it desirable to disconnect when doing so for long periods of time. The suspend/resume mechanism is an attractive solution for brief suspensions.

24.3.3 The Event Structure

The TAO Real-Time Event Service uses event structures that are defined in `$TAO_ROOT/orbsvcs/orbsvcs/RtecEventComm.idl` in the `RtecEventComm` module. Here are the basic definitions of `EventSet` and `Event`:

```
struct Event
{
  EventHeader header;
  EventData data;
};
typedef sequence<Event> EventSet;
```

24.3.3.1 The EventHeader Structure

Table 24-2 describes the members of the `EventHeader` structure.

Table 24-2 EventHeader Members

Type (typedef type)	Name	Description
`EventType (long)`	`type`	User-defined type field.
`EventSourceID (long)`	`source`	User-defined source id.
`long`	`ttl`	Time to live count for federations/gateways.
`Time (TimeBase::TimeT)`	`creation_time`	User-defined timing information.
`Time (TimeBase::TimeT)`	`ec_recv_time`	User-defined timing information.
`Time (TimeBase::TimeT)`	`ec_send_time`	User-defined timing information.

Setting the `type` and `source` members is generally sufficient for most simple applications.

The `ttl` member is a counter that is decremented each time an event is passed between individual event channels in a federation (see 24.3.8 for details).

When the counter reaches zero, the event is no longer passed to additional event channels in the federation.

Table 24-3 lists special values that are defined for `EventTypes`:

Table 24-3 EventType Special Values

Name	Description
`ACE_ES_EVENT_ANY`	Wild card value that matches all event types.
`ACE_ES_EVENT_SHUTDOWN`	Event type that can be used for shutdown events from the supplier.
`ACE_ES_EVENT_INTERVAL_TIMEOUT`	Event type for interval timeout events.
`ACE_ES_EVENT_DEADLINE_TIMEOUT`	Event type for deadline timeout events.
`ACE_ES_EVENT_UNDEFINED`	This value marks the beginning of the range of user-defined event types.

There are several other special values defined for internal use, all of which are values smaller than the value of the `ACE_ES_EVENT_UNDEFINED` preprocessor macro.

24.3.3.2 The EventData Structure

Table 24-4 describes the members of the `EventData` structure:

Table 24-4 EventData Members

Type (typedef type)	Name	Description
`long`	`pad1`	Pad field, improves alignment and performance.
`EventPayload(sequence<octet>)`	`payload`	User-defined payload using a sequence of octet. Users must supply their own marshaling.
`any`	`any_value`	User-defined payload using a `CORBA::Any` type.

The choice of whether to use the `any_value` or the `payload` member in a particular application is a design decision, based upon the type of application and the nature of the event data. In many situations, custom marshaling and demarshaling of the `EventPayload` may be more efficient than using a `CORBA::Any`.

24.3.3.3 Customizing the Event Structure

TAO provides several build options that allow removal of unneeded members from the event structure. Table 24-5 describes these options.

Table 24-5 Event Structure Build Options

Build Option	Description
TAO_LACKS_EVENT_CHANNEL_ANY	Removes the any_value member from the EventData structure.
TAO_LACKS_EVENT_CHANNEL_OCTET_SEQUENCE	Removes the payload member from the EventData structure.
TAO_LACKS_EVENT_CHANNEL_TIMESTAMPS	Removes the ec_recv_time and ec_send_time members from the EventHeader structure.

To take advantage of these optimizations, the TAO_RTEvent library and any executables using it (e.g., tao_rtevent) must be rebuilt. The flags can be set by adding them to the platform_macros.GNU file, uncommenting the appropriate lines in $TAO_ROOT/rules.tao.GNU, or passing them to make via the MAKEFLAGS environment variable or the command line. See Appendix A for more details on build flags and their use.

In addition, users can add their own members to the EventData structure in the file $TAO_ROOT/orbsvcs/orbsvcs/RtecDefaultEventData.idl. Once again, this requires that the TAO_RTEvent library and associated executables be rebuilt. To interoperate using the event channel, all processes must use the same event structure.

24.3.4 Filtering and Correlation

The TAO Real-Time Event Service gives consumers the capability to filter the events delivered over the event channel to specific consumers. Events can be filtered based on the event type and source id, as well as combinations of the type and source.

The above elements can also be combined in groups with a variety of semantics. *Conjunction* groups give the consumer the ability to delay the delivery of some events until all events in the group are available. Events that match the different elements of a conjunction group are queued until all elements of the group have been matched. Filters can also be combined in *disjunction* groups that deliver events every time *any* of the events in the

group are available. *Logical AND* groups require events to pass *all* filters in the group before they are delivered.

Conjunction, disjunction, and logical AND groups can also be nested within one another to create more complex filters (although the event channel may be configured so as to limit this nesting).

Negations allow individual filters (including groups) to have their logic *inverted* so that only events that fail a particular filter are passed to the consumer. The event channel architecture also supports *bit-mask* filters that enable faster filtering based on bit masks.

Source code for a simple filtering example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/EventServices/RTEC_Filter`.

24.3.4.1 Specifying and Constructing Filters

Filters are specified when connecting a consumer to the event channel via the `connect_push_consumer()` operation. The previous examples use an `ACE_ConsumerQOS_Factory` to construct the `ConsumerQOS` structure that was then passed to `connect_push_consumer()`.

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_type (ACE_ES_EVENT_ANY,  // Event Type
                 0);                // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

Here is the definition of the `ConsumerQOS` structure in IDL:

```
module RtecEventChannelAdmin {
  struct Dependency {
    RtecEventComm::Event event;
    RtecScheduler::handle_t rt_info;
  };
  typedef sequence<Dependency> DependencySet;
  struct ConsumerQOS {
    DependencySet dependencies;
    boolean is_gateway;
  };
  // Rest of the module...
};
```

The core of this structure is a sequence of events. Special event types are used to specify the different filter types. In 24.3.4.13, we discuss how to build custom filters.

When the `ConsumerQOS` is passed to the event channel, the event channel uses a filter builder to construct a tree of filter objects that is used to filter subsequent events. Each time an event is pushed, the event channel evaluates that event using the filter tree for each consumer. The tree of filters causes some events to be discarded (never delivered to that consumer), some to be queued (for potential later delivery to that consumer), and some to be immediately delivered.

The type of filter builder that the event channel uses is specified via the `-ECFiltering` option (see 24.5.2.8) and determines how the `ConsumerQOS` structure is translated into the filter object tree. For example, a *null* filter builder ignores the `ConsumerQOS` structure and builds a filter tree that passes all events. A *basic* filter builder supports all the basic types, but limits the nesting of filter groups (conjunction, disjunction, logical AND) to two levels. The *prefix* filter builder allows arbitrarily deep nesting, but requires additional information in the subscription to do so.

24.3.4.2 Filtering By Event Type

In the basic RTES example in 24.3.1, the consumer uses the special event type of `ACE_ES_EVENT_ANY` to accept events of all types:

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_type (ACE_ES_EVENT_ANY,  // Event Type
                0);                // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

By calling `insert_type()` with specific event types, consumers can elect to receive only events of those particular types:

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_type (MY_EVENT_TYPE,     // Event Type
                0);                // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

This consumer receives all events whose type field has a value of MY_EVENT_TYPE. All other events are not to be delivered to this consumer.

24.3.4.3 Filtering By Source ID

In the previous examples, no source IDs were specified in the QoS information so events from all sources are delivered by default. By calling insert_source(), the consumer can limit the delivery of events to those originating from a particular source or sources.

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_source (MY_SOURCE_ID,  // Source ID
                   0);            // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

The event channel now delivers to this consumer only those events whose source id field matches MY_SOURCE_ID.

24.3.4.4 Filtering By Source ID and Event Type Combinations

The consumer QoS factory also allows filtering based on specific Source Id/Event Type combinations via insert().

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert (MY_SOURCE_ID,   // Source ID
            MY_EVENT_TYPE,  // Event Type
            0);             // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

This causes all events of type MY_EVENT_TYPE with a source of MY_SOURCE_ID to be delivered to this consumer. All other combinations of event types and source identifiers are ignored.

Note *When calling* insert()*, be sure to pass the source identifier first and the event type second as these are easily swapped and the compiler does not provide a warning.*

24.3.4.5 Bit-Mask Filters

The bit-mask value filter allows users to apply (bitwise AND) bit masks to the source and event type fields and attempts to match the result with specified values. Here is an example of a consumer with a bit-mask value filter:

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_bitmasked_value (0xFFFFFFFF,  // Source mask
                            0x0000FFFF,  // Type mask
                            0x00000020,  // Source value
                            0x00000000)  // Type value
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

This code results in a filter that performs a bitwise AND of the source mask (`0xFFFFFFFF`) and the source id of the event and compares the result with source value passed (`0x00000020`). It also does a bitwise AND of the type mask (`0x0000FFFF`) and the type of the event and compares the result with the type value passed (`0x00000000`). The end result is a filter that passes all events with a source id of 32 (`0x00000020`) and an event type with none of the lower 16 bits set.

24.3.4.6 Disjunction Groups

So far, the examples in this chapter have utilized a disjunction group that contains a single filter. Invocations of `insert_type()`, `insert_source()`, and `insert()` create filters of the appropriate types. Each disjunction group can contain an unlimited number of these filters.

Disjunction groups cause the event channel to deliver an individual event whenever any of the filters contained within the group “match” the event. To start a disjunction group, invoke `start_disjunction_group()` and pass it the number of child filters that it will contain. The number of child filters argument can be omitted if the basic filter builder is used (see 24.3.4.10 for details).

Here is an example of a more complex disjunction group.

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (5);
qos.insert_type (MY_EVENT_1,     // Event Type
                 0);             // handle to the rt_info
qos.insert_type (MY_EVENT_2,     // Event Type
```

```
                 0);             // handle to the rt_info
qos.insert_source (MY_SOURCE_1,  // Source ID
                   0);           // handle to the rt_info
qos.insert_source (MY_SOURCE_2,  // Source ID
                   0);           // handle to the rt_info
qos.insert (MY_SOURCE_3,         // Source ID
            MY_EVENT_3,          // Event Type
            0);                  // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

This consumer receives all events of type MY_EVENT_1 and MY_EVENT_2, all events from MY_SOURCE_1 and MY_SOURCE_2, and all events of type MY_EVENT_3 from MY_SOURCE_3. Each of these events is delivered as it is pushed to the event channel.

24.3.4.7 Conjunction Groups

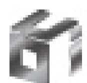
Construction of conjunction groups is accomplished in a similar manner:

```
ACE_ConsumerQOS_Factory qos;
qos.start_conjunction_group (2);
qos.insert_type (MY_EVENT_1,  // Event Type
                 0);          // handle to the rt_info
qos.insert_type (MY_EVENT_2,  // Event Type
                 0);          // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

Instead of delivering each matching event as it is received, the conjunction group causes the event channel to queue events until an event matching each filter in the conjunction is available. At that point, the event channel delivers all events in the group to the consumer (in the same EventSet). In the above example, if the event channel first receives an event of type MY_EVENT_1, the event channel will queue that event until an event of type MY_EVENT_2 is received. At that point, the event channel will deliver both events to the consumer via one call to the consumer's push() operation.

This feature gives the consumer the ability to correlate related events and process them as a group. In some applications, this may increase the performance of the system and reduce the workload for the consumer.

The conjunction only queues one event for each corresponding filter in the group. Any subsequent events that match that filter are discarded until the



conjunction is satisfied and its events delivered. In the above example, this means that once a MY_EVENT_1 type event is queued in the event channel, all subsequent events of this type are discarded until a MY_EVENT_2 event is received and the conjunction is satisfied.

24.3.4.8 Logical AND Groups

Construction of logical AND groups is done in a similar manner to disjunction and conjunction groups:

```
ACE_ConsumerQOS_Factory qos;
qos.start_logical_and_group (2);
// This bitmask matches events from any source with none of the high 16 bits of
// the event type set
qos.insert_bitmasked_value (0x00000000,   // Source mask
                            0xFFFF0000,   // Type mask
                            0x00000000,   // Source value
                            0x00000000);  // Type value
qos.insert_source (MY_SOURCE,    // Source ID
                   0);           // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

This code causes the consumer to only receive events that pass the source *and* bit-mask value filters specified (it passes all events from MY_SOURCE with none of the high 16 bits set on the event type).

24.3.4.9 Negating the Logic of Filters

Filter logic can be inverted with the use of negations. If a given tree of filters would normally pass one group of events and filter another, then negation of that tree of filters would pass the events formerly filtered and filter the ones formerly passed. Negations can only directly contain a single filter (although that filter may be a group that contains other filters). Here is an example of a negation filter.

```
ACE_ConsumerQOS_Factory qos;
qos.start_negation ();
qos.insert_source (MY_SOURCE,    // Source ID
                   0);           // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

This consumer receives events from any source other than MY_SOURCE.

24.3.4.10 Nesting Groups

The TAO Real-Time Event Service supports nesting of conjunction, disjunction, logical AND, and negation groups to arbitrarily deep levels. The default filter builder (basic) only supports two levels of nesting with the top level being restricted to an implicit disjunction group. The start of one group terminates the previous group. Here is an example:

```
ACE_ConsumerQOS_Factory qos;
qos.start_conjunction_group (2);
qos.insert_type (MY_EVENT_1, 0);
qos.insert_type (MY_EVENT_2, 0);
qos.start_conjunction_group (2);
qos.insert_type (MY_EVENT_3, 0);
qos.insert_type (MY_EVENT_4, 0);
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

This code creates two conjunction groups nested inside of the implicit disjunction group. This consumer is delivered pairs of events (types 1 and 2 together and types 3 and 4 together) as the event channel collects the appropriate matching events.

Because the basic filter builder assumes that the start of a subsequent group means the end of the previous one, the arguments to the functions that start groups can be omitted. In this case, the filter builder calculates the number of children. However, we strongly recommend the explicit use of the arguments to the start methods. If they are omitted and the prefix filter builder is specified at run time (via the `ECFiltering` service configuration option, see 24.5.2.8), the event channel will construct groups with no children and potentially create a filter with different semantics than those intended.

The prefix filter builder supports arbitrarily complex nesting, but requires the arguments to the start group functions that specify the number of children. Always specify the number of immediate children and not the children contained in enclosed groups. Here is an example of a more complex filter with the groups indented to show the intended structure.

```
ACE_ConsumerQOS_Factory qos;
qos.start_logical_and_group (2);
  qos.start_disjunction_group (2);
    qos.insert_source (MY_SOURCE_1, 0);
    qos.insert_source (MY_SOURCE_2, 0);
  qos.start_disjunction_group (2);
    qos.insert_type (MY_EVENT_1, 0);
```

```
    qos.insert_type (MY_EVENT_2, 0);
qos.start_logical_and_group (2);
  qos.start_negation ();
    qos.insert_source (MY_SOURCE_3, 0);
  qos.start_disjunction_group (2);
    qos.insert_type (MY_EVENT_3, 0);
    qos.insert_type (MY_EVENT_4, 0);
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

Note *Nesting disjunction groups does not make sense for any of the basic filter types discussed so far as it essentially creates one large disjunction group. The only time this structure is needed is when deadline timeouts are being specified. An example of this is shown in 24.3.5.2.*

24.3.4.11 Quick Rejection Filters Using Bit Masks

A bit-mask filter can be used to quickly reject certain events via bitwise AND operations with source and event type bit masks. Events that have non-zero values for the source and event type after applying the bit masks are passed to the enclosed filter. Events that result in a value of zero for either the source or event type are rejected. Bit-mask filters can only directly contain a single filter (although that filter may be a group that contains other filters). Here is an example of a consumer that uses a bit-mask filter.

```
ACE_ConsumerQOS_Factory qos;
qos.start_bitmask (0x0000000F,   // Source mask
                   0x0000000F);  // Type mask
qos.start_disjunction (2);
qos.insert_source (MY_SOURCE,    // Source ID
                   0);           // handle to the rt_info
qos.insert_source (MY_SOURCE_2,  // Source ID
                   0);           // handle to the rt_info
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

This consumer quickly filters out events that have none of the lower four bits set for the source or event type fields. Events with some of these bits set are passed to the disjunction filter. The end result is that all events from `MY_SOURCE` and `MY_SOURCE_2` with an event type that has any one of the lowest four bits set are delivered.

24.3.4.12 Null Filters

A null filter will match any event. This type of filter is useful in combination with the bit-mask filter when no further action beyond the bit-mask operation is desired. For example, the following filter group would simply accept all events that have at least one of the four lower order bits set on the source and event type:

```
ACE_ConsumerQOS_Factory qos;
qos.start_bitmask (0x0000000F,   // Source mask
                   0x0000000F);  // Type mask
qos.insert_null_terminator ();
supplier->connect_push_consumer (consumer.in (),
                                 qos.get_ConsumerQOS ());
```

This consumer receives all events that pass the bit-mask filter.

24.3.4.13 Constructing Filters By Hand

For most applications, consumer filters will be constructed using `ACE_ConsumerQOS_Factory`. However, in some cases it may be necessary to create custom filters. One such case would arise in an application that uses the TAO Real-Time Event Service remotely, but the application itself does not use TAO and is therefore unable to link with the TAO libraries. In this case, the `ACE_ConsumerQOS_Factory` operations would not be available to the application.

In an application that uses the factory to construct a filter, each call to a start or insert member function on the `ACE_ConsumerQOS_Factory` object causes the addition of a dependency structure to the `ConsumerQOS` object that the factory manages. This dependency structure is initialized with an `RtecEventComm::Event` structure that describes the filter being added. When `get_ConsumerQOS()` is called on the factory, it returns the managed `ConsumerQOS` object.

Construction of custom filters that do not use the factory is shown in the following example. This example uses the prefix filter builder. See 24.5.2.8 for how to configure this option.

```
// Create the consumer qos object (describes the filter)
RtecEventChannelAdmin::ConsumerQOS qos;
qos.is_gateway = 0;             // This is not a gateway subscription
qos.dependencies.length (2);    // set the length of the sequence
```

```
// Specify a disjunction group containing one filter
qos.dependencies[0].event.header.type = ACE_ES_DISJUNCTION_DESIGNATOR;
qos.dependencies[0].event.header.source = 1;  // # of filters in the group
qos.dependencies[0].rt_info = 0;

// Specify a filter that accepts events with type of MY_TYPE_1 from any source
qos.dependencies[1].event.header.type = MY_TYPE_1;
qos.dependencies[1].event.header.source = ACE_ES_EVENT_SOURCE_ANY;
qos.dependencies[1].rt_info = 0;
supplier->connect_push_consumer (consumer.in (), qos);
```

A list of the special event types (e.g., `ACE_ES_DISJUNCTION_DESIGNATOR`) can be found in `$TAO_ROOT/orbsvcs/orbsvcs/Event_Service_Constants.h`. More information about the structures associated with specific filter types can be found in the code for the `ACE_ConsumerQOS_Factory` class in the `$TAO_ROOT/orbsvcs/orbsvcs/Event_Utilities.*` files. Note that some filters (such as bit mask and bit-mask value filters) require more than one dependency structure.

Note *Manually building filters is tricky and extremely error prone. A helpful trick is to use the* `ACE_ConsumerQOS_Factory` *helper to construct the required filter on a platform with TAO, print out the contents of the resulting* `RtecEventChannelAdmin::ConsumerQOS` *structure, and use it to validate your manually constructed filter.*

24.3.5 Timeouts

The TAO Real-Time Event Service also supports the generation of timeout events based on various consumer/supplier criteria. The basic timeouts supported are *interval* timeouts and *deadline* timeouts. Interval timeouts generate a timeout event every time a fixed duration has elapsed. Deadline timeouts deliver timeout events only when certain other dependent events have not been received in the specified duration.

Timeouts are specified using the same filter mechanism as described in the previous section. This means they can be combined with source/event type filters and conjunction/disjunction groups. Such combinations provide dependencies between timeouts and source/event type filters that result in rich timeout semantics.

Full source code for this example is in the TAO source code distribution in the directory
`$TAO_ROOT/orbsvcs/DevGuideExamples/EventServices/RTEC_Filter`.

24.3.5.1 Interval Timeouts

Calling the `insert_time()` operation on the consumer QoS factory with an event type of `ACE_ES_EVENT_INTERVAL_TIMEOUT` specifies an interval timeout. You must also supply the interval desired in 100-nanosecond units.

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (1);
qos.insert_time(ACE_ES_EVENT_INTERVAL_TIMEOUT,
                100000000,   // 10^8 * 100 nanoseconds = 10 seconds
                0);          // handle to the rt_info
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

This consumer now receives events of type `ACE_ES_EVENT_INTERVAL_TIMEOUT` every 10 seconds. It receives no other events. If other events are desired as well, then additional filters can be added to the disjunction:

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (2);
qos.insert_type (MY_EVENT_1, 0);
qos.insert_time(ACE_ES_EVENT_INTERVAL_TIMEOUT,
                100000000,   // 10^8 * 100 nanoseconds = 10 seconds
                0);          // handle to the rt_info
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

This consumer now receives all events of type `MY_EVENT_1` as they are sent by the suppliers as well as interval timeout events every 10 seconds.

Placing an interval timeout in a conjunction group with other events causes the timeout event to be delivered with the other events in the group after they have all occurred. Consider the following example:

```
ACE_ConsumerQOS_Factory qos;
qos.start_conjunction_group (2);
qos.insert_type (MY_EVENT_1, 0);
qos.insert_time(ACE_ES_EVENT_INTERVAL_TIMEOUT,
                100000000,   // 10^8 * 100 nanoseconds = 10 seconds
                0);          // handle to the rt_info
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

If the timeout event occurs first, the event channel holds the timeout event until a MY_EVENT_1 type event is received. If a MY_EVENT_1 type event occurs first, the event channel holds the MY_EVENT_1 event until a timeout event is received. Both events are delivered together. After one event is queued, any subsequent events of the same type are discarded.

24.3.5.2 Deadline Timeouts

Deadline timeouts placed in a disjunction group give the developer the ability to schedule timeouts that occur when a fixed period of time has expired without any of the other events in the group being published. Deadline events are also created with the insert_time() member function, but require an event type of ACE_ES_EVENT_DEADLINE_TIMEOUT. The timeout value is still specified in 100-nanosecond units. Here is an example that defines a deadline timeout:

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (3);
qos.insert_type (MY_EVENT_1, 0);
qos.insert_type (MY_EVENT_2, 0);
qos.insert_time(ACE_ES_EVENT_DEADLINE_TIMEOUT,
                100000000,   // 10^8 * 100 nanoseconds = 10 seconds
                0);          // handle to the rt_info
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

In this example, an event of type ACE_ES_EVENT_DEADLINE_TIMEOUT is received by the consumer whenever 10 seconds have elapsed without an event of either type MY_EVENT_1 or MY_EVENT_2 having occurred. Each time an event of one of these types occurs, it is delivered to the consumer and the deadline timer is reset.

Placing a deadline timer in a conjunction group causes deadline timeout events whenever all the other filters in the group have not been satisfied before the deadline arrives. Repeating the previous example with a conjunction results in a consumer that only receives deadline timeout events whenever 10 seconds have elapsed without both MY_EVENT_1 and MY_EVENT_2 types of events occurring. Each time a matched pair of events is delivered to the consumer, the timer is reset.

Deadline timeouts are the only type of filter that benefits from the existence of nested disjunction groups. These nested groups allow the specification of

separate deadline timeout events for different groups of events. Consider the following filter specification:

```
ACE_ConsumerQOS_Factory qos;
qos.start_disjunction_group (3);
  qos.insert_type (MY_EVENT_1, 0);
  qos.insert_type (MY_EVENT_2, 0);
  qos.insert_time(ACE_ES_EVENT_DEADLINE_TIMEOUT, 100000000, 0);
qos.start_disjunction_group (3);
  qos.insert_type (MY_EVENT_3, 0);
  qos.insert_type (MY_EVENT_4, 0);
  qos.insert_time(ACE_ES_EVENT_DEADLINE_TIMEOUT, 50000000, 0);
supplier->connect_push_consumer (consumer.in (), qos.get_ConsumerQOS ());
```

This example delivers events of all four specified types to the consumer as well as timeout events whenever either 10 seconds elapse without events of either type 1 or 2 occurring or 5 seconds elapse without events of either type 3 or 4 occurring. When this consumer receives a timeout, there is no way for it to distinguish which deadline was exceeded. The only way to differentiate is to split it into two consumers, each with its own deadline timeout.

24.3.6 Creating and Configuring Event Channel Servants

The previous examples all used event channels that were in a remote process (the `tao_rtevent` server). The following section discusses the creation and management of event channel objects from within an application.

There are two separate mechanisms for configuring event channels, each of which provides a different set of configuration options. When creating your own event channels, you can set their attributes to configure certain behaviors. See 24.3.6.2 for the attributes and how to set them. You can always configure event channels using the service configurator. See 24.5 for the service configurator options and how to set them.

24.3.6.1 Local Event Channel Example

The following code shows the steps necessary to create and initialize a local event channel. First the following header files need to be included:

```
#include <orbsvcs/Event/EC_Event_Channel.h>
#include <orbsvcs/Event/EC_Default_Factory.h>
```

Next, the event channel's resource factory needs to be initialized and registered. This must occur before the ORB is initialized and is required to allow the event channel to be configured via the service configurator (see 24.5). Failure to initialize the factory results in any event channel configuration options in the service configurator file being ignored. Here is the initialization call:

```
TAO_EC_Default_Factory::init_svcs ();
```

After initializing the ORB, and activating the POA Manager, the application is ready to create the event channel object and register it with the POA.

```
// Get the POA/POA Manager and activate the POA Manager
CORBA::Object_var object = orb->resolve_initial_references ("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow (object.in ());
PortableServer::POAManager_var poa_manager = poa->the_POAManager ();
poa_manager->activate ();

// Create a local event channel and register it with the RootPOA.
TAO_EC_Event_Channel_Attributes attributes (poa.in (), poa.in ());
PortableServer::Servant_var<TAO_EC_Event_Channel> ec_impl =
  new TAO_EC_Event_Channel(attributes);
ec_impl->activate ();
PortableServer::ObjectId_var oid = poa->activate_object(ec_impl.in());
CORBA::Object_var ec_obj = poa->id_to_reference(oid.in());
RtecEventChannelAdmin::EventChannel_var ec =
  RtecEventChannelAdmin::EventChannel::_narrow(ec_obj.in());
```

The event channel is now ready to use. The only step remaining is to make the event channel available to interested consumers and suppliers. The examples in this section have been using the Naming Service to locate the event channel. The event channel can be bound in the Naming Service as follows:

```
const char* ecname = "MyName";
object = orb->resolve_initial_references("NameService");
CosNaming::NamingContextExt_var root_context =
                   CosNaming::NamingContextExt::_narrow(object.in());
CosNaming::Name_var name = root_context->to_name(ecname);
root_context->rebind(name.in(), ec.in());
```

The new event channel is now bound to "`MyName`" in the root naming context of the Naming Service. Other techniques (that are not shown here) of making the event channel implementation available to consumers and suppliers include:

- Returning the event channel object reference through an operation in the application's IDL interface.
- Writing the event channel's IOR as a string to a file.
- Using persistent object references.

24.3.6.2 Setting Attributes of the Event Channel

The `TAO_EC_Event_Channel_Attributes` object that is passed to the event channel constructor is used to set several event channel attributes. Table 24-6 provides a summary of the attributes that can be set:

Table 24-6 Event Channel Attributes

Name	Type	Default	Description
`supplier_poa`	`PortableServer::POA_ptr`	None	POA used by supplier admin and supplier proxies. This is typically the same POA the EC uses.
`consumer_poa`	`PortableServer::POA_ptr`	None	POA used by consumer admin and consumer proxies. This is typically the same POA the EC uses.
`consumer_reconnect`	`int`	`0`	Enables consumer reconnections when non-zero.
`supplier_reconnect`	`int`	`0`	Enables supplier reconnections when non-zero.
`disconnect_callbacks`	`int`	`0`	If not zero, the event channel sends disconnect callbacks when a disconnect operation is called on a proxy.
`scheduler`	`CORBA::Object_ptr`	nil	Scheduler that the EC should collaborate with. The default value causes the EC to not use a scheduler.

In the example in the previous section, the only attributes set were the supplier and consumer POAs:

```
TAO_EC_Event_Channel_Attributes attributes (poa.in (),   // Supplier POA
                                            poa.in ());  // Consumer POA
TAO_EC_Event_Channel* ec_impl = new TAO_EC_Event_Channel(attributes);
```

The event channel uses the supplier POA to activate `SupplierAdmin` and `ProxyPushSupplier` servants, and the consumer POA to activate `ConsumerAdmin` and `ProxyPushConsumer` servants.

All other attributes of the event channel are set using public data members of the `TAO_EC_Event_Channel_Attributes` class. For example:

```
TAO_EC_Event_Channel_Attributes attributes (poa.in (), poa.in ());
attributes.supplier_reconnect = 1;
attributes.consumer_reconnect = 1;
TAO_EC_Event_Channel* ec_impl = new TAO_EC_Event_Channel(attributes);
```

The example above enables suppliers and consumers to reconnect, thereby allowing suppliers to call `connect_push_supplier()` and consumers to call `connect_push_consumer()` multiple times without disconnecting as discussed in 24.3.2.3.

The `disconnect_callbacks` attribute controls whether the consumer and supplier disconnect callbacks are called when the corresponding disconnect operation is called on their proxy object. For example, if this attribute is set to true, when a consumer calls `disconnect_push_supplier()` on its proxy, the event channel invokes `disconnect_push_consumer()` on the consumer. Similar behaviors exist for push suppliers.

The `scheduler` attribute is a reference to the scheduler that the event channel uses. For more details, see the Scheduling Service documentation.

The default values for these attributes are defined as preprocessor macros in `$TAO_ROOT/orbsvcs/orbsvcs/Event/EC_Defaults.h`. You can use your own project-specific defaults by setting these macros in your `config.h` file and recompiling the event service.

24.3.7 Observers

The TAO Real-Time Event Channel architecture defines an `Observer` interface that allows for user-defined CORBA objects that are notified each time suppliers and consumers are connected to or disconnected from an event channel. This functionality is useful for monitoring subscriptions to aid in efficiently federating networks of event channels as discussed in 24.3.8. By default, the observer functionality is disabled and must be enabled using the `-ECObserver` service configurator option described in 24.5.2.9.

To implement custom gateways or other advanced features, you may need to develop your own observers. To create an observer, use the description in `$TAO_ROOT/orbsvcs/orbsvcs/RtecEventChannelAdmin.idl` to implement the `Observer` interface. To connect the observer, use the `append_observer()` operation in the `EventChannel` interface. While your observer is connected, and a consumer connects or disconnects, the event channel calls `update_consumer()` with a `ConsumerQOS` structure that represents the aggregate subscription information for all consumers. Likewise, each time a supplier connects or disconnects, `update_supplier()` is called with a `SupplierQOS` structure that holds the aggregate supplier information. To disconnect the observer, use the `remove_observer()` operation of the `EventChannel` interface.

Subscriptions with the `is_gateway` flag set to true do not have their subscription information passed to observers. This flag is set to false by default and can be set to true via the `is_gateway` data member of the `ConsumerQOS` structure.

```
ACE_ConsumerQOS_Factory qosFact;
// Setup the qos...
RtecEventChannelAdmin::ConsumerQOS qos = qosFact.get_ConsumerQOS();
qos.is_gateway = 1;
// Connect to the EC using the qos structure...
```

Gateways can be used to federate event channels (see 24.3.8).

24.3.8 Federating Event Channels

Using a single event channel in a large distributed system can often lead to unnecessarily long delivery times and unnecessarily high levels of network traffic. This is especially true when many of the consumers and suppliers that are communicating are located on the same machine or possibly even in the same process. Figure 24-2 shows such a system, where each node contains

consumers and suppliers and almost all events must make two trips across the network.

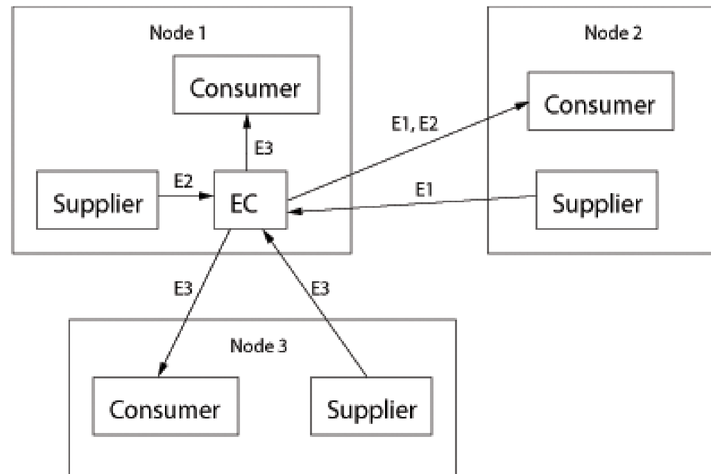


Figure 24-2 Single Event Channel System Architecture

By having event channels on each node in the network (and dividing the events published between them in some logical manner), you can avoid some of these problems. Figure 24-3 shows a hypothetical system with such an architecture. This configuration avoids some of the problems of the single event channel system. However, the architect must now specify the event channels to use for particular events and consumer and supplier processes must now communicate with several distinct event channels. Even with an optimal distribution of events on event channels, this configuration may produce additional overhead and network traffic

Federating event channels avoids many of these problems. When separate event channels are joined into a federation using one of the techniques described in the subsequent sections, the resulting federated event channel acts as one logical event channel. This frees you from allocating events to event channels or interacting with more than one event channel. It also avoids performance problems by delivering events locally, when possible. The Real-Time Event Service provides several mechanisms for federating event channels that differ mainly in the protocol used to transmit events between members of the federation. Federation mechanisms exist that send the events via CORBA messages, UDP, and IP multicast.

These mechanisms all utilize the normal consumer and supplier interfaces to receive events from one event channel and forward them to another. The CORBA and multicast mechanisms make use of the event channel's `Observer` interface that allows them to automatically tailor their consumer subscriptions based on the consumers that are connected to the event channel they are supplying. See 24.3.7 for a full description of the `Observer` interface..

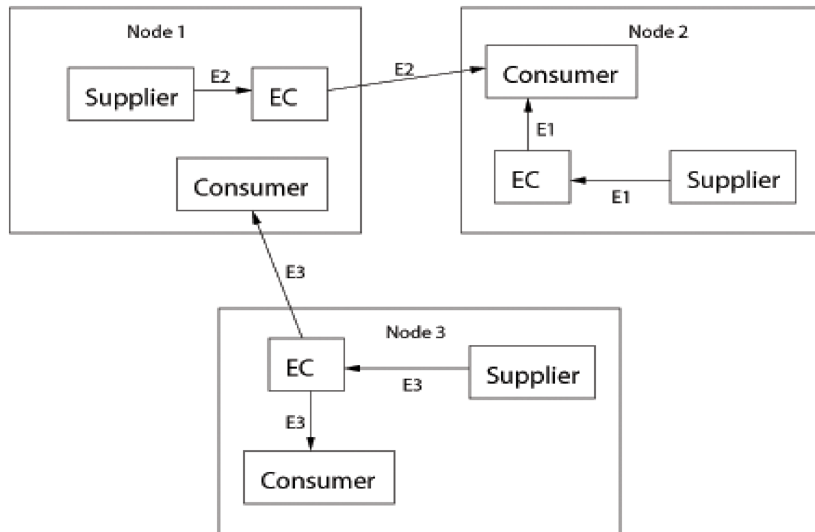
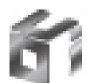


Figure 24-3 Multiple Event Channel System Architecture

To ensure that events do not infinitely loop between event channels in the federation, the time to live (`ttl`) field in the event header is used as a counter. Each time an event passes between event channels in a federation, the `ttl` flag is decremented. If the `ttl` value reaches zero, the event is no longer passed between event channels. Typically, you will want to set the `ttl` field to 1 when using federations. Events of local interest may use a value of 0.

24.3.8.1 Using the CORBA Gateway

TAO defines a generic gateway interface in the `TAO_EC_Gateway` class and provides one implementation of this interface, `TAO_EC_Gateway_IIOP`, that uses CORBA messages as the underlying communication mechanism. This gateway uses the existing consumer and supplier interfaces to connect to a pair



of event channels (one local and one remote). It transmits the events between the event channels by receiving them from a remote event channel as a consumer and sending them to the local event channel as a supplier. To use the consumer information from the local event channel for customizing its subscription information for the remote event channel, the gateway acts as an `Observer` on the local event channel. This ensures all necessary events (and no more) are passed to it for eventual delivery to the local consumers.

Figure 24-4 shows three event channels that have been federated using CORBA gateways.

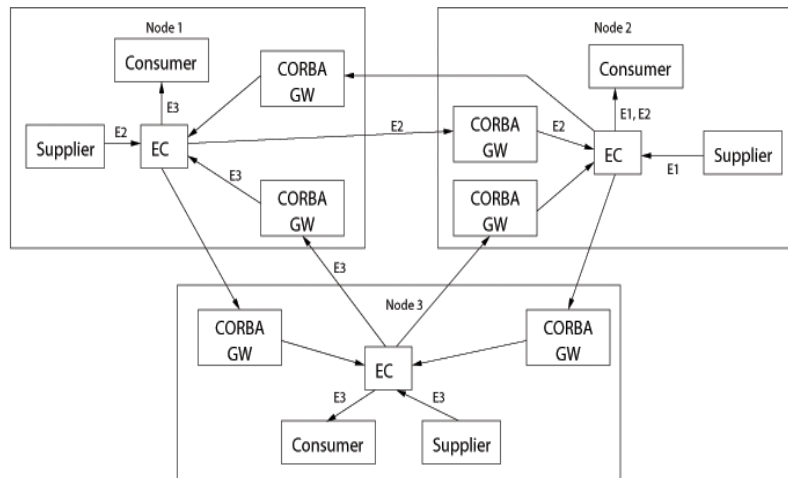


Figure 24-4 CORBA-Based Event Channel Federation

Note *Because the* `TAO_EC_Gateway_IIOP` *uses the Observer capabilities of the event channel, the Observer functionality must be activated as discussed in 24.5.2.9.*

Consumers and suppliers using the federation only connect to their local event channel and are unaware of the federation. The local event channel handles all events published from a local supplier to a local consumer. If any remote consumers are interested in them (subscribe to them), they are passed through the appropriate gateway to the corresponding remote event channel.

The following code shows how to create and use a CORBA based federation of two event channels. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/EventServices/RTEC_Federated`.

First include the appropriate header file:

```
#include <orbsvcs/Event/EC_Gateway_IIOP.h>
```

After the local event channel has been created and a reference to the remote event channel obtained, the gateway can be created and initialized:

```
RtecEventChannelAdmin::EventChannel_var local_ec;
RtecEventChannelAdmin::EventChannel_var remote_ec;

// Initialize the local event channel and connect to
// the remote event channel...

PortableServe::Servant_var<TAO_EC_Gateway_IIOP> gateway =
  new TAO_EC_Gateway_IIOP;
gateway->init(remote_ec.in(),                // Remote EC object reference
              local_ec.in());                // Local EC object reference
```

The gateway object is initialized with the remote and local event channel references.

To activate the gateway, it must be registered with the POA and attached as an observer to the local event channel. The `TAO_EC_Gateway_IIOP` class implements the `RtecEventChannelAdmin::Observer` interface that allows it to be passed into the `append_observer()` operation.

```
CORBA::Object_var poa_obj = orb->resolve_initial_references ("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow (poa_obj.in ());
PortableServer::ObjectId_var gateway_oid =
  poa->activate_object(gateway.in());
CORBA::Object_var gateway_obj = poa->id_to_reference(gateway_oid.in());
RtecEventChannelAdmin::Observer_var obs =
  RtecEventChannelAdmin::Observer::_narrow(gateway_obj.in());
RtecEventChannelAdmin::Observer_Handle local_ec_obs_handle =
  local_ec->append_observer (obs.in ());
```

The gateway is now observing the local event channel and each time a consumer is added or removed it receives the full subscription information for all consumers and updates its subscription on the remote event channel. When

the remote event channel delivers events to the gateway (acting as a consumer), it publishes them on the local event channel (acting as a supplier).

All suppliers and consumers now interact with the federated event channel by simply using their local event channel.

To fully federate a set of N event channels using CORBA-based gateways requires that N-1 gateways be collocated with each event channel (one for each remote event channel).

The CORBA gateway can be configured using the IIOP Gateway Factory object as described in 24.6.

24.3.8.2 Federating Event Channels with UDP

TAO's Real-Time Event Service supports two separate APIs for federating event channels using UDP. The first API has been supported by TAO for a number of years and uses a number of low-level classes that allow you to directly control all aspects of the federation. The second API was added to recent versions of TAO and allows a single `TAO_ECG_Mcast_Gateway` object to encapsulate all of the lower level objects of the first API. While not fully controllable, the gateway object provides a number of configuration options for commonly used variations. The example in this section uses the lower-level API. For more details on the gateway class and its usage, see the class documentation in `$TAO_ROOT/orbsvcs/orbsvcs/Event/ECG_Mcast_Gateway.h`.

To federate two event channels using the low-level UDP API, the `TAO_EGP_UDP_Sender`, `TAO_EGP_UDP_Receiver`, and `TAO_ECG_UDP_EH` classes are needed. To actually federate the channels, construct sender and receiver objects, attach each to an event channel, and connect the sender to the remote receiver. Figure 24-5 shows three event channels that have been federated using UDP. The sender object consumes events from its collocated event channel and transmits each event to a specified UDP port (associated with a remote event channel). The event handler (UDP EH) listens on a UDP port and passes each incoming event to the receiver. The receiver acts as an event supplier and pushes each event received onto its collocated event channel.

The following code shows how an individual event channel can be federated with another similarly configured event channel. Full source code for this example is in the TAO source code distribution in the directory

$TAO_ROOT/orbsvcs/DevGuideExamples/EventServices/RTEC_MCast_Federated.

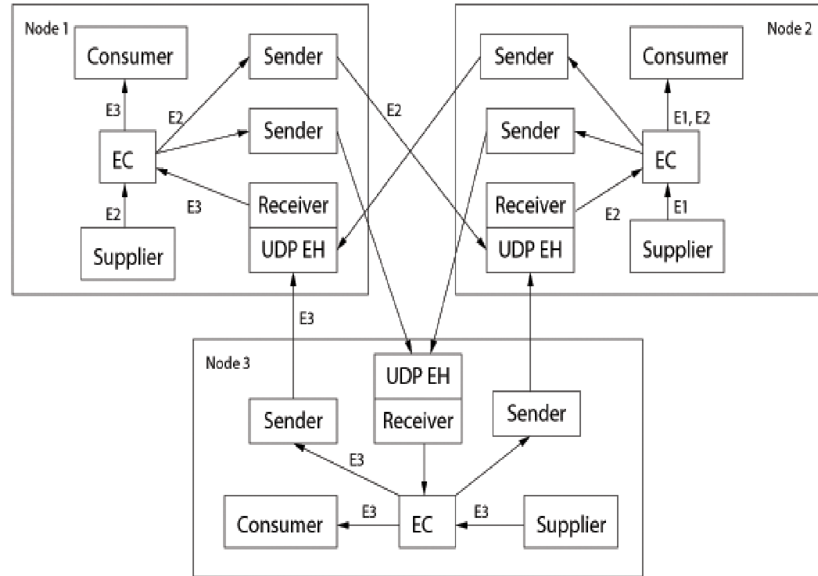


Figure 24-5 UDP-Based Event Channel Federation

First, include the files where the UDP federation classes are defined:

```
#include <orbsvcs/Event/ECG_UDP_Sender.h>
#include <orbsvcs/Event/ECG_UDP_Receiver.h>
#include <orbsvcs/Event/ECG_UDP_Out_Endpoint.h>
#include <orbsvcs/Event/ECG_UDP_EH.h>
```

After the event channel is created and activated, we need to create and activate an object that implements the `RtecUDPAdmin::AddrServer` interface. This object will be used by the sender object to determine where to send each event that the sender receives. The simple implementation used here (`SimpleAddressServer`) always returns the single address that it was passed in its constructor. That address should refer to the node and port where a corresponding event channel's event handler is listening.

```
u_short port = 12345;          // Port # of the remote EC's event handler
const char* address = "node2"; // The remote EC's node
ACE_INET_Addr send_addr (port, address);
```

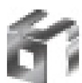
```
PortableServer::Servant_var<SimpleAddressServer> addr_srv_impl =
  new SimpleAddressServer(send_addr);
PortableServer::ObjectId_var addr_srv_oid =
  poa->activate_object(addr_srv_impl.in());
CORBA::Object_var addr_srv_obj = poa->id_to_reference(addr_srv_oid.in());
RtecUDPAdmin::AddrServer_var addr_srv =
  RtecUDPAdmin::AddrServer::_narrow(addr_srv_obj.in());
```

We now construct, initialize, and connect the sender object to the event channel as a consumer. We pass the object reference of our simple `RtecUDPAdmin::AddrServer` to the sender's `init()` function so it can use it to get the address to which it should send each event it receives:

```
// Create and initialize the sender object
TAO_EC_Servant_Var<TAO_ECG_UDP_Sender> sender = TAO_ECG_UDP_Sender::create();
TAO_ECG_UDP_Out_Endpoint endpoint;
if (endpoint.dgram ().open (ACE_Addr::sap_any) == -1) {
  std::cerr << "Cannot open send endpoint" << std::endl;
  return 1;
}
// Clone endpoint so sender can take ownership
sender->init (ec.in (), addr_srv.in (),
              new TAO_ECG_UDP_Out_Endpoint (endpoint));

// Setup the subscription and connect to the EC
ACE_ConsumerQOS_Factory cons_qos_fact;
cons_qos_fact.start_disjunction_group (1);
cons_qos_fact.insert (ACE_ES_EVENT_SOURCE_ANY, ACE_ES_EVENT_ANY, 0);
RtecEventChannelAdmin::ConsumerQOS sub = cons_qos_fact.get_ConsumerQOS ();
sender->open (sub);
```

Next, we construct, initialize, and connect the receiver object to the event channel as a supplier. Again, we pass the object reference to our simple `RtecUDPAdmin::AddrServer` to the receiver's `init()` function:

```
// Create and initialize the receiver
TAO_EC_Servant_Var<TAO_ECG_UDP_Receiver> receiver =
                                          TAO_ECG_UDP_Receiver::create();
receiver->init (ec.in (),
                new TAO_ECG_UDP_Out_Endpoint(endpoint),
                addr_srv.in ());

// Setup the registration and connect to the event channel
ACE_SupplierQOS_Factory supp_qos_fact;
supp_qos_fact.insert (MY_SOURCE_ID, MY_EVENT_TYPE, 0, 1);
RtecEventChannelAdmin::SupplierQOS pub = supp_qos_fact.get_SupplierQOS ();
receiver->open (pub);
```



We then create and connect the UDP event handler to the reactor, and initialize it with the address where it should listen.

```
// Create the appropriate event handler and register it with the reactor
TAO_ECG_UDP_EH* udp_eh = new TAO_ECG_UDP_EH (receiver.in());
udp_eh->reactor (orb->orb_core ()->reactor ());
u_short listenport = 12345;  // Listen port for this EC's event handler
ACE_INET_Addr local_addr (listenport);
if (udp_eh->open (local_addr) == -1) {
  std::cerr << "Cannot open EH" << std::endl;
}
```

Once we enter the event loop and the other event channel is similarly configured, this event channel then forms a federation with the other event channel. Consumers connected to either event channel can receive events from both. Each event pushed to an event channel is automatically sent to the other event channel (assuming its `ttl` field is one).

One way to connect a third event channel to this federation is to add a second sender object (to each existing event channel) and force it to use the address of the new event channel. In this type of configuration, each event channel in a federation of N event channels requires a receiver and N-1 senders.

For the sake of completeness, here is the implementation of the `SimpleAddressServer`. First, the header file:

```
#include <orbsvcs/RtecUDPAdminS.h>

class SimpleAddressServer : public POA_RtecUDPAdmin::AddrServer
{
public:
  SimpleAddressServer (const ACE_INET_Addr& address);
  virtual void get_addr (const RtecEventComm::EventHeader& header,
                         RtecUDPAdmin::UDP_Addr& address);

private:
  RtecUDPAdmin::UDP_Addr address_;
};
```

Next, the implementation of the constructor and the `get_addr()` operation:

```
#include "SimpleAddressServer.h"
#include <ace/INET_Addr.h>
```

```
SimpleAddressServer::SimpleAddressServer (const ACE_INET_Addr& address)
{
  this->address_.ipaddr = address.get_ip_address ();
  this->address_.port   = address.get_port_number ();
}

void SimpleAddressServer::get_addr (
  const RtecEventComm::EventHeader&,
  RtecUDPAdmin::UDP_Addr& address)
{
  address = this->address_;
}
```

24.3.8.3 Federating Event Channels with IP Multicast

The UDP and multicast federation mechanisms share much of the same infrastructure. Like UDP federations, multicast federations can be constructed using either a `TAO_ECG_Mcast_Gateway` object or a set of lower level objects that give developers direct control of the federation. Our example uses the lower level objects of the sender, receiver, and event handler and is very similar to that in the preceding section. For more information about using the gateway object, see the class documentation in `$TAO_ROOT/orbsvcs/orbsvcs/Event/ECG_Mcast_Gateway.h`.

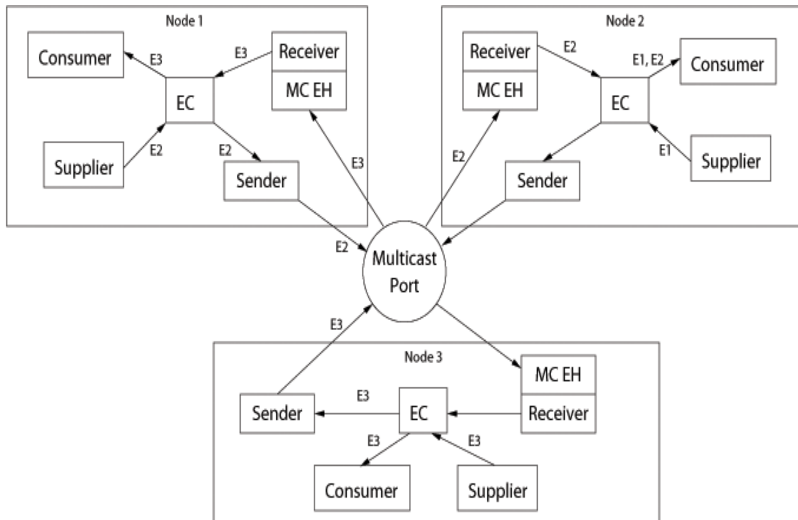


Figure 24-6 Multicast-Based Event Channel Federation

Figure 24-6 shows a multicast federation of three event channels. The main differences from the UDP example, lie in the address supplied to and returned by the address server (that should now be a multicast address) and the type of the event handler.

The example discussed in the previous section and found in the `$TAO_ROOT/orbsvcs/DevGuideExamples/EventServices/RTEC_Mcast_Federated` directory can also be used to form a multicast federation. The main code difference is found in the creation and initialization of the event handler. Note that the multicast event handler is no longer initialized with an address. It receives the address by querying the address server (via the receiver passed into its constructor).

```
TAO_ECG_Mcast_EH* mcast_eh = new TAO_ECG_Mcast_EH (receiver.in());
mcast_eh->reactor (orb->orb_core ()->reactor ());
if (mcast_eh->open (ec.in()) == -1) {
  std::cerr << "Cannot open EH" << std::endl;
}
eh = mcast_eh;
```

Because of the nature of multicast, you can use this example to federate an unlimited number of event channels (as long as they all use the same port/address combination). This makes the multicast approach the most scalable mechanism for federating large numbers of event channels.

Note *Because the* `TAO_ECG_Mcast_EH` *uses the Observer capabilities of the EC, the Observer functionality must be activated as discussed in 24.5.2.9.*

24.3.8.4 Choosing The Appropriate Federation Mechanism

Choosing the best federation mechanism for a project involves understanding the relative strengths (and weaknesses) of the existing mechanisms and evaluating them with respect to a project’s unique characteristics and forces.

The multicast federation approach has several unique advantages that make it applicable to many projects. It is the easiest federation type to set up and is also the most scalable in terms of memory usage and network traffic. Its main disadvantages are its use of an inherently unreliable protocol (that is often not supported on some platforms and networks) and potential problems with federating ECs over a complex network. It is appropriate for projects that

operate over a controlled network (that would make it fairly reliable) and need to conserve network bandwidth.

The UDP federation is more complex than the multicast federation, and thus does not scale as well. It requires more sender objects than the multicast mechanism for a full federation and thus will use more memory. It may also result in redundant network traffic. Its use of an inherently unreliable protocol is also a disadvantage. The UDP approach is appropriate for systems that cannot use multicast for some of the reasons cited above.

The CORBA federation uses a reliable protocol (by default it uses IIOP, which uses TCP/IP) but forces the addition of many gateway objects as the federation grows, like the UDP mechanism forces the addition of many senders. Thus, it is about as complex as the UDP mechanism. It may result in more network traffic than the UDP mechanism because of its use of a heavier weight transport protocol. It is especially appropriate for relatively small systems that require reliable event delivery.

24.4 tao_rtevent Command Line Options

The `tao_rtevent` server supplies the capability to start a single real-time event channel in its own process. You can configure it to make use of a local (in-process) or global scheduler. This server uses a local scheduler by default. It binds the Event Channel to a supplied name in the root naming context of the Naming Service. The Naming Service server must be running to use this server. The event channel created implements the `RtecEventChannelAdmin::EventChannel` interface. Table 24-7 describes the available command line options.

Table 24-7 tao_rtevent Command Line Options

Option	Description	Default
`-n` *`RTEC_name`*	Specifies the name with which to bind the Event Channel in the root naming context of the Naming Service.	`EventService`

Table 24-7 tao_rtevent Command Line Options

Option	Description	Default
`-s (global | local | none)`	Specifies a scheduler for this event channel. If `local`, a scheduler object is created in this process and used. If `global`, the Naming Service locates the scheduler using the "`ScheduleService`" name in the root naming context. If `none`, no scheduler is used.	`none`
`-o` *`filename`*	Specify the name of the file used for storing the IOR of the event channel.	Do not store the IOR in a file.
`-p` *`pid_filename`*	Specify the name of the file used for storing the process ID of this server.	Do not store the PID in a file.
`-q` *`object_id`*	Specify an Object ID to use for activating the event channel. When specified, this option also makes the event channel a persistent CORBA object.	Event channel is a transient CORBA object.
`-b`	Specifies that this event channel should enable bi-directional GIOP support.	Bi-directional GIOP support is not enabled.
`-a`	Use the thread per consumer dispatching strategy.	`-ECDispatchingStrategy` option determines strategy.
`-x`	Do not use the Naming Service. This simply creates an event channel.	Bind the EC in the Naming Service.

If a local scheduler is specified, the scheduler object is registered in the Naming Service under the root naming context with the name "`ScheduleService`." If a global scheduler is requested, the process that contains the scheduler should be started before running the event service. For further information see the appropriate Scheduling Service documentation.

While the thread per connection dispatching strategy can be specified through this command line (and programatically), other dispatching strategies are controlled via the service configurator options discussed in 24.5.1.1.

24.5 Event Channel Resource Factory

The event channel resource factory is responsible for creating many strategy objects that control the behavior of the event channel. The behavior of the

event channel is typically controlled by using the service configurator file to select the appropriate behaviors for the default factory implementation. In addition to the default factory implementation the RTES supplies a number of other implementation of this factory. Some are used to enable specific features that most applications do not use (such as real-time scheduling and dispatching) and others are minimal implementations that constrain run-time configuration. Optionally, the application developer could also provide a custom implementation of the event channel resource factory containing tailored or customized strategies. This section focuses on the options supported by the default event channel factory and some commonly used alternatives.

The RTES can use either of two event channel resource factories. They are both registered with the service configurator using the name `EC_Factory`, and are statically registered with the service configurator, so the static directive is used to supply initialization options to them. The options used to configure the two factory implementations differ slightly. These differences are noted in the documentation for each option. If you do nothing, the RTES uses the *default* event channel factory. You can select the *thread per consumer* (TPC) event channel factory either by passing the `-a` option to the `tao_rtevent` server as described in 24.4 or by explicitly calling the static member function `TAO_EC_TPC_Factory::init_svcs()`.

To change the behavior of the event channel factory, add a line similar to the line shown below to your service configuration file.

```
static EC_Factory "-ECFiltering basic"
```

For this option to be effective, you must make sure that one of the following function calls occur *before* the ORB is initialized (which one you use determines which factory your application uses):

```
TAO_EC_Default_Factory::init_svcs ();   // Initialize the default factory
TAO_EC_TPC_Factory::init_svcs();        // Initialize the TPC factory
```

This creates an event channel resource factory and statically registers it. If you forget to do this, the service configurator is not able to find the `EC_Factory` to initialize it.

The -ORBSvcConf option allows you to use file names other than svc.conf for service configurator initialization. See 17.13.63 for more information on this option.

Default values for many of the event channel resource factory's options are defined as preprocessor macros in $TAO_ROOT/orbsvcs/orbsvcs/Event/EC_Defaults.h. You can use your own project-specific defaults by setting these macros in your config.h file and recompiling the event service.

24.5.1 EC_Factory Option Overview

This section provides an overview of the configuration options supported by the default and TPC event channel factories. The following section provides detailed documentation of each of the individual options.

24.5.1.1 Dispatching

When the event channel is pushing events to interested consumers, the thread used to push the event is one decision that has far-reaching effects on the performance and behavior of the application. The event channel resource factory allows for selection of a dispatching strategy that defines how to push events received from suppliers to the interested consumers on the appropriate thread. The default event channel resource factory allows for reactive, priority, and multithreaded dispatching strategies. In addition, when a multithreaded dispatching strategy is selected, the number of threads to be used can be specified. Table 24-8 shows the options related to dispatching strategies.

Table 24-8 Dispatching related options

Option	Section	Description
-ECDispatching {**reactive** | mt}	24.5.2.5	Supply this option to select the dispatching strategy for supplier produced events.
-ECDispatchingThreads *nthreads*	24.5.2.6	Specify the number of threads to create and use for the multithreaded dispatching strategy. Defaults to one thread.
-ECDispatchingThreadFlags *flags:priority*	24.5.2.7	Specifies the thread flags and priority to use for creating dispatching threads. Affects only the multithreaded and thread per consumer dispatching strategies.

Table 24-8 Dispatching related options

Option	Section	Description
`-ECQueueFullServiceObject` *`service_name`*	24.5.2.14	Specifies a service name that is used for finding a service object that defines the strategy for handling full event dispatching queues.
`-ECTimeout {`**`reactive`**`}`	24.5.2.20	Supply this option to select the dispatching strategy for timeout events.

The reactive dispatching strategy delivers events on the same thread as they were received (or generated). This is usually the reactor's main thread. The multithreaded dispatching strategy creates a pool of threads and distributes each event on a randomly-selected member of the pool. The timeout strategy is used to determine how timeout events are dispatched. This option takes only a single value here but is extended by the non-default factories to provide additional strategies.

Use of the thread per consumer (TPC) event channel factory forces a dispatching strategy that creates a separate dispatching thread for each consumer. Each event received from a supplier is placed on a queue for each consumer that receives it. Each consumer's dispatching thread pushes the event to its consumer. When using the TPC factory, the `-ECDispatching` option is ignored.

24.5.1.2 Feature Control

Various features of the event channel can be disabled, configured, and replaced via the event channel resource factory. These include supplier and consumer filtering, observers, ORB identity, and scheduler interaction. Table 24-9 shows the options related to control of these event channel features.

Table 24-9 Event channel feature control options

Option	Section	Description
`-ECFiltering {null	`**`basic`**`	prefix}`	24.5.2.8	Controls the type of consumer filtering performed in the event channel.
`-ECSupplierFiltering {null	`**`per-supplier`**`}`	24.5.2.19	Controls the supplier-side filtering in the event channel. The null option disables filtering and sends all events to a global collection of consumers. Per-supplier filtering maintains separate consumer lists for each supplier.

Table 24-9 Event channel feature control options

Option	Section	Description
`-ECSupplierFilter`	24.5.2.19	Same as `-ECSupplierFiltering`.
`-ECObserver {`**`null`**` | basic | reactive}`	24.5.2.9	Enables the Observer functionality. Must be enabled to support Gateways and the federation of event channels.
`-ECScheduling {`**`null`**` | group}`	24.5.2.15	Effects coordination between the event channel and scheduling service to enable building the dependency list of consumers and suppliers.
`-ECUseORBId` *`orbid`*	24.5.2.22	Specifies the *`orbid`* of the ORB that the default factory uses.
`-ECConsumerValidateConnection` *`enabled`*	24.5.2.4	This option forces the event channel to make and validate the connection to the consumer process before fully processing the `connect_push_consumer` call.
`-ECTPCDebug`	24.5.2.21	Enables debugging messages when using the thread per consumer event channel resource factory.

The two filtering options can be disabled so as to improve efficiency in cases where they are not being used. To allow consumers to use nested filter groups, the `-ECFiltering` option must be set to `prefix`. The observer option needs to be enabled when an event channel process uses gateways or has observers attached to it.

The default factory requires an ORB for a variety of operations. It normally uses the default ORB (with a null string for the ORB identifier). Specify an `orbid` using the `-ECUseORBId` option to force the default factory to use a different ORB. Typically, this option is used to ensure that the default factory is using the same ORB as was used to activate the event channel.

24.5.1.3 Locking Options

The locking options allow the event channel resource factory to define the lock type desired for various components in the event channel. The default

factory allows specification of the lock type for consumer and supplier proxies using the options shown in Table 24-10.

Table 24-10 Locking options

Option	Section	Description
`-ECProxyConsumerLock {null	**thread**	recursive}`	24.5.2.10	Specifies the lock type for the consumer proxy object.
`-ECProxySupplierLock {null	**thread**	recursive}`	24.5.2.13	Specifies the lock type for the supplier proxy object.

These options can be set to null to increase performance if the event channel does not access the given components from multiple threads. The default values ensure that the event channel is thread-safe, but recursive locks may be required to avoid deadlocks in certain complex systems.

24.5.1.4 Consumer and Supplier Control Options

The following group of options allows the event channel resource factory to define how the event channel handles *dangling* (ill-behaved) suppliers and consumers. Consumers and suppliers that remain connected to the event channel when their CORBA objects are no longer accessible from the event channel process are considered ill-behaved. Such consumers and suppliers result when the consumer or supplier process fails to call disconnect, terminates abnormally, or has its node disconnected from the network. The default factory allows specification and configuration of the control policy for consumer and supplier proxies via the options shown in Table 24-11.

Table 24-11 Consumer and supplier control options

Option	Section	Description
`-ECConsumerControl {**null**	reactive}`	24.5.2.1	Defines the policy for handling ill-behaved consumers.
`-ECSupplierControl {**null**	reactive}`	24.5.2.16	Defines the policy for handling ill-behaved suppliers.
`-ECConsumerControlPeriod period`	24.5.2.2	Defines the polling period in microseconds for the reactive consumer control policy.
`-ECSupplierControlPeriod period`	24.5.2.17	Defines the polling period in microseconds for the reactive supplier control policy.
`-ECConsumerControlTimeout timeout`	24.5.2.3	Defines the timeout in microseconds that the reactive consumer control policy uses for pinging consumers.

Table 24-11 Consumer and supplier control options

Option	Section	Description
`-ECSupplierControlTimeout timeout`	24.5.2.18	Defines the timeout in microseconds that the reactive supplier control policy uses for pinging suppliers.

The default control policy of `null` leaves consumers and suppliers connected to the event channel even if the event channel is unable to access them. This allows consumers and suppliers to continue to be connected even in the face of intermittent communications.

The `reactive` control policy disconnects a consumer or supplier from the event channel the first time the event channel fails to deliver a request to it. It also periodically polls (by default every 5 seconds) all consumer and supplier proxies to ensure their continued connection. Failure to respond to the polling request results in disconnection of the proxy. If the polling period is set to 0, polling is completely disabled.

24.5.1.5 Proxy Collection Options

The proxy collection options define the types of collections used to hold consumer and supplier proxies. The default factory allows specification of the collection type for consumer and supplier proxies via the options shown in Table 24-12.

Table 24-12 Proxy collection options

Option	Section	Description
`-ECProxyPushConsumerCollection` *`flags`*	24.5.2.11	Define the characteristics of the collection used to store proxy consumers in the event channel.
`-ECProxyPushSupplierCollection` *`flags`*	24.5.2.12	Define the characteristics of the collection used to store proxy suppliers in the event channel.

The flags passed to these collection options fall into three separate groups, with each group specifying a different characteristic of the collection. A colon is used as a separator between flags (e.g., `mt:rb_tree:immediate`).

First, the lock type used to control access to the collection can be specified. The `st` flag allows specification of a null lock. The `mt` flag specifies a thread-safe lock. A thread-safe lock is specified by default.

The second characteristic is the actual collection type used. The `list` flag specifies that an ordered list collection is used. The `rb_tree` flag specifies that a collection using a red-black tree is used. The event channel uses an ordered list collection by default.

The third characteristic specifies how concurrent use of the collection is controlled, specifically the case where the collection is being iterated over while a client is attempting an operation that adds or removes a member of the collection. An example is concurrent distribution of events to push consumers (via iteration over the proxy push suppliers collection) while one of the consumers is attempting to disconnect itself. The default factory specifies four different strategies for this characteristic: `immediate`, `copy_on_read`, `copy_on_write`, and `delayed`.

The `immediate` flag causes each operation to block until it receives access to the collection. In the example above, the consumer attempting to disconnect blocks until the event distribution iteration completes. Note, that it is possible that the disconnect request may be processed in the same thread as the event distribution (via a nested upcall). If this occurs, immediate access is granted (the thread already has the lock for the collection) and the iterator may be invalidated. It is the developer's responsibility to ensure that the iterator is not invalidated. Using the `-ECDispatching` option (see 24.5.2.5) to establish a separate dispatching thread is the most common method for ensuring this validity. (In other words, the `immediate` collection update flag should not be used with `-ECDispatching reactive`.)

The `copy_on_read` flag causes the iterators to copy the collection before proceeding. This allows iterators to release the lock after the copy is made. Subsequent changes to the collection can occur while iteration is ongoing without affecting the iteration. In the above example, this means that the consumer can disconnect without harm to the event dispatching. The main disadvantage of this approach is that of the extra performance overhead incurred when the copy of the collection is allocated and replicated.

The `copy_on_write` flag causes any modifiers to the collection to make copies of the collection before proceeding. This means changes to the collection can occur while iteration is ongoing without affecting the iteration. In the above example, this means that the consumer can disconnect without harm to the event dispatching. The main disadvantage of this approach is that of the extra performance overhead incurred when the copy of the collection is allocated and replicated. Note that the `copy_on_write` strategy makes a

copy each time the collection is changed (connect, disconnect, reconnect, or shutdown) and the `copy_on_read` strategy makes copies each time the collection is iterated.

The `delayed` flag causes changes to the collection to be queued while iterations are ongoing. When all iterations are completed, the queued modifications are made. The event channel attributes of `busy_hwm` and `max_write_delay` allow bounds to be set on how many iterators access the collection at a time and how many iterators may access it before modification occurs. See 24.3.6.2 for details of these attributes and how to set them.

24.5.2 Event Channel Resource Factory Options

The remainder of this section describes the individual options interpreted by the default and TPC event channel factories. These options are applied to the default event channel resource factory by the service configurator as described in 24.5. In addition, we briefly point out which options are affected by the use of the two scheduling factories contained in `EC_Kokyu_Factory.h` and `EC_Sched_Factory.h`. See these factory implementations for further details on their use.

24.5.2.1 ECConsumerControl *control_policy*

Values for *control_policy*

`null` (default)	Do not discard dangling consumers.
`reactive`	Use a reactive policy to discard dangling consumers.

Description This option specifies the policy used when dealing with dangling consumers. The null control policy never disconnects ill-behaved consumers. The reactive policy disconnects consumers upon the first communication failure.

Usage Use the reactive control policy when consumers could possibly be destroyed without disconnecting. Use the default control policy, null, when all consumers are guaranteed to disconnect properly (or not at all) and you do not want to incur the overhead of the reactive policy.

Impact A null consumer control strategy causes degraded throughput when consumers are destroyed without first disconnecting. The reactive strategy requires slightly more overhead in normal operation, may result in consumers having to reconnect when the network quality is degraded, and requires slightly more memory.

See Also 24.5.2.2, 24.5.2.16

Example

```
static EC_Factory "-ECConsumerControl reactive"
```

24.5.2.2 ECConsumerControlPeriod *period*

Description Sets the period (in microseconds) that the reactive consumer control policy uses to poll the state of the consumers. The default period is 5000000 (5 seconds).

Usage For event channels using the reactive consumer control policy, use this option to control the time to wait between attempted *pings* on each consumer. The reactive consumer control strategy object pings the consumer by invoking `CORBA::Object::_non_existent()` on the consumer's object reference; this is a synchronous call. The `-ECConsumerControlPeriod` option is ignored when the consumer control policy is not `reactive`.

Impact Shorter periods require more bandwidth and processing to validate the existence of the consumers. Longer periods consume less of these resources. You can disable the ping altogether by setting the period to zero.

See Also 24.5.2.1, 24.5.2.17

Example

```
static EC_Factory "-ECConsumerControl reactive -ECConsumerControlPeriod 1000000"
```

24.5.2.3 ECConsumerControlTimeout *timeout*

Description Sets the relative round-trip timeout (in microseconds) that the reactive consumer control policy uses for polling consumers. The default timeout is 10000 (10 milliseconds).

Usage For event channels using the `reactive` consumer control policy, use this option to control the time the event channel waits for a consumer to respond to an attempted *ping*. The reactive consumer control strategy object pings the consumer by invoking `CORBA::Object::_non_existent()` on the consumer's object reference; this is a synchronous call. Failure to respond within the specified timeout period results in the event channel classifying that ping as a communication failure for that consumer. The `-ECConsumerControlTimeout` option is ignored when the consumer control policy is not `reactive`.

Impact Smaller timeout values may result in more timeout failures and consumers being disconnected more often. A larger timeout value means it takes longer to detect and remove dead consumers.

See Also 24.5.2.1, 24.5.2.2

Example

```
static EC_Factory "-ECConsumerControl reactive -ECConsumerControlTimeout 50000"
```

24.5.2.4 ECConsumerValidateConnection *enabled*

Values for *enabled*

0 (default)	The connection from the event channel to the consumer is lazily made, probably when the first event is pushed.
1	The connection is established during the consumer's `connect_push_consumer()` call.

Description Enabling this option forces the connection from the event channel to the consumer to be completed before the `connect_push_consumer()` call returns.

Usage Use this when connections are slow to establish (such as when running on a WAN).

Impact Connecting consumers can take much longer with this option enabled.

Example

```
static EC_Factory "-ECConsumerValidateConnection 1"
```

24.5.2.5 ECDispatching *dispatching_strategy*

Values for *dispatching_strategy*

reactive (default)	The reactive strategy delivers events to consumers on the same thread that received them.
priority	The priority strategy launches a thread for each priority and dispatches each event on the thread corresponding to its priority. This value is only allowed when the TAO_EC_Sched_Factory is used.
kokyu	The kokyu strategy is similar to the priority strategy but is integrated with the kokyu scheduler. This value is only allowed when the TAO_EC_Kokyu_Factory is used.
mt	The mt (multithreaded) strategy delivers events on separate threads from the one that received them, by randomly selecting a thread from a pool. The number of threads in the pool defaults to one, but can be controlled via the -ECDispatchingThreads option.

Description This option controls the dispatching strategy that the event channel uses. The default strategy is reactive.

Note *This option is ignored when the Thread Per Consumer (TPC) event channel resource factory is used. The TPC factory always uses a single dispatching thread for each consumer.*

Usage This strategy determines the order in which the event channel delivers events as well as its overall throughput. The multithreaded strategy should allow for greater performance than the reactive model. The reactive strategy is appropriate when it is acceptable and/or desired for all events to be delivered in the order they are received. The multithreaded strategy is especially effective at reducing the maximum event delivery latency and decoupling the suppliers from the consumers' execution time, especially in the collocated case.

The priority and kokyu strategies help prevent priority inversions in the event channel by delivering high priority events first. Each requires the loading of its respective library and factory. Using either with the default factory results in a warning message. They differ only in the scheduler that they use. The kokyu value can also be followed by an optional policy (sched_fifo or sched_rr) and scope (system or thread).

Impact The reactive model is inappropriate when slow processing by individual consumers can affect other users, greater throughput is desired, or the system requires strict prioritization of events. Note that the reactive strategy delivers events on the same thread where they are received. The ORB's configuration determines the receiving thread. The multithreaded strategy may increase the time required to dispatch an event in lightly loaded event channels and also require the allocation of additional resources.

See Also 19.3.5, 24.5.2.6

Examples

```
static EC_Factory "-ECDispatching mt"

static EC_Factory "-ECdispatching kokyu SCHED_OTHER -ECScheduling kokyu"
```

24.5.2.6 ECDispatchingThreads *nthreads*

Description By default, the multithreaded dispatching strategy creates one thread to use for the delivery of supplier-originated events to consumers. Use this option to specify a different number of threads to be created and used.

Usage Using the multithreaded dispatching strategy with the default of one thread provides the user the benefit of separating the dispatching thread from the receiving thread. This allows for a greater decoupling between suppliers and consumers. Use this option to specify additional dispatching threads which results in additional decoupling between consumers as well as potentially increased throughput.

Impact Specifying additional dispatching threads consumes additional resources.

See Also 24.5.2.5

Example

```
static EC_Factory "-ECDispatching mt -ECDispatchingThreads 5"
```

24.5.2.7 ECDispatchingThreadFlags *thread_flags:priority*

Description This option specifies the thread flags and priority for any dispatching threads that are created by the multithreaded or thread per consumer dispatching strategies. The full set of thread flags that can be passed to this option is:

```
THR_CANCEL_DISABLE
THR_CANCEL_ENABLE
THR_CANCEL_DEFERRED
THR_CANCEL_ASYNCHRONOUS
THR_BOUND
THR_NEW_LWP
THR_DETACHED
THR_SUSPENDED
THR_DAEMON
THR_JOINABLE
THR_SCHED_FIFO
THR_SCHED_RR
THR_SCHED_DEFAULT
THR_EXPLICIT_SCHED
THR_SCOPE_SYSTEM
THR_SCOPE_PROCESS
```

Multiple flags should be separated with the “|” character. The priority specified must be an integer and should be an allowed priority for that user id, platform, and set of thread flags.

See Also 24.5.2.5

Example

```
static EC_Factory "-ECDispatching mt -ECDispatchingThreadFlags THR_BOUND|
THR_NEW_LWP:5"
```

24.5.2.8 ECFiltering *filter_strategy*

Values for *filter_strategy*

`null`	Disables consumer-specified filtering. Each consumer is given a null filter that passes all events.
`basic` (default)	Enables consumer-specified filtering for the event channel. Each consumer installs filters as specified in the subscription. The basic strategy enables all filtering types but limits the nesting of filter groups to two levels.
`prefix`	Enables consumer-specified filtering for the event channel. Each consumer installs filters as specified in the subscription. The prefix strategy enables all filtering types with no limits on the nesting of filter groups.
`priority`	Enables consumer-specified filtering for the event channel and also collaborates with the scheduler to build the dependency graph. This value is only allowed when the `TAO_EC_Sched_Factory` is used.
`kokyu`	Enables consumer-specified filtering for the event channel and also collaborates with the kokyu scheduler to build the dependency graph. This value is only allowed when the `TAO_EC_Kokyu_Factory` is used.

Description This option specifies the type of filter builder that the event channel uses to construct the filter tree for each consumer. Specifying different strategies allows the disabling of filtering when it is not desired or the enabling of dependency graph building by the scheduling service.

Usage Either the basic or prefix filtering strategy is usually the desired strategy. The prefix strategy allows complex nesting of filter groups as discussed in 24.3.4.10, but requires filters to be constructed with additional *prefix* information. By specifying the null filtering strategy, the event channel can be caused to ignore the filters as specified in the subscriptions. The priority or kokyu filtering strategies should be enabled when the system is being used to collect dependency information. This would most likely be done in conjunction with the `-ECScheduling` option.

Impact Using the null strategy may cause consumers to process more events and result in extra network traffic. The priority strategy should only be used when collecting dependency information as it is otherwise superfluous. The basic strategy limits the complexity of the filters allowed and may result in the event channel misinterpreting certain complex consumer filters.

See Also 24.5.2.15

Example

```
static EC_Factory "-ECFiltering prefix"
```

24.5.2.9 ECObserver *observer_strategy*

Values for *observer_strategy*

`null` (default)	Disables the observer feature.
`basic`	Enables the observer feature.
`reactive`	A variant of the basic strategy, where observers are automatically removed from the set of observers when the are unreachable.

Description Observers are objects that can be connected to the event channel and notified whenever consumers and suppliers are connected and disconnected. This option is used to enable the notification of observers.

Usage By default, the observer feature is not enabled, which should allow for more efficient operation of the event channel. If the application defines and uses observers, the basic strategy must be used to make them effective. When gateway objects are utilized to federate event channels, then the basic strategy should be used because the gateways use observers internally in their implementation. The reactive strategy allows for unreachable observers to be removed from the set of observers.

Impact The basic strategy may result in slightly less efficient connection and disconnection. When the basic strategy is used with observers that are unreachable, it continues to try to notify them of each connect and disconnect. This can slow connects and disconnects.

The null strategy causes observers to not be notified and does not allow the use of gateways.

Example

```
static EC_Factory "-ECObserver basic"
```

24.5.2.10 ECProxyConsumerLock *lock_strategy*

Values for *lock_strategy*

`null`	Do not use any locking on the proxy consumers.
`thread` (default)	Use a thread-safe lock on the proxy consumers.
`recursive`	Use a recursive thread-safe lock on the proxy consumers.

Description This option defines the type of lock to use in synchronizing access to the proxy consumer objects.

Usage Single threaded applications can use the null lock to increase the efficiency of the consumer proxy. Multithreaded applications may need to set the lock to recursive in some cases where operations on the proxy consumer may cause recursive access to the proxy consumer. In all other situations, the thread lock should be used.

Impact The null lock causes problems in applications that access the proxy from more than one thread. The thread lock causes additional locking overhead that may not be needed in applications that restrict proxy access to a single thread. The recursive lock is even more expensive than the thread lock, but is required by applications that must recursively access the lock.

See Also 24.5.2.13

Example

```
static EC_Factory "-ECProxyConsumerLock recursive"
```

24.5.2.11 ECProxyPushConsumerCollection *flags*

Description This switch controls the type of collection the event channel uses to hold consumer proxies. The flags passed describe the characteristics of the desired collection. Colons should separate the flags (e.g., `mt:list`). The allowable flags are described in Table 24-13. Only one flag per type should be specified.

Table 24-13 Collection Type Flags

Flag	Type	Description
`mt` (default)	Synchronization	Use a thread-safe lock for the collection.
`st`		Use a null lock for the collection.
`list` (default)	Collection	Implement the collection using an ordered list.
`rb_tree`		Implement the collection using a red-black tree.
`copy_on_read`	Iterator	Before initiating an iteration of the collection, a copy of the complete collection is performed.
`copy_on_write`		Before initiating a modification to the collection, a copy of the complete collection is performed.
`delayed`		Changes that cannot be made immediately are queued for later execution.
`immediate` (default)		Threads block until they can execute a change to the collection.

For a more detailed discussion of the collection types see 24.5.1.5.

Usage Applications that can guarantee that a consumer proxy collection is only accessed from a single thread can specify the `st` flag to improve performance.

Event channels that connect and disconnect suppliers often and wish to optimize these operations (at the expense of iteration speed), should specify the `rb_tree` flag.

Applications that use the `immediate` flag must guarantee that the thread iterating over the proxy collection does not attempt to modify the collection, as this invalidates the iterator. Specifying a separate dispatching thread is one way to accomplish this. If you wish to minimize priority inversions between publication and supplier connections/disconnections, then use the `delayed` flag. Collections using the `copy_on_read` flag are only applicable for systems with small numbers of consumer proxies that require low latencies for proxy collection modifications. Collections using the `copy_on_write` flag are only applicable for systems with small numbers of consumer proxies that require low latencies for proxy collection iterations.

Impact The `mt` flag incurs additional overhead over the `st` flag during connection/disconnection of suppliers and iteration over the collection.

List-based collections result in slower updates to the collection. Red-black tree collections are slower during iteration over the collection.

Immediate update of consumer proxy collections (during connection or disconnection of suppliers) may cause priority inversions because of the long-lived locks involved. Copy on read collections incur dynamic allocation and copy costs for each iteration of the proxy collection. Copy on write collections incur dynamic allocation and copy costs for each modification to the proxy collection. Delayed updates to collections can result in long intervals between the requested change and its actual occurrence.

See Also 24.5.2.12

Example

```
static EC_Factory "-ECProxyPushConsumerCollection mt:delayed"
```

24.5.2.12 ECProxyPushSupplierCollection *flags*

Description This switch controls the type of collection the event channel uses to hold supplier proxies. The flags passed describe the characteristics of the desired collection. Colons should separate the flags (e.g., `mt:list`). The flags are described in Table 24-14. Only one flag per type should be specified.

Table 24-14 Collection Type Flags

Flag	Type	Description
`mt` (default)	Synchronization	Use a thread-safe lock for the collection.
`st`		Use a null lock for the collection.
`list` (default)	Collection	Implement the collection using an ordered list.
`rb_tree`		Implement the collection using a red-black tree.
`copy_on_read`	Iterator	Before initiating an iteration of the collection, a copy of the complete collection is performed.
`copy_on_write`		Before initiating a modification to the collection, a copy of the complete collection is performed.
`delayed`		Changes that cannot be made immediately are queued for later execution.
`immediate` (default)		Threads block until they can execute a change to the collection.

For a more detailed discussion of the collection types see 24.5.1.5.

Usage Applications that can guarantee that a supplier proxy collection is only accessed from a single thread can specify the `st` flag to improve performance.

Event channels that connect and disconnect consumers often and wish to optimize these operations (at the expense of iteration speed), should specify the `rb_tree` flag.

Applications that use the `immediate` flag must guarantee that the thread iterating over the proxy collection does not attempt to modify the collection, as this invalidates the iterator. Specifying a separate dispatching thread is one way to accomplish this. If you wish to minimize priority inversions between publication and consumer connections/disconnections, then use the `delayed` flag. Collections using the `copy_on_read` flag are only applicable for systems with small numbers of supplier proxies that require low latencies for proxy collection modifications. Collections using the `copy_on_write` flag are only applicable for systems with small numbers of supplier proxies that require low latencies for proxy collection iterations.

Impact The `mt` flag incurs additional overhead over the `st` flag during connection/disconnection of consumers and iteration over the collection.

List-based collections result in slower updates to the collection. Red-black tree collections are slower during iteration over the collection.

Immediate update of supplier proxy collections (during connection or disconnection of consumers) may cause priority inversions because of the long-lived locks involved. Copy on read collections incur dynamic allocation and copy costs for each iteration of the proxy collection. Delayed updates to collections can result in long intervals between the requested change and its actual occurrence.

See Also 24.5.2.11

Example

```
static EC_Factory "-ECProxyPushSupplierCollection mt:delayed"
```

24.5.2.13 ECProxySupplierLock *lock_strategy*

Values for *lock_strategy*

`null`	Do not use any locking on the proxy suppliers.
`thread` (default)	Use a thread-safe lock on the proxy suppliers.
`recursive`	Use a recursive thread-safe lock on the proxy suppliers.

Description This option defines the type of lock to use in synchronizing access to the proxy supplier objects.

Usage Single threaded applications can use the null lock to increase the efficiency of the push supplier. Multithreaded applications may need to set the lock to recursive in some cases where operations on the proxy supplier may cause recursive access to the proxy supplier. In all other situations, the thread lock should be used.

Impact The null lock causes problems in applications that access the proxy from more than one thread. The thread lock causes additional locking overhead that may not be needed in applications that restrict proxy access to a single thread. The recursive lock is even more expensive than the thread lock, but is required by applications that must recursively access the lock.

See Also 24.5.2.10

Example

```
static EC_Factory "-ECProxySupplierLock recursive"
```

24.5.2.14 ECQueueFullServiceObject *service_name*

Description This option specifies a service name that is used to find a Queue Full service object. That service implements a strategy for what to do when a dispatching queue is full. A dispatching queue is considered full when it reaches the high water mark (in number of events) defined by the constant TAO_EC_QUEUE_HWM, which has a default value of 16384.

By default, this option is set to `EC_QueueFullSimpleActions` which is a statically configured service we can configure with a separate static service configurator directive. It only takes two possible options, `wait` and `discard`. The default `wait` option causes any supplier threads wanting to queue an event onto a full dispatching queue to block and wait for the queue to fall below the high water mark. The `discard` option causes the supplier thread to discard the event being processed when it encounters a full dispatching queue.

This option only affects the multithreaded and thread per consumer dispatching strategies.

Usage The default service and associated `wait` strategy can cause events to "back up" in the event channel when the network or consumers are unable to process events fast enough. Eventually, this back pressure can affect the supplier processes. In this situation, you should either use the default service's `discard` strategy or provide your own service object that implements custom strategies for handling full dispatching queues.

See Also 24.5.2.5

Example There are two basic usage scenarios related to this option. If the simple Queue Full service object is sufficient for an application, then this option does not actually need to be set and the simple strategy can be configured through its own directive:

```
static EC_QueueFullSimpleActions "discard"
```

If the simple strategy is not sufficient, the application developer needs to write their own Queue Full service class and use this option to identify it. See the `TAO_EC_Queue_Full_Service_Object` class definition in `$TAO_ROOT/orbsvcs/orbsvcs/Event/EC_Dispatching_Task.h` for details. Your service needs to derive from this class and implement the `queue_full_action` member function.

```
static EC_Factory "-ECQueueFullServiceObject MyFancyQueueFullStrategy"
```

24.5.2.15 ECScheduling *scheduling_strategy*

Values for *scheduling_strategy*

null (default)	Disables collaboration with the scheduling service for dependency list collection.
group	Schedule/filter events as a set when evaluating for delivery.
priority	Enables collaboration with the scheduler for the building of the dependency list. This value is only allowed when the `TAO_EC_Sched_Factory` is used.
kokyu	Enables collaboration with the kokyu scheduler for the building of the dependency list. This value is only allowed when the `TAO_EC_Kokyu_Factory` is used.

Description This scheduling strategy controls the scheduling of events for delivery and other scheduler-related coordination. The null and group scheduling options specify whether events should be delivered individually or as groups, respectively. When the null scheduler is specified, events are always evaluated and delivered one at a time, regardless of how they are packaged by the supplier. When the group scheduler is used, events are kept in the group they were published in by the supplier and all events from the original set that pass a consumer's filter are delivered as a set to that consumer.

The priority and kokyu scheduling strategies use their respective scheduling libraries to schedule event delivery. These strategies are also used for the purpose of building the dependency lists for use by these schedulers.

Usage The priority and kokyu strategies are used when the application is using one of the real-time schedulers.

Impact The group strategy is more efficient when groups of events are published and delivered together.

Example

```
static EC_Factory "-ECScheduling group"
```

24.5.2.16 ECSupplierControl *control_policy*

Values for *control_policy*

`null` (default)	Do not discard dangling supplier.
`reactive`	Use a reactive policy to discard dangling supplier.

Description This option specifies the policy used when dealing with dangling suppliers. The null control policy never disconnects ill-behaved suppliers. The reactive policy disconnects suppliers at the first communication failure.

Usage Use the reactive control policy when suppliers may be destroyed without disconnecting. Use the default control policy (null) when you can guarantee that all suppliers disconnect properly (or not at all) and you do not want to incur the overhead of the reactive policy.

Impact A null supplier control strategy causes degraded throughput when suppliers are destroyed without first disconnecting. The reactive strategy requires slightly more overhead in normal operation, may result in suppliers inability to send messages to the event channel or having to reconnect when the network quality is bad, and requires slightly more memory.

See Also 24.5.2.1, 24.5.2.17

Example

```
static EC_Factory "-ECSupplierControl reactive"
```

24.5.2.17 ECSupplierControlPeriod *period*

Description Set the period (in microseconds) the reactive supplier control policy uses to poll the state of the suppliers. The default period is 5000000 (5 seconds).

Usage For event channels using the reactive supplier control policy, use this option to control the time to wait between attempted *pings* on each supplier. The reactive supplier control strategy object pings the supplier by invoking `CORBA::Object::_non_existent()` on the supplier's object reference; this is a synchronous call. The `-ECSupplierControlPeriod` option is ignored when the supplier control policy is not `reactive`.

Impact Shorter periods require more bandwidth and processing to validate the existence of the suppliers. Longer periods consume less of these resources. You can disable the ping altogether by setting the period to zero.

See Also 24.5.2.2, 24.5.2.16

Example

```
static EC_Factory "-ECSupplierControl reactive -ECSupplierControlPeriod 1000000"
```

24.5.2.18 ECSupplierControlTimeout *timeout*

Description Sets the relative round-trip timeout (in microseconds) used for polling suppliers. This timeout is used both for polling pull model suppliers and by the `reactive` supplier control strategy. The default timeout is 10000 (10 milliseconds).

Usage For event channels using the `reactive` supplier control policy, use this option to control the time the event channel waits for a supplier to respond to an attempted *ping*. For pull suppliers, use this option to control the time the event channel waits for each `pull()` call to a pull supplier. The reactive supplier control strategy object pings the supplier by invoking `CORBA::Object::_non_existent()` on the supplier's object reference; this is a synchronous call. Failure to respond within the specified timeout period results in the event channel classifying that ping as a communication failure for that supplier.

Impact Smaller timeout values may result in more timeout failures and suppliers being disconnected more often. A larger timeout value means it takes longer to detect and remove dead suppliers.

See Also 24.5.2.16, 24.5.2.17

Example

```
static EC_Factory "-ECSupplierControl reactive -ECSupplierControlTimeout 50000"
```

24.5.2.19 ECSupplierFiltering *supplier_filter_strategy*

Values for *supplier_filter_strategy*

`null`	Disables supplier-based filtering. This results in the event channel keeping a global collection of consumers and attempting a push to each of them.
`per-supplier` (default)	Enables supplier-based filtering. This results in the event channel maintaining a (smaller) collection of interested consumers for each supplier and only attempting a push to this subset of consumers.

Description This option controls supplier-based filtering and does not affect which consumers receive events. It does affect the internal data structures of the event channel that determine its memory footprint and performance.

Note *This option can also be specified as* `-ECSupplierFilter` *and accepts the same values.*

Usage Generally, the per-supplier strategy produces the best performance. The null strategy allows for less memory usage (one global list of supplier proxies in the event channel instead of separate ones for each supplier) and may perform essentially the same for cases where all consumers receive events from all or nearly all of the suppliers.

Impact The per-supplier strategy consumes (relatively) large amounts of memory for systems with high numbers of both consumers and suppliers, especially for the case where all consumers receive events from all suppliers. It also increases the connection and disconnection times for consumers and suppliers. The null strategy decreases overall performance in cases where large numbers of consumers and suppliers exist, and each consumer is only interested in the events published by a small number of suppliers.

When suppliers don't properly specify their event publication via the `SupplierQOS` structure at connection time, the per-supplier strategy may sometimes cause events to not be delivered to all eligible consumers.

Example

```
static EC_Factory "-ECSupplierFiltering null"
```

24.5.2.20 ECTimeout *timeout_dispatching_strategy*

Values for *timeout_dispatching_strategy*

`reactive` (default)	The reactive strategy delivers the timeout events to consumers on the main reactor thread.
`priority`	The priority strategy launches a thread for each priority and dispatches each timeout event on the thread corresponding to its priority. This option is not yet implemented.
`kokyu`	The kokyu strategy launches a thread for each priority and dispatches each timeout event on the thread corresponding to its priority. This option is not yet implemented.

Description This option controls what thread is used to dispatch timeout events to the consumers.

Usage The reactive strategy is the only one currently implemented in the default factory. The reactive strategy uses on of ORB's processing threads to push the timeout events. The priority and kokyu strategies provide the ability to ensure that high priority timeouts are delivered first by using their respective scheduling libraries.

Impact The reactive strategy may cause priority inversions. The priority and kokyu strategies may cause an increase in the time required to deliver a timeout event on lightly loaded event channels. The priority and kokyu strategies may cause the allocation of additional resources.

See Also 24.5.2.5

Example

```
static EC_Factory "-ECTimeout reactive"
```

24.5.2.21 ECTPCDebug

Description This option enables the printing of additional debug information when the thread per consumer (TPC) dispatching strategy is used. It is only available when the TPC factory is used. See 24.5 for how to specify this factory.

Usage This option is intended for use in debugging event channel issues when using the TPC dispatching strategy.

Impact The additional debug messages slow down the event channel.

Example

```
static EC_Factory "-ECTPCDebug"
```

24.5.2.22 ECUseORBId *orb-id*

Description Sets the name of the ORB that the default factory implementation uses. The default factory creates strategy objects that use this ORB to perform remote invocations and to gain access to the ORB's reactor.

Usage This option is only useful in applications that create multiple ORBs and activate the event channel in one of them. Use it to ensure that the objects created by the default factory use the same ORB as the event channel and related objects.

Impact This option may cause the creation of a new ORB (and associated resources), if the ORB with the given name has not been initialized.

Example

```
static EC_Factory "-ECUseORBId Orb2"
```

24.6 The IIOP Gateway Factory

The IIOP gateway factory is responsible for creating strategy objects that control the behavior of the CORBA gateways used for federating event channels as described in 24.3.8.1. The behavior of the CORBA (or IIOP) gateway is controlled by using the service configurator file to select the appropriate behaviors for the default factory implementation.

This resource factory is registered with the service configurator using the name `EC_Gateway_IIOP_Factory`. It is statically registered with the service configurator, so the static directive is used to supply initialization options to it. To change the behavior of the IIOP gateway factory, add a line similar to the line shown below to your service configuration file.

```
static EC_Gateway_IIOP_Factory "-ECGIIOPConsumerECControl reactive"
```

For these options to be effective, you must make sure that the following function call occurs in every process containing CORBA gateways *before* the ORB is initialized:

```
TAO_EC_Gateway_IIOP_Factory::init_svcs ();
```

This creates an IIOP gateway factory and statically registers it. If you forget to do this, the service configurator is not able to find the `EC_Gateway_IIOP_Factory` to initialize it.

The `-ORBSvcConf` option allows you to use file names other than `svc.conf` for service configurator initialization. See 17.13.63 for more information on this option.

Default values for many of the IIOP gateway factory's options are defined as preprocessor macros in `$TAO_ROOT/orbsvcs/orbsvcs/Event/ECG_Defaults.h`. You can use your own project-specific defaults by setting these macros in your `config.h` file and recompiling the event service.

24.6.1 Option Overview

This section provides an overview of the configuration options supported by the default `EC_Gateway_IIOP_Factory`. Table 24-15 presents a brief summary of the available options for this factory followed by short

descriptions of these options. The following section provides full documentation for each option.

Table 24-15 IIOP Gateway Factory Options

Option	Section	Description
`ECGIIOPConsumerECControl {null | reactive | reconnect}`	24.6.2.1	Define the policy for handling consumer event channels that can't be communicated with.
`ECGIIOPConsumerECControlPeriod period`	24.6.2.2	Define the period in microseconds of the reactive and reconnect control policies.
`ECGIIOPConsumerECControlTimeout timeout`	24.6.2.3	Round-trip timeout in microseconds for the consumer event channel ping.
`ECGIIOPUseConsumerProxyMap {0 | 1}`	24.6.2.4	Flag that specifies whether a map of consumer proxies should be used or a single consumer proxy.
`ECGIIOPUseORBId orbid`	24.6.2.5	Specifies the id of the ORB that the factory uses.
`ECGIIOPUseTTL {0 | 1}`	24.6.2.6	Flag that specifies whether the ttl field of the event is used to limit gateway propagation.

The consumer control policy works similarly to the control policies on the event channel, but in the gateway's case they determine how the downstream or consumer event channel is treated when the gateway fails to invoke a request on it. The control period and timeout affect the period and round-trip timeout of the periodic pings performed by the reactive and reconnect control policies.

Consumer proxy map and TTL (time to live) features can each be disabled for applications that would benefit from their avoidance. Most applications should keep the defaults. Specification of an alternate ORB ID for the factory is beneficial when the gateway's process is using multiple ORBs.

24.6.2 IIOP Gateway Factory Options

The remainder of this section describes the individual options interpreted by the IIOP Gateway factory. These options are applied to the IIOP Gateway factory by the service configurator as described in 24.6.

24.6.2.1 ECGIIOPConsumerECControl *control_policy*

Values for *control_policy*

`null` (default)	Do not discard consumer event channels that cannot be contacted.
`reactive`	Use a reactive policy to discard consumer event channels that cannot be contacted.
`reconnect`	Attempts to periodically reconnect to event channels after they cannot be contacted.

Description This option specifies the policy used by the gateway when dealing with consumer event channels that suffer failures when the gateway pushes events on them. The consumer event channel is the event channel that is connected as a consumer to the gateway. The null control policy never disconnects consumer event channels that suffer communication failures and continues to try to push events to those channels. The reactive policy discards consumer event channels upon their first communication failure and never reconnects them. The reconnect control policy also disconnects consumer event channels after their first failure, but attempts to periodically reconnect to that event channel. Both the reactive and reconnect policies periodically poll consumer event channels to determine if they are still available. See the `ECGIIOPConsumerECControlPeriod` and `ECGIIOPConsumerECControlTimeout` options for details of this polling.

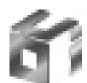
Usage Use the reconnect control policy for the greatest robustness when using IIOP gateways. Use the default control policy, null, when you can guarantee the reliability of your event channels and you do not want to incur the overhead of the reactive or reconnect policies.

Impact A null consumer control strategy causes degraded throughput when consumer event channels crash unexpectedly or cannot be contacted. Since the reactive strategy never reconnects consumer channels, it causes permanent breaks in the federation. The reconnect strategy requires slightly more overhead in normal operation, may result in reconnections when the network quality is degraded, and requires slightly more memory. When using the reconnect policy, persistent object references for event channels are required.

See Also 24.6.2.2, 24.6.2.3

Example

```
static EC_Gateway_IIOP_Factory "-ECGIIOPConsumerECControl recconnect"
```

24.6.2.2 ECGIIOPConsumerECControlPeriod *period*

Description Set the period (in microseconds) the reactive and reconnect control policies use to poll the state of the consumer event channels. The default period is 5000000 (5 seconds).

Usage For gateways using the reactive or reconnect control policies, use this option to specify the time to wait between attempted *pings* on each consumer event channel. The reactive and reconnect control strategy objects ping the consumer event channel by invoking `CORBA::Object::_non_existent()` on the event channel's object reference; this is a synchronous call. This option is ignored when the control policy is `null`.

Impact Shorter periods require more bandwidth and processing to validate the existence of the event channels. Longer periods consume less of these resources. You can disable the ping altogether by setting the period to zero.

See Also 24.6.2.1, 24.6.2.3

Example

```
static EC_Gateway_IIOP_Factory "-ECGIIOPConsumerECControl recconnect
-ECGIIOPConsumerECControlPeriod 1000000"
```

24.6.2.3 ECGIIOPConsumerECControlTimeout *timeout*

Description Sets the relative round-trip timeout (in microseconds) that the reactive and reconnect control policies use for polling consumer event channels. The default timeout is 10000 (10 milliseconds).

Usage For gateways using the reactive or reconnect control policies, use this option to control the time the gateway waits for a consumer event channel to respond to an attempted *ping*. These control strategies ping the consumer by invoking `CORBA::Object::_non_existent()` on the consumer event channel's object reference; this is a synchronous call. Failure to respond within the specified timeout period results in the control policy classifying that ping as a communication failure for that consumer. This option is ignored when the consumer control policy is null.

Impact Smaller timeout values may result in more timeout failures and consumer event channels being disconnected more often. A larger timeout value means it takes longer to detect and remove event channels that have crashed or become isolated.

See Also 24.6.2.1, 24.6.2.2

Example

```
static EC_Gateway_IIOP_Factory "-ECGIIOPConsumerECControl recconnect
-ECGIIOPConsumerECControlTimeout 50000"
```

24.6.2.4 ECGIIOPUseConsumerProxyMap *flag*

Values for *flag*

0	Use a single consumer proxy for all events the gateway processes.
1 (default)	Use a consumer proxy map with a separate consumer proxy for each source ID the gateway processes.

Description How many consumer proxies the gateway uses on the consumer event channel affects the performance of your federated event channel application. Using the consumer proxy map means the gateway uses a separate consumer proxy for each source ID and allows the consumer event channel to use supplier filtering to improve efficiency. It also means that changes in the subscription set of the gateway can be handled in a more incremental manner. Not using the map means only a single consumer proxy is used and it supplies all events to the consumer event channel.

Usage By default, the consumer proxy map is enabled; pass a value of zero to disable it. Most applications probably want this feature enabled but some applications with memory-usage concerns may benefit from disabling it.

Impact Using the consumer proxy map consumes additional resources in the gateway, but should be more efficient in most applications. Disabling the consumer proxy map consumers less resources, but usually at a performance cost.

See Also 24.5.2.19

Example

```
static EC_Gateway_IIOP_Factory "-ECGIIOPUseConsumerProxyMap 0"
```

24.6.2.5 ECGIIOPUseORBId *orbid*

Description Sets the name of the ORB used by the gateway factory. The factory creates strategy objects that use this ORB to perform remote invocations and to gain access to the ORB's reactor.

Usage This option is only useful in applications that create multiple ORBs and activate the gateways in one of them. Use it to ensure that the objects created by the IIOP gateway factory use the same ORB as the event channel and related objects. When not specified, the default ORB is used.

Impact This option may cause the creation of a new ORB (and associated resources), if the ORB with the given name has not been initialized.

See Also 24.5.2.22

Example

```
static EC_Gateway_IIOP_Factory "-ECGIIOPUseORBId myorb"
```

24.6.2.6 ECGIIOPUseTTL *flag*

Values for *flag*

`0`	Do not use the ttl field. All events pushed to the gateway are pushed to the consumer event channel.
`1` (default)	The ttl field of the event header structure is used to limit the number of gateways an event can pass through.

Description This option determines whether the gateway uses the ttl field of the event header structure to limit the number of gateways that an event can pass though. When enabled, as it is by default, the ttl field is decremented each time an event passes through a gateway and if the ttl field is zero the gateway does not pass the event. Passing zero to this option disables this behavior and means all events are passed through all gateways.

Usage For normal gateway behavior, leave the ttl feature enabled. Disabling it may make things simpler for suppliers because they no longer have to set the ttl field which requires knowledge of the federation topology.

Impact When disabling the ttl functionality events cycle forever in topologies with recursive loops in the federation. This option has negligible effect on performance.

Example

```
static EC_Gateway_IIOP_Factory "-ECGIIOPUseTTL 0"
```

CHAPTER 25

Notification Service

25.1 Introduction

The OMG Notification Service version 1.1 specification (OMG Document formal/04-10-13) extends the OMG Event Service (Chapter 23) with many improvements, including: a well-defined event structure; event filtering; discovery of event types required or offered; configurable quality of service properties; and an optional event type repository. The basic architectural elements of the Event Service (consumers, suppliers, event channels, proxy consumers and suppliers, consumer and supplier administration interfaces) are preserved in the Notification Service specification. In fact, implementations of the Notification Service can still support existing Event Service consumers and suppliers without recompilation. New architectural elements (filters, event channel and filter factories, new structured supplier and consumer interfaces) have been added, and these newer features can be accessed using the familiar programming model introduced in the Event Service.

TAO's implementation of the OMG Notification Service supports a useful subset of the specification. It supports the push-model of event communication only; there is no support for the pull model. Certain optional

features from the specification are also unsupported, including the event type repository and typed events. Other differences between the specification and TAO's implementation will be noted in the relevant sections of this chapter.

25.2 Notification Service Architecture

Figure 25-1 shows the architecture of the Notification Service.

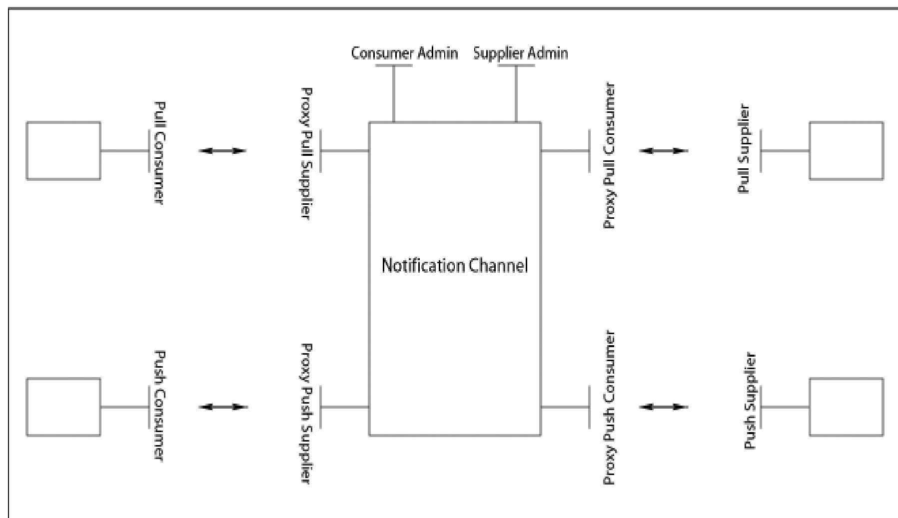
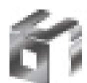


Figure 25-1 Notification Service Architecture

The Notification Service is hierarchical in structure, with a single `EventChannelFactory` supplying one or more `EventChannel` objects. Each event channel, in turn, supplies one or more `ConsumerAdmin` and `SupplierAdmin` objects, and each administration object supplies zero or more consumers or suppliers. Each of these objects is assigned a unique identifier that can be used to retrieve its object reference. Each "parent" interface provides an operation that may be used to enumerate its "children," and each child provides an attribute for accessing its parent.

The primary purpose of the administration (admin) interfaces is to allow grouping of consumers and suppliers with common filtering and quality of service (QoS) properties. As consumers and suppliers are created, they inherit the applicable QoS properties of their parent admin object. If you later change the QoS properties of an admin, its children will not be updated. Filtering



works differently. The consumers and suppliers do not inherit the parent's filters; instead, the admin filters are always *combined* with its children's filters using either a logical `AND` or a logical `OR` operation. Updating the filtering constraints on an admin object has an immediate affect on the event processing of its children.

Consumers and suppliers are derived from one of four different hierarchies. The original Event Service interfaces can still be used to communicate untyped events as simple CORBA `any`s. In addition, the Notification Service defines three new sets of interfaces; one for use with untyped events, one for structured events, and one for batches of events.

Regardless of the specific consumer and supplier interfaces used, the same basic model of communication is supported. Suppliers push events to the event channel and the event channel asynchronously pushes these events to connected consumers. Each event may pass through filters before being forwarded to consumers, and QoS properties may affect the processing of events as they travel through the various elements of the Notification Service architecture

The Notification Service supports mixing and matching of the various consumer and supplier interfaces. For example, you can supply events as sequences of structured events using the `SequenceProxyPushSupplier` interface, and consume them as individual `StructuredEvent`s using the `StructuredProxyPushConsumer` interface. There are well-defined translations among each of the event propagation mechanisms.

25.3 Notification Service Features

The Notification Service provides several new features, relative to the Event Service, for event-based communications, including:

- Structured Events
- Event Filtering
- Subscriptions
- Batched Events
- Quality of Service Properties

These features are discussed in the next several subsections.

25.3.1 The Structured Event Type

Like the Event Service, the Notification Service defines a model for communicating untyped events in the form of CORBA `any`s. Untyped events can be used with the new filtering and QoS features, but it is far easier to use the new `CosNotification::StructuredEvent` type. The structured event type has a well-defined data structure comprising three fixed header fields, a variable-length header portion of name-value pairs, a variable-length filterable body portion also of name-value pairs, and a remaining body portion, or payload, that consists of a single CORBA `any`.

Figure 25-2 shows the `CosNotification::StructuredEvent` type.

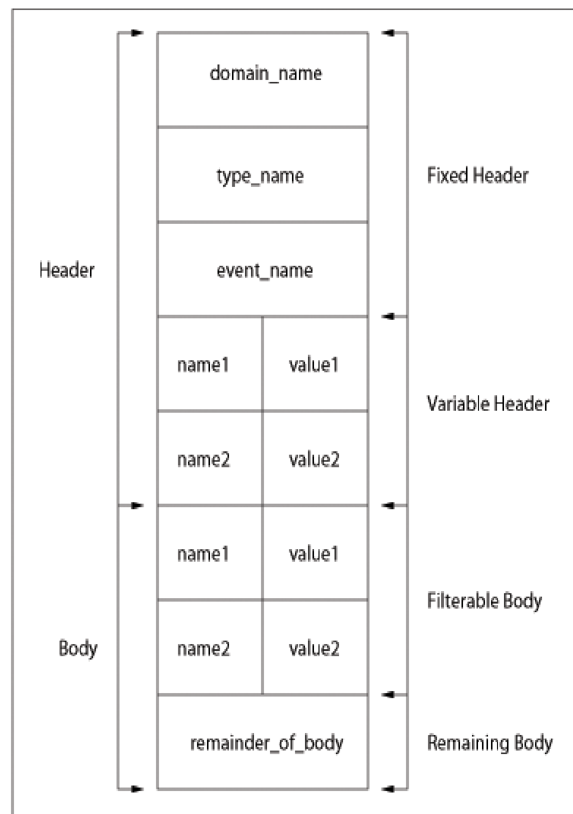


Figure 25-2 CosNotification::StructuredEvent

QoS settings for priority, timeout, and reliability may be specified in the variable header portion. The variable body portion can be easily used in

filtering constraints. The `StructuredEvent` is intended for use with applications that require strongly-typed events. Users of these events are able to map their application-specific events into a common data structure, thereby making various optimizations possible during event processing.

The Notification Service specification defines a `CosNotifyComm` module that contains `StructuredPushSupplier` and `StructuredPushConsumer` interfaces for communicating structured events.

25.3.1.1 The Structured Event Header

The structured event's header has two parts: a fixed header portion containing fields to identify the event's type and instance; and an optional list of name-value pairs. The three fixed header fields are all strings:

- `domain_name`—To identify the vertical industry domain to which this event belongs, such as telecommunications, finance, or health-care.
- `type_name`—To further classify the event within its domain.
- `event_name`—To identify a specific event instance.

Together, the `domain_name` and `type_name` fields are known as the *event type*, and have special meaning for filtering and subscription processing.

The optional header portion consists of a sequence of zero or more name-value pairs in which QoS settings can be carried. TAO supports two well-known settings for this portion of the header:

- `priority`—To specify an integer priority value for the event.
- `timeout`—To specify a relative time after which the event will be discarded if it has not been delivered.

Other name-value pairs in the header can be used to carry data or application-specific header information, but you will normally use the name-value pairs in the body portion, instead.

25.3.1.2 The Structured Event Body

The data content of an event is mapped into the event body. The structured event body is divided into two parts:

- The filterable body portion is intended to carry the most interesting fields of the event, upon which the consumer is most likely to base filtering

decisions. It comprises a sequence of name-value pairs, in which each name is a string and each value is a CORBA `any`.

- The remaining body area is defined as a single CORBA `any` and is intended to carry large blocks of data related to the event. Although this field is considered separate from the filterable body area of the event, there are no restrictions against additional filtering on the contents of this data.

25.3.2 EventBatch Data Type

The Notification Service specification defines a sequence of `StructuredEvents` called a `CosNotification::EventBatch`. Within the `CosNotifyComm` module are `SequencePushSupplier` and `SequencePushConsumer` interfaces for communicating with batched events.

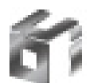
The `StructuredEvents` in an `EventBatch` received from a supplier are handled independently inside the Notification Service. There is no functional difference between using a series of CORBA calls to send one `StructuredEvent` at a time, and using a single CORBA call to send a batch of `StructuredEvents`. There may, however, be a performance benefit to sending a batch of events due to the reduced number of CORBA calls.

The QoS properties `MaximumBatchSize` and `PacingInterval` can be applied to consumers that register to receive event batches. `MaximumBatchSize` is used to specify how many individual `StructuredEvents` will be queued, and `PacingInterval` specifies how long `StructuredEvents` can be queued, before being delivered as a batch. These and other QoS properties are discussed in detail in 25.3.5.

There is no correlation between incoming batches and outgoing batches. For example: if a supplier pushes five events in a batch, a consumer should not expect to receive these five events in a single batch. They may be split across batch boundaries; intermixed with events from other consumers, etc.

25.3.3 Event Filtering

The most notable improvement the Notification Service specification introduces over the Event Service is the introduction of event filtering. Event filtering allows a consumer to subscribe to a precise set of events. TAO supports the Extended Trader Constraint Language (ETCL) filtering grammar as defined by the OMG. ETCL allows applications to create complex



expressions to describe which events should be allowed to pass through an element of the Notification Service architecture. Filters are usually applied to admin objects, but may also be used with `ProxyConsumer` or `ProxySupplier` objects.

Note *The filters described above are known as Forwarding filters, because they are used to determine which events should be forwarded to the next layer of the architecture. The Notification Service specification also defines a type of filters known as Mapping filters, which can be used to modify an event's properties as it passes through objects. Mapping filters are not supported in TAO's implementation of the Notification Service.*

Besides the Extended Trader Constraint Language (ETCL), TAO's implementation of the Notification Service also supports the original Trader Constraint Language (TCL) as defined by the Object Trader Service specification. For more information on standard TCL, see *Advanced CORBA Programming with C++*, 19.10. For more information on ETCL, see 2.4.2 in the Notification Service specification.

TAO's implementation of the Notification Service specification supports filtering for both untyped events and structured events. Filtering with structured events is easier than with untyped events, because you can simply add named fields into the variable body portion of the events and define filtering constraints using ETCL that reference the named fields.

Note *Filtering of untyped events is nominally supported in TAO, but is not recommended.*

25.3.3.1 Using Event Filtering

Depending upon the characteristics of your application, you can use filters to improve performance. For example, you could use a filter object with a supplier to ensure that the supplier publishes only those events that match a certain set of constraints, and to avoid populating the notification channel with unnecessary events. Similarly, you could use a filter object with a consumer to inform the notification channel of the types of events the consumer wants to receive and thereby avoid burdening the consumer with having to process unwanted events. In both cases, filtering can also help prevent unnecessary

network bandwidth consumption because unwanted events will not be sent at all. Keep in mind, however, that event filtering imposes a computational burden on the processes in which the notification channel itself is operating, over and above its normal responsibilities of receiving and dispatching events.

Filter objects are used to filter events based on a set of constraints provided by the application developer. Filter objects can be attached to an administration object or to individual proxy objects. When a filter object is attached to an administration object, event receiving and forwarding by all the associated proxies is affected. When a change is made to a filter object that is attached to an administration object, all the associated proxies are affected by the change. If you want to control the filtering behavior of each proxy individually, you can attach a filter to each proxy. In this way, the filter affects only the event receiving and forwarding behavior of that individual proxy.

Filters are first-class CORBA objects. Filter objects can be collocated with the notification channel process itself or they can be distributed in their own address spaces. The event channel provides a factory interface to create filter objects, and filters created with this factory reside in the same address space as the notification channel itself. You could also create filter objects separately from the event channel and attach them, by object reference. However, filtering via remote filter objects may introduce a significant performance penalty since each filter's `match()` operation would be invoked in a distributed fashion.

25.3.3.2 The Inter-Group Filter Operator

Note that each proxy can have two sets of filter objects associated with it, those that are associated with its managing administration object and those that are associated with that proxy itself. The `InterFilterGroupOperator` flag can be used to control whether each proxy created by the administration object will perform a logical `AND` or a logical `OR` on the results of the two filter sets in making its event receiving and forwarding decisions. If no filters are attached to an object, the default behavior is to pass all events.

The inter-group filter operator for TAO's default consumer and supplier admin objects is the OR operator. A common error is to use the default consumer admin (by calling event channels `default_consumer_admin` attribute), apply no filters to the admin, and apply filters to the supplier proxy. Because the admin object has no filters, it passes all objects. Because the

inter-group filter operator is OR, the supplier proxy passes all events to the consumer, *regardless* of the supplier proxy's filters.

One way to avoid this behavior is to always create your own admin objects using `new_for_consumers()` and specifying the AND filter operator. You can also force the default admin objects to apply the AND filter operator by using the `-DefaultConsumerAdminFilterOp` (see 25.7.3.4) and `-DefaultSupplierAdminFilterOp` (see 25.7.3.5) service configurator options.

Note *The Notification Service Specification does not define which filter operator should be used for the default admins. If your application is dependent on this, it may affect its portability to other Notification Service implementations.*

25.3.4 Offers and Subscriptions

Offers and subscriptions provide a mechanism to allow consumers to be notified whenever the set of offered event types changes, and for suppliers to be notified whenever the set of event types required by consumers changes. By using these offer and subscription notifications, applications can create *adaptive* suppliers and consumers that can change their filtering constraints dynamically to adapt to changes in the types of events actually being used in the system. Unlike filters, which work automatically, it is up to the application developer to make use of offer and subscription information in consumer and supplier implementations.

Note *The Notification Service specification (section 2.6.5) defines a mechanism that ties filtering to offer- and subscription-change notifications, but this feature is not supported in TAO.*

25.3.4.1 Offer Changes

A supplier uses the `offer_change()` operation to specify the event types it offers to the notification channel. The information passed via the `offer_change()` operation is used by the notification channel to aggregate a list of all the event types that its connected suppliers offer. For example, if three suppliers all offer event type “A”, then the channel only needs to notify the consumers of a change if all three suppliers stop supporting that event type. If a supplier decides to add a new event type or to remove an existing

event type from its offer, it can use the `offer_change()` operation again to inform the notification channel of the change in the types of events it supplies.

The `offer_change()` operation can be invoked on the proxy consumer object or on the supplier administration object. Invoking `offer_change()` on the proxy consumer affects only the offer for that particular supplier. Invoking the operation on the supplier administration object, on the other hand, means the change in event types being offered will be shared by all the proxy consumer objects that were created by that `SupplierAdmin` object.

The notification channel invokes `offer_change()` on the consumer whenever a supplier changes its offered set of events and this causes the channel's set of offered events to change.

25.3.4.2 Subscription Changes

Similarly, a consumer uses the `subscription_change()` operation to inform the notification channel of the event types it is interested in receiving. The notification channel uses this information to aggregate a list of all the event types its connected consumers require. For example, if three consumers all want event type "A", then the channel only needs to notify the suppliers of a change if all three consumers stop wanting that event type. If the consumer is later required to receive a new event type, or to stop receiving a particular event type, it can use the `subscription_change()` operation again to inform the notification channel of the change in the types of events it requires.

The `subscription_change()` operation can be invoked on the consumer administration object or on the proxy supplier object. Invoking it on the proxy supplier affects only the subscription for that particular consumer, whereas invoking it on the consumer administration object means the change in event types being subscribed to will be shared by all the proxy supplier objects that were created by that `ConsumerAdmin` object.

The notification channel invokes `subscription_change()` on the supplier whenever a consumer changes its subscribed set of events and this causes the channel's set of subscribed events to change.

25.3.4.3 Obtaining Offered and Subscribed Event Types

A supplier can discover the set of event types the consumers of an event channel require by invoking the `obtain_subscription_types()` operation on its proxy consumer. Similarly, a consumer can discover the set of

event types suppliers of the channel offer by invoking the `obtain_offered_types()` operation on its proxy supplier.

Once connected to the notification channel, consumers and suppliers are updated with offer and subscription information via the `offer_change()` and `subscription_change()` operations, respectively.

Note *These updates can be disabled via the* `-NoUpdates` *service configurator option described in 25.7.3.13.*

Using these operations, you can create suppliers that are able to respond to the needs of consumers by producing events the consumers require and to stop producing events that are no longer required by any consumers of the notification channel. Likewise, you can create consumers that are able to dynamically change their subscriptions to start receiving new event types added by suppliers or to discontinue their interest in event types that are no longer produced by the suppliers.

25.3.5 QoS Support

Another major feature of the Notification Service that is not found in the Event Service is the inclusion of interfaces to control QoS characteristics of the event delivery. These interfaces include the ability to get and set QoS properties at the event channel, admin, proxy, and event levels.

Note that the Notification Service style of asynchronous event communication does not provide a one-button QoS setting to solve all QoS-related problems. Since there is no direct communication between suppliers and consumers of events, it is not possible to set QoS at just one place such that it covers the entire event communication pathway. Instead, QoS properties must be set at all three conceptual points through which an event can be transmitted—from the supplier to the event channel, within the event channel itself, and from the event channel to the consumer. These three points must be used together to achieve correct quality of service end-to-end.

In addition, many of the following properties also require application of a proper threading model in order for the QoS properties to have their full effect. See 25.7.1 for a discussion of the threading model and related run-time configuration options.

25.3.5.1 Supported QoS Properties

TAO's implementation of the Notification Service supports a subset of the QoS properties defined by the specification as well as some TAO-specific properties. The following specification-defined properties are *not* supported:

- StartTime
- StopTime
- StartTimeSupported
- StopTimeSupported

Note *TAO does not support use of the* `validate_qos()` *and* `validate_event_qos()` *operations, and it does not always raise the* `UnsupportedQoS` *and* `UnsupportedAdmin` *exceptions where applicable.*

The name of each property defined by the specification is contained in a string constant in `$TAO_ROOT/orbsvcs/orbsvcs/CosNotification.idl`. Each property also has a specific IDL type for its value and it's your responsibility to put the correct type of value in each property's `any`. The above file also contains constant values that predefine values for certain properties. TAO-specific property details are similarly defined in `$TAO_ROOT/orbsvcs/orbsvcs/NotifyExt.idl`. The following sections describe TAO's supported QoS properties along with the associated IDL data type and any predefined values.

25.3.5.2 Notification QoS Properties

The following specification-defined QoS properties are established using the `set_qos()` operation and accessed using the `get_qos()` operation.

Timeout

This property is only supported on a per-event basis. If an event has not been delivered by the specified relative time, then it will be discarded. The data type of this property is `TimeBase::TimeT`, which is a typedef to an unsigned long long with units of 100s of nanoseconds (10E-7 seconds).

Priority

This property is only supported on a per-event basis, and is used to control the order of the events delivered to the consumers. The default value is 0; any integer in the range -32767 to 32767 is valid. The data type is a short. Constants exist for `LowestPriority`, `HighestPriority`, and `DefaultPriority`.

OrderPolicy

This property is used by a proxy to arrange the events in its dispatch queue. When events are delivered as published, and not queued, it has no effect. This property only applies to structured events, either individually or within sequences. The following constants define the only valid values which can be assigned to the `OrderPolicy` property.

- `AnyOrder`: According to the specification, events can be delivered in any implementation-specific order. In TAO's implementation this is equivalent to `FifoOrder`. This is the default value for this policy.
- `FifoOrder`: Events are delivered in the order of their arrival.
- `PriorityOrder`: Events are ordered based on their priority, the highest priority events being delivered first.
- `DeadlineOrder`: Events are ordered based on their expiration timeouts, the events with the shortest timeouts being delivered first.

DiscardPolicy

This property defines the order in which the events are discarded by a proxy or the event channel when their internal buffers overflow. This property applies on a per-channel basis only if it is set on a channel that also has the `RejectNewEvents` admin property set to false. The following constants define the only valid values which can be assigned to the `DiscardPolicy` property.

- `AnyOrder`: According to the specification, events can be discarded in any implementation-specific order. In TAO's implementation this is equivalent to `FifoOrder`. This is the default value for this policy.
- `FifoOrder`: Events are discarded in the order of their arrival.
- `LifoOrder`: Events are discarded in the inverted order of their arrival.

- `PriorityOrder`: Events are discarded based on their priority, the lowest priority events being discarded first.
- `DeadlineOrder`: Events are discarded based on their expiration times, the events with the earliest deadline being discarded first.

MaxEventsPerConsumer

This property defines a bound to the maximum number of events the channel will queue on behalf of a given consumer. This property can only be set on proxy suppliers and has a default value of zero, which means that no maximum will be enforced. The data type for this property is long.

When setting this property to non-zero values, it should always be larger than the MaxQueueLength property value of its channel. Setting it to equal or larger values can result in some consumers being starved.

MaximumBatchSize

This property defines the maximum number of events that will be delivered within each sequence of events. It applies only to sequence proxy suppliers, and defaults to zero, which means that no maximum will be enforced. The data type for this property is long.

PacingInterval

This property defines the maximum period of time the channel will collect individual events into a sequence before delivering the sequence to the consumer. The time starts when a new event arrives for an idle consumer. This property applies only to sequence proxy suppliers, and defaults to zero, which means that no timeout will be enforced. If the number of events received within a given `PacingInterval` equals or exceeds `MaximumBatchSize`, the consumer will receive a sequence of events whose length equals `MaximumBatchSize`. The data type of this property is `TimeBase::TimeT`, which is a typedef to an unsigned long long with units of 100s of nanoseconds (10E-7 seconds).

ConnectionReliability

This property can be set to either `Persistent` or `BestEffort`. If it is set to `Persistent` for an `EventChannel` the Notification Service saves the admins, proxy suppliers, and proxy consumers created in that channel in persistent storage. When the Notification Service starts up, it reloads this

connection information from persistent storage and uses it to reestablish the connections that were active when it last ran.

Setting ConnectionReliabilty to `Persistent` is only valid when Topology Persistence is configured for the Notification Service as described in 25.7.4.

EventReliability

This property can be set for a channel or for individual events. It can be set to either `BestEffort` or `Persistent`. When it is set to `Persistent` for a channel, events delivered through that channel will be delivered reliably, even if the Notification Service or a consumer fails and must be restarted.

Events delivered through a reliable channel are delivered reliably unless the `EventReliability` property for the event is explicitly set to `BestEffort`.

An unreliable channel is one for which the `EventReliability` property is not specified, or explicitly set to `BestEffort`. The `EventReliability` property for an event delivered through an unreliable channel is ignored.

Event reliability is only available when Event Persistence is configured for the Notification Service as described in 25.7.5.

The `EventReliability` property should be set to `Persistent` only when the `ConnectionReliability` property is also set to `Persistent`. This is not checked by the current implementation of the TAO Notification Service, but a check may be added in the future.

25.3.5.3 Notification Administration Properties

The following specification-defined QoS properties apply only to an EventChannel. They are established using the `set_admin()` operation and accessed using the `get_admin()` operation.

Note *The* `set_qos()` *operation is used to set most properties, but these additional QoS properties are supported only by the* `EventChannel` *interface and use the* `set_admin()` *operation, instead. This distinction is important, because setting an admin property using* `set_qos()` *will appear to work, but will have no effect.*

MaxQueueLength

This property specifies the maximum number of events the notification event channel will queue internally before it starts discarding events. The events will be discarded according to the `DiscardPolicy` QoS parameter or `RejectNewEvents` property. The data type for this property is long.

MaxConsumers

This property defines the maximum number of consumers that can be connected to the notification event channel at a time. If this number is exceeded, then an `IMP_LIMIT` exception is raised. You must be careful to correctly disconnect consumers when using this property to avoid reaching the limit due to inactive consumers remaining attached to the channel. The data type for this property is long.

MaxSuppliers

This property defines the maximum number of suppliers that can be connected to the notification event channel at a time. If this number is exceeded, then an `IMP_LIMIT` exception is raised. You must be careful to correctly disconnect suppliers when using this property to avoid reaching the limit due to inactive suppliers remaining attached to the channel. The data type for this property is long.

Note *TAO's Notification Service implementation differs from the specification in that it raises the* `CORBA::IMP_LIMIT` *exception instead of the* `CosNotifyChannelAdmin::AdminLimitExceeded` *exception in the cases described above.*

RejectNewEvents

This property specifies how the event channel should handle events when the number of events exceeds the value associated with the `MaxQueueLength` property. When `RejectNewEvents` is set to true, any attempt to push new events to the channel will result in the `IMP_LIMIT` CORBA system exception being raised. When this property is set to false, any attempt to push new events to the channel will result in a queued event being discarded according to the value of the `DiscardPolicy` property. The data type for this property is boolean.

25.3.5.4 TAO-Specific RT CORBA Properties

The following TAO-specific RT CORBA QoS properties are established using the `set_qos()` operation and accessed using the `get_qos()` operation. Both of the properties defined here define the allocation of threading resources for proxies. Supplier proxies use these threading resources to dispatch events to consumers. Consumer proxies use them to process incoming events. The threading architecture of the notification channel and related configuration options are discussed in more detail in 25.7.1. Please note that the usage and behavior of these options is dependent on the Real-Time Notification features described in 25.3.8 and are affected by whether the associated library is loaded.

ThreadPool

This property defines a thread pool that a proxy allocates and uses for processing events. It can be set on channel or admin objects, but mainly affects the proxies created under those objects. Setting this property for a consumer proxy overrides the default number of threads defined by the `-SourceThreads` option and setting it for a supplier proxy overrides the default number of threads defined by the `-DispatchingThreads` option. The data type of this property is a structure named `ThreadPoolParams` which is defined in `$TAO_ROOT/orbsvcs/orbsvcs/NotifyExt.idl`. When used without the RT Notification library, all of the fields of this structure are ignored except `static_threads` which is an integer value used as the number of threads to allocate for that proxy's thread pool. When used with the RT Notification library, all of the fields are used to initialize an RT CORBA thread pool and other RT CORBA policies for the POA used to activate the proxy object. See 25.3.8 for further details on using this property with the RT CORBA features.

ThreadPoolLanes

This property defines a thread pool with lanes that a proxy allocates and uses for processing events. It can be set on channel or admin objects, but mainly affects the proxies created under those objects. Setting this property for a consumer proxy overrides the default number of threads defined by the `-SourceThreads` option and setting it for a supplier proxy overrides the default number of threads defined by the `-DispatchingThreads` option. The data type of this property is a structure named `ThreadPoolLanesParam` which is defined in `$TAO_ROOT/orbsvcs/orbsvcs/NotifyExt.idl`. This

property requires the loading of the RT Notification library and results in an exception when used without this library. When used with the RT Notification library, the fields are used to initialize an RT CORBA thread pool with lanes and other RT CORBA policies for the POA used to activate the proxy object. See 25.3.8 for further details on using this property with the RT CORBA features.

Note *When the Notification Service creates threads for a thread pool, it specifies a default priority as part of the thread creation parameters. On HP-UX version 10 and later, the process owner must be a member of a group that has the* `RTSCHED` *privilege in order to specify a priority for a new thread. Without this privilege, the thread creation operation will return an error (*`EPERM`*). Other operating systems do not exhibit this behavior.*

On HP-UX, you can add the `RTSCHED` *privilege to all members of a group with the* `setprivgrp(1M)` *command. You must be super-user to execute this command.*

For example, for a group named "`corbausers`*", the command would be entered as follows:*

```
# setprivgrp corbausers RTSCHED
```

The user that starts the Notification Service should be a member of this group.

By default, the effects of the `setprivgrp` *command are lost after a reboot. See* `<http://www.faqs.org/faqs/hp/hpux-faq/>` *for information on how to ensure the privilege group changes become permanent.*

25.3.5.5 Other TAO-Specific Properties

BlockingPolicy

Use this property to set a blocking timeout for use when the notification channel's queue is full. The default behavior, when this property is not specified, is to discard an event when the queue is full (based on the values of `RejectNewEvents` and `DiscardPolicy`). When this property is used, the channel blocks for the specified timeout while waiting for the queue to have space for the event. If the timeout expires an event is discarded as in the default case. The data type of this property is `TimeBase::TimeT`, which is a

typedef to an unsigned long long with units of 100s of nanoseconds (10E-7 seconds).

You must use multithreaded dispatching when you specify this option (specified via -DispatchingThreads). In addition, to avoid deadlocks when using this property, be sure to configure the ORB so that dispatching threads do not process incoming events. Here is a suitable service configurator file for a notification channel process that uses the `BlockingPolicy` property:

```
static Client_Strategy_Factory "-ORBWaitStrategy rw -ORBTransportMuxStrategy
exclusive -ORBConnectStrategy blocked"
static Resource_Factory "-ORBFlushingStrategy blocking"
static Notify_Default_Event_Manager_Objects_Factory "-DispatchingThreads 1"
```

25.3.5.6 Accessing and Modifying QoS Properties

The QoS properties mentioned above can be set at the following levels: event channel, admin objects, proxy objects and structured event. If a property is set at more than one level, the property value set at the lower level applies. It is your responsibility to apply these policies in a meaningful way considering the asynchronous nature of event communication.

Table 25-1 indicates the levels at which each property can be set.

Table 25-1 Levels at Which Setting Each QoS Property is Supported

Property	Event	Proxy	Admin	Channel (QoS)	Channel (Admin)
`BlockingPolicy`		x	x	x	
`ConnectionReliability`		x	x	x	
`DiscardPolicy`		x	x	x	
`EventReliability`	x			x	
`MaxConsumers`					x
`MaxEventsPerConsumer`		x			
`MaximumBatchSize`		x			
`MaxQueueLength`					x
`MaxSuppliers`					x
`OrderPolicy`		x	x	x	
`PacingInterval`		x			
`Priority`	x				

Table 25-1 Levels at Which Setting Each QoS Property is Supported

Property	Event	Proxy	Admin	Channel (QoS)	Channel (Admin)
RejectNewEvents					x
ThreadPool			x	x	
ThreadPoolLanes			x	x	
Timeout	x				

The following code fragment shows how to set the `OrderPolicy` and `DiscardPolicy` properties:

```
CosNotification::QoSProperties qos(2);
qos.length(2);

qos[0].name = CORBA::string_dup (CosNotification::OrderPolicy);
qos[0].value <<= CosNotification::FifoOrder;

qos[1].name = CORBA::string_dup (CosNotification::DiscardPolicy);
qos[1].value <<= CosNotification::LifoOrder;

// qos_admin can be any object whose interface derives from
// CosNotification::QoSAdmin, including EventChannel, ConsumerAdmin,
// SupplierAdmin, ProxyConsumer, ProxySupplier, etc.
qos_admin->set_qos(qos);
```

Setting QoS in a Structured Event Header

As mentioned previously, you can also insert QoS properties into the variable header fields of a `StructuredEvent`, as follows:

```
CosNotification::StructuredEvent event;

// Populate the event's fixed header fields.
CosNotification::FixedHeader& fh = event.header.fixed_header;
fh.event_type.domain_name = CORBA::string_dup("Messenger");
fh.event_type.type_name = CORBA::string_dup("message");
fh.event_name = CORBA::string_dup("a_message");

//Populate the event's variable header fields with the qos properties.
CosNotification::OptionalHeaderFields& vh = event.header.variable_header;
vh.length(1);
vh[0].name = CORBA::string_dup(CosNotification::Timeout);
// TimeT is in 10ths of a microsecond (100 nanoseconds).
const TimeT one_minute = (TimeBase::TimeT)(60 * 1000 * 1000 * 10);
vh[0].value <<= one_minute;
```

25.3.5.7 Negotiating QoS and Conflict Resolution

TAO's implementation of the Notification Service does not support the `validate_qos()` and `validate_event_qos()` operations, and it does not always raise `UnsupportedQoS` exceptions when expected, you are advised to ensure QoS properties are valid before they are set.

get_qos() Operation

This operation returns the current QoS properties that are set on a given object, including properties that were set by default. The returned properties may even include invalid properties that are ignored by the Notification Service. You can invoke the `get_qos()` operation as follows:

```
QoSProperties_var props = qos_admin->get_qos();
```

25.3.5.8 QoS-Related Exceptions

UnsupportedQoS Exception

Some operations described above may raise this exception if parameters passed to it are QoS properties that are not supported. The exception contains a sequence of erroneous QoS properties and their value ranges. Each included property has an associated error. Table 25-2 describes possible values for the `UnsupportedQoS` error codes. It is taken from the Notification Service specification.

Table 25-2 UnsupportedQoS Error Codes

Error Code	Meaning
`UNSUPPORTED_PROPERTY`	This property is not supported by this implementation for this type of target object.
`UNAVAILABLE_PROPERTY`	This property cannot be set at (to any value) in the current context (i.e., in the context of other QoS properties).
`UNSUPPORTED_VALUE`	The value requested for this property is not supported by this implementation for this type of target object. A range of values which would be supported is returned.
`UNAVAILABLE_VALUE`	The value requested for this property is not supported in the current context. A range of values which would be supported is returned.

Table 25-2 UnsupportedQoS Error Codes

Error Code	Meaning
BAD_PROPERTY	This property name is unrecognized. The implementation knows nothing about it.
BAD_TYPE	The type supplied for the value of this property is incorrect.
BAD_VALUE	An illegal value is supplied for this property. A range of values which would be supported is returned.

BAD_QOS System Exception

This exception is raised during transmission of a structured event from a supplier to the notification event channel when the QoS properties indicated in the header of the event cannot be satisfied by the channel.

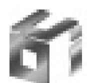
25.3.6 Connection Reliability

Normally when the Notification Service starts executing, it creates an Event Channel Factory. Based on command line options it may also create an Event Channel. After that it waits for clients (consumers and suppliers) to create and initialize event channels, admins, proxies, filters, etc. required to deliver events. The collection of all of these objects created inside the notification service is called the notification service *topology*.

If connection reliability is configured for the Notification Service it will save this topology information, including information about the connections to clients, in persistent storage. When it restarts, it can reload this information, reestablish the connections to the clients (if they are still available), and be ready to deliver events without the need for the clients to reconfigure the topology.

To enable connection reliability, several things must happen:

- Topology persistence must be configured via the Service Configurator. For more details on configuring topology persistence, see 25.7.4.
- Client reconnection to the Notification Service must be enabled. For details see 25.7.3.2.
- The `ConnectionReliability` QoS property must be set to `Persistent` for those portions of the topology that should be saved.



- The clients must be written to take advantage of the ability to reconnect to existing objects in the Notification Service. Among other things this may involve registering with the Notification Service's reconnection registry. See 25.3.6.1 for details.

Each portion of the topology inherits the `ConnectionReliability` property from its parent so setting `ConnectionReliability` to `Persistent` for an event channel automatically makes all admins and proxies contained in that channel reliable.

`ConnectionReliability` may be disabled at the Admin or Proxy level by specifying the `ConnectionReliability` property as `BestEffort` for the object that should not be saved. The converse, enabling `ConnectionReliability` for an `Admin` in a non-reliable `EventChannel` or a `Proxy` in a non-reliable `Admin`, is not possible. Unless the parent is saved, the child object cannot be saved.

TAO's implementation of the Notification Service is designed to allow the persistent topology to be reloaded on another computer, even if it has a different architecture or operating system.

Here is a sample `svc.conf` file from a TAO connection reliability tests (`$TAO_ROOT/orbsvcs/tests/Notify/Reconnecting/ns_mt_topo.conf`):

```
static TAO_CosNotify_Service "-DispatchingThreads 2 -SourceThreads 2
-AllowReconnect"
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist:_make_TAO_Notify_XML_Topology_Factory() "-base_path
./reconnect_test"
```

The `-AllowReconnect` option on the `TAO_CosNotify_Service` is required for connection reliability. The `-base_path` option on the Topology Factory tells the topology persistence code where to load the topology from when starting and where to save the topology as it changes.

25.3.6.1 The Reconnection Registry

Specifying Connection Reliability and configuring persistent topology support provides a basic level of connection reliability. Clients that desire a greater ability to recover their connection after a Notification Service restart can register to receive notification of the restart using a new, TAO-specific feature called the Reconnection Registry.

The `EventChannelFactory` in TAO implements the following IDL interface:

```
module NotifyExt
{
  /**
   * \brief An interface that handles registration of suppliers and consumers.
   *
   * This registry should be implemented by an EventChannelFactory and
   * will call the appropriate reconnect methods for all ReconnectionCallback
   * objects registered with it.
   */
  interface ReconnectionRegistry
  {
    typedef long ReconnectionID;
    ReconnectionID register_callback(in ReconnectionCallback reconection);

    void unregister_callback (in ReconnectionID id);

    /// Check to see if the ReconnectionRegistry is alive
    boolean is_alive ();
  };
}
```

Clients that wish to use the Reconnection Registry must implement the following interface:

```
module NotifyExt
{
  /**
   * \brief An interface which gets registered with a ReconnectionRegistry.
   *
   * A supplier or consumer must implement this interface in order to
   * allow the Notification Service to attempt to reconnect to it after
   * a failure.  The supplier or consumer must register its instance of
   * this interface with the ReconnectionRegistry.
   */
  interface ReconnectionCallback
  {
    /// Perform operations to reconnect to the Notification Service
    /// after a failure.
    void reconnect (in Object new_connection);

    /// Check to see if the ReconnectionCallback is alive
    boolean is_alive ();
  };
}
```

Clients should narrow the reference the `EventChannelFactory` to a `NotifyExt::ReconnectionRegistry` and then call the `register_callback()` operation. After doing so, they will receive a call to their `reconnect()` operation when the Notification Service has restarted and is ready for use.

25.3.7 Event Reliability

The default behavior of the notification service is to provide *best-effort* delivery of events. If anything goes wrong delivering an event to a particular consumer, events for that consumer are silently discarded to keep the overall system operating. This behavior can be changed to provide reliable event delivery by configuring event persistence in the Service Configurator (see 25.7.5) and specifying the `Persistent` setting for the `EventReliability` QoS property as necessary.

For event reliability to work, persistent `EventReliability` must be set at the `EventChannel` level. It may be disabled on a per-event basis by specifying `BestEffort` for the `EventReliability` property of a given event. This works for Structured or Sequence events, but not for Any events which have no property settings.

Notice that it is not possible to enable `EventReliability` for an event unless it is being sent through a reliable `EventChannel`. Any attempt to do so is silently ignored.

Event reliability also depends on connection reliability (see 25.3.6). If event persistence is configured, but topology persistence is not, the Notification Service will not start.

TAO's implementation of the Notification Service is designed to allow the persistent event information to be reloaded on another computer, even if it has a different architecture or operating system.

Here is a sample `svc.conf` file from a TAO event reliability test (`$TAO_ROOT/orbsvcs/tests/Notify/Reconnecting/ns_mt_both.conf`):

```
static TAO_CosNotify_Service "-DispatchingThreads 2 -SourceThreads 2
-AllowReconnect"
#
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist:_make_TAO_Notify_XML_Topology_Factory() "-base_path
./reconnect_test"
```

```
dynamic Event_Persistence Service_Object*
TAO_CosNotification_Serv:_make_TAO_Notify_Standard_Event_Persistence()
"-file_path ./event_persist.db"
```

The TAO_CosNotify_Service and Topology Factory configurations are identical to the example show in the Connection Reliability section. The Event Persistence service object's `-file_path` option is used to specify the directory where events will be persisted.

25.3.7.1 The Impact of Using Event Reliability

Setting event reliability for an event causes the event and information about its delivery to be written to an event persistence file. When the notification service is restarted, it first reloads topology information from the topology persistence file and uses this to recreate the internal structures that support event delivery (Channels, Admins, Proxies, and other objects). It then reads any undelivered events from the event persistence file and reactivates them for delivery to the appropriate consumers. As soon as the consumers reconnect to the notification service, they begin to receive the events that were in route at the time the notification service stopped running.

A supplier that pushes a reliable event to the notification service will not receive a response to the `push()` invocation until the event has been safely stored in the event persistence file.

Reliable events that are being delivered to a consumer that becomes unavailable are held until the connection is reestablished to the consumer or the consumer is restarted (on the same, or a different computer).

25.3.8 Real-Time CORBA Support

Applications utilizing the Real-Time CORBA features described in Chapter 8 encounter several challenges when they wish to use the Notification Service. In general, the service needs to honor their quality of service requirements as events pass through the notification channel. TAO supports these applications by bundling a set of related features in the `TAO_RT_Notification` library. These features include:

- New QoS properties to support
  - RT CORBA thread pool usage in the notification channel
  - Specifying the RT CORBA priority model of the notification channel

- Optimized event processing
- Optimized collocation between proxies

25.3.8.1 Enabling RT CORBA Support

In order to use the RT Notification features you need to load the `TAO_RT_Notification` library and then specify your configuration via the `ThreadPool` and/or `ThreadPoolLanes` properties. Typically, the library is loaded via a dynamic service configurator directive such as this:

```
dynamic TAO_Notify_Service Service_Object *
TAO_RT_Notification:_make_TAO_RT_Notify_Service () ""
```

This factory is derived from the normal `CosNotify_Service` factory and takes all the same options as described in 25.7.1 (although not all are necessary or applicable when using the `RT_Notify_Service` factory).

25.3.8.2 Thread Pool Property

The `ThreadPool` property contains a `ThreadPoolParams` structure as its data element. Here is the relevant IDL:

```
module NotifyExt
{
  typedef short Priority;
  const Priority minPriority = 0;
  const Priority maxPriority = 32767;

  enum PriorityModel
  {
    CLIENT_PROPAGATED,
    SERVER_DECLARED
  };
  struct ThreadPoolParams
  {
    PriorityModel priority_model;
    Priority server_priority;

    unsigned long stacksize;
    unsigned long static_threads;
    unsigned long dynamic_threads;
    Priority default_priority;
    boolean allow_request_buffering;
    unsigned long max_buffered_requests;
    unsigned long max_request_buffer_size;
```

```
  };
};
```

The members of this structure are used to create two POA policies for a new POA that is used to activate enclosed objects. The `PriorityModelPolicy` uses the `priority_model` and `server_priority` members to define the RT CORBA priority model that the POA uses (server-declared or client-propagated). The remaining members are used to construct a thread pool and `ThreadpoolPolicy` for that POA. Additional details on these POA policies and their usage is available in 8.3.7 and 8.3.8.

When this property is applied on an event channel, the event channel uses this POA (and its thread pool) for all enclosed admin and proxy objects. When applied to an admin object, the admin object uses the POA (and its thread pool) for all enclosed proxy objects.

25.3.8.3 Thread Pool Lanes Property

The `ThreadPoolLanes` property contains a `ThreadPoolLanesParams` structure as its data element. Here is the relevant IDL:

```
module NotifyExt
{
  struct ThreadPoolLane
  {
    PriorityModel priority_model;
    Priority server_priority;

    Priority lane_priority;
    unsigned long static_threads;
    unsigned long dynamic_threads;
  };

  typedef sequence <ThreadPoolLane> ThreadPoolLanes_List;

  struct ThreadPoolLanesParams
    {
      PriorityModel priority_model;
      Priority server_priority;

      unsigned long stacksize;
      ThreadPoolLanes_List lanes;
      boolean allow_borrowing;
      boolean allow_request_buffering;
      unsigned long max_buffered_requests;
      unsigned long max_request_buffer_size;
```

```
  };
};
```

Most of this is identical to the ThreadPool property with the exception that we are now specifying a thread pool with multiple lanes in place of a monolithic thread pool. Each lane runs at a specific priority and processes events of that priority. Refer to 8.3.7 for details about thread pools with lanes.

25.3.9 Monitoring and Control of the Notification Service

TAO's Notification Service supports a separate *Monitoring and Control Interface* that goes beyond the standard OMG-defined interfaces to give remote applications the ability to query and track the state and progress of existing notification channels. In addition to monitoring, the interface also allows applications to control certain aspects of the notification channel. This includes the ability to remove and destroy event channels, consumer admin objects, supplier admin objects, consumer proxies, and supplier proxies. Applications can also remotely shut down an entire notification channel factory (and all of it notification channels).

25.3.9.1 TAO Monitoring Types

Monitoring and Control support in the TAO Notification Service is based upon ORB-level monitoring and control capabilities defined at the TAO level. Applications using these capabilities with the Notification Service also need to link with the TAO_Monitor library.
This library defines a statistic as a particular data element that is being monitored at run-time. Each statistics is identified by a unique name within its monitoring interface. The basic monitoring data types are defined in $TAO_ROOT/tao/Monitor/Monitor_Types.pidl. When retrieving a statistic by name, the basic data type returned is a structure named Monitor::Data:

```
module Monitor
{
  enum DataType { DATA_NUMERIC, DATA_TEXT };
  union UData switch (DataType) {
    case DATA_NUMERIC: Numeric num;
    case DATA_TEXT: NameList list;
  };
  struct Data
  {
```

```
    Name itemname;
    UData data_union;
  };
  typedef sequence<Data> DataList;
};
```

This data type contains the name of the statistic and information about either a numeric or text statistic. The text statistic returns a sequence of string values for that statistic. The numeric statistic returns a history of values as well as some cumulative statistical measures since it was last cleared. The `DataList` type is used when multiple statistics are retrieved at once. Here are the relevant IDL definitions for the text-based statistics:

```
module Monitor
{
  typedef string Name;
  typedef CORBA::StringSeq NameList;
};
```

Here are the relevant IDL definitions for the numeric-based statistics:

```
module Monitor
{
  struct DataValue
  {
    /// The number of samples currently taken into account
    TimeBase::TimeT timestamp;
    double value;
  };
  typedef sequence<DataValue> DataValueList;
  struct Numeric
  {
    DataValueList dlist;
    unsigned long count;
    double average;
    double sum_of_squares;
    double minimum;
    double maximum;
  };
};
```

The TAO Monitor library also defines some filtering and constraint types related to statistics, but the TAO Notification Service interfaces do not use these.

25.3.9.2 TAO Notification Monitoring and Control Interface

The Notification Service's Monitoring and Control interface is named `CosNotification::NotificationServiceMonitorControl` and is defined in `NotificationServiceMC.idl` in the directory `$TAO_ROOT/orbsvcs/orbsvcs/Notify/MonitorControl/`.
Applications using this interface should link with the `TAO_CosNotification_MC` library or inherit from the `notification_mc` MPC base project. The MPC base project also has the advantage of automatically linking in the TAO Monitoring library.
Here are the statistics-related operations from the `NotificationServiceMonitorControl` interface:

```
module CosNotification
{
  interface NotificationServiceMonitorControl
  {
    exception InvalidName { Monitor::NameList names; };

    Monitor::NameList get_statistic_names ();

    Monitor::Data get_statistic (in string name)
      raises (InvalidName);

    Monitor::DataList get_statistics (in Monitor::NameList names)
      raises (InvalidName);

    Monitor::DataList get_and_clear_statistics (in Monitor::NameList names)
      raises (InvalidName);

    void clear_statistics (in Monitor::NameList names)
      raises (InvalidName);
```

The `get_statistics_names()` operation retrieves the list of statistic names currently supported by this interface. This list grows and shrinks as entities (such as event channels) are created and destroyed. 25.3.9.3 discusses the set of statistics currently supported by this interface. The remaining operations above take either a single statistic name or a sequence of statistic names as a parameter. Passing a statistic name that is not currently supported results in the `CosNotification::InvalidName` exception. The `clear_statistics()` and `get_and_clear_statistics()` operations remove any statistics history for the specified statistics.

The remaining operations in this interface allow applications to shutdown event channels, remove consumer and supplier admin objects, remove consumers and suppliers, and shutdown the entire notification channel factory.

```
    // Control commands begin here
    void shutdown_event_channel (in string name)
      raises (InvalidName);

    void remove_consumer (in string name)
      raises (InvalidName);

    void remove_supplier (in string name)
      raises (InvalidName);

    void remove_consumeradmin (in string name)
      raises (InvalidName);

    void remove_supplieradmin (in string name)
      raises (InvalidName);

    oneway void shutdown ();
  };
};
```

The operations above are passes names as parameter to identify the particular entity to operate upon. By default, these names are stringified versions of the normal Notification Service identifiers (`ChannelID`, `ProxyID`, `AdminID`). Users can also explicitly name these entities using the TAO-specific mechanism as described in 25.3.9.4.

25.3.9.3 Supported Statistics

The statistics names supported by the Notification Service are organized hierarchically, like Unix directories. At the top of this hierarchy is the notification channel factory. You can retrieve the full list of notification channel factories supported by the monitoring interface by using the `FactoryNames` statistic. This is a text statistic and returns a set of factory names. These names each serve as the prefix for statistic names related to that factory. For example, assuming a factory name of `NotifyEventChannelFactory`, the creation time of that factory can be retrieved with the `NotifyEventChannelFactory/CreationTime` statistic name. Assuming an event channel named `EC1` is in this factory, the statistic name of `NotifyEventChannelFactory/EC1/SupplierCount` can be

used to get the count of suppliers in that event channel. When you call `get_statistics_names()` the full set of statistics supported are returned, without regard to this hierarchy.

The following tables define the available statistics for the different entities within the Notification Service. Table 25-3 describes the statistics associated with the event channel factory. For the purposes of these statistics, activity is determined by the number of consumers and suppliers in a notification channel. A channel with no consumers and no suppliers is considered inactive. A channel with at lease one consumer or supplier is considered active.

Table 25-3 Event Channel Factory Statistics

Statistic	Type	Description
`ActiveEventChannelCount`	Numeric	The number of *active* event channels currently in the event channel factory.
`InactiveEventChannelCount`	Numeric	The number of *inactive* event channels currently in the event channel factory.
`ActiveEventChannelNames`	Text	The names of all *active* event channels currently in the event channel factory.
`InactiveEventChannelNames`	Text	The names of all *inactive* event channels currently in the event channel factory.
`CreationTime`	Numeric	Creation time (in POSIX time) of the event channel factory.

Table 25-4 describes the statistics associated with each event channel.

Table 25-4 Event Channel Statistics

Statistic	Type	Description
`CreationTime`	Numeric	Creation time (in POSIX time) of the event channel.
`ConsumerCount`	Numeric	The number of consumers connected to this event channel.
`SupplierCount`	Numeric	The number of suppliers connected to this event channel.
`ConsumerNames`	Text	The names of consumers connected to this event channel.
`SupplierNames`	Text	The names of suppliers connected to this event channel.
`TimedoutConsumerNames`	Text	The names of any consumers timed out of this event channel.

Table 25-4 Event Channel Statistics

Statistic	Type	Description
`ConsumerAdminCount`	Numeric	The number of consumer admin objects connected to this event channel.
`SupplierAdminCount`	Numeric	The number of supplier admin objects connected to this event channel.
`ConsumerAdminNames`	Text	The names of consumer admin objects connected to this event channel.
`SupplierAdminNames`	Text	The names of supplier admin objects connected to this event channel.
`QueueSize`	Numeric	The cummulative size of the event channel's queues in bytes.
`QueueElementCount`	Numeric	The total number of events currently in the event channel's queues.
`OldestEvent`	Numeric	The time stamp (in POSIX time) of the oldest event in the event channel's queues. Reports a value of zero if the queues are empty.
`SlowestConsumers`	Text	Returns a list of consumer names that are connected to the consumer admin with the largest queue size.

25.3.9.4 Using the Extended Interfaces to Name Entities

When the monitor and control interface is used with the existing Notification Service code, all reporting is done using the normal Notification Service identifiers (`ChannelID`, `ProxyID`, `AdminID`). This functionality works to distinguish particular entities, like one event channel, from another. It does not help an application reliably relate specific reported statistics between execution runs or to application-level entities. In order to do this, we need to assign names to the different entities and not depend on the arbitrary system assigned identifiers.

Because the standard OMG-defined interfaces do not support the naming of entities, TAO's Notification Service offers extended interfaces that allow this capability. These interfaces are defined in `$TAO_ROOT/orbsvcs/orbsvcs/Notify/MonitorControlExt/NotifyMonitoringExt.idl`:

```
module NotifyMonitoringExt
{
```

```
exception NameAlreadyUsed {};
exception NameMapError {};

interface SupplierAdmin: NotifyExt::SupplierAdmin
{
  CosNotifyChannelAdmin::ProxyConsumer
    obtain_named_notification_push_consumer (
                         in CosNotifyChannelAdmin::ClientType ctype,
                         out CosNotifyChannelAdmin::ProxyID proxy_id,
                         in string name)
    raises (CosNotifyChannelAdmin::AdminLimitExceeded,
            NameAlreadyUsed,
            NameMapError);
};

interface ConsumerAdmin: NotifyExt::ConsumerAdmin
{
  CosNotifyChannelAdmin::ProxySupplier
    obtain_named_notification_push_supplier (
                         in CosNotifyChannelAdmin::ClientType ctype,
                         out CosNotifyChannelAdmin::ProxyID proxy_id,
                         in string name)
    raises (CosNotifyChannelAdmin::AdminLimitExceeded,
            NameAlreadyUsed,
            NameMapError);
};

interface EventChannel: CosNotifyChannelAdmin::EventChannel
{
  CosNotifyChannelAdmin::ConsumerAdmin
    named_new_for_consumers(
                  in CosNotifyChannelAdmin::InterFilterGroupOperator op,
                  out CosNotifyChannelAdmin::AdminID id,
                  in string name)
    raises (NameAlreadyUsed,
            NameMapError);

  CosNotifyChannelAdmin::SupplierAdmin
    named_new_for_suppliers(
                  in CosNotifyChannelAdmin::InterFilterGroupOperator op,
                  out CosNotifyChannelAdmin::AdminID id,
                  in string name)
    raises (NameAlreadyUsed,
            NameMapError);
};

interface EventChannelFactory: NotifyExt::EventChannelFactory
{
  CosNotifyChannelAdmin::EventChannel
```

```
      create_named_channel (
                  in CosNotification::QoSProperties initial_qos,
                  in CosNotification::AdminProperties initial_admin,
                  out CosNotifyChannelAdmin::ChannelID id,
                  in string name)
      raises(CosNotification::UnsupportedQoS,
             CosNotification::UnsupportedAdmin,
             NameAlreadyUsed,
             NameMapError);
  };
};
```

The `NotifyMonitoringExt` module defines extended versions of the event channel factory, event channel, consumer admin, and supplier admin interfaces that allow the user to name entities as they are created. Each is directly derived from the existing Notification interfaces defined in the `NotifyExt` module and indirectly derived from the standard Notification Service interfaces. They add new operations for creating entities that differ from the existing operations only by the addition of a name to the parameter list. For example, instead of creating event channels with the `create_channel()` operation in the `CosNotifyChannelAdmin::EventChannelFactory` interface, you can use the `create_named_channel()` operation in the `NotifyMonitoringExt::EventChannelFactory`. Any channels subsequently created always have their statistics reported with the assigned name instead of the `system-generated ChannelID` value.

To use the interface extensions, first, narrow the entity objects you are using to the extended interface types (in place of the Notification Service standard interfaces). Then, you are ready to use the “named” interfaces. For example, to create a named event channel:

```
#include "orbsvcs/Notify/MonitorControlExt/NotifyMonitoringExtC.h"

  // Get the event channel factory object from somewhere and then
  // narrow to the extended interface.
  CORBA::Object_var obj = ...;
  NotifyMonitoringExt::EventChannelFactory_var notify_ext_factory =
    NotifyMonitoringExt::EventChannelFactory::_narrow (obj.in());

  CosNotifyChannelAdmin::ChannelID id;
  CosNotification::QoSProperties qos;
  CosNotification::AdminProperties admin_prop;
  CosNotifyChannelAdmin::EventChannel_var ec =
      notify_ext_factory->create_named_channel(qos, admin_prop, id, "MyEC";
```

25.3.9.5 Initializing and Connecting to the Monitoring Interface

Because the Monitoring and Control capabilities of the event channel are not enabled in the basic Notification Service libraries, this functionality needs to be enabled through use of dynamically loaded factories in a service configurator file. Here is an example service configurator file:

```
dynamic TAO_MonitorAndControl Service_Object *
TAO_CosNotification_MC:_make_TAO_MonitorAndControl () "-o mc.ior -ORBArg
-ORBInitRefNameService=file://ns.ior"

dynamic TAO_MC_Notify_Service Service_Object *
TAO_CosNotification_MC_Ext:_make_TAO_MC_Notify_Service () ""
```

The `TAO_MonitorAndControl` factory enables the monitoring and control functionality in the notification server. This service creates its own ORB and uses it for the monitoring and control CORBA objects. Table 25-5 shows the configuration options that this factory takes.

Table 25-5 TAO_MonitorAndControl Service Options

Option	Description	Default
`-o` *`file`*	Specifies a file to write the Monitor and Control object's IOR to.	Do not write the IOR to a file.
`-ORBArg` *`opts`*	Specifies ORB-initialization options to pass to the monitor and control ORB when it is created.	ORB is created with the default configuration.
`-NoNameSvc`	Do not register the Monitor and Control object with the Naming Service.	Monitor and Control object is registered under the name "TAO_MonitorAndControl"

The TAO_MC_Notify_Service factory enables the extended notification interfaces discussed in 25.3.9.4. It does not take any configuration options.

Listed below is an example that starts the `tao_cosnotification` server with this configuration. Note that we have configured both the event channel factory and monitor and control objects to use the same Naming Service.

```
$TAO_ROOT/orbsvcs/Notify_Service/tao_cosnotification -ORBSvcConf mc.conf \
                 -ORBInitRefNameService=file://ns.ior
```

25.4 Using the Notification Service

We now look at an example that illustrates the use of Notification Service features. A basic example using structured events is introduced in 25.4.2. In 25.4.3, we discuss how consumer and supplier connections can be managed. We extend the basic example in 25.4.4 by adding offer publication to the supplier and event type subscription to the consumer. Filtering is added to the supplier and consumer in 25.4.5. In 25.4.6, we add QoS properties to the example. In 25.4.7, we show how to transmit batched events. Finally, in 25.4.8, we show how you can collocate a notification event channel in the same address space as a supplier. For the sake of clarity, error checking in these examples has been kept to a minimum. Full source code for these examples is in the TAO source code distribution in subdirectories under `$TAO_ROOT/orbsvcs/DevGuideExamples/NotifyService/`.

25.4.1 Building Notification Service Applications

Table 25-6 lists the set of libraries containing TAO's Notification Service implementation.

Table 25-6 TAO Notification Service Libraries

Library	Description
`TAO_CosNotification`	Client-side IDL-generated C++ code (stubs)
`TAO_CosNotification_Skel`	Server-side IDL-generated C++ code (skeletons)
`TAO_CosNotification_Serv`	Notification Service server implementation (required for processes that contain notification channels)
`TAO_CosNotification_Persist`	Persistence-related features (required for notification channel server processes that set `ConnectionReliability` and/or `EventReliability` to `Persist`)
`TAO_RT_Notification`	Features related to Real-Time CORBA (required for notification channel server processes that use RT CORBA with the Notification Service)

Most processes that use TAO's Notification Service must link with the `TAO_CosNotification` and `TAO_CosNotification_Skel` libraries. Processes that do not implement any notification-related servants (such as some push suppliers) may be able to only link the `TAO_CosNotification`

library. Processes that contain event channel factories and event channels must also link `TAO_CosNotification_Serv`. These processes must also link either `TAO_CosNotification_Persist` or `TAO_RT_Notification` if they use the persistence or real-time features, respectively.

MPC projects for processes that use the Notification Service can simply inherit from the `notify_skel` base project. MPC projects for notification channel servers can simply inherit from the `notify_serv` base project. For example, here is the mpc file for the Notification Service example in `$TAO_ROOT/orbsvcs/DevGuideExamples/Notification/Messenger` (discussed in the next section):

```
project(*Server): namingexe, portableserver, notification_skel {
  requires += exceptions
  Source_Files {
    StructuredEventSupplier_i.cpp
    MessengerServer.cpp
    Messenger_i.cpp
  }
}

project(*Client): namingexe, notification {
  requires += exceptions
  Source_Files {
    MessengerC.cpp
    MessengerClient.cpp
  }
}

project(*Consumer): namingexe, portableserver, notification_skel {
  requires += exceptions
  IDL_Files {
  }
  Source_Files {
    MessengerConsumer.cpp
    StructuredEventConsumer_i.cpp
  }
}
```

For more information on MPC, see Chapter 4.

25.4.2 A Basic Example

Our example extends the Messenger example discussed in Chapter 3 to use the Notification Service. Suppose, for example, that the `MessengerServer` is to publish the messages it receives so that other interested consumers can receive

them. One way to do this is for the `MessengerServer` to become a client of another set of CORBA objects that are interested in receiving the messages sent to it. In this scenario, the `MessengerServer` obtains an object reference for each interested object, then invokes an operation on each object to forward each message it receives. However, this approach is inflexible and inefficient because:

- The `MessengerServer` must know about all objects that are interested in receiving the messages *a priori*, thereby making it difficult to dynamically add or remove them at run time.
- The `MessengerServer` must spend processing time forwarding the messages to these interested objects.

The asynchronous style of event communication used by the Notification Service addresses these issues. It allows suppliers to send messages (events) to consumers that are interested in receiving them, yet neither the consumers nor the suppliers need to know about one another. Moreover, it allows applications to dynamically add or remove suppliers and consumers without impacting other objects in the system. Finally, it transfers responsibility for dispatching events to consumers from suppliers to the notification channel rather than imposing this processing overhead on the suppliers themselves.

We now modify our `MessengerServer` example to send an event to consumers via a notification channel each time it receives a message from a client so that other objects interested in receiving this event can subscribe to the notification channel and receive it. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NotifyService/Messenger/`.

25.4.2.1 Starting the tao_cosnotification Server

This example uses the `tao_cosnotification` and `tao_cosnaming` servers in TAO. By default, the `tao_cosnotification` server creates a single notification channel factory object and binds it in the root naming context of the Naming Service to the name `NotifyEventChannelFactory`. The notification channel factory is used to create notification channels. The Naming Service and Notification Service servers must be started in the following order before running this example:

```
$TAO_ROOT/orbsvcs/Naming_Service/tao_cosnaming
$TAO_ROOT/orbsvcs/Notify_Service/tao_cosnotification
```

Command line options for the `tao_cosnotification` server are covered in 25.6.

25.4.2.2 Implementing the Structured Push Supplier Interface

The `CosNotifyComm::StructuredPushSupplier` IDL interface is implemented by our supplier class `StructuredEventSupplier_i`. The `StructuredPushSupplier` interface contains the following two operations:

- `disconnect_structured_push_supplier()` allows the notification channel to inform the supplier that it has been disconnected from the channel. Typically, this is called when the channel is destroyed.
- `subscription_change()` allows the notification channel to inform the supplier of changes to the event types its connected consumers are interested in receiving.

```
#include <orbsvcs/CosNotifyCommS.h>

class StructuredEventSupplier_i :
    public virtual POA_CosNotifyComm::StructuredPushSupplier
{
  public:
    // Constructor
    StructuredEventSupplier_i(CORBA::ORB_ptr orb);

    // Override operations from StructuredPushSupplier interface.
    virtual void disconnect_structured_push_supplier () ;
    virtual void subscription_change (
         const CosNotification::EventTypeSeq& events_added,
         const CosNotification::EventTypeSeq& events_removed);
  private:
    CORBA::ORB_var orb_;
};
```

The `StructuredEventSupplier_i` constructor simply duplicates the ORB reference passed to it and stores it in its data member.

```
StructuredEventSupplier_i::StructuredEventSupplier_i(CORBA::ORB_ptr orb)
  : orb_(CORBA::ORB::_duplicate(orb)) { }
```

The `disconnect_structured_push_supplier()` operation is called when the supplier is being disconnected from the notification channel. Our implementation deactivates the supplier object.

```
void StructuredEventSupplier_i::disconnect_structured_push_supplier ()
 {
  CORBA::Object_var obj = orb_->resolve_initial_references ("POACurrent");
  PortableServer::Current_var current =
    PortableServer::Current::_narrow (obj.in());
  PortableServer::POA_var poa = current->get_POA ();
  PortableServer::ObjectId_var objectId = current->get_object_id ();
  poa->deactivate_object (objectId.in());
}
```

The `subscription_change()` operation is called by the supplier's consumer proxy object to inform the supplier of changes to the subscription information by the notification channel's consumers. Its implementation does nothing for now, but will be expanded later when we address offers and subscriptions.

```
void StructuredEventSupplier_i::subscription_change (
  const CosNotification::EventTypeSeq&,
  const CosNotification::EventTypeSeq&)
{
  // More to come...
}
```

25.4.2.3 Developing the Structured Event Supplier

Next, we modify the implementation of our `MessengerServer` so that it can behave as a supplier of structured events. Each time it receives a message, it will create a new structured event and populate it with information from the message, then push this event to the notification channel via its consumer proxy.

First, we initialize the ORB and obtain and activate the `RootPOA` as usual:.

```
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main(int argc, char* argv[])
{
  try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();
```

Next, we create a `Messenger_i` servant and activate it in the `RootPOA`. We then export its object reference as a string and wait for client requests:

```
    PortableServer::Servant_var<Messenger_i> messenger_servant =
      new Messenger_i(orb.in());
    PortableServer::ObjectId_var oid =
      poa->activate_object(messenger_servant.in());
    CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
    CORBA::String_var str = orb->object_to_string(messenger_obj.in());
    std::ofstream iorFile("Messenger.ior");
    iorFile << str.in() << std::endl;
    iorFile.close();
    std::cout << "IOR written to file Messenger.ior" << std::endl;
    orb->run();
    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
  }
  return 0;
}
```

We now turn our attention to the `Messenger_i` class that implements the Messenger interface. Our class now acts as an event supplier. We have added an additional `#include` directive to access the Notification Service channel administration definitions. We have also added new fields, one to hold the object reference of our structured proxy push consumer, and another to hold an ORB reference.

```
#include "MessengerS.h"
#include <orbsvcs/CosNotifyChannelAdminC.h>

class Messenger_i : public virtual POA_Messenger
{
  public:
    // Constructor and destructor.
    Messenger_i (CORBA::ORB_ptr orb);
    virtual ~Messenger_i ();

    // Override operations from the Messenger interface.
    CORBA::Boolean send_message (
      const char* user_name,
      const char* subject,
      char*& message);
```

```
  private:
    CosNotifyChannelAdmin::StructuredProxyPushConsumer_var
      structured_proxy_consumer_;
    CORBA::ORB_var orb_;
};
```

Our modified implementation of the `Messenger_i` class follows:

```
#include "StructuredEventSupplier_i.h"
#include "Messenger_i.h"
#include <orbsvcs/CosNotifyChannelAdminC.h>
#include <orbsvcs/CosNotifyCommC.h>
#include <orbsvcs/CosNamingC.h>
```

In the constructor, we now initialize the ORB. We then use the Naming Service to access the Notification Service's notification channel factory:

```
Messenger_i::Messenger_i (CORBA::ORB_ptr orb)
  : orb_(CORBA::ORB::_duplicate(orb))
{
  try {
    CORBA::Object_var naming_obj =
      orb_->resolve_initial_references ("NameService");
    CosNaming::NamingContext_var naming_context =
      CosNaming::NamingContext::_narrow(naming_obj.in());
```

When we started the `tao_cosnotification` server, it bound the notify channel factory in the root naming context of the Naming Service to the name "`NotifyEventChannelFactory`." We resolve the factory using this name:

```
    CosNaming::Name name;
    name.length(1);
    name[0].id = CORBA::string_dup("NotifyEventChannelFactory");
    CORBA::Object_var obj = naming_context->resolve(name);

    CosNotifyChannelAdmin::EventChannelFactory_var notify_factory =
      CosNotifyChannelAdmin::EventChannelFactory::_narrow(obj.in());
    if (CORBA::is_nil(notify_factory.in())) {
      std::cerr << "Unable to find notify factory" << std::endl;
    }
```

We use the notification channel factory to create a new notification channel. The `create_channel()` operation takes the following parameters as input:

- The initial QoS values for the channel.

- The initial administrative property values. This is a sequence of name/value pairs, where each name is a `string` and each value is a `CORBA::Any`.
- The `ChannelID` as an `out` parameter. The `ChannelID` identifies the notification channel within the context of the notification event channel factory.

```
CosNotifyChannelAdmin::ChannelID id;
CosNotification::QoSProperties initial_qos;
CosNotification::AdminProperties initial_admin;

CosNotifyChannelAdmin::EventChannel_var notify_channel =
  notify_factory->create_channel (initial_qos,
                                  initial_admin,
                                  id);
if (CORBA::is_nil(notify_channel.in())) {
  std::cerr << "Unable to create notification channel" << std::endl;
}
```

There are several ways to provide access to this notification channel. For this example, we bind it in the root naming context of the Naming Service to the name “`MessengerChannel`”:

```
name[0].id = CORBA::string_dup("MessengerChannel");
naming_context->rebind(name, notify_channel.in());
```

Alternatively, the supplier can export `ChannelID`, which could then be used by consumers to get a reference to the notification channel using the `get_event_channel()` operation of the notification channel factory.

The Notification Service allows any number of `SupplierAdmin` objects to be associated with a notification channel. Normally each `SupplierAdmin` object is responsible for creating and managing proxy consumers with a common set of QoS property settings and filter objects. These proxy consumer objects are used by their suppliers to push events onto the notification channel. The `new_for_suppliers()` operation is used to obtain a `SupplierAdmin` object. This operation takes the following parameters:

- `InterFilterGroupOperator`, which can be either `AND_OP` or `OR_OP`. For more information on this parameter, see 25.3.3.2.
- `AdminID`, an `out` parameter that identifies the `SupplierAdmin` object.

```
CosNotifyChannelAdmin::InterFilterGroupOperator ifgop =
  CosNotifyChannelAdmin::AND_OP;
CosNotifyChannelAdmin::AdminID adminid;
CosNotifyChannelAdmin::SupplierAdmin_var supplier_admin =
  notify_channel->new_for_suppliers (ifgop, adminid);
if (CORBA::is_nil (supplier_admin.in())) {
  std::cerr << "Unable to access supplier admin" << std::endl;
}
```

Note *The above code creates a new supplier admin each time it is run. In applications with long running notification channels, this can lead to an accumulation of supplier admins and degraded performance. In a production application, you would want to either call* `destroy()` *on the admin when you are done with it or use* `get_supplieradmin()` *or* `get_all_supplier_admins()` *on the event channel to check for an existing admin object.*

For the push supplier to push events onto the notification channel, we must obtain a proxy push consumer object reference. We get a reference to a proxy push consumer object from the `SupplierAdmin` object by using the `obtain_notification_push_consumer()` operation. This operation takes two parameters:

- `ClientType` identifies the type of events our supplier will produce, which can be `ANY_EVENT`, `STRUCTURED_EVENT`, or `SEQUENCE_EVENT`.
- `ProxyID` is an `out` parameter that identifies the proxy object.

In our example, the supplier is producing structured events, so we specify `ClientType` to be `STRUCTURED_EVENT`. Note that the return type of the `obtain_notification_push_consumer()` operation is `CosNotifyChannelAdmin::ProxyConsumer`. We must narrow it to `CosNotifyChannelAdmin::StructuredProxyPushConsumer` since our supplier will be producing structured events. The supplier will use this proxy consumer to push structured events to the notification channel:

```
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxyConsumer_var proxy_consumer =
  supplier_admin->obtain_notification_push_consumer(
    CosNotifyChannelAdmin::STRUCTURED_EVENT,
    proxy_id);

structured_proxy_consumer_ =
```

```
      CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(
        proxy_consumer.in());
    if (CORBA::is_nil(structured_proxy_consumer_.in())) {
     std::cerr << "Unable to obtain structured proxy push consumer" << std::endl;
    }
```

We now create an instance of our `StructuredEventSupplier_i` push supplier servant class. We activate the supplier servant in the `RootPOA` of our new ORB:

```
    PortableServer::Servant_var<StructuredEventSupplier_i> supplier_servant =
      new StructuredEventSupplier_i(orb_.in());
    CORBA::Object_var poa_obj = orb_->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(poa_obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();
    PortableServer::ObjectId_var objectId =
      poa->activate_object(supplier_servant.in());
```

Using the `connect_structured_push_supplier()` operation, we now connect our supplier to the consumer proxy object. This operation takes an object reference to the supplier as a parameter:

```
    CORBA::Object_var supplier_obj = poa->id_to_reference(objectId.in());
    CosNotifyComm::StructuredPushSupplier_var supplier =
      CosNotifyComm::StructuredPushSupplier::_narrow(supplier_obj.in());
    structured_proxy_consumer_->
      connect_structured_push_supplier(supplier.in());
  }
  catch (CORBA::Exception& ex) {
    std::cerr << ex << std::endl;
  }
}
```

Next, we extend the implementation of our Messenger's `send_message()` operation. In addition to printing information about the message, we create a new structured event and populate it with the contents of the message. The contents of the structured event are shown in 25.3.1. For this example, the event type domain is `OCI_TAO` and the event type name is `examples`. We invoke the `push_structured_event()` operation of the structured push consumer proxy object to push the event to the notification channel:

```
CORBA::Boolean Messenger_i::send_message (
  const char* user_name,
  const char* subject,
```

```
  char*& message)
{
  std::cerr << "Message from: " << user_name << std::endl;
  std::cerr << "Subject:      " << subject << std::endl;
  std::cerr << "Message:      " << message << std::endl;

  // Create a structured event.
  CosNotification::StructuredEvent event;

  // Populate the event's fixed header fields.
  event.header.fixed_header.event_type.domain_name =
    CORBA::string_dup("OCI_TAO");
  event.header.fixed_header.event_type.type_name =
    CORBA::string_dup("examples");
  event.header.fixed_header.event_name =
    CORBA::string_dup("myevent");

  // Populate the event's filterable body fields.
  event.filterable_data.length (3);
  event.filterable_data[0].name = CORBA::string_dup("Message from:");
  event.filterable_data[0].value <<= (const char *)user_name;
  event.filterable_data[1].name = CORBA::string_dup("Subject:");
  event.filterable_data[1].value <<= (const char *)subject;
  event.filterable_data[2].name = CORBA::string_dup("Message:");
  event.filterable_data[2].value <<= (const char *)message;

  // Push the event to the notification channel.
  structured_proxy_consumer_->push_structured_event(event);
  return true;
}
```

This simple example omits two elements that may be necessary in production applications. Both are related to process shutdown and any cleanup which would occur if `shutdown()` was called on the ORB.

- We have left no way to disconnect from the proxy push consumer. This means our proxy consumer will likely live in the event channel forever after our process exits. In a production application, you should be sure to call `disconnect_structured_push_consumer()` on the `StructuredProxyPushConsumer` to allow the event channel to destroy this unused object.
- We “leak” the servant object. The solution here is that we should use reference-counted servants, so the POA automatically deletes any servants when the POA is deleted.

25.4.2.4 Implementing the Structured Push Consumer Interface

The CosNotifyComm::StructuredPushConsumer IDL interface is implemented by the class StructuredEventConsumer_i. The three operations in the StructuredPushConsumer interface are as follows:

- push_structured_event() delivers structured events from the notification channel to the consumer.
- disconnect_structured_push_consumer() called when the notification channel disconnects the structured push consumer from its associated proxy supplier.
- offer_change() allows the notification channel to inform the consumer of changes to the event types its connected suppliers intend to produce.

```
#include <orbsvcs/CosNotifyCommS.h>

class StructuredEventConsumer_i :
  public virtual POA_CosNotifyComm::StructuredPushConsumer
{
  public:
    // Constructor.
    StructuredEventConsumer_i(CORBA::ORB_ptr orb);

    // Override operations from StructuredPushConsumer interface.
    virtual void push_structured_event(
      const CosNotification::StructuredEvent& event);
    virtual void disconnect_structured_push_consumer();
    virtual void offer_change (
      const CosNotification::EventTypeSeq& events_added,
      const CosNotification::EventTypeSeq& events_removed);
  private:
    CORBA::ORB_var orb_;
};
```

The StructuredEventConsumer_i constructor simply duplicates the ORB reference passed to it and stores it in its data member:

```
StructuredEventConsumer_i::StructuredEventConsumer_i(CORBA::ORB_ptr orb)
  : orb_(CORBA::ORB::_duplicate(orb)) { }
```

The push_structured_event() operation is called for each event that matches the consumer's subscription information. Our implementation simply extracts and prints each element from the filterable body fields of the structured event:

```
void StructuredEventConsumer_i::push_structured_event(
  const CosNotification::StructuredEvent& event)
{
  const char* value;
  for (int i=0; i<event.filterable_data.length(); ++i) {
    if (event.filterable_data[i].value >>= value) {
      std::cout << event.filterable_data[i].name << "\t" << value << std::endl;
    }
  }
}
```

The `disconnect_structured_push_consumer()` operation is called when the consumer is being disconnected from the notification channel. Our implementation deactivates the consumer object from its POA:

```
void StructuredEventConsumer_i::disconnect_structured_push_consumer()
{
  CORBA::Object_var obj = orb_->resolve_initial_references ("POACurrent");
  PortableServer::Current_var current =
    PortableServer::Current::_narrow (obj.in());
  PortableServer::POA_var poa = current->get_POA ();
  PortableServer::ObjectId_var objectId = current->get_object_id ();
  poa->deactivate_object (objectId.in());
}
```

The `offer_change()` operation is called by the consumer's supplier proxy object to inform the consumer of changes in the event types offered by the suppliers of the notification channel. Its implementation does nothing for now, but it will be expanded later when we address offers and subscriptions.

```
void StructuredEventConsumer_i::offer_change(
  const CosNotification::EventTypeSeq&,
  const CosNotification::EventTypeSeq&)
{
  // More to come...
}
```

25.4.2.5 Developing the Structured Event Consumer

Next, we create a `MessengerConsumer` application to find the notification channel created by the supplier, create an instance of our structured push consumer implementation class, connect the consumer to the channel, and process events:

```
#include "StructuredEventConsumer_i.h"
#include <orbsvcs/CosNotifyChannelAdminC.h>
#include <orbsvcs/CosNotifyCommC.h>
#include <orbsvcs/CosNamingC.h>

int main(int argc, char* argv[])
{
  try
  {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var naming_obj =
      orb->resolve_initial_references ("NameService");
    if (CORBA::is_nil(naming_obj.in())) {
      std::cerr << "Unable to find Naming Service" << std::endl;
      return 1;
    }
    CosNaming::NamingContext_var naming_context =
      CosNaming::NamingContext::_narrow(naming_obj.in());
```

When the supplier created the notification channel, it bound it in the Naming Service's root naming context to the name `MessengerChannel`. We now resolve the notification channel from the Naming Service:

```
    CosNaming::Name name;
    name.length (1);
    name[0].id = CORBA::string_dup("MessengerChannel");
    CORBA::Object_var notify_channel_obj = naming_context->resolve(name);
    CosNotifyChannelAdmin::EventChannel_var notify_channel =
      CosNotifyChannelAdmin::EventChannel::_narrow(notify_channel_obj.in());
    if (CORBA::is_nil (notify_channel.in())) {
      std::cerr << "Unable to find the notification channel" << std::endl;
      return 1;
    }
```

Alternatively, we could use the `get_event_channel()` operation on the notification channel factory to look up the notification channel using the `CosNotifyChannelAdmin::ChannelID` returned when the supplier created the channel.

Any number of `ConsumerAdmin` administration objects are allowed to be associated with a notification channel. Normally each `ConsumerAdmin` object is responsible for creating and managing proxy suppliers with a common set of QoS property settings and filter objects. These proxy supplier objects push events to their consumers. A `ConsumerAdmin` object is obtained using the `new_for_consumers()` operation. This operation takes the following parameters:

- `InterFilterGroupOperator`, which can be either `AND_OP`, or `OR_OP`. For more information on this parameter, see 25.3.3.2.
- `AdminID`, an `out` parameter that identifies the `ConsumerAdmin` object.

```
CosNotifyChannelAdmin::AdminID adminid;
CosNotifyChannelAdmin::InterFilterGroupOperator ifgop =
  CosNotifyChannelAdmin::AND_OP;
CosNotifyChannelAdmin::ConsumerAdmin_var consumer_admin =
  notify_channel->new_for_consumers (ifgop, adminid);
if (CORBA::is_nil (consumer_admin.in())) {
  std::cerr << "Unable to access consumer admin" << std::endl;
}
```

Note *The above code creates a new consumer admin each time it is run. In applications with long running notification channels, this can lead to an accumulation of consumer admins and degraded performance. In a production application, you would want to either call* `destroy()` *on the admin when you are done with it or use* `get_consumeradmin()` *or* `get_all_consumer_admins()` *on the event channel to check for an existing admin object.*

For the push consumer to connect to the notification channel and begin receiving events, we must obtain a proxy push supplier object reference. We get a reference to a proxy push supplier from the `ConsumerAdmin` object using the `obtain_notification_push_supplier()` operation. This operation takes two parameters:

- `ClientType` identifies the type of events our consumer wishes to receive, which can be `ANY_EVENT`, `STRUCTURED_EVENT`, or `SEQUENCE_EVENT`.
- `ProxyID` is an `out` parameter that identifies the proxy object.

In our example, the consumer is receiving structured events, so we specify `ClientType` to be `STRUCTURED_EVENT`. Note that the return type of the `obtain_notification_push_supplier()` operation is defined as `CosNotifyChannelAdmin::ProxySupplier`. We must narrow it to `CosNotifyChannelAdmin::StructuredProxyPushSupplier` since our consumer will be receiving structured events. The consumer will connect to this proxy supplier to begin receiving structured events from the notification channel:

```
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxySupplier_var proxy_supplier =
  consumer_admin->obtain_notification_push_supplier(
    CosNotifyChannelAdmin::STRUCTURED_EVENT,
    proxy_id);
CosNotifyChannelAdmin::StructuredProxyPushSupplier_var
  structured_proxy_supplier =
    CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(
      proxy_supplier.in());
if (CORBA::is_nil (structured_proxy_supplier.in())) {
 std::cerr << "Unable to obtain structured proxy push supplier" << std::endl;
  return 1;
}
```

We now create an instance of our `StructuredEventConsumer_i` push consumer servant class. We then activate the servant in the `RootPOA` and obtain its object reference. We then connect our consumer to the proxy push supplier using the `connect_structured_push_consumer()` operation. This operation takes an object reference to the consumer as a parameter:

```
PortableServer::Servant_var<StructuredEventConsumer_i> consumer_servant =
  new StructuredEventConsumer_i(orb.in());
CORBA::Object_var poa_obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(poa_obj.in());
PortableServer::ObjectId_var oid =
  poa->activate_object(consumer_servant.in());
CORBA::Object_var consumer_obj = poa->id_to_reference(oid.in());
CosNotifyComm::StructuredPushConsumer_var consumer =
  CosNotifyComm::StructuredPushConsumer::_narrow(consumer_obj.in());
structured_proxy_supplier->connect_structured_push_consumer(consumer.in());
```

We also add information about the event types the consumer is interested in receiving to the proxy push supplier using the `subscription_change()` operation. We add the structured events identified by the domain name “OCI_TAO” and type name “examples”, and remove all other event types from the subscription:

```
CosNotification::EventTypeSeq added;
CosNotification::EventTypeSeq removed;
added.length(1);
removed.length(1);

added[0].domain_name = CORBA::string_dup("OCI_TAO");
added[0].type_name = CORBA::string_dup("examples");

removed[0].domain_name = CORBA::string_dup("*");
```

```
    removed[0].type_name = CORBA::string_dup("*");

    structured_proxy_supplier->subscription_change(added, removed);
```

Note the use of the wildcard character “*” (an asterisk) in the domain and type name fields in the sequence of event types to be removed. This indicates a match with *all* event types in *all* domains. Therefore, all event types are removed except for the event type that was added. As before, we could also use “`%ALL`” with the same meaning.

Next, we activate the POA and enter the ORB event loop so we can receive events:

```
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();
    orb->run();
  } catch (CORBA::Exception& ex) {
    std::cerr << ex << std::endl;
    return 1;
  }
  return 0;
}
```

When an event is pushed to the notification channel, the operation `push_structured_event()` will be invoked on our consumer. This consumer process is assumed to run indefinitely and does not bother to call `disconnect_structured_push_supplier()` on its proxy. Real applications should ensure that the disconnect operation is called before the orb is shut down and the process exits. Failure to do so results in “leaking” the proxy object in the notification channel server.

25.4.3 Managing Connections

This section briefly describes how consumers and suppliers can manage their connections to the Notification Service.

25.4.3.1 Connecting and Disconnecting Consumers

Our example used the `connect_structured_push_consumer()` operation to connect to its structured proxy push supplier:

```
  structured_proxy_supplier->connect_structured_push_consumer (consumer.in());
```

The consumer remains connected to the notification channel until the corresponding `disconnect_structured_push_supplier()` operation is invoked:

```
structured_proxy_supplier->disconnect_structured_push_supplier ();
```

This disconnects the consumer from the notification channel and causes the structured proxy push supplier to be destroyed.

25.4.3.2 Connecting and Disconnecting Suppliers

Suppliers are connected and disconnected using analogous operations on the structured proxy push consumer object:

```
structured_proxy_consumer->connect_structured_push_supplier (supplier.in ());

// Do all the publication required...

structured_proxy_consumer->disconnect_structured_push_consumer ();
```

TAO's Notification Service cleans up resources associated with the supplier when disconnect is called. Failure to call disconnect causes these resource to be leaked.

25.4.3.3 Suspending and Resuming Consumer Connections

Consumers that want to stop receiving events for a period of time can use the `suspend_connection()` operation. Events published by a supplier while the consumer has suspended the connection will be held in a queue until the consumer invokes the `resume_connection()` operation. The `resume_connection()` operation will cause any queued events to be delivered to the consumer as delivery returns to normal.

Using these operations is more efficient than alternative approaches. One alternative is to just stay connected to the notification channel and ignore the events as they are pushed to the consumer. Suspending is much more efficient because the supplier stops pushing events to that consumer and events that are sent to a suspended connection are not lost.

Another approach is to disconnect from the event channel, then later reconnect to the same channel. Events that are waiting to be delivered to the consumer, when the consumer disconnects, and events that are sent to the channel after

the disconnection, will never be delivered to this consumer. When the consumer reconnects to the channel it will begin receiving new events sent to that channel.

A framework for the use of these operations is as follows:

```
structured_proxy_supplier->suspend_connection ();

 // Do something else for a while...

structured_proxy_supplier->resume_connection ();
```

25.4.3.4 Destroying the Notification Channel

Notification channels consume resources within the process in which they are created. You can invoke the `destroy()` operation to destroy a notification channel and release its resources when it is no longer needed.

```
notify_channel->destroy ();
```

When a notification channel is destroyed, all currently connected consumers and suppliers are notified via their respective disconnect operations. This allows consumers and suppliers to clean up any resource they may be holding.

25.4.4 Using Offers and Subscriptions

In this section, we show how to add offers and subscriptions to our simple example. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NotifyService/OfferSubscriptions/`.

25.4.4.1 Adding Publication of Offers to the Supplier

The collection of event types a supplier produces is called an *offer*. A supplier can inform consumers of the notification channel about the types of events it will be producing by invoking the `offer_change()` operation. Later, it can invoke the `offer_change()` operation again to add or remove event types from its offer. The notification channel aggregates the offers of all its suppliers so consumers can be informed of what types of events are currently published via the channel. The `offer_change()` operation can be invoked on the supplier's consumer proxy object or on its `SupplierAdmin` object.

Here, we show how to add publication of offers to the consumer proxy object. The `offer_change()` operation takes two parameters:

- A sequence of event types the supplier will *add* to its offer (i.e., the event types it will *start* supplying).
- A sequence of event types the supplier will *remove* from its offer (i.e., the event types it will *no longer* supply).

The type of these parameters is `CosNotification::EventTypeSeq`. Each event type in the sequence is a structure containing two strings:

- The *domain_name* identifies the vertical industry domain in which the event is defined (e.g., telecommunications, finance, health-care).
- The *type_name* categorizes the event within the industry domain.

In the following example, we use the `offer_change()` operation to add the structured event type identified by domain name `OCI_TAO` and type name `examples` to our supplier's offer, and to remove all other event types from the offer:

```
// Add one event type to the offer.
CosNotification::EventTypeSeq events_added;
events_added.length (1);
events_added[0].domain_name = CORBA::string_dup ("OCI_TAO");
events_added[0].type_name = CORBA::string_dup ("examples");

// Remove all other event types from the offer.
CosNotification::EventTypeSeq events_to_be_removed;
events_removed.length (1);
events_removed[0].domain_name = CORBA::string_dup ("*");
events_removed[0].type_name = CORBA::string_dup ("*");

// Change the offer.
structured_proxy_consumer->offer_change (events_added, events_removed);
```

Invoking the `offer_change()` operation on the supplier's consumer proxy object, as in the above example, affects only that supplier's offer. Its offer will be aggregated with the offers of all the other suppliers of the notification channel and published as the offer of events for the channel as a whole.

On the other hand, invoking the `offer_change()` operation on the supplier administration object, as in the following example, affects the publication of offers from all the suppliers that share the supplier administration object:

```
// Change the offer.
supplier_admin->offer_change (events_added, events_removed);
```

Note that this does not necessarily change the *actual* types of events the suppliers can produce and push onto the channel. Suppliers can still produce any event type whether or not the event type is published in the notification channel's offer. The `offer_change()` operation only affects the aggregate of the event types published for the channel (and therefore visible to consumers via the `obtain_offered_types()` operation).

25.4.4.2 Adding Subscriptions to the Consumer

The collection of event types in which a consumer is interested is called a *subscription*. A consumer can inform notification channel suppliers about the types of events it requires by invoking the `subscription_change()` operation. Later, it can invoke the `subscription_change()` operation again to add new event types or remove event types from its subscription. The notification channel aggregates the subscriptions of all its consumers so suppliers can be informed of what types of events consumers of the channel require. The `subscription_change()` operation can be invoked on the consumer's supplier proxy object or on its `ConsumerAdmin` object.

Here, we show how to add a subscription to the notification event consumer. The `subscription_change()` operation takes two parameters:

- A sequence of event types the consumer will *add* to its subscription (i.e., the event types it wants to *start* receiving).
- A sequence of event types the consumer will *remove* from its subscription (i.e., the event types it *no longer* wishes to receive).

The type of these parameters is `CosNotification::EventTypeSeq`, described in 25.4.4.1.

In the following example, we use the `subscription_change()` operation to add the structured event type identified by domain name `OCI_TAO` and type name `examples` to our consumer's subscription, and to remove all other event types from the subscription:

```
// Add one event type to the subscription.
CosNotification::EventTypeSeq events_added;
events_added.length (1);
events_added[0].domain_name = CORBA::string_dup ("OCI_TAO");
events_added[0].type_name = CORBA::string_dup ("examples");
```

```
// Remove all other event types from the subscription.
CosNotification::EventTypeSeq events_removed;
events_removed.length (1);
events_removed[0].domain_name = CORBA::string_dup ("*");
events_removed[0].type_name = CORBA::string_dup ("*");

// Change the subscription
structured_proxy_supplier->subscription_change (
  events_added, events_removed);
```

Invoking the `subscription_change()` operation on the consumer's supplier proxy object, as in the above example, affects only that consumer's subscription. Its subscription will be aggregated with the subscription of all other consumers of the notification channel and published as the subscription of events for the channel as a whole.

On the other hand, invoking the `subscription_change()` operation on the consumer administration object, as in the following example, affects the publication of subscriptions from all the consumers that share the consumer administration object:

```
// Change the subscription.
consumer_admin->subscription_change(events_added, events_removed);
```

Note that this does not necessarily change the *actual* types of events the consumers can receive from the channel. Consumers can still receive any event type (that passes all filters) whether or not the event type is in the notification channel's aggregate subscription. However, you may choose to implement the operation `subscription_change()` in your suppliers such that they stop producing events that are no longer required by any consumers of the channel, as indicated in the aggregate of all the consumers' subscriptions. In that case, the `subscription_change()` operation may in fact change the actual types of events the consumers will receive.

For example, our structured event supplier class might implement the `subscription_change()` operation as follows to determine if we should continue to produce events for the messages our `MessengerServer` receives. (Assume that the field `produce_message_events_` is a boolean flag indicating whether or not message events are to be produced. For brevity, we are not checking the "`%ALL`" wildcard.)

```
void StructuredEventSupplier_i::subscription_change (
```

```
  const CosNotification::EventTypeSeq& events_added,
  const CosNotification::EventTypeSeq& events_removed)
{
  // Check if we are to produce "examples" events.

  // Check the list of removed event types.
  int i;
  for (i=0; i<events_removed.length(); ++i) {
    if (!strcmp(events_removed[i].domain_name, "OCI_TAO") &&
        !strcmp(events_removed[i].type_name, "examples")) {
      produce_message_events_ = false;
    }
    else if (!strcmp(events_removed[i].domain_name, "*") &&
             !strcmp(events_removed[i].type_name, "*")) {
      produce_message_events_ = false;
    }
  }

  // Check the list of added event types.
  for (i=0; i<events_added.length(); ++i) {
    if (!strcmp(events_added[i].domain_name, "OCI_TAO") &&
        !strcmp(events_added[i].type_name, "examples")) {
      produce_message_events_ = true;
    }
    else if (!strcmp(events_added[i].domain_name, "*") &&
             !strcmp(events_added[i].type_name, "*")) {
      produce_message_events_ = true;
    }
  }
}
```

We could then modify the Messenger servant's `send_message()` operation from 25.4.2.3 to check the `produce_message_events_` flag before producing an event.

```
CORBA::Boolean Messenger_i::send_message (
  const char* user_name,
  const char* subject,
  char*& message
  )
{
  std::cerr << "Message from: " << user_name << std::endl;
  std::cerr << "Subject:      " << subject << std::endl;
  std::cerr << "Message:      " << message << std::endl;

  if (produce_message_events_ == true) {
    // Create and send a structured event as before...
  }
  return true;
```

```
}
```

25.4.4.3 Obtaining Offer and Subscription Information

Suppliers invoke the `obtain_subscription_types()` operation on the proxy consumer to get the list of event types the consumers connected to the notification channel require. Consumers invoke `obtain_offered_types()` on the proxy supplier to get a list of event types the suppliers connected to the notification channel produce. Each of these operations takes an input parameter of type `CosNotifyChannelAdmin::ObtainInfoMode` that controls what information is returned from the operation as well as whether subsequent automatic updates of subscription information (via the supplier operation `subscription_change()`) or offer information (via the consumer operation `offer_change()`) are enabled or disabled.

Here is a portion of the IDL definitions within the `CosNotifyChannelAdmin` module showing the definitions of these operations and the `ObtainInfoMode` type:

```
module CosNotifyChannelAdmin
{
  enum ObtainInfoMode {
    ALL_NOW_UPDATES_OFF,
    ALL_NOW_UPDATES_ON,
    NONE_NOW_UPDATES_OFF,
    NONE_NOW_UPDATES_ON
  };

  interface ProxyConsumer :
    CosNotification::QoSAdmin,
    CosNotifyFilter::FilterAdmin
  {
    CosNotification::EventTypeSeq obtain_subscription_types (
      in ObtainInfoMode mode);
  };

  interface ProxySupplier :
    CosNotification::QoSAdmin,
    CosNotifyFilter::FilterAdmin
  {
    CosNotification::EventTypeSeq obtain_offered_types (
      in ObtainInfoMode mode);
  };
};
```

Table 25-7 lists the possible values for the `ObtainInfoMode` parameter.

Table 25-7 ObtainInfoMode Parameter Values

Value	Description
`ALL_NOW_UPDATES_OFF`	Returns the current list of subscription (offer) types known by the target proxy consumer (supplier). Subsequent automatic sending of subscription (offer) update information is disabled.
`ALL_NOW_UPDATES_ON`	Returns the current list of subscription (offer) types known by the target proxy consumer (supplier). Subsequent automatic sending of subscription (offer) update information is enabled.
`NONE_NOW_UPDATES_OFF`	The invocation does not return any data. Subsequent automatic sending of subscription (offer) update information is disabled.
`NONE_NOW_UPDATES_ON`	The invocation does not return any data. Subsequent automatic sending of subscription (offer) update information is enabled.

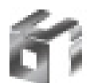
For example, to get a list of all event types currently required by consumers of the channel, and to enable (via `subscription_change()`) subscription updates, our supplier would invoke `obtain_subscription_types()` on its consumer proxy object:

```
CosNotification::EventTypeSeq_var event_type_seq =
  structured_proxy_consumer->obtain_subscription_types (
    CosNotifyChannelAdmin::ALL_NOW_UPDATES_ON);
```

To get a list of all event types currently offered by suppliers of the channel, and to disable future offer updates via `offer_change()`, our consumer would invoke the `obtain_offered_types()` operation on its supplier proxy object:

```
CosNotification::EventTypeSeq_var event_type_seq =
  structured_proxy_supplier->obtain_offered_types (
    CosNotifyChannelAdmin::ALL_NOW_UPDATES_OFF);
```

Before `obtain_offered_types()` or `obtain_subscription_types()` are called, offer and subscription updates are enabled. If you do not wish to receive these updates, you must explicitly disable them as described above.



25.4.5 Adding Event Filtering

In this section, we add event filtering to our simple example. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NotifyService/Filtering/`.

25.4.5.1 Adding Event Filtering to the Supplier

Suppose we want to ensure that only events matching certain criteria are supplied to the notification channel. We want events not matching the criteria to be discarded. For example, assume we want to allow only those events for which the *Subject* field exactly matches the string `urgent`. Recall that we can filter on any field in the optional header part, filterable body part, or remaining body part of the structured event. To accomplish this, we can create a filter object and attach it to the supplier administration object. This filter would be shared by all the suppliers associated with that `SupplierAdmin` object. To apply a filter to a single supplier, we would attach it to that supplier's proxy consumer. Both of these approaches depend on our use of an admin object created with the AND inter-group filter operator:

```
CosNotifyChannelAdmin::AdminID adminid;
CosNotifyChannelAdmin::SupplierAdmin_var supplier_admin =
  ec->new_for_suppliers (CosNotifyChannelAdmin::AND_OP, adminid);
```

Each notification channel provides a filter factory for creating filter objects. Use the `default_filter_factory()` operation to obtain an object reference to the channel's filter factory object:

```
CosNotifyFilter::FilterFactory_var filter_factory =
  notify_channel->default_filter_factory ();
```

Next, the factory must create a filter object. The factory's `create_filter()` operation takes one parameter representing the name of the constraint language the filter object will use to express filtering constraints. The Notification Service specification defines `EXTENDED_TCL` as the standard value for specifying ETCL filters. TAO's Notification Service also understands `ETCL`, but we'll use the standard string for portability reasons:

```
CosNotifyFilter::Filter_var filter =
           filter_factory->create_filter ("EXTENDED_TCL");
if (CORBA::is_nil (filter.in())) {
  std::cerr << "Unable to create filter object" << std::endl;
```

```
    return 1;
  }
```

Next, create a sequence of constraints. Each constraint is a data structure with two members.

- A sequence of event types indicating the types of events to which the constraint applies. Its type is `CosNotification::EventTypeSeq`, described in 25.4.4.1.
- A string specifying a boolean expression in the chosen constraint grammar that will be applied to the contents of each event if it is of a type contained in the sequence of event types in the first element.

By convention, if the event type sequence has a length of zero, or if it contains a single element in which the domain and event type names are empty strings, the boolean expression applies to all event types. Wildcard characters ("`*`" and "`%ALL`") are also allowed in the event type fields.

Use the `add_constraints()` operation to add the constraints to the filter object. Then add the filter object to the supplier administration object or the proxy consumer object using the `add_filter()` operation.

Here we create a constraint list with a single element. The constraint expression `$.Subject =='urgent'` is applied to all events regardless of the event's type.

```
CosNotifyFilter::ConstraintExpSeq constraint_list;
constraint_list.length (1);
constraint_list[0].event_types.length (0);
constraint_list[0].constraint_expr =
  CORBA::string_dup ("$.Subject == 'urgent'");
filter->add_constraints (constraint_list);

// Apply the filter to all suppliers sharing the same SupplierAdmin object.
supplier_admin->add_filter (filter.in());
```

or

```
// Apply the filter to only one supplier's proxy consumer object.
structured_proxy_consumer->add_filter (filter.in());
```

The result is that only events with a field `Subject` that has the value `urgent` will be forwarded to the notification channel.

The results of multiple constraints within a filter object are ORed together to determine if the event matches any of the filtering constraints. If the result is true, the event has passed the filter. If there are multiple filters added to a proxy or supplier administration object, the event is evaluated against each filter until one of the filters returns true or all the filters return false. If all the filters return false, the event is discarded. If one of the filters returns true, the event passes and is forwarded to the notification channel.

25.4.5.2 Adding Event Filtering to the Consumer

Filtering can also be applied to the consumer side of Notification Service event communication. With consumer-side filtering, a structured event consumer can specify the precise set of events it is interested in receiving, based upon filtering criteria. Only events that meet these criteria will be forwarded to the consumer by the consumer's proxy supplier. As on the supplier side, we need to ensure that we use the AND inter-group filter operator:

```
CosNotifyChannelAdmin::AdminID adminid;
CosNotifyChannelAdmin::ConsumerAdmin_var consumer_admin =
  ec->new_for_consumers(CosNotifyChannelAdmin::AND_OP, adminid);
```

For example, suppose our consumer object wants to receive events only for those messages that originated from "sysadmin@company.com". We can add a filter on the consumer side to filter incoming events accordingly. Similar to what we did in the supplier, we now add a filter object to the consumer administration object.

```
CosNotifyFilter::FilterFactory_var filter_factory =
  notify_channel->default_filter_factory ();

CosNotifyFilter::Filter_var filter =
     filter_factory->create_filter ("EXTENDED_TCL");
if (CORBA::is_nil (filter.in())) {
  std::cerr << "Unable to create filter object" << std::endl;
  return 1;
}

CosNotifyFilter::ConstraintExpSeq constraint_list;
constraint_list.length (1);
constraint_list[0].event_types.length (0);
constraint_list[0].constraint_expr =
  CORBA::string_dup ("$.From == 'sysadmin@company.com'");
filter->add_constraints (constraint_list);
```

```
// Apply the filter to all consumers sharing the same ConsumerAdmin object.
consumer_admin->add_filter (filter.in());
```

or

```
// Apply the filter to only one consumer's proxy supplier object.
structured_proxy_supplier->add_filter (filter.in());
```

Our consumer will now receive only those events for messages originating from the `MessengerClient` at `sysadmin@company.com`.

25.4.6 Adding QoS Properties

Next, we extend the `MessengerServer` code from our basic example to use certain QoS properties. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NotifyService/QoSProperties/`.

We can add notification administrative QoS properties to the event channel created in the `Messenger_i` class. In this example, we add the `MaxQueueLength`, `MaxSuppliers`, and `MaxConsumers` properties and set their values to 7, 5, and 5 respectively:

```
CosNotifyChannelAdmin::EventChannelFactory_var notify_factory =
  CosNotifyChannelAdmin::EventChannelFactory::_narrow (obj.in ());

CosNotifyChannelAdmin::ChannelID id;
CosNotification::QoSProperties initial_qos;
CosNotification::AdminProperties initial_admin;

initial_admin.length (3);

initial_admin[0].name =
  CORBA::string_dup (CosNotification::MaxQueueLength);
initial_admin[0].value <<= (CORBA::Long)7;

initial_admin[1].name = CORBA::string_dup (CosNotification::MaxSuppliers);
initial_admin[1].value <<= (CORBA::Long)5;

initial_admin[2].name = CORBA::string_dup (CosNotification::MaxConsumers);
initial_admin[2].value <<= (CORBA::Long)5;

CosNotifyChannelAdmin::EventChannel_var ec =
  notify_factory->create_channel (initial_qos, initial_admin, id);
```

Since we set the MaxQueueLength to be 7, the notification channel will queue the incoming events internally until the number of events exceeds 7. Once the number of events exceeds this limit the channel will start to discard the queued events. We restrict the number of suppliers that can be connected to the notification event channel at a time to be 5. If an attempt is made to connect a supplier when the number of suppliers connected to the channel is 5, IMP_LIMIT exception is raised. We also restrict the number of consumers that can be connected to the notification event channel at a time to be 5. If an attempt is made to connect a consumer when the number of consumers connected to the channel is 5, then IMP_LIMIT exception is raised.

QoS properties can be added at the proxy level too. To illustrate this, we add the OrderPolicy property to the proxy supplier object and set its value to FifoOrder in our MessengerConsumer:

```
CosNotifyChannelAdmin::StructuredProxyPushSupplier_var supplier_proxy;
supplier_proxy =
  CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow (
    proxy_supplier.in());

CosNotification::QoSProperties properties (1);

properties.length (1);
properties[0].name = CORBA::string_dup (CosNotification::OrderPolicy);
properties[0].value <<= CosNotification::FifoOrder;

supplier_proxy->set_qos (properties);
supplier_proxy->connect_structured_push_consumer(consumer.in());
```

Since we set the OrderPolicy property to FifoOrder, the proxy supplier will arrange the events in its dispatch queue in the same order as the events were received.

25.4.7 Transmitting an EventBatch

The Notification Service also defines interfaces required to transfer more than one structured event using one operation, in the form of a sequence of structured events known as an EventBatch. In this section, we extend our example to show how to use batched events. Full source code for this example is in the TAO source code distribution in the directory $TAO_ROOT/orbsvcs/DevGuideExamples/NotifyService/EventSequence/.

We modify the MessengerServer to create an EventBatch whenever it receives a message from a client and send it through a notification event channel to consumers. To transmit event batches, we need to implement the SequencePushSupplier and SequencePushConsumer interfaces, which define the behavior of objects that transmits event batches using push-style communication. Our EventSequenceSupplier_i class implements the SequencePushSupplier interface and our EventSequenceConsumer_i class implements the SequencePushConsumer interface.

25.4.7.1 Implementing the Sequence Push Supplier Interface

The CosNotifyComm::SequencePushSupplier IDL interface is implemented by our supplier class EventSequenceSupplier_i. The SequencePushSupplier interface has two operations: disconnect_sequence_push_supplier() and subscription_change().

The disconnect_sequence_push_supplier() operation is called when the supplier is being disconnected from the notification channel. Our implementation deactivates the supplier object from its POA:

```
void EventSequenceSupplier_i::disconnect_sequence_push_supplier ( )
{
  CORBA::Object_var obj = orb_->resolve_initial_references ("POACurrent");
  PortableServer::Current_var current =
    PortableServer::Current::_narrow (obj.in());
  PortableServer::POA_var poa = current->get_POA ();
  PortableServer::ObjectId_var objectId = current->get_object_id ();
  poa->deactivate_object (objectId.in());
}
```

The subscription_change() operation is called by the supplier’s consumer proxy object to inform the supplier of changes to the subscription information by the notification channel’s consumers. A specific implementation is not necessary for this example:

```
void EventSequenceSupplier_i::subscription_change (
  const CosNotification::EventTypeSeq& added,
  const CosNotififcation::EventTypeSeq& removed)
{
}
```

25.4.7.2 Developing the EventBatch Supplier

Next, we implement our MessengerServer as a supplier of an EventBatch. Each time it receives a message, it will create a new EventBatch and populate it with the information from the message, then push this EventBatch to the notification channel via its consumer proxy.

In the MessengerServer class, we initialize the ORB and get a reference to the RootPOA. We create a Messenger_i servant and activate it in the RootPOA. Once the servant is activated, we convert the servant's object reference to a string and write it to the file Messenger.ior as usual:

```
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main(int argc, char* argv[])
{
  try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var obj = orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj.in());
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    PortableServer::Servant_var<Messenger_i> messenger_servant =
      new Messenger_i(orb.in());
    PortableServer::ObjectId_var oid =
      poa->activate_object(messenger_servant.in());
    CORBA::Object_var messenger_obj = poa->id_to_reference(oid.in());
    CORBA::String_var str = orb->object_to_string(messenger_obj.in());
    std::ofstream iorfile("Messenger.ior");
    iorfile << str.in() << std::endl;
    std::cout << "IOR written to file Messenger.ior" << std::endl;
    orb->run();
    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
    std::cerr << ex << std::endl;
    return 1;
  }
  return 0;
}
```

Now we turn our attention to the Messenger_i class that implements the Messenger interface. It now acts as an EventBatch supplier. In the constructor of the Messenger_i class we create a new notification channel

and bind it to the root naming context of the Naming Service with the name `MessengerChannel`:

```
Messenger_i::Messenger_i (CORBA::ORB_ptr orb)
  : orb_(CORBA::ORB::_duplicate(orb))
{
  try {
    CORBA::Object_var naming_obj =
      orb_->resolve_initial_references ("NameService");
    CosNaming::NamingContext_var naming_context =
      CosNaming::NamingContext::_narrow(naming_obj.in());

    CosNaming::Name name;
    name.length (1);
    name[0].id = CORBA::string_dup("NotifyEventChannelFactory");
    CORBA::Object_var obj = naming_context->resolve(name);

    CosNotifyChannelAdmin::EventChannelFactory_var notify_factory =
      CosNotifyChannelAdmin::EventChannelFactory::_narrow(obj.in());
    if (CORBA::is_nil(notify_factory.in())) {
      std::cerr << "Unable to find notify factory" << std::endl;
    }

    CosNotifyChannelAdmin::ChannelID id;
    CosNotification::QoSProperties initial_qos;
    CosNotification::AdminProperties initial_admin;

    CosNotifyChannelAdmin::EventChannel_var notify_channel =
      notify_factory->create_channel (initial_qos,
                                      initial_admin,
                                      id);
    if (CORBA::is_nil(notify_channel.in())) {
      std::cerr << "Unable to create notification channel" << std::endl;
    }

    name[0].id = CORBA::string_dup("MessengerChannel");
    naming_context->rebind(name, notify_channel.in());
```

Once we create the notification channel we obtain a `SupplierAdmin` object using `new_for_suppliers()` operation:

```
    CosNotifyChannelAdmin::InterFilterGroupOperator ifgop =
      CosNotifyChannelAdmin::AND_OP;
    CosNotifyChannelAdmin::AdminID adminid;
    CosNotifyChannelAdmin::SupplierAdmin_var supplier_admin =
      notify_channel->new_for_suppliers (ifgop, adminid);
    if (CORBA::is_nil (supplier_admin.in())) {
      std::cerr << "Unable to access supplier admin" << std::endl;
```

```
}
```

Next, we obtain a proxy push consumer object so that the push supplier can push events to the notification channel. We obtain a reference to a proxy push consumer object by invoking `obtain_notification_push_consumer()` on the `SupplierAdmin` object. The parameters passed to this operation are `ClientType` and `ProxyID`. `ClientType` identifies the types of events our supplier will produce. `ProxyID` is an `out` parameter that identifies the proxy object. Since our supplier is producing a sequence of structured events, we specify `ClientType` to be `SEQUENCE_EVENT`.

We obtain a `CosNotifyChannelAdmin::ProxyConsumer` object reference from the `obtain_notification_push_consumer()` operation. We narrow it to `CosNotifyChannelAdmin::SequenceProxyPushConsumer` since our supplier will be producing sequences of structured events. The supplier uses this proxy consumer to push sequences of structured events to the notification channel:

```
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxyConsumer_var proxy_consumer =
  supplier_admin->obtain_notification_push_consumer(
    CosNotifyChannelAdmin::SEQUENCE_EVENT,
    proxy_id);

sequence_proxy_consumer_ =
  CosNotifyChannelAdmin::SequenceProxyPushConsumer::_narrow(
    proxy_consumer.in());
if (CORBA::is_nil(sequence_proxy_consumer_.in())) {
  std::cerr << "Unable to obtain sequence proxy push consumer" << std::endl;
}
```

Next, we create an instance of our `EventSequenceSupplier_i` push supplier servant class and activate it in the `RootPOA`. Finally, we connect our supplier to the consumer proxy object:

```
PortableServer::Servant_var<EventSequenceSupplier_i> supplier_servant =
  new EventSequenceSupplier_i(orb_.in());
CORBA::Object_var poa_obj = orb_->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(poa_obj.in());
PortableServer::POAManager_var mgr = poa->the_POAManager();
mgr->activate();
PortableServer::ObjectId_var objectId =
poa->activate_object(supplier_servant.in());

CORBA::Object_var supplier_obj = poa->id_to_reference(objectId.in());
```

```
    CosNotifyComm::SequencePushSupplier_var supplier =
      CosNotifyComm::SequencePushSupplier::_narrow(supplier_obj.in());
    sequence_proxy_consumer_->
      connect_sequence_push_supplier(supplier.in());
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
  }
}
```

In the `send_message()` operation of `Messenger_i` class we print the information about the message received. Then, we create a new structured event and populate it with the contents of the message. For this example, we set the event type domain to be `OCI_TAO` and the event type name to be `examples`. We then create an `EventBatch` from the single structured event we created to illustrate how we can transmit an `EventBatch` without adding much complexity. We invoke the `push_structured_events()` operation of the sequence push consumer proxy object to push the `EventBatch` to the notification channel:

```
CORBA::Boolean Messenger_i::send_message (
  const char* user_name,
  const char* subject,
  char*& message
  )
{
  std::cerr << "Message from: " << user_name << std::endl;
  std::cerr << "Subject:      " << subject << std::endl;
  std::cerr << "Message:      " << message << std::endl;

  try
  {
    // Create a structured event.
    CosNotification::StructuredEvent event;

    // Populate the event's fixed header fields.
    event.header.fixed_header.event_type.domain_name =
      CORBA::string_dup("OCI_TAO");
    event.header.fixed_header.event_type.type_name =
      CORBA::string_dup("examples");
    event.header.fixed_header.event_name =
      CORBA::string_dup("myevent");

    // Populate the event's filterable body fields.
    event.filterable_data.length (3);
    event.filterable_data[0].name = CORBA::string_dup("Message from:");
    event.filterable_data[0].value <<= (const char *)user_name;
```

```
    event.filterable_data[1].name = CORBA::string_dup("Subject:");
    event.filterable_data[1].value <<= (const char *)subject;
    event.filterable_data[2].name = CORBA::string_dup("Message:");
    event.filterable_data[2].value <<= (const char *)message;

    // Create and populate an EventBatch.
    // (We simply put 4 copies of the same event into the sequence.)
    CosNotification::EventBatch events;
    events.length(4);
    events[0] = event;
    events[1] = event;
    events[2] = event;
    events[3] = event;

    // Push the event to the notification channel.
    sequence_proxy_consumer->push_structured_events(events);
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return false;
  }
  return true;
}
```

25.4.7.3 Implementing the Sequence Push Consumer Interface

The `CosNotifyComm::SequencePushConsumer` IDL interface is implemented by the class `EventSequenceConsumer_i`. The `SequencePushConsumer` interface contains the three operations: `push_structured_events()`, `disconnect_sequence_push_consumer()`, and `offer_change()`.

The `push_structured_events()` operation is invoked for each event sequence that matches the consumer's subscription information. Our implementation simply extracts and prints the filterable body fields of the structured events contained in the `EventBatch`:

```
void EventSequenceConsumer_i::push_structured_events (
  const CosNotification::EventBatch& events)
{
  const char* value;
  for (unsigned int n=0; n<events.length(); ++n) {
    for (unsigned int i=0; i<events[n].filterable_data.length(); ++i) {
      events[n].filterable_data[i].value >>= value;
     std::cout << events[n].filterable_data[i].name << "\t" << value << std::endl;
    }
```

```
  }
}
```

The `disconnect_sequence_push_consumer()` operation is invoked when the consumer is being disconnected from the notification channel. Our implementation deactivates the consumer object from its POA:

```
void EventSequenceConsumer_i::disconnect_sequence_push_consumer ()
{
  CORBA::Object_var obj = orb_->resolve_initial_references ("POACurrent");
  PortableServer::Current_var current =
    PortableServer::Current::_narrow (obj.in());
  PortableServer::POA_var poa = current->get_POA ();
  PortableServer::ObjectId_var objectId = current->get_object_id ();
  poa->deactivate_object (objectId.in());
}
```

The `offer_change()` operation is invoked by the consumer's supplier proxy object to inform the consumer of changes in the event types offered by the suppliers of the notification channel. A specific implementation is not necessary for this example:

```
void StructuredEventConsumer_i::offer_change(
  const CosNotification::EventTypeSeq&,
  const CosNotification::EventTypeSeq&)
{
}
```

25.4.7.4 Developing the Event Consumer

Next, we create a `MessengerConsumer` application to find the notification channel created by the supplier. In this application we create an instance of the `EventSequenceConsumer` push consumer class, connect the consumer to the channel, and process events.

In the `MessengerConsumer` class, we initialize the ORB, locate the Naming Service, then resolve the `MessengerChannel` that was created by our supplier:

```
#include <orbsvcs/CosNamingC.h>
#include <orbsvcs/CosNotifyChannelAdminC.h>
#include <orbsvcs/CosNotificationC.h>
#include <iostream>
```

```
int main(int argc, char* argv[])
{
  try
  {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var naming_obj =
      orb->resolve_initial_references ("NameService");
    if (CORBA::is_nil(naming_obj.in())) {
      std::cerr << "Unable to find Naming Service" << std::endl;
      return 1;
    }
    CosNaming::NamingContext_var naming_context =
      CosNaming::NamingContext::_narrow(naming_obj.in());

    CosNaming::Name name;
    name.length (1);
    name[0].id = CORBA::string_dup("MessengerChannel");
    CORBA::Object_var notify_channel_obj = naming_context->resolve(name);
    CosNotifyChannelAdmin::EventChannel_var notify_channel =
      CosNotifyChannelAdmin::EventChannel::_narrow(notify_channel_obj.in());
    if (CORBA::is_nil (notify_channel.in())) {
      std::cerr << "Unable to find the notification channel" << std::endl;
      return 1;
    }
```

Next, we obtain a `ConsumerAdmin` object using the `new_for_consumers()` operation:

```
    CosNotifyChannelAdmin::AdminID adminid;
    CosNotifyChannelAdmin::InterFilterGroupOperator ifgop =
      CosNotifyChannelAdmin::AND_OP;
    CosNotifyChannelAdmin::ConsumerAdmin_var consumer_admin =
      notify_channel->new_for_consumers (ifgop, adminid);
    if (CORBA::is_nil (consumer_admin.in())) {
      std::cerr << "Unable to access consumer admin" << std::endl;
    }
```

We then use the `obtain_notification_push_supplier()` operation to obtain a proxy supplier. The parameters passed to this operation are `ClientType` and `ProxyID`. The `ClientType` parameter identifies the types of events our supplier will produce. The `ProxyID` parameter is an `out` parameter that identifies the proxy object. Since our consumer is receiving a sequence of structured events, we specify `ClientType` to be `SEQUENCE_EVENT`.

We obtain a `CosNotifyChannelAdmin::ProxySupplier` object from the `obtain_notification_push_supplier()` operation. We narrow it to

`CosNotifyChannelAdmin::SequenceProxyPushSupplier` since our supplier will be receiving sequences of structured events. The consumer will connect to this proxy supplier to begin receiving sequences of structured events from the notification channel:

```
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxySupplier_var proxy_supplier =
  consumer_admin->obtain_notification_push_supplier(
    CosNotifyChannelAdmin::SEQUENCE_EVENT,
    proxy_id);
CosNotifyChannelAdmin::SequenceProxyPushSupplier_var
  sequence_proxy_supplier =
    CosNotifyChannelAdmin::SequenceProxyPushSupplier::_narrow(
      proxy_supplier.in());
if (CORBA::is_nil (sequence_proxy_supplier.in())) {
  std::cerr << "Unable to obtain sequence proxy push supplier" << std::endl;
  return 1;
}
```

We create an instance of our `EventSequenceConsumer_i` push consumer servant class and activate it in the `RootPOA`. We then obtain its object reference and connect it to the proxy push supplier using the `connect_sequence_push_consumer()` operation. We pass the consumer's object reference as a parameter this operation. Information about the event types that the consumer is interested in receiving is also passed to the proxy push supplier using the `subscription_change()` operation:

```
PortableServer::Servant_var<EventSequenceConsumer_i> consumer_servant =
  EventSequenceConsumer_i(orb.in());
CORBA::Object_var poa_obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa =
  PortableServer::POA::_narrow(poa_obj.in());
PortableServer::ObjectId_var oid =
  poa->activate_object(consumer_servant.in());
CORBA::Object_var consumer_obj = poa->id_to_reference(oid.in());
CosNotifyComm::SequencePushConsumer_var consumer =
  CosNotifyComm::SequencePushConsumer::_narrow(consumer_obj.in());
structured_proxy_supplier->connect_sequence_push_consumer(consumer.in());

CosNotification::EventTypeSeq added;
CosNotification::EventTypeSeq removed;
added.length(1);
removed.length(1);

added[0].domain_name = CORBA::string_dup("OCI_TAO");
added[0].type_name = CORBA::string_dup("examples");
```

```
    removed[0].domain_name = CORBA::string_dup("*");
    removed[0].type_name = CORBA::string_dup("*");

    sequence_proxy_supplier->subscription_change(added, removed);
```

Finally, we activate the POA and enter the ORB event loop so we can receive events. When an event is pushed to the notification channel, the operation `push_structured_events()` will be invoked on our consumer.

```
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();
    orb->run();
    orb->destroy();
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Caught a CORBA exception: " << ex << std::endl;
    return 1;
  }
  return 0;
}
```

25.4.8 Collocated Notification Channels

The previous examples used the `NotifyEventChannelFactory` object provided by the `tao_cosnotification` server to create notification channels. Thus, the channel factory and the notification channels have all been located in a separate process (the `tao_cosnotification` server) from the consumers or suppliers. For performance reasons, there may be situations in which you want to create and manage local, or *collocated*, notification channels. For example, you may want to collocate a notification channel with a particular supplier to avoid a network hop for each event originating from that supplier. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NotifyService/SupplierSideNC/`.

25.4.8.1 Collocated Notification Channel Example

Here we show how to create and use a local notification channel factory. We then use this factory to create a notification channel collocated with a supplier.

First, include the following header file that defines the notification channel factory implementation (servant) class:

```
#include <orbsvcs/Notify/Notify_EventChannelFactory_i.h>
```

Next, in your program's `main()` function, initialize the ORB and obtain and activate the `RootPOA` as usual:

```
int main(int argc, char* argv[])
{
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var poa_object = orb->resolve_initial_references("RootPOA");

   PortableServer::POA_var poa =
      PortableServer::POA::_narrow(poa_object.in());
   PortableServer::POAManager_var mgr = poa->the_POAManager();
   mgr->activate();
```

The `TAO_Notify_EventChannelFactory_i::create()` operation is now used to create an instance of TAO's notification event channel factory implementation class. The `create()` function not only creates a factory servant instance, but also activates it in a POA (provided as a parameter) and returns the factory's object reference:

```
    CosNotifyChannelAdmin::EventChannelFactory_var notify_factory =
      TAO_Notify_EventChannelFactory_i::create(poa.in());
```

Next, we use this local factory to create our notification channel using the `create_channel()` operation exactly as before (see 25.4.2.3). However, the new notification channel will be a local object:

```
    CosNotifyChannelAdmin::ChannelID id;
    CosNotification::QoSProperties initial_qos;
    CosNotification::AdminProperties initial_admin;

    CosNotifyChannelAdmin::EventChannel_var notify_channel =
      notify_factory->create_channel(initial_qos,
                                     initial_admin,
                                     id);
```

The rest of the program requires no additional changes to use this collocated notification channel.

25.4.9 Real-Time Notification Example

In this section, we'll discuss the modifications to the basic messenger example from 25.4.2 that are necessary to use the RT CORBA features with the Notification Service. We present selected snippets of this code with our additions and changes in a bold font. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/NotifyService/RTNotify/`.

25.4.9.1 Notification Server Configuration

The notification server needs to be started with the RT CORBA and RT Notification libraries loaded. We accomplish this be placing these directives:

```
dynamic TAO_RT_ORB_Loader Service_Object *TAO_RTCORBA:_make_TAO_RT_ORB_Loader ()
"-ORBPriorityMapping continuous"

dynamic TAO_Notify_Service Service_Object *
TAO_RT_Notification:_make_TAO_RT_Notify_Service () ""
```

in the `notify.conf` file and launching the server with this configuration file:

```
$TAO_ROOT/orbsvcs/Notify_Service/tao_cosnotification -ORBSvcConf notify.conf
```

This dynamically loads the RT Notification library and enables the RT CORBA-related features in the notification service.

25.4.9.2 Messenger Server Changes

The Messenger Server process acts as our event supplier and we modify it here to allocate a thread pool with lanes for our consumer proxy and to set the RT CORBA priority for event publication.

First, we need to set the `ThreadPoolLanes` property on the supplier admin. This is done before creating the consumer proxy:

```
CosNotifyChannelAdmin::SupplierAdmin_var supplier_admin =
  ec->new_for_suppliers (ifgop, adminid);

NotifyExt::ThreadPoolLanesParams tpl_params;

tpl_params.priority_model = NotifyExt::CLIENT_PROPAGATED;
tpl_params.server_priority = DEFAULT_PRIORITY;
tpl_params.stacksize = 0;
tpl_params.allow_borrowing = 0;
```

```
tpl_params.allow_request_buffering = 0;
tpl_params.max_buffered_requests = 0;
tpl_params.max_request_buffer_size = 0;
tpl_params.lanes.length (2);
tpl_params.lanes[0].lane_priority   = LOW_PRIORITY;
tpl_params.lanes[0].static_threads  = 2;
tpl_params.lanes[0].dynamic_threads = 0;
tpl_params.lanes[1].lane_priority   = HIGH_PRIORITY;
tpl_params.lanes[1].static_threads  = 2;
tpl_params.lanes[1].dynamic_threads = 0;
CosNotification::QoSProperties qos;
qos.length(1);
qos[0].name = CORBA::string_dup (NotifyExt::ThreadPoolLanes);
qos[0].value <<= tpl_params;

supplier_admin->set_qos(qos);

CosNotifyChannelAdmin::ProxyID supplieradmin_proxy_id;

CosNotifyChannelAdmin::ProxyConsumer_var proxy_consumer =
  supplier_admin->obtain_notification_push_consumer(
    CosNotifyChannelAdmin::STRUCTURED_EVENT,
    supplieradmin_proxy_id);
```

This causes the supplier admin to create a POA with the Priority Model and Thread Pool policies that is used to activate any subsequently created consumer proxies. When we subsequently use our consumer proxy to begin publishing events they are placed into the correct lanes of the thread pool based on the priority of the incoming `push_structured_event()` operation. Here are the modifications related to setting the priority for event publication:

```
CORBA::Object_var current_obj =
  this->orb_->resolve_initial_references ("RTCurrent");
RTCORBA::Current_var current = RTCORBA::Current::_narrow (current_obj.in ());

current_->the_priority(HIGH_PRIORITY);

// Set up the events for publication...
consumer_proxy_->push_structured_event(event);
```

Obtaining and narrowing the `RTCurrent` object can be done once for the process and stored for use in all threads. The stored `RTCurrent` object reference can then be used to set the priority for each thread where push operations are called.

25.4.9.3 Messenger Consumer Changes

The consumer side changes involve setting up a thread pool for the supplier proxy and setting up the proper RT CORBA policies for the consumer object's activation.

Setting up the thread pool for the supplier proxy is analogous to what we saw above for the supplier-side:

```
CosNotifyComm::StructuredPushConsumer_var consumer =
  CosNotifyComm::StructuredPushConsumer::_narrow (consumer_obj.in ());

NotifyExt::ThreadPoolLanesParams tpl_params;

tpl_params.priority_model = NotifyExt::CLIENT_PROPAGATED;
tpl_params.server_priority = DEFAULT_PRIORITY;
tpl_params.stacksize = 0;
tpl_params.allow_borrowing = 0;
tpl_params.allow_request_buffering = 0;
tpl_params.max_buffered_requests = 0;
tpl_params.max_request_buffer_size = 0;
tpl_params.lanes.length (2);
tpl_params.lanes[0].lane_priority   = LOW_PRIORITY;
tpl_params.lanes[0].static_threads  = 2;
tpl_params.lanes[0].dynamic_threads = 0;
tpl_params.lanes[1].lane_priority   = HIGH_PRIORITY;
tpl_params.lanes[1].static_threads  = 2;
tpl_params.lanes[1].dynamic_threads = 0;
CosNotification::QoSProperties qos;
qos.length(1);
qos[0].name = CORBA::string_dup (NotifyExt::ThreadPoolLanes);
qos[0].value <<= tpl_params;

consumer_admin->set_qos(qos);

CosNotifyChannelAdmin::ProxyID consumeradmin_proxy_id;

CosNotifyChannelAdmin::ProxySupplier_var proxy_supplier =
  consumer_admin->obtain_notification_push_supplier(
     CosNotifyChannelAdmin::STRUCTURED_EVENT,
     consumeradmin_proxy_id);
```

In order for our consumer to be enabled for RT CORBA and honor the priorities that the events carry through the event channel, we need to create the necessary RT CORBA policies, construct an RT POA with them, and then use this POA to activate our consumer. The code below illustrates this for our example.

```
CORBA::Object_var poa_object =
  orb->resolve_initial_references("RootPOA");

PortableServer::POA_var poa =
  PortableServer::POA::_narrow (poa_object.in());

CORBA::Object_var rtorb_obj = orb->resolve_initial_references ("RTORB");
RTCORBA::RTORB_var rt_orb = RTCORBA::RTORB::_narrow (rtorb_obj.in ());

// Create an RT POA with a lane at the given priority.
CORBA::Policy_var priority_model_policy =
  rt_orb->create_priority_model_policy (RTCORBA::CLIENT_PROPAGATED,
                                        DEFAULT_PRIORITY);

RTCORBA::ThreadpoolLanes lanes (2);
lanes.length (2);

lanes[0].lane_priority   = LOW_PRIORITY;
lanes[0].static_threads  = 2;
lanes[0].dynamic_threads = 0;
lanes[1].lane_priority   = HIGH_PRIORITY;
lanes[1].static_threads  = 2;
lanes[1].dynamic_threads = 0;

// Create a thread-pool.
CORBA::ULong stacksize = 0;
CORBA::Boolean allow_request_buffering = false;
CORBA::ULong max_buffered_requests = false;
CORBA::ULong max_request_buffer_size = false;
CORBA::Boolean allow_borrowing = false;

// Create the thread-pool.
RTCORBA::ThreadpoolId threadpool_id =
  rt_orb->create_threadpool_with_lanes (stacksize,
                                        lanes,
                                        allow_borrowing,
                                        allow_request_buffering,
                                        max_buffered_requests,
                                        max_request_buffer_size);

// Create a thread-pool policy.
CORBA::Policy_var lanes_policy =
  rt_orb->create_threadpool_policy (threadpool_id);

CORBA::PolicyList poa_policy_list(2);
poa_policy_list.length (2);
poa_policy_list[0] = priority_model_policy;
poa_policy_list[1] = lanes_policy;
```

```
PortableServer::POAManager_var poa_manager = poa->the_POAManager ();

PortableServer::POA_var rt_poa = poa->create_POA ("RT POA",
                                                  poa_manager.in (),
                                                  poa_policy_list);

PortableServer::Servant_var<StructuredEventConsumer_i>  servant =
  new StructuredEventConsumer_i(orb.in());

PortableServer::ObjectId_var objectId =
  rt_poa->activate_object (servant.in());

CORBA::Object_var consumer_obj =
  rt_poa->id_to_reference (objectId.in ());

CosNotifyComm::StructuredPushConsumer_var consumer =
  CosNotifyComm::StructuredPushConsumer::_narrow (consumer_obj.in ());
```

Now, when our consumer is connected to the event channel, events published through the event channel propagate their priority with them and this priority is used to select the correct lane for processing in the consumer process.

25.5 Compatibility with the Event Service

The Notification Service is backwards compatible with the Event Service. Therefore, existing Event Service applications can interoperate with Notification Service applications. For example, Event Service consumers and suppliers can connect to a Notification Service event channel using the basic IDL interfaces defined by the Event Service.

25.6 tao_cosnotification Command Line Options

The full path of the `tao_cosnotification` server is:

```
$TAO_ROOT/orbsvcs/Notify_Service/tao_cosnotification
```

The `tao_cosnotification` server provides a single notification event channel factory in its own process. By default, it binds the factory to the name `NotifyEventChannelFactory` in the root naming context of the Naming Service. The Naming Service must already be running to use the `tao_cosnotification` server.

The `tao_cosnotification` server accepts various command line options to control aspects of its initialization. Table 25-8 describes these command line options.

Table 25-8 tao_cosnotification Command Line Options

Option	Description	Default
`-?`	Displays the available options.	None
`-Factory` *`factory_name`*	Specifies the name with which to bind the notification channel factory in the root naming context of the Naming Service. The *factory_name* is also used to bind the factory object in the IOR table when -Boot is specified.	`NotifyEventChannelFactory`
`-Boot`	Specifies that *factory_name* should be registered with the IOR table. This allows the use of `corbaloc` style URLs and also TAO's Implementation Repository. (See Chapter 28.)	Disabled
`-NameSvc`	Binds the notification event channel factory with the Naming Service.	Enabled
`-NoNameSvc`	Specifies that the Naming Service should not be used.	Disabled
`-IORoutput` *`file_name`*	Specifies the name of the file for storing the notification channel factory's IOR as a string.	`stderr`
`-Channel`	Specifies that a notification channel should be created in addition to the notification channel factory.	Disabled
`-ChannelName` *`channel_name`*	Specifies the name with which to bind the notification channel, if created, in the root naming context of the Naming Service.	`NotifyEventChannel`
`-Notify_TPReactor` *`nthreads`* `-RunThreads` *`nthreads`*	Specifies the number of worker threads in the thread pool for TP reactor. The ORB should be instructed to use the TP reactor via the service configurator. Note: The `Notify_TPReactor` option is deprecated and is only included for backwards compatibility	1

Table 25-8 tao_cosnotification Command Line Options

Option	Description	Default
`-LoggingInterval` *`interval`*	Enables the Notification Service to use the ACE Logging Service. The interval is in seconds and is used for a timer that the logging service needs. The ACE logging service must be configured via a separate directive in the `svc.conf` file.	No logger timer is registered; ACE logging service can't be used by the notification service.
`-Timeout` *`time`*	Applies the round trip timeout policy to the ORBs the Notification Service is using.	No timeout. Calls block indefinitely.
`-UseSeparateDispatchingORB` *`(0 | 1)`*	Causes the NS to use a separate ORB for dispatching events to consumers. This helps alleviate “back pressure” issues for some performance-intensive applications.	The same ORB is used to process CORBA requests and dispatch events.

25.7 Notification Service Configuration Options

To allow the Notification Service to be used in a wide variety of situations, its behavior can be configured via three separate factories using the service configurator. See 16.3 for a general discussion of service configurator usage. The majority of the notification service configuration options, especially threading controls, are available via the Notify Service Factory as discussed in 25.7.1. The topology persistence feature is configured via the Connection Reliability Factory (see 25.7.4) and the event persistence feature is controlled via the Event Reliability Factory (see 25.7.5).

25.7.1 Notify Service Factory Options

The options specified for the notify service factory allow the Notification Service to use multiple threads and other asynchronous techniques to reduce the coupling between the Notification Service and its clients. These options are applied via the service configurator to the Notify Service Factory. This factory was previously called `Notify_Default_Event_Manager_Objects_Factory`, but has now been mercifully renamed to `TAO_CosNotify_Service`. You can still use both names in your service configuration files, but we recommend using the latter and use it exclusively in the ensuing examples. The options described in this

section can also be passed via the RT Notify Service Factory when it is used to load the RT CORBA Notification features as described in 25.3.8.

These options appear in the following line in the service configurator file (e.g., `svc.conf`):

```
static TAO_CosNotify_Service "options"
```

25.7.2 Threading Options

By default, the notification channel processes events immediately, or reactively. The thread that processes the incoming publication of a `push()` operation is also used for all processing of that event, including subsequent `push()` invocations on the consumers. Immediate processing results in less overhead, but it results in coupling between clients as the behavior of one client can have an adverse effect on other clients. A slow consumer delays delivery of the same event to other consumers and may block publication of other events by the supplier. Various options below add additional thread breaks in the processing pipeline by adding thread pools at different locations.

A notification channel server can operate in one of two basic threading modes. This mode changes the behavior of some of the following options. By default, different proxies (consumer and supplier) share the same server threading resources in the *shared resource model*. By contrast, when the `-AllocateTaskPerProxy` option is used, we are using the *task per proxy model* where each proxy is allocated its own threading resources.

There are two "branch points" during the processing of an event, the Source point and the Dispatch point. At each of these branch points, the event can be processed immediately (on the current thread) or it can be queued for later processing by one thread from a pool of threads. Use the `-DispatchingThreads` option to specify the number of threads to allocate at the Dispatch point and the `-SourceThreads` option to specify the number of threads at the Source point. When separate source threads are used, the ORB thread processing a supplier's `push()` request limits itself to queueing the event for processing on a source thread. When dispatching threads are used, the outgoing `push()` invocation on the consumer occurs on the dispatching thread. All other processing is done before the dispatching thread break (by either the source or ORB thread). The number of ORB threads allocated can be specified via the `-RunThreads` option as discussed in 25.6.

When using the task per proxy model, the specified number of source threads are created for each consumer proxy (or supplier) and the specified number of dispatching threads are created for each supplier proxy (or consumer).

The options specified via the service configurator act as the global defaults for all channels and proxies in a process but can be overridden for individual proxies via the `ThreadPool` and `ThreadPoolLanes` properties as discussed in 25.3.5.4 and 25.3.8.

Previous versions of the Notification Service allowed for the number of threads to be specified at two additional branch points, Lookup and Listener Evaluation. These extra branch points were previously honored as separate points but are now collapsed into the existing branch points. The Lookup Evaluation point has been merged into the Source point and the Listener Evaluation point has been merged into the Dispatch point. The corresponding `-ListenerThreads` and `-LookupThreads` options have been deprecated and should no longer be used. If specified, these options result in a warning message and the threads specified may be added to the remaining branch points. Listener threads are always added to the dispatch thread pool. When the task per proxy model is used, lookup threads are added to the source thread pool. Otherwise, lookup threads are ignored.

Originally, it was necessary to enable multithreading at one of these branch points, then specify the number of threads available. This has been streamlined so that specifying the number of threads automatically enables multithreading. Because of this change the following options, are deprecated, but still recognized and ignored: `-MTDispatching`, `-MTSourceEval`, `-MTLookup`, and `-MTListenerEval`. You should no longer use these options as they now result in warnings and will eventually be removed.

The `-AsynchUpdates` option, if specified, creates a separate thread responsible for delivering subscription information to clients. Unless you are using subscription information to control what information is generated by suppliers based on what information the consumer subscriptions, you can ignore this option.

Option	Section	Description
`-AllocateTaskPerProxy`	25.7.3.1	Specifies that the Notification Service should allocate thread resources on a per proxy basis.

Option	Section	Description
`-AsynchUpdates`	25.7.3.3	Causes subscription and publication updates to be sent asynchronously.
`-DispatchingThreads nthreads`	25.7.3.6	Specifies the number of dispatching threads to use.
`-ListenerThreads nthreads`	25.7.3.7	Deprecated
`-LookupThreads nthreads`	25.7.3.8	Deprecated
`-MTDispatching`	25.7.3.9	Deprecated
`-MTListenerEval`	25.7.3.10	Deprecated
`-MTLookup`	25.7.3.11	Deprecated
`-MTSourceEval`	25.7.3.12	Deprecated
`-SourceThreads nthreads`	25.7.3.14	Specify the number of threads to allocate for processing incoming requests.

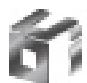
25.7.3 Other Options

The `-AllowReconnect` option controls whether consumer and supplier proxies allow subsequent connect calls to proxies that are already connected.

The `-DefaultConsumerAdminFilterOp` and `-DefaultSupplierAdminFilterOp` options allow developers to override the inter-group filter operator that the default admin objects are created with. This operator is used to determine whether the proxy and admin filter sets are evaluated.

The `-UseSeparateDispatchingORB` option specifies that the notification channel should use a separate ORB for dispatching events to consumers. This option is useful for avoiding the nested upcalls in busy notification channels.

The `-ValidateClient` option causes the notification service to validate clients' liveliness. Each consumer and supplier is checked for activity since the last liveliness check. If the client has been active it is marked as alive, if not a "ping" is done on the object reference. If the "ping" fails, the consumer or supplier is disconnected. The `-ValidateClientInterval` controls the



interval between validation passes. The `-ValidateClientDelay` allows for a delay, before the initial validation pass.

Option	Section	Description
`-AllowReconnect`	25.7.3.2	Allows consumers and suppliers to reconnect to existing proxies.
`-DefaultConsumerAdminFilterOp` *(AND | OR)*	25.7.3.4	Specifies the inter-group filter operator for the default consumer admin object.
`-DefaultSupplierAdminFilterOp` *(AND | OR)*	25.7.3.5	Specifies the inter-group filter operator for the default supplier admin object.
`-NoUpdates`	25.7.3.13	Globally disables subscription and publication updates.
`-UseSeparateDispatchingORB` *(0 | 1)*	25.7.3.15	Use a separate ORB to dispatch events to consumers.
`-ValidateClient`	25.7.3.16	Enable validation of clients' liveliness and removal of dead clients.
`-ValidateClientDelay` *seconds*	25.7.3.17	Specifies delay in seconds before the initial client liveliness validation.
`-ValidateClientInterval` *seconds*	25.7.3.18	Specifies the interval in seconds between client validation passes.

25.7.3.1 AllocateTaskPerProxy

Description This option specifies that the Notification Service should allocate thread resources (i.e., for dispatching and source processing) on a per proxy basis. Using this option affects the behavior of the `-SourceThreads` and `-DispatchingThreads` options as well as some of the QoS Properties.

Usage The notify service factory provides options to enable multithreading and allocate threads for event dispatching and source processing. By default, the requested number of threads is allocated for each consumer and supplier admin object in the notification channel. Different consumer and supplier proxies that are created in that admin object share the same pool of threads.

For example, multithreaded event dispatching is enabled by the `-DispatchingThreads` option (see 25.7.3.6) which also specifies the number of threads allocated for dispatching. By default, the number of threads used for dispatching is whatever is supplied to the `-DispatchingThreads` option for each consumer admin object, regardless of the number of proxies. Use the `-AllocateTaskPerProxy` option to specify that the specified number of threads should be allocated for each supplier proxy. Similarly, this option also alters the source processing threads from being allocated on a per-proxy basis to a per-admin basis.

Impact Allocating threads on a per-proxy basis consumes more resources than allocating them on a per-admin basis. Large numbers of consumers and suppliers can cause creation of excessive numbers of threads.

Events are queued within the notification channel on a per-task basis. This means that when this option is not specified, all consumers share the same event queue and any queue-based QoS properties (`DiscardPolicy`, `OrderPolicy`, `MaxEventsPerConsumer`, `BlockingPolicy`) function on a per-admin basis. When this option is specified, each proxy applies these QoS properties independently to their own queue.

See Also 25.7.3.6, 25.7.3.14

Example

```
static TAO_CosNotify_Service "-AllocateTaskPerProxy -DispatchingThreads 5"
```

25.7.3.2 AllowReconnect

Description This option enables the non-standard behavior necessary to allow clients to reconnect to an existing proxy in the notification service. It should be enabled if topology persistence is configured.

Impact According to the OMG specification for the notification service, a client that attempts to connect to a proxy that is already in use should receive a `CosEventChannelAdmin::AlreadyConnected` exception. Unfortunately, this prevents recovery from certain types of failure. When this option is enabled, clients can reconnect to an existing proxy.

Setting this option may allow a rogue client to “steal” a connection that is being used by another client. This should be taken into consideration in designing a reliable system.

Example

```
static TAO_CosNotify_Service "-AllowReconnect"
```

25.7.3.3 AsynchUpdates

Description This option specifies that the Notification Service should send subscription updates to suppliers and publication (offer) updates to consumers asynchronously. If this option is used, a separate thread will be allocated for dispatching these subscription/publication updates.

Usage Use the `-AsynchUpdates` option to enable asynchronous sending of subscription and publication updates. The number of threads used for sending these updates defaults to one and is not configurable.

Impact When subscription or publication changes occur, the Notification Service dispatches updates to registered listeners. By default, these updates are performed on the same thread as that on which the subscription or publication changes occur. Using a separate thread for sending these updates decouples consumers making subscription changes from suppliers receiving subscription updates, and decouples suppliers making publication changes from consumers receiving publication updates. Using a separate thread for sending subscription or publication updates can improve performance. The allocation of a separate thread for these updates consumes additional resources.

See Also 25.4.4, 25.7.3.13

Example

```
static TAO_CosNotify_Service "-AsynchUpdates"
```

25.7.3.4 DefaultConsumerAdminFilterOp *op_type*

Values for *op_type*

`OR` (default)	Use the OR operator as the inter-group filter operator of the default consumer admin. This means that events published to the consumers in the default consumer admin only need to pass either the admin's or proxy's filters.
`AND`	Use the OR operator as the inter-group filter operator of the default consumer admin. This means that events published to the consumers in the default consumer admin need to pass both the admin's and proxy's filters.

Description For most consumer admin objects, the inter-group filter operator is set when the object is constructed. The default consumer admin is implicitly created by the event channel and is by default given the OR operator. This option allows application developers to override this default.

Usage Using this option to set the AND operator tends to give the semantics that most users expect. The current TAO default of OR is being kept for backward compatibility reasons.

See Also 25.7.3.5

Example

```
static TAO_CosNotify_Service "-DefaultConsumerAdminFilterOp AND"
```

25.7.3.5 DefaultSupplierAdminFilterOp *op_type*

Values for *op_type*

`OR (default)`	Use the OR operator as the inter-group filter operator of the default supplier admin. This means that events published by the suppliers in the default supplier admin only need to pass either the admin's or proxy's filters.
`AND`	Use the OR operator as the inter-group filter operator of the default supplier admin. This means that events published by the suppliers in the default supplier admin need to pass both the admin's and proxy's filters.

Description For most supplier admin objects, the inter-group filter operator is set when the object is constructed. The default supplier admin is implicitly created by the event channel and is by default given the OR operator. This option allows application developers to override this default.

Usage Using this option to set the AND operator tends to give the semantics that most users expect. The current TAO default of OR is being kept for backward compatibility reasons.

See Also 25.7.3.4

Example

```
static TAO_CosNotify_Service "-DefaultSupplierAdminFilterOp AND"
```

25.7.3.6 DispatchingThreads *nthreads*

Description This option specifies the number of threads to be created and used for multithreaded dispatching of events.

Usage Using this option gives you the benefit of separating the dispatching thread from the processing thread. This separation allows for greater decoupling between suppliers and consumers. If this option is not used, then the same thread used to process the event is also used to dispatch it to the consumers. By default, the number of threads specified for this option are allocated as a pool for each consumer admin object. When used with the `-AllocateTaskPerProxy` option, the specified number of threads is allocated per supplier proxy, which is effectively per consumer.

Impact Specifying additional dispatching threads results in increased decoupling between consumers and suppliers. It may also increase throughput for heavy loads. However, additional dispatching threads consume more resources and may introduce higher event latencies due to context switches.

See Also 25.7.3.1

Example

```
static TAO_CosNotify_Service "-DispatchingThreads 5"
```

25.7.3.7 ListenerThreads *nthreads [DEPRECATED]*

Description This option is deprecated.

This option specifies the number of threads to be created and used for *listener* (proxy supplier) filter evaluation.

Usage This activity now occurs on the dispatching threads and the threads specified by this option are now simply treated as additional dispatching threads. You should no longer use this option and should simply define your total desired dispatching threads directly using `-DispatchingThreads`.

Impact See 25.7.3.6 for dispatching threads impact.

See Also 25.7.3.6

Example

```
static TAO_CosNotify_Service "-ListenerThreads 5"
```

25.7.3.8 LookupThreads *nthreads [DEPRECATED]*

Description This option is deprecated.

This option specifies the number of threads to be created and used for subscription lookup.

Usage This activity now occurs on the source thread. If `-AllocateTaskPerProxy` is specified, then this option is ignored. If `-AllocateTaskPerProxy` is not specified, then the threads specified by this option are simply treated as additional source processing threads. You should no longer use this option and should simply define your total desired source processing threads directly using `-SourceThreads`.

Impact See 25.7.3.14 for source processing threads impact

See Also 25.7.3.14

Example

```
static TAO_CosNotify_Service "-LookupThreads 5"
```

25.7.3.9 MTDispatching *[DEPRECATED]*

Description This option is deprecated.

This option was previously required to force the notification channel to use a separate dispatching thread pool. This functionality is now invoked implicitly when using the `-DispatchingThreads` option.

Usage Use the `-DispatchingThreads` option to specify the number of dispatching threads and to force multithreaded dispatching.

Impact See 25.7.3.6 for dispatching threads impact.

See Also 25.7.3.6

Example This example forces multithreaded dispatching with a pool of one dispatching thread.

```
static TAO_CosNotify_Service "-DispatchingThreads 1"
```

25.7.3.10 MTListenerEval *[DEPRECATED]*

Description This option is deprecated.

This option was previously required to force the notification channel to use a separate listener evaluation thread pool. This functionality is no longer allocated its own thread pool.

Usage Use the `-SourceThreads` and `-DispatchingThreads` options to allocate the desired thread pools.

Impact N/A

See Also 25.7.3.6, 25.7.3.14

Example N/A

25.7.3.11 MTLookup *[DEPRECATED]*

Description This option is deprecated.

This option was previously required to force the notification channel to use a separate listener evaluation thread pool. This functionality is no longer allocated its own thread pool.

Usage Use the `-SourceThreads` and `-DispatchingThreads` options to allocate the desired thread pools.

Impact N/A

See Also 25.7.3.6, 25.7.3.14

Example N/A

25.7.3.12 MTSourceEval *[DEPRECATED]*

Description This option is deprecated.

This option was previously required to force the notification channel to use a separate source evaluation thread pool. This functionality is now invoked implicitly when using the `-SourceThreads` option.

Usage Use the `-SourceThreads` option to specify the number of source evaluation threads and to force multithreaded processing.

Impact See 25.7.3.14 for dispatching threads impact.

See Also 25.7.3.14

Example

```
static TAO_CosNotify_Service "-SourceThreads"
```

25.7.3.13 NoUpdates

Description This option globally disables subscription and publication updates.

Usage When specified, this option causes all subscription and publication updates to be disabled. Any consumers or suppliers subscribing to these updates will never receive any.

Impact Consumers and suppliers that depend on subscription and publication updates may not function correctly. There should be a small decrease in processor resource usage when this option is enabled.

See Also 25.7.3.3

Example

```
static TAO_CosNotify_Service "-NoUpdates"
```

25.7.3.14 SourceThreads *nthreads*

Description This option specifies the number of threads to be created and used for multithreaded source processing of events.

Usage Using this option gives you the benefit of separating the source processing thread from the ORB's thread that is used to process an incoming `push()` operation. This separation allows for greater decoupling between suppliers and consumers. If this option is not used, then the same thread used to process the incoming event is also used for consumer proxy filter evaluation and fan out to the receiving supplier proxies. By default, the number of threads specified for this option are allocated as a pool for each supplier admin object. When used with the `-AllocateTaskPerProxy` option, the specified number of threads is allocated per consumer proxy, which is effectively per supplier.

Impact Specifying additional source processing threads results in increased decoupling between consumers and suppliers. It may also increase throughput for heavy loads. However, additional source processing threads consume more resources and may introduce higher event latencies due to context switches.

See Also 25.7.3.1

Example

```
static TAO_CosNotify_Service "-SourceThreads 5"
```

25.7.3.15 UseSeparateDispatchingORB *enabled*

Values for *enabled*

`0` (default)	Use the same ORB to process CORBA requests and dispatch events to consumers.
`1`	Create a second ORB and use it exclusively for dispatching events to consumers.

Description Normally, the notification service uses the same ORB for processing incoming CORBA requests (such as push calls from suppliers) as it does for dispatching (push calls to consumers). Because of the way TAO processes nested upcalls (see 15.4.4 for details), heavily loaded servers can in some situations become deeply nested and exhibit a variety of undesirable characteristics. These include application deadlocks, excessive memory usage, and excessive event latencies.

Enabling this option, causes the use of a separate ORB for event dispatching. Because this ORB has no activated objects in it, there should never be nested upcalls on it. This helps avoid many of the above characteristics.

Usage Use this option when you see indications of excessive nested upcalls in your notification channel server.

Impact There is a small increase in resource usage when two ORBs are being used.

Example

```
static TAO_CosNotify_Service "-UseSeparateDispatchingORB 1"
```

25.7.3.16 ValidateClient

Description This option causes the notification channel to validate the liveliness of client objects (consumers and suppliers). If the client has been active since the last validation, it is marked as alive. If it has not been active, its object reference is "pinged" using a call to `_non_existent()`. Failure to contact the client causes it to be disconnected from the notification channel.

The period between validation passes is controlled by the `-ValidateClientInterval` option. The default value for that option is zero, which indicates that only a single validation pass will occur.

Usage This option should be used to prevent "dead" clients from accumulating in the notification channel and degrading performance.

Impact There is a small amount of processing and bandwidth overhead for this feature.

See Also 25.7.3.17, 25.7.3.18

Example

```
static TAO_CosNotify_Service "-ValidateClient"
```

25.7.3.17 ValidateClientDelay *seconds*

Description This option specifies a delay, in seconds, that occurs before the initial start of the validator task that is enabled by the `-ValidateClient` option. The default delay is zero, meaning the validation timer will begin immediately after the notification channel factory is created.

Usage A long delay is useful when the notification channel is repopulating its topology from a persistent store.

See Also 25.7.3.16, 25.7.3.18

Example

```
static TAO_CosNotify_Service "-ValidateClient -ValidateClientDelay 30"
```

25.7.3.18 ValidateClientInterval *seconds*

Description This option specifies the interval in seconds between the validation passes that are enabled by the `-ValidateClient` option. The default for this option is zero, which indicates that only one pass will be made. A non-zero value is used as the interval between the end of one pass and the beginning of the next.

Usage Use this option to enable periodic validation of the notification service's clients.

See Also 25.7.3.16, 25.7.3.17

Example

```
static TAO_CosNotify_Service "-ValidateClient -ValidateClientInterval 10"
```

25.7.4 Connection Reliability: Topology Persistence Options

To support setting the QoS property `ConnectionReliabilty` to `Persistent` (as discussed in 25.3.6), we must specify the `Topology_Factory` that the notification channel uses. The only topology factory currently supported is the XML topology factory which is located in the `TAO_CosNotification_Persist` library. We typically load this library and factory using a dynamic directive in the service configurator file. Here is the general form of this directive (which should all appear on one line of the configuration file):

```
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist:_make_XML_Topology_Factory() "options"
```

This dynamically loads the XML Topology Factory as the `Topology_Factory` and configures it with the specified options. The rest of this section documents the various options that this factory allows.

Application developers are also free to write their own topology factories, but that is left as an exercise to the reader.

Option	Section	Description
`-v`	25.7.4.6	Enable verbose logging for topology factory configuration.
`-base_path`	25.7.4.2	Specifies base path to use for saving and loading topology files.
`-save_base_path`	25.7.4.5	Specifies a specific base path to use for saving topology files.
`-load_base_path`	25.7.4.3	Specifies a specific base path to use for loading topology files.
`-backup_count`	25.7.4.1	Specifies the number of topology backup files to keep.
`-no_timestamp`	25.7.4.4	Suppresses the writing of a time stamp in the topology file.

25.7.4.1 backup_count

Description This option specifies how many previous versions of the XML file will be retained.

Usage This option is not required. The default value is 1.

The default value, 1, means that only the *file_path*`.000` file will be kept. If a higher number is specified, then older versions will be kept. The older versions will be named *file_path*`.001`, *file_path*`.002`, and so on.

Impact Under normal circumstances, only one backup file is required; setting this number to a larger value lets the system keep a brief history of topology changes. Since the XML files are roughly human-readable, this can be used as a diagnostic tool for problems related to Notification Service topology persistence.

See Also 25.7.4.2

Example

```
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist:_make_XML_Topology_Factory() "-base_path
/safe/topology -backup_count 2"
```

25.7.4.2 base_path *file_path*

Description This option determines where persistent topology information is loaded from and stored.

Usage The argument for this option is a fully qualified path name without an extension. Three extensions will be appended to this path to create three file names: `.new`, `.xml`, and `.000`.

This option is not required. Its default value is `./Notification_Service_Topology`.

Impact Saved topology information will be written to *file_path*`.new` file. Information in a topology persistence file with a `.new` extension is not necessarily complete and will not be used to restore the topology.

When the *file_path*`.new` file is complete, the previous *file_path*`.000` file (if any) will be deleted, the previous *file_path*`.xml` file (if any) will be renamed to *file_path*`.000` and the *file_path*`.new` file will be renamed to *file_path*`.xml`.

The assumption is that a file system rename operation is atomic. If this assumption holds, then at any time the file *file_path*`.xml` (if it exists) contains the most recent complete save. If *file_path*`.xml` does not exist, then *file_path*`.000` contains the most recent complete save. If neither of these files exist, the saved topology information is not available.

See Also 25.7.4.1, 25.7.4.5, 25.7.4.3

Example

```
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist:_make_XML_Topology_Factory() "-base_path
/safe/topology"
```

25.7.4.3 load_base_path *file_path*

Description This option specifies the base path to be used for the files that load the topology state. When used with the `-store_base_path` option they provide an alternative to the `-base_path` option. They allow the file from which topology information is loaded at Notification Service startup time to be different from the file to which this information is saved as the system runs.

Usage This option is not required. There is no default value.

This option should not be used if the `-base_path` option is used.

If this option is used the `-save_base_path` option should also be used.

Impact This option is mostly used for developer testing. A system administrator may find an interesting use for this option, possibly involving script files that rename the XML files during recovery from a Notification Service failure.

See Also 25.7.4.2, 25.7.4.5

Example

```
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist:_make_XML_Topology_Factory() "-load_base_path
/safe/topology_in -save_base_path /safe/topology_out"
```

25.7.4.4 no_timestamp

Description This option suppresses the time stamp that is normally written to the XML file in which the topology is saved.

Usage Use this option when you don't want the timestamp written such as when you wish to automate comparisons of XML topology persistence files.

Impact The time stamp is for information only and is not needed for correct functioning of the topology persistence.

This option is intended primarily for testing the persistent topology implementation.

Example

```
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist:_make_XML_Topology_Factory() "-no_timestamp"
```

25.7.4.5 save_base_path *file_path*

Description This option specifies the base path to be used for the files that save the topology state. When used with the `-load_base_path` option they provide an alternative to the `-base_path` option. They allow the file from which topology information is loaded at Notification Service startup time to be different from the file to which this information is saved as the system runs.

Usage This option is not required. There is no default value.

This option should not be used if the `-base_path` option is used.

If this option is used the `-load_base_path` option should also be used.

Impact This option is mostly used for developer testing. A system administrator may find an interesting use for this option, possibly involving script files that rename the XML files during recovery from a Notification Service failure.

See Also 25.7.4.2, 25.7.4.3

Example

```
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist:_make_XML_Topology_Factory() "-load_base_path
/safe/topology_in -save_base_path /safe/topology_out"
```

25.7.4.6 v

Description This option enables verbose logging. This option and any option that appears after it will be written to the log (normally the console) as it is processed. This is intended to help diagnose and document the Topology Persistence settings. This output is also enabled when the TAO debug level is greater than zero.

The default is to configure Topology Persistence silently.

Usage This should be the first option for the `Topology_Factory` in the service configurator file so that all options will be displayed.

Example

```
dynamic Topology_Factory Service_Object*
TAO_CosNotification_Persist:_make_XML_Topology_Factory() "-v"
```

25.7.5 Event Reliability: Event Persistence Options

To support setting the QoS property `EventReliability` to `Persistent` (as discussed in 25.3.7), we must specify the `Event_Persistence` factory that the notification channel uses. The only event persistence factory currently supported is the standard event persistence factory and we typically load this factory using a dynamic directive in the service configurator file. Here is the general form of this directive (which should all appear on one line of the configuration file):

```
dynamic Event_Persistence Service_Object*
TAO_CosNotification:_make_Standard_Event_Persistence() "options"
```

This dynamically loads the standard event persistence factory as the `Event_Persistence` factory and configures it with the specified options. The rest of this section documents the various options that this factory allows.

Application developers are also free to write their own event persistence factories, but that is left as an exercise to the reader.

Option	Section	Description
`-v`	25.7.5.3	Enable verbose logging for event persistence factory configuration.
`-file_path`	25.7.5.2	Specifies the file to use for saving and loading event persistence.
`-block_size`	25.7.5.1	Specifies a specific base path to use for saving topology files.

25.7.5.1 block_size

Description This option gives the block size in bytes for the device on which the event reliability file is stored.

Usage This option is not required. The default value is 512.

Impact For both performance and reliability reasons, it is important that the value matches the physical characteristics of the device.

For Windows and for “normal” drives (IDE drives and most SCSI drives) on Linux and UNIX, the correct value is almost always 512.

For special purpose SCSI drives or devices in a RAID array, the correct block size is determined by the values used to format the disk and configure the system. See information provided by your vendor and/or system administrator.

Example

```
dynamic Event_Persistence Service_Object*
TAO_CosNotification:_make_Standard_Event_Persistence() "-file_path /safe/events
-block_size 1024"
```

25.7.5.2 file_path

Description This option gives the completely qualified name for the file in which persistent event information will be stored.

This is a required option. There is no default value.

Impact The file should be configured on a reliable device that supports synchronized writes (i.e., flushing the operating system's write cache). A device that is suitable for storing a reliable database would be appropriate for storing this file. The file will be subject to a relatively high number of small (single block) write requests, but very few, if any, read requests.

If the file does not exist, a new file will be created.

If the file does exist, and if topology is successfully loaded, the events from this file will be reloaded and redelivered automatically.

Example

```
dynamic Event_Persistence Service_Object*
TAO_CosNotification:_make_Standard_Event_Persistence() "-file_path /safe/events"
```

25.7.5.3 v

Description This option enables verbose logging. This option and any option that appears after it will be written to the log (normally the console) as it is processed. This is intended to help diagnose and document the Event Persistence settings. This output is also enabled when the TAO debug level is greater than zero.

The default is to configure Event Persistence silently.

Usage This should be the first option for the `Event_Persistence` factory in the service configurator file so that all options will be displayed.

Example

```
dynamic Event_Persistence Service_Object*
TAO_CosNotification:_make_Standard_Event_Persistence() "-v -file_path events"
```

CHAPTER 26

Interface Repository

26.1 Introduction

The Interface Repository (IFR) is a CORBA service defined by the OMG in Chapter 14 of the CORBA 3.1 specification (OMG Document formal/08-01-04). The IFR manages information related to IDL entities. It is able to maintain relationships between such things as modules, interfaces, and operations, as well as all of the supporting entities, such as typedefs, primitives, and structs. The IFR has its own set of interfaces, used to store, manage, and retrieve information in the repository. See 14.5 and 14.7 of CORBA 3.1 for a full description of the IDL defining the IFR

Why is it important to have a repository of information about IDL interfaces and types? It gives CORBA objects the means for self description. It is similar to Java's Reflection API. Through the IFR, it is possible to obtain an otherwise unknown object reference, and to be able to make invocations on it. This is useful when building applications such as system management or debugging tools, middleware gateways or protocol translators. It is also useful when used in conjunction with, for instance, the Trader service. For example, you may have a number of Objects which offer a "print" capability to deliver

data to a printer, each one tuned to a specific kind of job. You may not wish to compile in a number of similar stubs since your application may be older than the objects with which it wishes to interact. Using the IFR, it is possible to determine which object has the appropriate interface.

26.2 Using the Interface Repository

The IFR maintains an object model representing the contents of interfaces and related IDL components. The IFR may hold the contents of many IDL files, all anchored by a single `CORBA::Repository` object. The repository has an interface that allows for addition of modules, interfaces, exceptions, typedefs, anything that can legally exist outside the context of a module in IDL. The Repository also has a query interface allowing for retrieval of this information. See 26.4 for an example use of this query interface. The full details of the IFR's storage and retrieval interface are beyond the scope of this book (see 14.5 and 14.7 in the CORBA specification).

In addition to the repository, there are interfaces that describe all of the IDL constructs. For instance, `ModuleDef` is used to describe a module, `InterfaceDef` for interfaces. These definition interfaces are all part of the `CORBA` domain, and they all inherit from either `Container`, `Contained`, or both.

The definition of any entity which may contain another entity, such as a `ModuleDef`, or `InterfaceDef`, derives from `Container` (for the sake of brevity, since all of these types are defined in the CORBA module, the prefix "`CORBA::`" will be omitted). A `Container` has an interface which allows searching over the contents of the container, with control over the scope of the search, and the level of detail returned.

The definition of any entity which may be contained by another, which includes all but the repository itself, derives from `Contained`. These interfaces all inherit a common set of attributes including a reference to the enclosing container, a Repository ID string uniquely identifying this entity, and the name given to the entity in the original IDL.

The CORBA specification describes the Interface Repository as a tool for the "management of a collection of related objects' interface definitions." It does not require, however, that all the IFR Objects in a repository be "related." If they are unrelated, or if a repository is used as a long-running catch-all, an

ambiguity could arise. In IDL, interfaces may be forward declared. Starting with CORBA 2.3, it was legal to have the forward declaration of an interface in one IDL file, and its full definition in another. This organization poses a potential problem. Suppose that a Repository is used to hold interfaces from a broad range of objects. It might be possible for the same Repository ID to refer to more than one entity. This might happen as the result of using a forward declaration of an interface in one file, and the full interface in another. Or it might be that modules and prefixes are not sufficiently unique, and two interfaces end up with the same repository Id. In this case, the TAO tools, described below, will simply replace an existing entity with a new one. This choice was made in the interest of interoperability, since it is the behavior chosen by most other ORB vendors who have implemented an IFR.

26.3 TAO's Interface Repository Implementation

TAO's implementation of the Interface Repository consists of two processes. The first, `tao_ifr_service`, is meant to be always available. It houses the Repository objects and responds to queries. The other is the IFR loader, `tao_ifr`, a utility used to interpret IDL files and load the results into the IFR. It also provides other useful maintenance services. The source for both of these applications is located in `$TAO_ROOT/orbsvcs/IFR_Service`.

26.3.1 tao_ifr_service

This is the application supplied with TAO that is host to Repository objects. The `tao_ifr_service` application gives a measure of control over the repository. As with any TAO application, `tao_ifr_service` is subject to all of the ORB configuration options documented in Chapter 17. Table 26-1 lists the additional command line options that are specific to `tao_ifr_service`.

Table 26-1 Interface Repository Service Command Line Options

Option	Description	Default
`-o` *`filename`*	Specify a *filename* for storing the IOR from this server.	IOR is written to `if_repo.ior`
`-p`	Make IFR service persistent using a memory-mapped file.	IFR does not use memory-mapped files.
`-b` *`filename`*	Specify a *filename* to use as backing store with -p option.	`ifr_default_backing_store` is used for the mapping file,

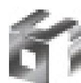


Table 26-1 Interface Repository Service Command Line Options

Option	Description	Default
`-l`	Lock repository during operation invocations.	No locking is used.
`-r`	Use the Windows registry as backing store, if available. Overridden by `-p`.	Do not use the windows registry as a backing store.
`-m (0 | 1)`	Specify whether the IFR Service should listen for multicast service discovery requests. If `-m 1` is used, it listens for multicast requests.	Do not listen for multicast service location requests.

When started, `tao_ifr_service` always writes the Repository IOR to a file. By default, the file named `if_repo.ior` is used. The name of this file may be changed by using the `-o` option. Clients may also locate the IFR service via IP multicast. When multicast support is enabled with `-m 1`, the default port on which the Interface Repository listens for service discovery multicast requests is 10020. This is defined in `$TAO_ROOT/tao/default_ports.h`. The environment variable `InterfaceRepoServicePort` can be used to override this default. In addition, the clients may also locate the service by placing a stringified object reference in the `InterfaceRepositoryIOR` environment variable.

By default, the repository's state is maintained in the process's heap. It is frequently desirable to use persistent storage, allowing the Repository to withstand a process termination and not be populated each time it is started. The `-p` option is used to enable the use of a memory-mapped file for the repository's state. The name of the backing store file may be set using the `-b` option. When run on a Win32 based platform, the tao_ifr_service may use the windows registry as its database, when started with `-r`. Use of the memory-mapped file takes precedence over the registry. In other words, `-p` takes precedence over `-r`.

The final command line option to discuss is the -l option, which activates the use of mutex locks in a multithreaded tao_ifr_service. This is recommended when multiple updates may occur concurrently.

26.3.2 tao_ifr

The `tao_ifr` executable is TAO's interface repository loader and makes the task of populating the IFR as simple as possible. It works by reading IDL files, supplied as command line arguments, then adding (or removing) the entities contained within. It uses the same *front end* and structural code as the TAO IDL compiler, `tao_idl`. The front end is responsible for parsing and processing the IDL files into internal data structures. The differences between the IFR loader and IDL compiler lie in their *back ends*. The `tao_ifr` back end populates the Interface Repository server with the IDL information it processes. Like the IDL compiler, the IFR loader is applied to one or more IDL files. The general form of the command line for `tao_ifr` is:

```
tao_ifr [-ORB options][ifr options] idl_file [idl_file ...]
```

See Chapter 17 for information on the ORB initialization options, all of which are available here. Because they share a common front end, options intended for this code are identical to those of the TAO IDL compiler. Table 26-2 describes all of front end-related command line options that the IFR loader shares with the IDL compiler.

Table 26-2 tao_ifr Front End Options

Option	Description
`-v`	Traces `tao_ifr` processing stages.
`-V`	Prints the version info for tao_ifr and exits.
`-d`	Outputs a dump of the Abstract Symbol Tree (AST).
`-u`	Prints the list of options and exits.
`-E`	Invokes the preprocessor and exits.
`-D`*`macro_def`*	Preprocessor defines *macro_def*
`-U`*`macro_name`*	Preprocessor undefines *macro_name*.
`-I`*`include_path`*	*include_path* is passed to the preprocessor.
`-A`*`assertion`*	*assertion* (a local implementation-specific escape) is passed to the preprocessor.
`-Yp,`*`path`*	Specifies a path to the preprocessor, rather than what was used to build `tao_ifr`.
`-t` *`dir_name`*	Temporary directory to be used by the `tao_ifr` front end. If not specified, then `ACE_DEFAULT_TEMP_DIR_ENV` or `/tmp` is used.

Table 26-2 tao_ifr Front End Options

Option	Description
`-Cw`	Output a warning if two identifiers in the same scope differ in spelling only by case.
`-Ce`	Output an error if two identifiers in the same scope differ only by case. This is the default if `-Cw` is not specified.
`-w`	Suppress warning messages.
`-Wp,arg1,arg2,...`	Passes the arguments to the preprocessor.

These command line options can be broken into two groups. The first group is essentially for debugging; these are `-v`, `-V`, and `-d`. The remainder control the processing of the IDL files. All of the options above are identical to those of the IDL compiler.

Table 26-3 describes all of command line options that are unique to the interface loader.

Table 26-3 tao_ifr Back End Options

Option	Description
`-Si`	Suppress processing of included IDL files.
`-L`	Enable locking at IDL file level.
`-r`	Contents of IDL file(s) are removed from, not added to, the repository.
`-T`	Allows duplicate typedefs in IDL files.

By default, the IFR loader processes all IDL found in the files specified to it and all their included files and adds this information to the repository. The `-Si` option cause the IFR loader to only add information from IDL files specified on the command line and not from included files. The `-r` option causes all processed IDL elements to be removed from the IFR repository. The -T option allows duplicate typedefs to be processed, by ignoring any typedefs after the first one of an identifier. In the absence of the -T option, the second and subsequent typedefs of the same identifier will be reported as an error.

The TAO IFR loader uses the same techniques for obtaining a reference to the Repository as any other CORBA process. It asks the ORB to resolve the initial reference of "`InterfaceRepository`." The ORB will try an initial reference supplied on the command line (see 17.13.36). Use "`InterfaceRepository`" for the Object ID, and use "`file://if_repo.ior`" for the Object URL. Of

course, if `tao_ifr_service` was started with -o *filename*, then use that file name instead. Otherwise, the ORB may locate the IOR via the `InterfaceRepositoryIOR` environment variable or an IP multicast request (assuming the IFR server was started with multicast enabled).

26.4 Example IFR Client

Now we take a look at how an application can use the IFR to obtain information about an interface. This program, called `IFRBrowser`, obtains a reference to the Repository, then traverses all of the entities, writing them to the console. Full source code for this example is in the TAO source code distribution in the directory `$TAO_ROOT/orbsvcs/DevGuideExamples/InterfaceRepo/`.

26.4.1 Annotated Source Code

The following code is from `IFRBrowser.cpp`.

The CORBA specification does not define a location for the IFR interface headers. All of the interfaces this browser uses are defined in `IFR_Basic.idl`, found in `$TAO_ROOT/tao/IFR_Client`. Therefore this browser needs the client side include file shown below. The `Log_Msg.h` include file provides access to the ACE logging facility that provides the `ACE_DEBUG` macro.

```
#include <ace/Log_Msg.h>
#include <tao/IFR_Client/IFR_BasicC.h>
#include <tao/ORB.h>

const char* programLabel = "IFR Browser";
```

The following forward declarations are for methods used in `main()`, but defined further below. They represent the methods used to output certain elements of the IFR.

```
void listContents(const CORBA::ContainedSeq& repoContents);
void listInterface(CORBA::InterfaceDef_ptr interfaceDef);
void listOperation(CORBA::OperationDescription* operationDescr);
void listParameter(CORBA::ParameterDescription* parameterDescr);

const char* decodeTypeCode(const CORBA::TypeCode_ptr typeCode);
const char* decodeParameterMode(CORBA::ParameterMode mode);
```

```
const char* decodeOperationMode(CORBA::OperationMode mode);
```

The main function performs all of the common CORBA application initialization, obtains a reference to the Interface Repository, then lets the `listRepo()` function handle the actual work.

```
int main(int argc, char* argv[])
{
  try
  {
    CORBA::ORB_var orb =
      CORBA::ORB_init(argc, argv);

    CORBA::Object_var obj =
      orb->resolve_initial_references("InterfaceRepository");

    CORBA::Repository_var ifrRepo =
      CORBA::Repository::_narrow(obj.in());

    if(CORBA::is_nil(ifrRepo.in())) {
      ACE_DEBUG((LM_ERROR,
              ACE_TEXT("(%N) failed to narrow interface repository reference.\n")
                 ));
      return -1;
    }
```

The Repository may hold many elements. The operation `contents()` retrieves all of these repository-level elements in a single sequence. The description `kind` value `CORBA::dk_all` indicates that all elements are to be included in the list. The kind value may be used to constrain the list to only particular kinds of elements, such as typedefs or interfaces.

```
    CORBA::ContainedSeq_var repoContents =
      ifrRepo->contents(CORBA::dk_all,1);

    ACE_DEBUG((LM_INFO,
             ACE_TEXT("%s: the interface repository contains %d elements.\n"),
             programLabel,
             repoContents.length()
            ));
    listContents(repoContents.in());

    return 0;
  }
  catch(CORBA::Exception& ex) {
    ex._tao_print_exception("Main block");
  }
```

```
  return 1;
}
```

The `listContents()` function iterates over the sequence of contained objects, some of which may in turn contain other objects. The Repository may directly contain anything that can be defined outside the context of a module, which traditionally includes constants, type definitions, exceptions, interfaces, valuetypes, value boxes, and modules. With the addition of the CORBA Component Model (CCM) this can also include things such as components and homes. This example only supports constants, typedefs, exceptions, interfaces, and modules. It ignores all other constructs.

```
void listContents(const CORBA::ContainedSeq& repoContents)
{
  //
  // List the contents of each element.
  //
  for(unsigned int i = 0; i < repoContents.length(); ++i) {
    switch(((repoContents[i])->describe())->kind) {
    case CORBA::dk_Constant:
      ACE_DEBUG((LM_INFO,
                 ACE_TEXT("%s: element[%d] is a constant definition.\n"),
                 programLabel,
                 i + 1
                ));
      break;
    case CORBA::dk_Typedef:
      ACE_DEBUG((LM_INFO,
                 ACE_TEXT("%s: element[%d] is a typedef definition.\n"),
                 programLabel,
                 i + 1
                ));
      break;
    case CORBA::dk_Exception:
      ACE_DEBUG((LM_INFO,
                 ACE_TEXT("%s: element[%d] is an exception definition.\n"),
                 programLabel,
                 i + 1
                ));
      break;
    case CORBA::dk_Interface:
      {
        ACE_DEBUG((LM_INFO,
                   ACE_TEXT("%s: element[%d] is an interface definition.\n")
                   ACE_TEXT("%s: listing element[%d]...\n"),
                   programLabel,
                   i + 1,
```

```
                 programLabel,
                 i + 1
                ));
```

When the kind of element encountered is an interface, `IFRBrowser` narrows the `contained` reference to an `InterfaceDef`, which is the definition of an interface. Then, `listInterface()` is used to output the particulars for the interface.

```
        CORBA::InterfaceDef_var interfaceDef =
          CORBA::InterfaceDef::_narrow(repoContents[i].in());
        listInterface(interfaceDef.in());
        break;
      }
    case CORBA::dk_Module:{
      ACE_DEBUG((LM_INFO,
                 ACE_TEXT("%s: element[%d] is a module definition.\n"),
                 programLabel,
                 i + 1
                ));
```

As with the interface description, a module may also contain other elements. No special function is needed for a module as the interface `ModuleDef` is nothing more than the empty inheritance of `Container` and `Contained`. The `Contained` interface has all the necessary identity information access.

```
        CORBA::ModuleDef_var moduleDef =
          CORBA::ModuleDef::_narrow(repoContents[i].in());
        CORBA::ContainedSeq_var moduleContents =
          moduleDef->contents(CORBA::dk_all,1);
        CORBA::String_var moduleId = moduleDef->id();
        CORBA::String_var moduleName = moduleDef->name();
        ACE_DEBUG((LM_INFO,
             ACE_TEXT("%s: \n  // %s\n  module %s\n{\n")
             ACE_TEXT("%s: the module contains %d elements.\n"),
             programLabel,
             moduleId.in(),
             moduleName.in(),
             programLabel,
             repoContents.length(),
            ));
        listContents(moduleContents.in());
      break;
     }
    default:
      break;
    }
```

```
  }
}
```

When an interface is encountered above, `listInterface()` is used to output all of the particulars. Note below, that a specialized interface is used to obtain the identity information. This differs from the `ModuleDef`, which relies on the inherited `Contained` interface for identity. Since `InterfaceDef` also inherits from `Contained`, the name and repository Id could be accessed through the `InterfaceDef` directly. The `FullInterfaceDescription` provides access to the basic identity as well as to other interface-specific information, thus providing “one stop shopping” for all of the interface’s details.

```
void listInterface(CORBA::InterfaceDef_ptr interfaceDef)
{
  CORBA::InterfaceDef::FullInterfaceDescription_var fullDescr =
    interfaceDef->describe_interface();

  const char* interfaceName =
    fullDescr->name;
  const char* interfaceRepoId =
    fullDescr->id;

  ACE_DEBUG((LM_INFO,
             ACE_TEXT("%s:\n\t// %s\n\tinterface %s\n\t{"),
             programLabel,
             interfaceRepoId,
             interfaceName
            ));


  unsigned int operationsCount;
  if((operationsCount =
      fullDescr->operations.length()
      ) > 0
     )
  {
    for(unsigned int i = 0; i < operationsCount; ++i) {
      listOperation(&(fullDescr->operations[i]));
    }
  }

  unsigned int attributesCount;
  if((attributesCount =
      fullDescr->attributes.length()
      ) > 0
     )
```

```
  {
    ACE_DEBUG((LM_INFO,
               ACE_TEXT("%s: %s has %d attribute(s).\n"),
               programLabel,
               interfaceName,
               attributesCount
             ));
  }

  ACE_DEBUG((LM_INFO, "\n\t}\n"));
}
```

For each operation in an interface, the `listOperation()` method lists the repository Id, name, return type, and all of the parameters.

```
void listOperation(CORBA::OperationDescription* operationDescr)
{
  const char* operationName =
    operationDescr->name;
  const char* operationRepoId =
    operationDescr->id;
  const char* operationResult =
    decodeTypeCode(operationDescr->result.in());
  const char* operationMode =
    decodeOperationMode(operationDescr->mode);

  ACE_DEBUG((LM_INFO,
             ACE_TEXT("\n\t\t// %s \n\t\t%s %s %s"),
             operationRepoId,
             operationResult,
             operationName,
             operationMode
            ));

  CORBA::ParDescriptionSeq* params =
    &(operationDescr->parameters);
  CORBA::ULong paramsCount = params->length();
  if (paramsCount > 0) {
    ACE_DEBUG((LM_INFO, "\n\t\t(\n\t\t"));
    for (CORBA::ULong i =0; i < paramsCount; ++i) {
      listParameter(&((*params)[i]));
      if(i < (paramsCount - 1)) {
        ACE_DEBUG((LM_INFO, ",\n\t\t"));
      }
    }
    ACE_DEBUG((LM_INFO, "\n\t\t);\n"));
  }
  else {
    ACE_DEBUG((LM_INFO, "();\n"));
```

```
  }
}
```

Finally, each parameter to an operation is listed, one per line, based on the contents of the supplied parameter description. This information could also be used to populate an argument list on a dynamic `request` object, if DII were used to interact with an object implementing this interface.

```
void listParameter(CORBA::ParameterDescription* parameterDescr)
{
  const char* typCode =
    decodeTypeCode(parameterDescr->type.in());
  const char* paramMode =
    decodeParameterMode(parameterDescr->mode);
  ACE_DEBUG((LM_INFO,
             ACE_TEXT("%s %s %s"),
             paramMode,
             typCode,
             parameterDescr->name.in()
            ));
}
```

The remaining helper functions are used to return strings based on various codes.

```
const char* decodeTypeCode(const CORBA::TypeCode_ptr typeCode)
{
  const char* code;
  if(typeCode->equivalent(CORBA::_tc_void))
    code = "void";
  else if(typeCode->equivalent(CORBA::_tc_boolean))
    code = "boolean";
  else if(typeCode->equivalent(CORBA::_tc_string))
    code = "string";
  else
    code = "";
  return code;
}

const char* decodeParameterMode(CORBA::ParameterMode mode)
{
  const char* paramMode;
  switch(mode) {
  case CORBA::PARAM_IN:
      paramMode = "in";
      break;
  case CORBA::PARAM_OUT:
```

```
        paramMode = "out";
        break;
    case CORBA::PARAM_INOUT:
        paramMode = "inout";
        break;
    default:
        paramMode = "";
    }
    return paramMode;
}

const char* decodeOperationMode(CORBA::OperationMode mode)
{
    return (mode == CORBA::OP_NORMAL) ? "synchronous" : "asynchronous";
}
```

26.4.2 Run the Example

For this example, we have a simple IDL file, `test.idl`. The implementation of the IDL is not important to this example. However, this IDL file shows several major components of IDL, including a module, an interface and an operation.

```
module warehouse
{
  interface inventory
  {
    boolean getCDinfo (in string artist,
                       inout string title,
                       out float price);
  };
};
```

The following commands show how to start the Interface Repository service, and how to populate the repository with the elements of the IDL file shown above.

```
tao_ifr_service -m 1&

tao_ifr test.idl
```

The first command starts the `tao_ifr_service` with multicast discovery enabled. We will rely on clients using multicast to locate the Interface Repository. The `tao_ifr_service` is normally a long-running process, so this example runs it in the background (assuming some Unix command shell).

When first run, the Repository is empty. The `tao_ifr` command above reads the contents of `test.idl` and populates the Repository. At this point, we run the `IFRBrowser`.

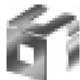
```
IFRBrowser
```

`IFRBrowser` uses multicast to locate the Repository, then iterates over the entire contents of the Repository, writing the results as shown.

```
IFR Browser: the interface repository contains 1 elements
IFR Browser: element[1] is a module definition.
IFR Browser:
// IDL:warehouse:1.0
module warehouse
{
IFR Browser: the module contains 1 elements.
IFR Browser: element[1] is an interface definition.
IFR Browser: listing element[1]...
        // IDL:warehouse/inventory:1.0
        interface inventory
        {
                // IDL:warehouse/inventory/getCDinfo:1.0
                boolean getCDinfo synchronous
                (
                in string artist,
                inout string title,
                out price
                );
         }
}
```

Alternatively, we could run this example using the `InterfaceRepositoryIOR` environment variable. The following code assumes the use of bash (or another similar shell environment):

```
tao_ifr_service -o ifr.ior&

export InterfaceRepositoryIOR=file://ifr.ior

tao_ifr test.idl

IFRBrowser
```



CHAPTER 27

TAO Security

Preface

Support for security in CORBA has been through a number of iterations and resulting specifications. TAO has supported subsets of many of these specifications and provides many tools and interfaces that application developers can use to build and deploy secure applications.

The TAO 1.6a implementation remains in a state of transition. Previous versions of TAO included a partial implementation of CORBA Security and SSLIOP. The implementation in TAO 1.6a is based on newer versions of the CORBA Security specification. Unfortunately, the implementation is incomplete, so some features are not available. Further, some features that were available in previous releases may not be available as the implementation currently exists. Specifically, *Policy Enforcing* applications (see 27.8, “Security Policy Enforcing Application”) will very likely not work properly, though all *Security Unaware* (see *27.6, “Security Unaware Application”)* and many *Policy Changing* (see *27.7, “Security Policy Controlling Application”*) applications will work. The rest of this chapter documents these working (or mostly working) features.

Why is CORBA Security changing? In the original specification for the CORBA Security Service (OMG Document security/00-12-02), the OMG defined a model for security by specifying sets of features required for security. These feature sets were further divided into feature packages (see Table 27-1). The specification also defined IDL interfaces (effectively, APIs) for accessing these various feature sets from within applications.

One of the more critical feature packages was "Common Secure Interoperability," or "CSI," which was supposed to allow different vendors' implementations of security to interoperate, so that a secure CORBA application need not be deployed in an ORB-homogenous environment. Unfortunately, as implementations for CSI emerged and evolved (one of these implementations was in TAO), developers recognized that the original CSI package merely addressed the transport layer and was inadequate for truly interoperable security systems. Thus, the OMG augmented this feature package with CSIv2 (Chapter 10 of CORBA 3.1, Part 2), which layers atop CSIv1 and communicates privileges under which requests are made.

As developers progressed on implementing CSIv2, they also realized that the existing APIs (IDL modules `SecurityLevel1` and `SecurityLevel2`) were insufficient to use with CSIv2. Thus, as one of the principal members of the CSIv2 task force developed a Java implementation of CORBA Security, he concurrently created `SecurityLevel3`, an API which, in the developer's words, is "much more robust...than OMG's Security Level 2 API, and has well defined semantics." We anticipate that `SecurityLevel3` will eventually be pushed into an official OMG specification. Also, recognizing the deficiencies of the Security Levels 1 and 2 APIs, developers of other security implementations for other ORBs began implementing CSIv2 under Security Level 3 APIs. TAO contains one of those implementations.

The TAO 1.6a release remains between these implementations. The newer `SecurityLevel3` module replaces `SecurityLevel1` and `SecurityLevel2`, and there is no design for coexistence. Although many in the TAO community seem to be interested in using security features, few have shown interest in sponsoring research or development activity in that area. That, of course, means that development moves forward on a time-permitting basis from volunteers who already have other sponsored work.

TAO's security implementation will continue to grow and improve, hopefully accelerated by active sponsorship from the community-at-large.

27.1 Introduction

This chapter introduces you to various features of TAO's Security Service:

- The OMG's Security Service Specification.
- Modern cryptographic techniques.
- The *Secure Sockets Layer* (SSL) protocol.
- TAO's fundamental capabilities with respect to distributed object system security.

Since we cannot provide a complete treatment of security in a single chapter, we have not attempted it. Ultimately, you are responsible for determining the security needs of your system, the suitability of TAO's feature set to those needs, and the correctness of the implementations in that context. This chapter provides guidance on how to use many of the features available in TAO, but not all.

We have chosen code examples that are faithful to the specification and to TAO's design objectives. We have chosen not to present examples that demonstrate how to circumvent known issues and problems because such examples would too quickly become irrelevant; for this sort of information we refer you to the release notes included with each source distribution.

27.1.1 Road Map

In this chapter, we explore the topic of security from the perspective of an application with security needs, as well as from the perspective of the features available in TAO's implementation. While the chapter is designed to be consumed as a whole, you may find it beneficial to read certain sections independently.

If you want to learn more about...

- *background information* required to make good decisions about using TAO's security service, Sections 27.2, "Introduction to CORBA Security" and 27.3, "Secure Sockets Layer Protocol" provide overviews on core concepts in CORBA Security and the core technologies in SSL, the security technology used in TAO's implementation.

- *dealing with certificates* required for using SSL, Section 27.4, "Working with Certificates" discusses the pragmatics of managing certificates for use with SSL and TAO's security service.
- *building/compiling* TAO's security service implementation with OpenSSL, look at Section 27.5, "Building ACE and TAO Security Libraries."
- *code that uses CORBA Security features*, Sections 27.6, "Security Unaware Application" and 27.7, "Security Policy Controlling Application" both give simple examples of applications at various levels of participation in security. Full source code for the examples in this chapter can be found in the TAO source code distribution in `$TAO_ROOT/orbsvcs/DevGuideExamples/Security` and subdirectories therein.

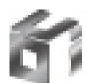
Please note that in addition to this chapter, in order to fully leverage the power of TAO's Security Service implementation, you must understand its foundation, the OMG's Security Service version 1.8 specification (OMG Document formal/02-03-11).

This chapter presents capabilities that allow an application to participate at two levels: *security unaware* and *security policy controlling*. The chapter opens with an introduction to the security service specification. This is followed by an overview of the SSL protocol and a summary of the associated security technology concepts. We then explain how to build the ACE and TAO security libraries, how to configure the ORB's security services, and how to work with certificates used by SSL. The chapter closes with working examples of TAO's security features and a summary of the ORB configuration options related to security.

27.2 Introduction to CORBA Security

The OMG's Security Service version 1.8 specification (OMG Document formal/02-03-11) is a comprehensive treatment of security as it relates to distributed object systems and applications. This specification defines a robust feature set and architecture for the implementation of secure, CORBA-based, distributed object systems. The specification:

- Summarizes potential threats faced by a distributed object system.



- Identifies issues arising from the nature of distributed object systems.
- Describes an abstract *Security Reference Model.*
- Describes an *Implementation Architecture.*
- Defines interfaces for use by object system implementers, application developers, and security administrators.

The specification does not, however, describe any specific security technology. Its purpose is to identify the essential features of a secure distributed object system and describe an architecture that can be integrated with current security technologies, such as the *Secure Sockets Layer Protocol* (SSL), and new security technologies that may emerge in the future.

Allowing application developers to choose the extent to which an application participates in protecting itself is an essential aspect of *CORBA Security*. There are three levels of participation for an application:

1. *None:* an application may rely completely on the enclosing object system for its protection, remaining ignorant of the existence of any security.
2. *Policy configuration:* an application can set security policies that control the protective measures employed by the object system. However, that application is not obligated to take an active role in the enforcement of that policy, deferring enforcement to the object system itself.
3. *Policy configuration and context inspection:* an application can take an active role by setting policy and by using security context information, obtained at run time, to control access to and use of its services.

This section summarizes essential aspects of the specification:

- The security reference model.
- Feature packages.
- The implementation architecture.

Other aspects of the specification, such as the various application developer interfaces, are discussed in the various examples.

The reference model sets the overall context for CORBA Security. Distributed object systems are deployed under a variety of circumstances and in numerous application domains. Consequently, a distributed object technology, such as CORBA, may be called upon to satisfy a broad range of requirements. The security reference model's purpose is to define a feature set that satisfies the potential security needs of distributed object systems without specifying a

feature solution for any specific system. Some features are purely functional, meant to address the needs of secure applications. Other features are architectural in nature, meant to give object system implementers flexibility when selecting security technology and implementing security services.

The features described by the reference model are organized into a number of feature packages. Object system implementers balance development costs, system capability, and feature compliance by choosing among the various feature packages.

The implementation architecture describes a means of realizing the reference model in a way that will satisfy the needs of various user communities including:

- End users.
- Application developers.
- System administrators and security managers.
- Object system implementers.

The architecture defines the responsibilities of the various components that must cooperatively realize the features set forth by the reference model and establishes the structural boundaries that separate these components.

27.2.1 CORBA Security Reference Model

The reference model defines a set of features that may be needed by an arbitrary distributed object system. No single set of features is appropriate for all systems. Instead, the feature set summarized here forms a collection of resources from which a system can choose based on its needs.

27.2.1.1 “Principal” Identification and Authentication

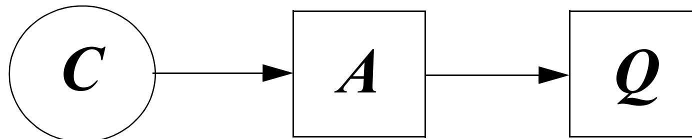
Each human user or independently operating software entity, referred to as a *principal*, that attempts to gain access to or use a secure system shall first establish the right to do so (the CORBA Security concept of *principal* is completely different from the GIOP 1.0 *principal*; we only speak of the CORBA Security concept in this chapter). A principal establishes their right to use a secure system by presenting an identity, a claim of “who they are,” and evidence to justify their claim, proof that “they are who they claim to be.” Each principal has at least one identity but may have multiple identities.

For example, a principal may have an identity used to gain access to the system, called an *access identity*, and a separate identity to represent that same principal in system audit records, called an *audit identity*. A principal may have other privileges that further quantify their rights within the system. Identities are referred to as *identity attributes*. Privileges are referred to as *privilege attributes*.

Various mechanisms may be used to identify and authenticate principals. The mechanism used by a particular system may be an integral component but this is not necessarily the case. A secure system may rely on external mechanisms, such as operating system login procedures, to identify and authenticate principals. Most importantly, the model does not define any specific means of identifying and authenticating a principal, and leaves this to be defined by other parties, such as the designer for a specific implementation of CORBA Security, or even an additional OMG specification.

27.2.1.2 Delegation of Attributes

Consider the scenario where object *A*, in the course of executing an operation on behalf of client *C*, needs to invoke an operation on another object *Q*. In an



unsecure system, this is neither unusual, nor does it pose a problem. However, in a secured system where *C->A* is a secure invocation, there is a question regarding which credentials *A* should use when performing its invocation on *Q*.

A target object (such as *A*) acting as an intermediary between a principal (such as *C*) that initiates an operation, called an *initiating* or *originating principal,* and another object (such as *Q*) may present its own identity and privilege attributes or it may present those of the initiating principal. In the first case, the intermediary object assumes the role of originating principal with respect to the new target object, somewhat akin to how a deputy temporarily assumes the authority of a sheriff. In the second case, the intermediary object actually

mediates between the originating principal and the new target object. Security policies enforced at the new target are based on the identity and privilege attributes presented by the intermediate object.

Simple delegation allows any intermediate object to present the originating principal's identity and privilege attributes to subsequent targets without restriction. An intermediate object can effectively impersonate the originating principal. Other forms of delegation allow the originating principal to control the extent to which their identity and privilege attributes can be delegated by intermediate objects.

27.2.1.3 Non-Repudiation

A user of a secure system, including a human user, a software component acting on behalf of a human user, and a software component acting independently, shall be prevented from denying their actions within the system. A secure system may generate and record irrefutable evidence of principals' activities to prevent any principal from denying their actions. This capability requires application level participation and so cannot be provided to security unaware applications.

27.2.1.4 Auditing

Users of a secure system shall be accountable for their actions. Secure systems make and keep records of actions that are relevant to system security. At the system level, these events may include authentication of a principal, a change in security policy, an operation invocation, and so forth. Application level events may also be recorded depending upon an application's needs. Event records support system security audits by associating each security relevant event with an initiating principal. Audit policies may be needed to restrict the volume of recorded events because the number of security relevant events may be quite large.

27.2.1.5 Transparent Protection

A secure system shall automatically protect all applications, according to applicable security policy, including those applications that choose to remain unaware of security policies and their enforcement. Applications need not take an active role in the specification or enforcement of security policies. Moreover, applications shall not accidentally circumvent security policy through ordinary use of system services. For example, an application should

not be able to circumvent security policy, intentionally or unintentionally, merely by invoking an operation via an object reference.

27.2.1.6 Application Control of Security Policy

An application may alter security policy. A secure system shall enforce default security policies unless and until those policies are modified by an application. Subsequently, a secure system shall enforce the security policies specified by the application. An application that modifies security policy is not, however, obligated to enforce such policy. The burden of policy enforcement lies with the object system. Of course, modifying a security policy constitutes use of the system. Therefore, any application attempting to modify security policy must have the authority to do so.

27.2.1.7 Application Enforcement of Security Policy

Applications may take an active part in the enforcement of security policy. This is particularly true if the object system lacks the mechanisms to enforce a desired policy. A principal's identity and privilege attributes, represented by a *credential*, are accessible, at the target, during the course of an invocation. An application may use a principal's attributes, obtained from the credential, to manage that principal's use of the system.

27.2.1.8 Secure Messaging

Principals invoking operations from remote systems shall be subject to identification and authentication. During a secure invocation, secure object systems first establish a secure association between the client and target. This association includes sufficient context information to enable security policy enforcement, throughout the invocation, by client and server ORBs. Secure systems shall therefore provide a means for identifying and authenticating any principal invoking an operation from a remote system. Moreover, messages exchanged between systems within the context of a secure association may be protected from unauthorized disclosure and tampering.

27.2.1.9 Administration

A secure system shall provide a means for administering security policies. Objects are organized into domains for the purpose of defining and administering security policies. Policies are defined with respect to a domain.

Polices are applied to objects based upon the domain or domains of which they are a member.

27.2.2 CORBA Security Feature Packages

Note *This section refers to the now-obsolete SecurityLevel1 and SecurityLevel2 modules (see “Preface” on page -961), but the information contained herein remains relevant to the discussion of the general features demanded by the CORBA Security Specification.*

As mentioned in 27.2.1, no single set of features is appropriate for all systems, just as there’s not a single set of features available on a particular model car. The specification permits leeway for an implementation of CORBA Security to provide only those features of the reference model (described in 27.2.1.1 through 27.2.1.9) it deems relevant to its target audience. However, for convenience, the specification defines certain “feature packages” with common groupings of reference model features. Feature packaging permits implementations with varying degrees of conformance to the specification.

Table 27-1 CORBA Security Feature Package Quick Reference

Package Type	Package Name	In TAO?	What’s this?	Section
Main	Level 1	✔	Simple, basic security options, such as secure communication. Defined in `Security` and `SecurityLevel1`.	27.2.2.1
	Level 2	partial	Advanced control on security. Defined in `SecurityAdmin` and `SecurityLevel2`.	
Optional	Non-Repudiation	✘	Supports generation of non-repudiation evidence. Defined in `NRModule`.	27.2.2.2
Security Replaceability	ORB Services	partial	Security implementation provided via interceptors and not in the ORB Core.	27.2.2.3
	Security Services	✘	Security implementation provided external to ORB, but not via interceptors.	

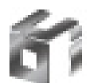


Table 27-1 CORBA Security Feature Package Quick Reference

Package Type	Package Name	In TAO?	What's this?	Section
Common Secure Interoperability	CSI 0	✔	Security policies based on identity only. No privileges attributes transmitted or support for delegation.	27.2.2.4
	CSI 1	✔	Security policies based on identity only. No privilege attributes transmitted. Delegation optional, but unavailable in TAO.	
	CSI 2	✘	Security policies based on identity *and* privileges. Privilege attributes transmitted. Delegation required, and is application-controlled.	
Interoperability	SECIOP	✘	ORB supports SECIOP.	27.2.2.5
	SECIOP + DCE-CIOP	✘	ORB supports SECIOP and DCE-CIOP.	27.2.2.6
Security Mechanism	SPKM/GSSAPI	✘	Public-key protocol allowing identity-based policies. No delegation support. Requires SECIOP.	27.2.2.7
	GSS Kerberos	✘	Secret-key protocol allowing identity--based policies. Unrestricted delegation supported. Requires SECIOP.	
	CSI ECMA	✘	Secret- and public-key protocol allowing identity- and privilege-based policies. Controlled delegation supported. Requires SECIOP.	
	SSL	✔	Public-key protocol allowing identity-based policies. No delegation support. Cannot use SECIOP.	

27.2.2.1 Main Functionality Packages

Level 1

This package provides the most basic set of security capabilities. Security features are provided to all applications whether or not they participate in security policy enforcement. Principals are identified and authenticated although not necessarily by the ORB.

The ORB provides for secure communication between client and server. Secure communications features include establishing trust between client and server and protecting messages from disclosure and tampering. Appropriate access controls are applied to all secure invocations.

Interfaces and data types are defined in the `Security` and `SecurityLevel1` modules.

Other capabilities and constraints include:

- An intermediate object can delegate the originating principal's credentials or present its own credentials to subsequent targets.
- Applications participating in security policy enforcement can access principals' credentials via the operation `SecurityLevel1::Current::get_attributes()` (discussed later in this chapter). Attribute values obtained via this operation may then be used to control access to and use of the system, at the invocation level, and to capture information for audit events.
- Level 1 provides control of security policy via the operation `CORBA::Object::set_policy_overrides()` (also discussed later in this chapter).
- Security administration interfaces are omitted at Level 1.
- Other feature packages, including the Optional and Security Replaceability packages may also be supported at Level 1.

Level 2

This package provides additional interfaces for accessing principals' credentials and additional capabilities for security policy control and administration.

Interfaces and data types are defined in the `SecurityAdmin` and `SecurityLevel2` modules in addition to the Level 1 modules.

27.2.2.2 Optional Packages

Non-repudiation

This is the only optional package. This package provides features and interfaces to support the generation of non-repudiation evidence.

Interfaces and data types are defined in the `NRModule`.

Note *TAO does not support non-repudiation, so this package is not discussed in this chapter.*

27.2.2.3 Security Replaceability Packages

There are many ways to integrate a CORBA Security implementation with an ORB. An obvious way is to tightly couple the two such that the ORB implementation does not stand independent of the Security implementation. However, if more flexibility is desired perhaps because the provider of the ORB implementation is different from the provider of the Security implementation, there must be looser coupling.

The specification provides two standard packages to enable looser coupling at different levels. These packages specify whether or not the ORB's security services can be replaced and, if so, the mechanism for incorporating additional or alternative services. The specification does not define *how* the replacement actually occurs, e.g., loading shared libraries, recompilation of source, etc.; these details are left open to the ORB implementer.

The two specified packages are:

- **ORB Services Replaceability**: the ORB invokes security services via specified interceptors and in a specified sequence. The use of other specific interfaces, such as `CORBA::Object::get_policy()`, is required in the interceptors' implementation. This places all security service code outside of the ORB core.
- **Security Services Replaceability**: the ORB invokes security services via specified replaceability interfaces. The use of interceptors is optional. Unlike the ORB Services Replaceability Package, this package allows implementation of security services without specific knowledge of how the ORB works. It is similar to ORB Services Replaceability in that it positions all security service code outside of the ORB core.

27.2.2.4 Common Secure Interoperability (CSI) Packages

These packages classify an ORB's capabilities regarding security policy definition and the delegation of credentials. Classifying capabilities in this manner allows ORBs to make decisions regarding peer capabilities when establishing secure associations and handling secure invocations.

- **CSI 0**: Security policies are based on identity attributes only. Privilege attributes are not transmitted from client to server. Delegation of credentials is not supported.
- **CSI 1**: Security policies are based on identity attributes only. Privilege attributes are not transmitted from client to server. Delegation of credentials, if supported, is unrestricted. A principal cannot control whether or not an intermediary object delegates its credentials nor the extent to which credentials are delegated.
- **CSI 2**: Security policies are based on identity and privilege attributes. Privilege attributes are transmitted from client to server along with identity attributes. Delegation of credentials is supported. A principal can control whether or not an intermediary object delegates its credentials and, when permitted, the extent to which delegation occurs.

27.2.2.5 SECIOP Interoperability Package

This package specifies that the ORB generates and uses security information in IORs and exchanges messages with other ORB's via the SECIOP protocol using GIOP/IIOP (with specified security enhancements).

27.2.2.6 SECIOP + DCE-CIOP Interoperability

An ORB that realizes this package supports SECIOP over GIOP/IIOP and secure communications, based on DCE security services, via the DCE-CIOP protocol. ORBs that support this package meet all requirements for standard secure interoperability.

27.2.2.7 Security Mechanism Packages

- **Simple Public Key GSS-API Mechanism (SPKM) Protocol**. This protocol allows identity based policies, without delegation, using public-key cryptography. (Public key cryptography techniques are

described later in this chapter.) This protocol requires the SECIOP extensions to the IIOP protocol.

- **Generic Security Service (GSS) Kerberos Protocol**. This protocol allows identity-based policies with unrestricted delegation using secret key cryptography. This protocol also requires the SECIOP extensions to IIOP.
- **CSI ECMA Protocol**. This protocol supports identity- and privilege-based security policies with controlled delegation. Secret and public key cryptography techniques are used in this protocol. This protocol also requires the SECIOP extensions to IIOP.
- **Secure Sockets Layer (SSL) Protocol**. SSL allows identity-based policies without delegation. SSL is independent of GIOP/IIOP and so does not require SECIOP extensions to IIOP.

Note *TAO's security features are based on SSL.*

27.2.3 CORBA Security Implementation Architecture

The specification addresses the security architecture in four respects:

1. Basic environmental protection and communications.
2. Application components.
3. ORB Core, ORB Services, and Communications Protocols.
4. Security Technology.

27.2.3.1 Basic Environmental Protection and Communications

Basic environmental protection refers to matters such as the host platform, the operating system, the physical plant, and so forth, that are outside the scope of

any software system. As shown in Figure 27-1, a secure system operates within a processing environment.

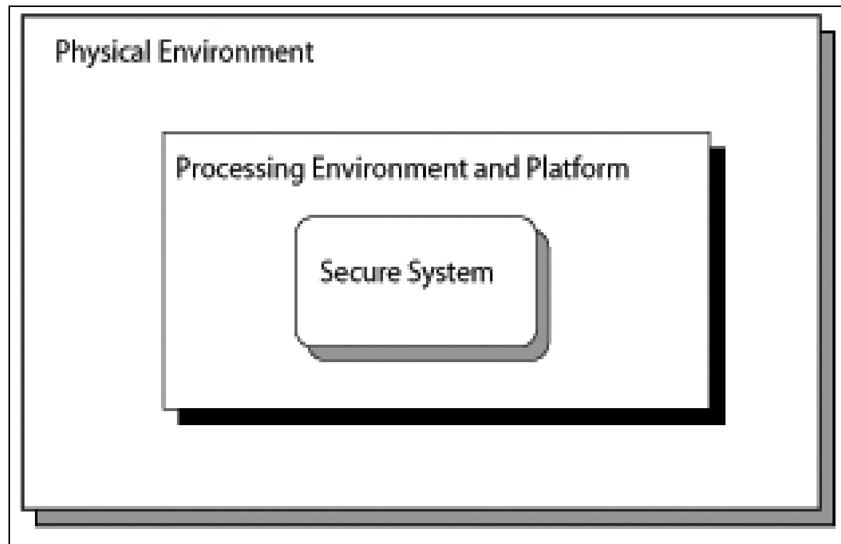
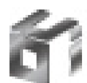


Figure 27-1 System Environments

The processing environment consists of the host, the operating system, and other system and application software. A secure system often depends upon the processing environment's security in one way or another. For example, a secure system may rely on the processing environment to identify and authenticate users by trusting user credentials obtained from or provided by the processing environment.

Every processing environment exists within a physical environment. Different physical environments offer varying degrees of protection. Consider the different levels of protection offered by:

- A street-side kiosk.
- A home office.
- An ordinary office building.
- An office protected by an electronic access control system.
- A secure room protected by armed security guards and multiple electronic access control systems.



A street-side kiosk environment offers little if any protection. An isolated room surrounded by armed guards and multiple access control systems offers a great deal of protection. In each of the examples cited here, the protection provided by the physical environment is an important factor to consider when choosing a processing environment. In turn, the degree of protection afforded by a processing environment is an important factor in the design of a secure software system.

Most distributed systems depend upon interactions between components that are housed by separate processing environments. Many distributed systems involve interactions among multiple components housed by separate processing environments that are located in separate physical environments. Secure communications facilities offer protection from disclosure of and tampering with messages exchanged between systems using those facilities. However, many systems are deployed in circumstances where secure communications facilities are impractical or simply unavailable. Therefore, a secure system may employ cryptographic technology for the express purpose of protecting inter-process communications.

Compromising a secure system may require penetration of the physical and processing environments, penetration of the communications facilities, or both. Furthermore, users who are authorized to enter the physical environment, access the processing environment, and use a system may compromise the system accidentally or perhaps deliberately. Consequently, in addition to protective measures taken by the system itself, action is also required to mitigate threats to the physical and processing environments. The CORBA security architecture is limited to the needs of software systems. It does not address the issues related to processing environments, physical environments, or secure communications facilities.

27.2.3.2 Security Architecture

Figure 27-2 depicts the fundamental relationships among:

- Application layer components.
- The ORB Core.
- ORB Services, including security services (note that “ORB Services” are not equivalent to “CORBA Services”, such as `CosNaming`).
- Communications protocols.

- Security technology components

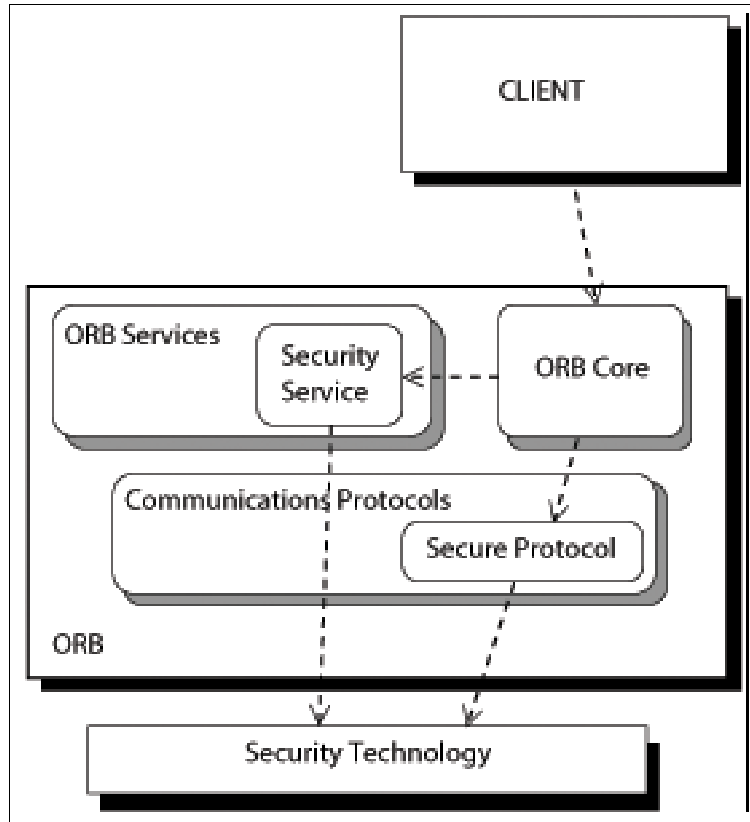


Figure 27-2 Security Architecture

The ORB Core provides the application layer, depicted here by `CLIENT`, with a coherent representation of a distributed object system. From the application's perspective, a client object simply invokes an operation on a target object via an object reference. The ORB Core arranges for communications between the client and the target objects as needed to carry out the requested operation. The ORB Core also has the responsibility to invoke ORB Services as each request is processed. This responsibility includes invoking security services as specified by applicable security policy.

When message protection policy requires message integrity or confidentiality, the ORB uses a secure communications protocol to establish trust with its peers, and to protect messages exchanged with its peers. The separation of the

ORB core, ORB services, and communications protocols is not evident to the application layer. The different policies that apply to an object reference control the way the ORB processes an invocation, but not the way a client makes an invocation.

When a client invokes an operation on an object reference, the client-side ORB selects the security services applied at the client. The server-side ORB selects the security services applied at the target. If the policies or capabilities at the client and target are inconsistent or incompatible, the server may negotiate a different set of capabilities with the client prior to processing the request or may refuse to process the request altogether.

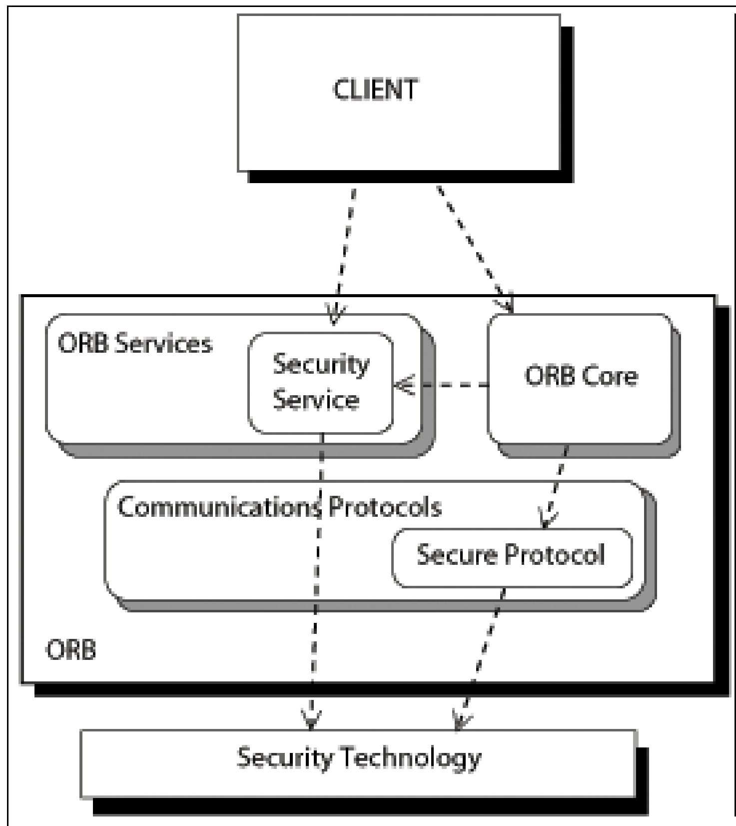


Figure 27-3 Security Aware Application

A security aware application, i.e., an application that controls security policy or participates in the application of security policy, may invoke security services directly as depicted in Figure 27-3.

Security aware applications can alter security policy via `CORBA::Object::set_policy_overrides()` and obtain references to security services via `CORBA::ORB::resolve_initial_references()`.

Note that the security architecture maintains a clear distinction between:

- Security services and the underlying security technology.
- Secure protocols and the supporting security technology.

This allows ORB implementers to choose a security technology most appropriate to their needs and their target audience. It also affords an opportunity for growth as security technology evolves and new security technologies emerge. The specification does, however, address the integration of specific security technology, such as SSL, with the ORB and how that effects the ORB's compliance with the specification.

It is also important to be aware that the architecture does not specify interfaces for applications to access the security technology used by the ORB. This does not prohibit an application's use of such technology. It simply places the responsibility for such interfaces outside the scope of the ORB.

27.2.3.3 Security Context Information

Before a client can invoke an operation on a target object, a security association must be established between the client and the target object through a specific object reference. This association is sometimes referred to as the *binding* between the client and the target as shown in Figure 27-4.The ORB is responsible for creating and managing security associations. A security association's lifetime may not exceed that of the process in which it was created although it may be shorter. Security policy and other environmental factors specify how the ORB creates secure associations.

Context information specific to the client's use of the object reference is associated with the binding and represented at both the client and target. Some of this context information can be accessed at the target via security services

objects such as the `Current` objects. However, this information is not accessible by the client.

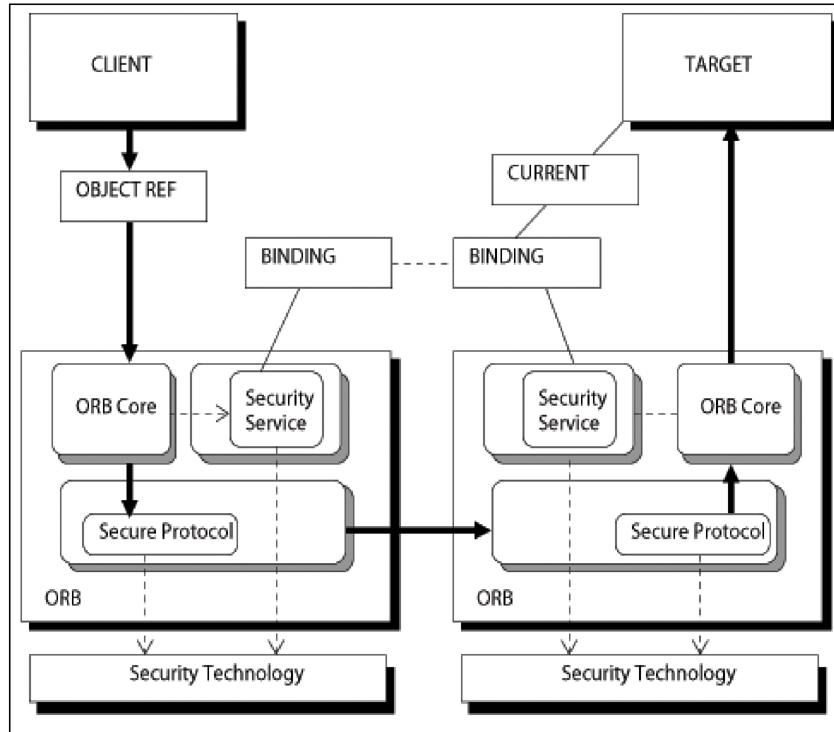


Figure 27-4 Security Binding

27.2.3.4 TAO's Security Service Architecture

Note *Some figures and text in this section refer to SecurityLevel1 and SecurityLevel2 Current objects. For reasons explained in the Preface, these do not exist in TAO 1.6a. However, we leave them in this sections' discussion as the concept remains relevant in SecurityLevel3.*

TAO's security service architecture, depicted in Figure 27-5, is consistent with the OMG's specification.

TAO employs SSL as the underlying security technology. SSLIOP is a secure communications protocol that uses SSL to establish secure associations and to provide message protection.

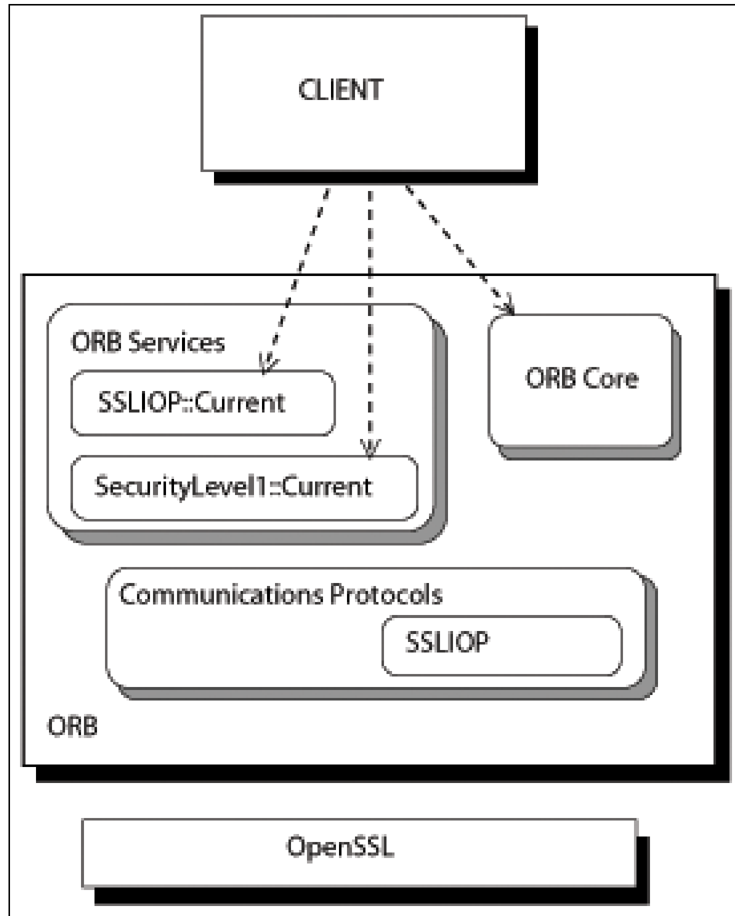


Figure 27-5 TAO's Security Service Architecture

Security context information is made available at the target via several objects:

- `SecurityLevel1::Current`
- `SecurityLevel2::Current`
- `SSLIOP::Current`

References to these objects are obtained via
`CORBA::ORB::resolve_initial_references()`.

Note `SSLIOP::Current` *is a TAO-specific extension unaffected by the Security Level changes.*

Security policies that control the establishment of trust between client and target, and the level of message protection may be set by an application via `CORBA::Object::set_policy_overrides()`.

27.3 Secure Sockets Layer Protocol

SSL supports secure communications between two endpoints, including peer authentication, message integrity, and message confidentiality. *Authentication* guarantees that a peer's identity is genuine. *Message integrity* guarantees that messages cannot be modified in transit without detection. *Message confidentiality* guarantees that only a message's intended recipient can read it.

SSL uses both public-key and secret-key cryptography techniques. Public-key techniques protect information exchanged between client and server to establish a secure session. Secret-key cryptography protects messages exchanged after a secure session is established. Public-key cryptography provides better authentication and key exchange mechanisms than secret-key cryptography, but is more computationally intensive and therefore less efficient. Secret-key cryptography is better suited to bulk message protection.

There are three versions of SSL, referred to as SSLv1, SSLv2, and SSLv3. There is also a closely-related specification, Transport Layer Security (TLS), which is based upon SSLv3 and similar in capability. The difference between SSL and TLS is that TLS is a standard from the Internet Engineering Task Force, while SSL is a *de facto* standard originally defined, published, and implemented by Netscape Corporation. Throughout this chapter, when we refer to SSL we are referring to SSLv3, but the discussions are applicable to TLS as well. OpenSSL, the SSL implementation used by TAO, supports both SSLv3 and TLS.

This section opens with a conceptual introduction to secret-key and public-key cryptography. It goes on to describe SSL's architecture and its various protocol elements. The section concludes with an example SSL session. This

material is introductory only. It is not a complete treatment of cryptography or SSL. It provides conceptual support for topics covered throughout the remainder of this chapter. If you are already familiar with these topics, you can skip ahead to 27.5.

27.3.1 Secret-Key Cryptography

Secret-key cryptography, sometimes called *symmetric encryption*, is based on a single key that is known by a message's originator and its receiver. As shown in Figure 27-6, a message's originator converts the message from a legible form, called *plaintext*, to an illegible form, called *ciphertext*, using the secret key and a mathematical algorithm. The message's receiver uses the same key and a companion algorithm to convert the ciphertext into the original plaintext. A third party attempting to eavesdrop on the exchange between originator and receiver cannot read the message because it is encrypted.

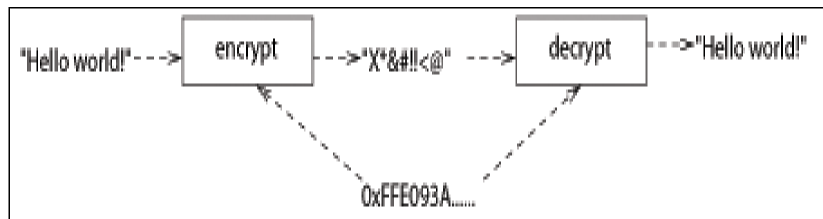
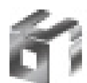


Figure 27-6 Symmetric Encryption/Decryption

Secret-key cryptography provides message confidentiality only. There is no inherent mechanism to detect that a message was modified in transit. Secret-key technology also carries with it the problem of key exchange. A message originator cannot make a message private unless the message's receiver knows the key. Therefore, the key must be exchanged in advance, preferably using a communications medium other than that used to exchange encrypted messages.

Some secure protocols, Kerberos for example, resolve the key exchange problem using a third-party system called a *key server*. A key server knows the secret key of each system that it supports. A message originator first asks the key server for a key and identifies the intended recipient. The key server sends the message originator a *ticket* that contains a new key and a message for the intended recipient. The key server encrypts the enclosed message, which also contains the new key, using the intended recipient's key. The key



server encrypts the ticket using the message originator's key. The message originator now has the new key and a private means of conveying it to the intended recipient.

27.3.2 Public-Key Cryptography

Public-key cryptography uses a different encryption scheme than secret-key cryptography. Public-key cryptography also includes additional mechanisms that provide for message integrity, identity authentication, and key exchange at run time. The following sections describe both the encryption method and additional capabilities offered by public-key cryptography.

27.3.2.1 Asymmetric Encryption

Public-key cryptography is based on *asymmetric encryption*, which uses two related keys called a *key-pair*. As shown in Figure 27-7, data encrypted using one key can only be deciphered using its companion key and vice versa.

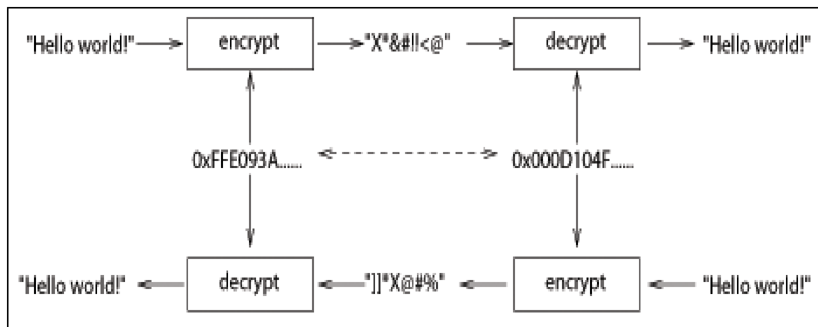


Figure 27-7 Asymmetric Encryption/Decryption

A key-pair's owner makes one key public and keeps the other key private. A message encrypted using a party's *public key* can be decrypted only with that party's *private key*. A message originator can send a private message by encrypting it using the receiver's public key. Only a holder of the receiver's private key can read such a message. A message originator can also encrypt a message using its private key. This does not prevent disclosure of the message, because any party holding the originator's public key can read it, but it deters tampering with the message. Only a holder of the originator's private key can create a message that deciphers correctly using the corresponding public key.

Message Integrity

Message integrity is provided through the use of *message digests*. A message digest is an algorithm that yields a fixed length string from an arbitrary length message. Ideally, a digest would be unique based on the input message, but this is not practical. Therefore, most digest algorithms are devised so the probability of two input messages producing the same digest is very low.

Since more than one input message can yield the same digest, it's not possible to derive the input message from the digest alone. Although not unique to a particular message, a message digest can effectively serve as a message's fingerprint when it is enclosed with the message from which it was generated. A message's receiver can test the message's integrity by computing a digest and comparing it with the digest that was enclosed with the message. If the two digests are identical, the message was received intact. Otherwise, the message was modified in transit and is not trustworthy.

In practice, message digests are used to compute *message authentication codes* and to create *digital signatures*. A message authentication code (MAC) is computed using a message and a key. MACs have the same properties as message digests:

- They cannot be deciphered to discover the original message or the key.
- It is hard to produce two messages that yield the same MAC.
- They can be used to verify a message's integrity.

A digital signature confirms the authenticity of a message's originator. A digital signature is created by encrypting a message's digest (or MAC) using the message originator's private key. The message's receiver first obtains the digest using the originator's public key and then compares it with a locally generated digest. If the two digests are comparable, it proves that the public key's owner sent the message and that the message was received intact. Moreover, the signature is applicable only to a single message; it cannot be used to forge other messages.

Identity Binding and Certificates

Public-key cryptography relies on the association between a key-pair's public and private components to verify identities. A message that is successfully deciphered using a known public key implies that the originator holds the private key, and therefore that the message was originated by the public key's

owner. For this reason, it is crucial to *bind* a key-pair's owner with the owner's identity.

A *certificate* binds a key-pair holder to his public key. To obtain a certificate, an individual presents substantive proof of their identity to a certificate issuing authority. The *certificate authority* (CA), once satisfied of the requestor's identity, creates and signs an electronic document containing:

- The certificate holder's identity.
- The certificate holder's public key.
- The certificate issuer's identity.

Identities are represented by a *distinguished name*, a hierarchical naming structure defined by the X.509 specification that gives each entity a unique identifier. The digital signature affixed to a certificate attests to the issuing authority's identity. A certificate holder offers their certificate as proof of his identity and a means of conveying his public key.

Certificate Authorities

A certificate is accepted as genuine if it is signed by a trusted certificate authority. A certificate is authenticated by obtaining the issuing authority's public key, from the issuing authority's own certificate, and using it to decipher the signature. If the signature is authentic, the indicated authority issued the certificate. If the system trusts that issuing authority, then the certificate is considered authentic and the owner's identity accepted. Otherwise, the issuing authority's certificate is authenticated using the same procedure. This recursive process results in a chain of certificates, all of which are subject to acceptance or rejection.

Certificate authorities vouch for the identity of others. This includes situations where one certificate authority vouches for another. However, as a practical matter, some certificate authorities are not subject to authentication. These authorities vouch for themselves and sign their own certificates. When a *self-signed certificate* is encountered during the authentication process, the process has reached the decision point. If the system trusts the authority that issued the self-signed certificate, it accepts the entire chain of certificates. Otherwise, it rejects them all.

27.3.3 SSL Architecture

SSL, depicted in Figure 27-8, consists of four separate protocols:

- The Handshake Protocol.
- The Change Cipher Spec Protocol.
- The Alert Protocol.
- The Record Protocol.

The Handshake Protocol establishes a secure session between two endpoints. During the process, the handshake protocol:

- Negotiates the protocol version.
- Negotiates the *cipher suite*, the set of cryptographic algorithms used during message exchange.
- Exchanges keys.
- Authenticates peers during the handshake process.

The record protocol processes packets, according to the cipher suite, as they move between upper layer protocols and the transport layer. The alert protocol reports important events such as pending connection closure and processing errors. The change cipher spec protocol signals a change in the current cipher suite.

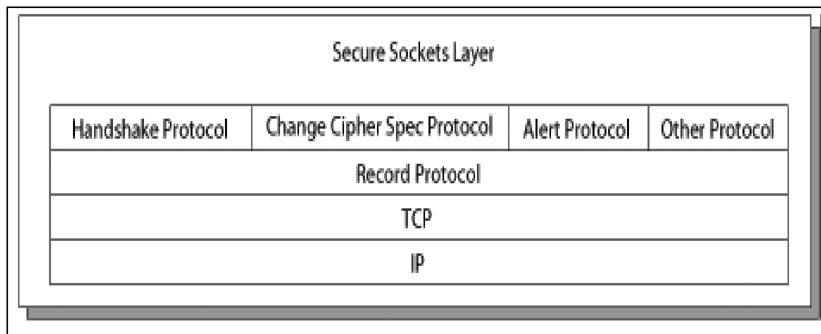


Figure 27-8 SSL Architecture

SSL depends upon a lower level transport protocol to provide reliable message exchange between endpoints. SSL does not have intrinsic support for transport error detection and correction. TCP/IP is the most commonly used transport protocol, although other protocols are sometimes used. TAO has a pluggable protocol called SSLIOP that encapsulates SSL over TCP/IP.

The following sections describe each protocol's role and present an example SSL session.

27.3.3.1 Handshake Protocol

The handshake protocol's role is to:

- Negotiate a protocol version.
- Negotiate a cipher suite.
- Authenticate peers.
- Establish seed data used to generate secret keys.

This section describes essential aspects of the process. The example session presented later contains a detailed sequence diagram.

Client and server negotiations take the form of a client request followed by the server's selection. The client indicates the highest protocol version it is capable of supporting and the server replies with the selected version. The client sends the server a list of the cipher suites it is capable of supporting, listing its preference first on the list, and the server responds with the selected cipher suite. There are no rules that govern the server's selection mechanism; this is left to the implementers.

A cipher suite defines a combination of server authentication method, key exchange method, digest algorithm, and bulk encryption algorithm. Cipher suites are defined by the protocol specification, they are not arbitrary. A session's cipher suite informs the record protocol what protective measures to apply to the packet stream between client and server. It is initially empty, which indicates that no protective measures are applied. However, critical information exchanged during negotiations between client and server is protected using public-key cryptography techniques and therefore confidential even when the cipher suite is empty.

To establish the bulk encryption keys the client and server first exchange two chunks of random data. The client generates a chunk and sends it to the server, and vice versa. Next, the client computes the pre-master secret, a new value based on both random data chunks, and sends it to the server. Client and server then compute the various encryption keys using both chunks of random data and the pre-master secret. The resulting keys do not need to be exchanged since client and server have the same seed data and use the same algorithms.

To protect against certain kinds of attack, client and server exchange MACs that are computed over all messages exchanged during the handshake protocol. This guarantees that none of the messages exchanged during the handshake process can be altered without detection.

27.3.3.2 Record Protocol

The record protocol applies protective measures to packets prior to their transmission and reverses the process for received packets. Figure 27-9 depicts the downstream process, the protection of application packets moving toward the transport layer.

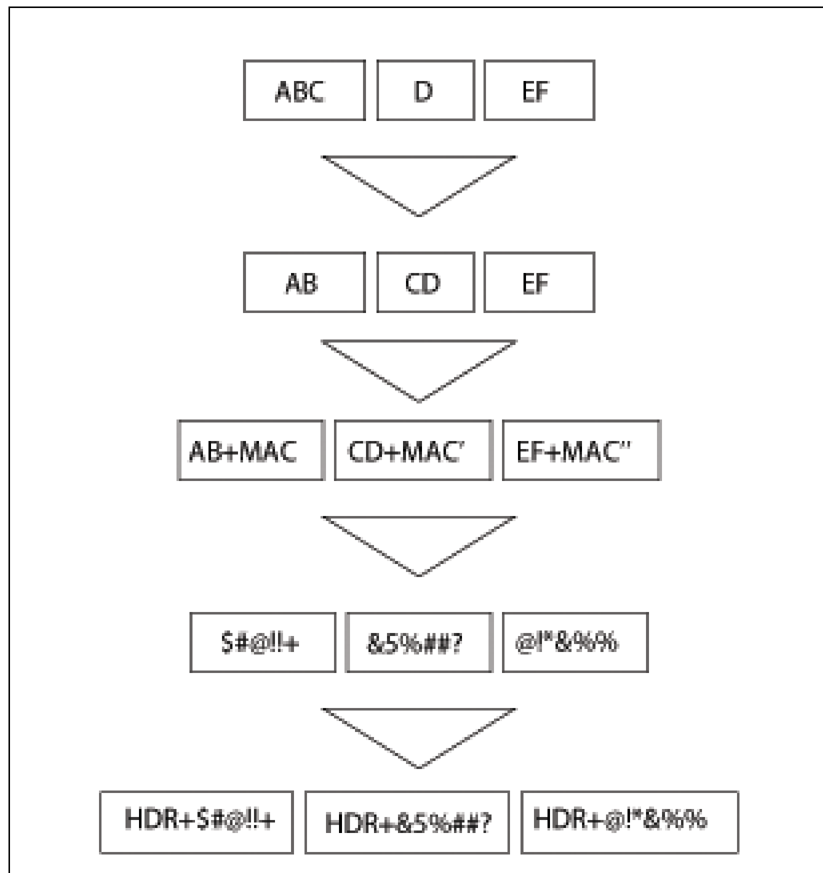


Figure 27-9 Downstream Packet Processing

The protocol first re-organizes application layer packets into limited-size fragments. Packet boundaries are not preserved. A packet may be split into multiple fragments or combined with other packets to form a single fragment. A MAC is computed and appended to each fragment. The fragments are then encrypted, a header is prefixed, and the resulting record is passed to the transport layer.

Records moving up from the transport layer, depicted in Figure 27-10, are decrypted, authenticated, and passed to the appropriate upper layer protocol.

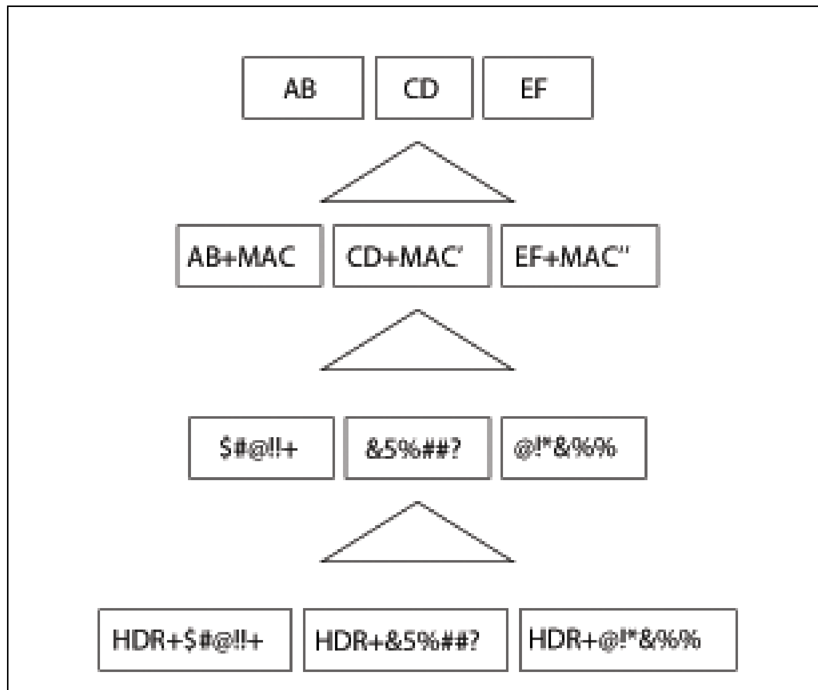


Figure 27-10 Upstream Packet Processing

The record protocol does not differentiate between upper layer protocols with respect to record protection. Protective measures, when in force, are applied to all packets including those associated with one of the other SSL protocols.

The SSL specification allows records to be compressed prior to the encryption step but does not specify any compression algorithms. Compression is not discussed in this chapter because it is not supported in many SSL implementations, although it is supported by OpenSSL.

27.3.3.3 Change Cipher Spec Protocol

The change cipher spec protocol consists of a single message. It is sent during the handshake protocol after the session parameters have been negotiated. It indicates that the sender will apply the protective measures specified by the cipher suite to all subsequent messages.

27.3.3.4 Alert Protocol

The alert protocol serves two purposes. A `close_notify` warning signals its receiver that the sender is closing the connection. All other messages indicate that the sender encountered an error condition.

27.3.3.5 Example Session

Figure 27-11 depicts a typical SSL session:

1. `ClientHello`: the client initiates the handshake protocol by sending the `ClientHello` message. This message indicates the highest protocol version supported by the client, lists the cipher suites supported by the client with the client's preference appearing first, and contains a chunk of random data.
2. `ServerHello`: the server responds to a `ClientHello` with a `ServerHello`. This message contains the server's selections for protocol version and cipher suite. It also contains a chunk of random data.
3. `Certificate` (to client): the `Certificate` message contains a list of ASN1-encoded certificates. The server's certificate appears first. Each subsequent certificate identifies the authority that issued the preceding certificate. The server's certificate is always sent to the client unless the chosen cipher suite does not require it.
4. `ServerHelloDone`: this message indicates that the server has completed this phase of the handshake protocol.
5. `ClientKeyExchange`: this message contains the pre-master secret or, for some cipher suites, data needed to compute the pre-master secret.
6. `ChangeCipherSpec` (to server): this message indicates that all subsequent messages will be protected according to the chosen cipher suite.
7. `Finished` (to server): this message contains a MAC that was computed over all preceding handshake messages.
8. `ChangeCipherSpec` (to client): this message indicates that all subsequent messages will be protected according to the chosen cipher suite.
9. `Finished` (to client): this message contains a MAC that was computed over all preceding handshake messages.

   A secure session is now established between client and server.
10. The client and server exchange application data.
11. `Alert: close_notify`: this message informs the server that the client is closing the connection.

The connection is closed.

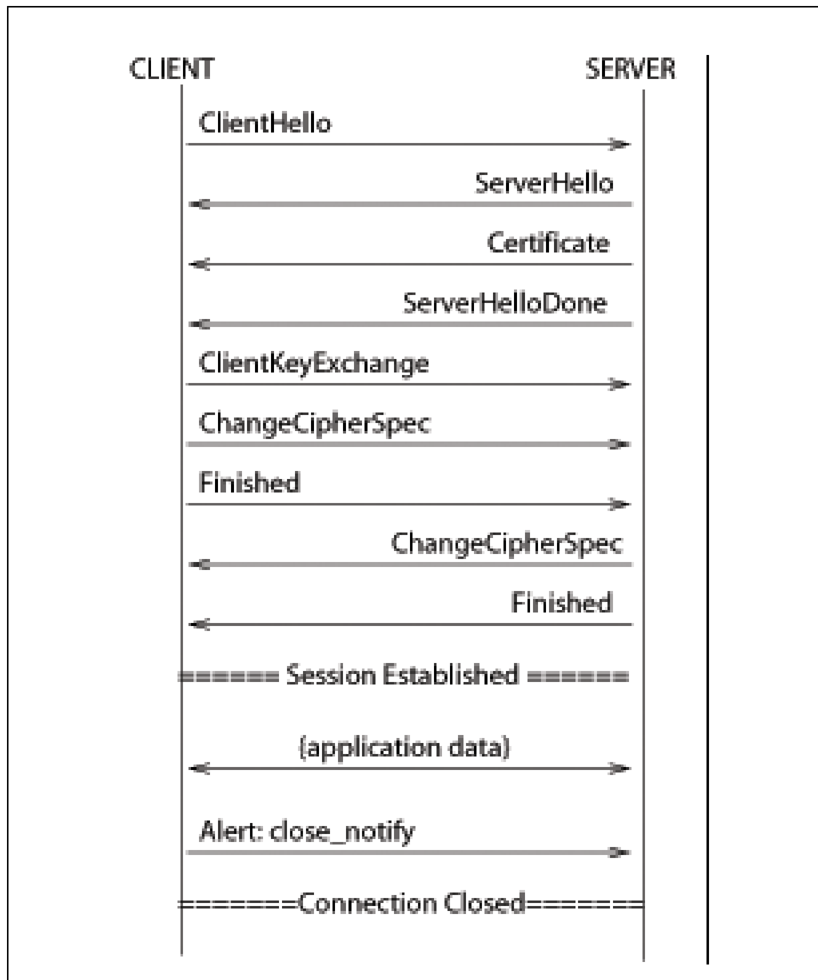


Figure 27-11 Example SSL Session

27.4 Working with Certificates

To develop and test applications that use TAO's security services, you will need to create certificates. OpenSSL includes a tool called `openssl` for creating, manipulating and inspecting keys, certificate requests, and certificates. OpenSSL also includes a Perl script called `CA.pl` to facilitate certain operations, including in particular those operations supporting the creation and management of certificates. Of interest here are the methods for:

- Creating your own certificate authority.
- Generating certificate requests.
- Creating and signing certificates.

Certificates may appear in either ASN1 or *Privacy Enhanced Mail* (PEM) format. `CA.pl`'s default behavior produces certificates in PEM format and all certificates presented in this chapter appear in this format.

Note *Some OpenSSL packages vary the name and/or location of this script slightly. You will have to determine the location of the script in your installation of OpenSSL.*

In the examples that follow:

`user input is presented in this font`

`system output is presented in this font`

27.4.1 Environment Setup

`openssl`, and therefore `CA.pl`, use a configuration file to specify certain options. The path to a suitable configuration file may be passed as an option to many `openssl` commands. The configuration file can also be specified by an environment variable called `OPENSSL_CONF`.

The source distribution includes a default version of the aforementioned configuration file called `openssl.cnf`. The location of this file varies with the installation. On UNIX and UNIX-like platforms, you may find it in the `ssl` directory that appears in the path to `CA.pl`. On Windows, you will find it in the release's `apps` directory.

The examples assume use of the default configuration file and the `OPENSSL_CONF` environment variable.

27.4.2 Create Certificate Authority

To create a new local certificate authority use:

```
$ CA.pl -newca
```

Here is an example:

```
$ CA.pl -newca
CA certificate filename (or enter to create)CR
Making CA certificate ...
Using configuration from /usr/local/ssl/openssl.cnf
Generating a 1024 bit RSA private key
.............................++++++
....................++++++
writing new private key to './demoCA/private/cakey.pem'
Enter PEM pass phrase:taosecurity
Verifying password - Enter PEM pass phrase:taosecurity
-----
You are about to be asked to enter information that will be incorporated into your
certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Missouri
Locality Name (eg, city) []:St. Louis
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Object Computing, Inc.
Organizational Unit Name (eg, section) []:TAO
Common Name (eg, YOUR name) []:John Smith
Email Address []:smith_j@ociweb.com
$
```

This command:

- Builds a local directory hierarchy, `./demoCA`, for use by the other certificate authority commands.
- Creates the CA's private key.
- Stores the CA's private key in `./demoCA/private/cakey.pem`.
- Creates the CA's self-signed certificate.
- Stores the CA's certificate in `./demoCA/cacert.pem`.

During this process, you will be prompted for the CA's identity attributes and a pass phrase to guard use of the CA's certificate. You will be prompted for the CA's pass phrase each time you sign certificate requests.

You are now ready to create a create a certificate request.

27.4.3 Create Certificate Request

To create a new certificate request, use:

```
$ CA.pl -newreq
```

Here is an example:

```
$ CA.pl -newreq
Using configuration from /usr/local/ssl/openssl.cnf
Generating a 1024 bit RSA private key
....++++++
..........++++++
writing new private key to 'newreq.pem'
Enter PEM pass phrase:taosecurity
Verifying password - Enter PEM pass phrase:taosecurity
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Missouri
Locality Name (eg, city) []:St. Louis
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Object Computing, Inc.
Organizational Unit Name (eg, section) []:TAO
Common Name (eg, YOUR name) []:John Smith
Email Address []:smith_j@ociweb.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:corbasecurity
An optional company name []:OCI
Request (and private key) is in newreq.pem
$
```

This command:

- Generates a new private key.

- Generates a new certificate request.
- Stores the certificate request and private key in `newreq.pem`.

During this process, you will be prompted for the user's attributes and a pass phrase to protect the user's certificate. The completed request document will appear similar to the following:

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,CB2F8D7A7D2FFA06

FTVP97A35gWIuTIZctfEqP2JllwdswKttmjI6clPTyfWmTK3439CuXI7OFgX09ME
A1dG2EnFCWiMneVpx/GKan0g5vpefNYvC1O/JjkRtQSUwJO+xgLLdSk7pluuIA6b
uiUVaiCezvBsLYl9g9UWH9pKUsVbrtrICedIK5C+d97ed3A5GEZXEMf8+jU/FM5M
s4A+FT2Uq2SHj47xrTfBPcemYZjoTEUzfxrovevdqbCFLhf9FfMyODRe9y2fQp5g
yK8TGV5miPcPOoDqRTkspbq9Is3yFXXnIdT8JYQxgHpJzyQVed4G3zZYiPhM4kZG
Xy9lA6L+0NZUzRMALNJUqWxj2IIFnkRJ1VBC5meyO10aieqRLylr7JUzTQVA5xf/
YQfcciMugHAU0n4+uLVdehgnLmpolfa+EJh7rETxnUQyQrnrDvOucMYnFDmuYLoy
2SUri9OszuhXzX1P1TzfEcD/LASJpf/NAgHRXDn39D+MXnxAVdmZmY0xPVzmeCDz
MtmjhG++fujjOc9XpEIvvJ7e3Baq3xBVLFn7FhMZQUnrNRHyikCvh39WiE8eJl2I
e1GDLucuVscD4PD5WdMYdMbFnH6YaRFL0XrBETe8hvCn2oEXtUO4QxHARG2fw9if
4skWWcSg5PPejDxEBOQZwQwxO8tnF1BWOzDnkQHvc6HMowlvejvXY9ZC6cwTTnUG
NEcNJwUFRZ8s7ZErVzBOB6NAF4q+UUb6N+EQBcsfHYtyleuDwhulafjkXnXl3uc4
rXLdK9t2ufcWpnyLLZ1g/MZaTyCay6ddXntUs2DueHr0FOmAy8Iq8g==
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE REQUEST-----
MIICDjCCAXcCAQAwgZsxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhNaXNzb3VyaTES
MBAGA1UEBxMJU3QuIExvdWlzMR8wHQYDVQQKExZPYmplY3QgQ29tcHV0aW5nLCBJ
bmMuMQwwCgYDVQQLEwNUQU8xEzARBgNVBAMTCkpvaG4gU21pdGgxITAfBgkqhkiG
9w0BCQEWEnNtaXRoX2pAb2Npd2ViLmNvbTCBnzANBgkqhkiG9w0BAQEFAAOBjQAw
gYkCgYEAqxIMACmYKHipD8GB9PNbARx04Yt0MDm6q7shbU/i1qndGo8yYgYuDcG1
cxF0dm6PLLJzuz5XysuiIGBVzCD86b2qZxPq1yCaSjxgM9YMwAw61K3hxF4hwwWY
EPwBxzzJ9117ioo7yVHovcO2bpCuphhO/X1lUdwXdiGH8giPC3sCAwEAAaAyMBIG
CSqGSIb3DQEJAjEFEwNPQ0kwHAYJKoZIhvcNAQkHMQ8TDWNvcmJhc2VjdXJpdHkw
DQYJKoZIhvcNAQEEBQADgYEAcpg+VrzTdhVUE+buFwAZAc9LwHjs5xade9wU5lWp
sWbixoC6MP+UK9x6BQvjYMTWCEjXGm7IkHr34w6SWhXozTM+bn1AS+0vX2ezXMeU
jXtvWfNdTr2je56Xv9kS3e/CyQPclVGZTJThRJuFB4I2DoZt8PtyclT8AOX9+AcZ
dA4=
-----END CERTIFICATE REQUEST-----
```

The file must remain as is to serve as a certificate request. The file can also serve as a private key in this form. However, the private key portion can be copied into a separate file. (Lines delimiting the private key are required.)

You are now ready to sign the request and create a certificate.

27.4.4 Sign Certificate Request and Issue Certificate

To sign a certificate request and issue a certificate, use:

```
$ CA.pl -sign
```

Here is an example:

```
$ CA.pl -sign
Using configuration from /usr/local/ssl/openssl.cnf
Enter PEM pass phrase:taosecurity
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number: 1 (0x1)
        Validity
            Not Before: Sep  9 23:08:46 2005 GMT
            Not After : Sep  9 23:08:46 2006 GMT
        Subject:
            countryName               = US
            stateOrProvinceName       = Missouri
            localityName              = St. Louis
            organizationName          = Object Computing, Inc.
            organizationalUnitName    = TAO
            commonName                = John Smith
            emailAddress              = smith_j@ociweb.com
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                AB:9E:C0:4C:75:2B:89:C3:2F:4F:D9:71:91:85:D1:A5:C5:D2:67:BB
            X509v3 Authority Key Identifier:
               keyid:BF:8B:5B:ED:36:97:84:40:C8:BA:C2:B1:CD:C4:55:37:8A:55:F1:7D
                DirName:/C=US/ST=Missouri/L=St. Louis/O=Object Computing, Inc.
                    /OU=TAO/CN=John Smith/emailAddress=smith_j@ociweb.com
                serial:A1:D5:25:21:ED:7E:39:33

Certificate is to be certified until Sep  9 23:08:46 2006 GMT (365 days)
Sign the certificate? [y/n]: y


1 out of 1 certificate requests certified, commit? [y/n] y
Write out database with 1 new entries
Data Base Updated
Signed certificate is in newcert.pem
```

This command:

- Processes the certificate request held in `newreq.pem`.
- Generates a new public key and certificate.
- Stores the certificate in the file `./newcert.pem`.

The completed certificate will appear similar to the following:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=US, ST=Missouri, L=St. Louis, O=Object Computing, Inc.,
            OU=TAO, CN=John Smith/emailAddress=smith_j@ociweb.com
        Validity
            Not Before: Sep  9 23:08:46 2005 GMT
            Not After : Sep  9 23:08:46 2006 GMT
        Subject: C=US, ST=Missouri, L=St. Louis, O=Object Computing, Inc.,
            OU=TAO, CN=John Smith/emailAddress=smith_j@ociweb.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:ab:12:0c:00:29:98:28:78:a9:0f:c1:81:f4:f3:
                    5b:01:1c:74:e1:8b:74:30:39:ba:ab:bb:21:6d:4f:
                    e2:d6:a9:dd:1a:8f:32:62:06:2e:0d:c1:b5:73:11:
                    74:76:6e:8f:2c:b2:73:bb:3e:57:ca:cb:a2:20:60:
                    55:cc:20:fc:e9:bd:aa:67:13:ea:d7:20:9a:4a:3c:
                    60:33:d6:0c:c0:0c:3a:d4:ad:e1:c4:5e:21:c3:05:
                    98:10:fc:01:c7:3c:c9:f7:5d:7b:8a:8a:3b:c9:51:
                    e8:bd:c3:b6:6e:90:ae:a6:18:4e:fd:7d:65:51:dc:
                    17:76:21:87:f2:08:8f:0b:7b
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                AB:9E:C0:4C:75:2B:89:C3:2F:4F:D9:71:91:85:D1:A5:C5:D2:67:BB
            X509v3 Authority Key Identifier:
               keyid:BF:8B:5B:ED:36:97:84:40:C8:BA:C2:B1:CD:C4:55:37:8A:55:F1:7D
                DirName:/C=US/ST=Missouri/L=St. Louis/O=Object Computing,
Inc./OU=TAO/CN=John Smith/emailAddress=smith_j@ociweb.com
                serial:A1:D5:25:21:ED:7E:39:33

    Signature Algorithm: md5WithRSAEncryption
        57:5c:a1:20:14:64:88:d5:05:21:b1:8e:6a:32:28:5b:94:07:
        87:00:b6:1f:c6:00:71:47:a5:7b:7f:44:81:82:16:aa:8c:04:
```

```
        e4:03:70:b6:fc:5a:6a:9f:c3:27:bc:35:0d:e1:a5:79:22:ce:
        05:32:44:e1:cc:40:03:62:65:86:de:73:c3:e6:6c:e6:8d:f0:
        4d:dd:2c:65:fe:72:23:b0:4d:de:f2:a7:d0:41:4d:55:85:40:
        8f:08:4f:95:93:53:75:7a:21:34:5a:a8:83:ae:9a:da:10:ef:
        7e:d4:96:52:13:85:5e:52:db:cc:9d:a7:74:52:4e:05:6f:1d:
        b6:b3
-----BEGIN CERTIFICATE-----
MIID3zCCA0igAwIBAgIBATANBgkqhkiG9w0BAQQFADCBmzELMAkGA1UEBhMCVVMx
ETAPBgNVBAgTCE1pc3NvdXJpMRIwEAYDVQQHEwlTdC4gTG91aXMxHzAdBgNVBAoT
Fk9iamVjdCBDb21wdXRpbmcsIEluYy4xDDAKBgNVBAsTA1RBTzETMBEGA1UEAxMK
Sm9obiBTbWl0aDEhMB8GCSqGSIb3DQEJARYSc21pdGhfakBvY2l3ZWIuY29tMB4X
DTA1MDkwOTIzMDg0NloXDTA2MDkwOTIzMDg0NlowgZsxCzAJBgNVBAYTAlVTMREw
DwYDVQQIEwhNaXNzb3VyaTESMBAGA1UEBxMJU3QuIExvdWlzMR8wHQYDVQQKExZP
YmplY3QgQ29tcHV0aW5nLCBJbmMuMQwwCgYDVQQLEwNUQU8xEzARBgNVBAMTCkpv
aG4gU21pdGgxITAfBgkqhkiG9w0BCQEWEnNtaXRoX2pAb2Npd2ViLmNvbTCBnzAN
BgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAqxIMACmYKHipD8GB9PNbARx04Yt0MDm6
q7shbU/i1qndGo8yYgYuDcG1cxF0dm6PLLJzuz5XysuiIGBVzCD86b2qZxPq1yCa
SjxgM9YMwAw61K3hxF4hwwWYEPwBxzzJ9117ioo7yVHovcO2bpCuphhO/X1lUdwX
diGH8giPC3sCAwEAAaOCAS8wggErMAkGA1UdEwQCMAAwLAYJYIZIAYb4QgENBB8W
HU9wZW5TU0wgR2VuZXJhdGVkIENlcnRpZmljYXRlMB0GA1UdDgQWBBSrnsBMdSuJ
wy9P2XGRhdGlxdJnuzCB0AYDVR0jBIHIMIHFgBS/i1vtNpeEQMi6wrHNxFU3ilXx
faGBoaSBnjCBmzELMAkGA1UEBhMCVVMxETAPBgNVBAgTCE1pc3NvdXJpMRIwEAYD
VQQHEwlTdC4gTG91aXMxHzAdBgNVBAoTFk9iamVjdCBDb21wdXRpbmcsIEluYy4x
DDAKBgNVBAsTA1RBTzETMBEGA1UEAxMKSm9obiBTbWl0aDEhMB8GCSqGSIb3DQEJ
ARYSc21pdGhfakBvY2l3ZWIuY29tggkAodUlIe1+OTMwDQYJKoZIhvcNAQEEBQAD
gYEAV1yhIBRkiNUFIbGOajIoW5QHhwC2H8YAcUele39EgYIWqowE5ANwtvxaap/D
J7w1DeGleSLOBTJE4cxAA2Jlht5zw+Zs5o3wTd0sZf5yI7BN3vKn0EFNVYVAjwhP
lZNTdXohNFqog66a2hDvftSWUhOFXlLbzJ2ndFJOBW8dtrM=
-----END CERTIFICATE-----
```

This file can serve as a certificate in this form or the certificate portion can be copied to a separate file. (Lines delimiting the certificate are required.)

27.4.5 Removing Key Pass Phrases

By default, keys are protected by the PEM pass phrase provided when the certificate was created. This pass phrase must be supplied whenever an application utilizing the key is launched. To run processes in the background, remove the pass phrase as follows:

```
$ openssl rsa -in key-with-password.pem -out key-without-password.pem
```

The file `key-with-password.pem` holds a private key protected by a password. The key with password removed is stored in the file `key-without-password.pem`.

27.4.6 Certificate Commands Summary

Environment variables:

- `PATH` includes directories containing `openssl` and `CA.pl`.
- `OPENSSL_CONF` specifies path to configuration file.
- `openssl.conf` is the default configuration file provided with the OpenSSL source release.

Commands:

- `CA.pl -newca` establishes a new local certificate authority.
- `CA.pl -newreq` creates a new private key and certificate request.
- `CA.pl -sign` creates a new public key and a signed certificate.
- `openssl rsa -in key-with-password.pem -out key-without-password.pem` removes the pass phrase from a private key.

27.5 Building ACE and TAO Security Libraries

This section discusses construction of executables that use TAO’s security features and the supporting libraries; `libACE_SSL`, `libTAO_Security`, and `libTAO_SSLIOP`.

The ACE SSL library (`libACE_SSL`) wraps OpenSSL’s implementation of SSL in a manner consistent with ACE’s Inter-Process Communication Service Access Point (IPC_SAP) architecture and, consequently, TAO’s pluggable protocol architecture. SSLIOP (`libTAO_SSLIOP`) is a secure communications protocol, founded on the ACE SSL wrappers, which supports:

- Peer authentication.
- Message integrity.
- Message confidentiality.

TAO’s Security services library (`libTAO_Security`) provides security aware applications with:

- Access to security attributes.
- A means to control message protection level.

- A means to control the establishment of trust between clients and servers.

OpenSSL is the foundation of TAO's security service. It may already be installed on your platform, particularly if you are working with a Linux platform. However, on many other UNIX and UNIX-like platforms, and on Windows platforms, you may need to obtain, build, and install OpenSSL yourself. OpenSSL source is freely available from the OpenSSL web site: `<http://www.openssl.org>`. The source release includes some general instructions, the README file, and files with specific instructions for several platforms, including the INSTALL file for UNIX-like platforms and the INSTALL.W32 file for Windows platforms.

The following sections address:

- Building the security libraries.
- Testing the libraries.
- Building executables.

Note *The instructions for building the security libraries have changed since TAO 1.3a.*

27.5.1 Building Security Libraries: UNIX Variants

ACE and TAO security libraries can be built individually or along with other ACE and TAO libraries. The following settings support construction of the security-related libraries and application executables:

- `SSL_ROOT` (in `platform_macros.GNU` or as an environment variable): specifies the path to the OpenSSL installation directory.
- `ssl` (in `platform_macros.GNU`): indicates whether or not to build the security-related libraries. Specifying `ssl=1` causes the security-related libraries to be built.
- `ssl` (in MPC's `default.features`): must be set to 1 (`ssl=1`) for MPC to generate project files to build SSL-related projects. See 4.3.2.3 for more information on MPC's feature file.

For more information on these and other build settings, see Appendix A.

With these settings assigned correctly, you can build the security libraries simply by building the normal TAO libraries. You can also build the TAO libraries individually with the following commands:

```
cd $TAO_ROOT/orbsvcs/orbsvcs
make -f GNUmakefile.Security
make -f GNUmakefile.SSLIOP
```

`libTAO_SSLIOP` depends upon `libTAO_Security`, so be sure it is built prior to building `libTAO_SSLIOP`.

27.5.2 Building Security Libraries: Microsoft Visual Studio for Windows

TAO's source release contains pre-defined project and workspace/solution files for building ACE and TAO libraries, including those related to TAO's security service. In the `%TAO_ROOT%` directory, `TAO_ACE.sln` includes projects needed to build the various security-related libraries.

The following settings support construction of the security-related libraries and application executables:

- `SSL_ROOT` (environment variable): specifies the path to the OpenSSL installation directory.
- `ssl` (in MPC's `default.features`): must be set to 1 (`ssl=1`) for MPC to generate project files to build SSL-related projects. See 4.3.2.3 for more information on MPC's feature file.

For more information on these and other build settings, see Appendix A.

The MPC-generated project settings establish the appropriate OpenSSL include and library search paths based on the `SSL_ROOT` environment variable (e.g., `$(SSL_ROOT)/include`, `$(SSL_ROOT)/inc32`, and `$(SSL_ROOT)/lib`), so you do not need to add these locations to the Visual Studio tools options settings.

To build the security libraries, open the `TAO_ACE.sln` solution file in Visual Studio and simply build the `SSLIOP` project. Since the `SSLIOP` project depends upon the `SSL` and `Security` projects, building this one project will

build all the necessary security libraries: `ACE_SSL`, `TAO_Security`, and `TAO_SSLIOP`.

27.5.3 Testing the Security Libraries

Tests for TAO's security services are located in `$TAO_ROOT/orbsvcs/tests/Security`. `README` files describe each test and provide other useful information such as instructions for running the test manually. A perl script, `run_test.pl`, is provided to execute each test. `Makefiles` and project files are provided in most, if not all, cases.

27.5.4 Building Security Unaware Executables

TAO's support for security-unaware applications, i.e., the secure communications protocol, is implemented in the `TAO_SSLIOP` library. Thus, a security-unaware application must link with this library if the executable is statically linked. Otherwise, no additional steps are required because the library is loaded dynamically by the service configurator.

MPC projects for security-unaware applications can simply inherit from the `ssl` base project. For example, here is the file for the `SecurityUnawareApp` example in `$TAO_ROOT/orbsvcs/DevGuideExamples/Security/SecurityUnawareApp`:

```
project(*Server): taoexe, portableserver, security, ssl {
  requires += exceptions
  Source_Files {
    Messenger_i.cpp
    MessengerServer.cpp
  }
}

project(*Client): taoexe, security, ssl {
  requires += exceptions
  Source_Files {
    MessengerC.cpp
    MessengerClient.cpp
  }
}
```

27.5.5 Building Security Aware Executables

TAO's support for security aware applications that wish to set security policy or access security context data at run time is implemented in the

TAO_Security library and the TAO_SSLIOP library. OpenSSL libraries are also required for security-aware applications that wish to access the security attributes:

- For UNIX and UNIX-like platforms, the OpenSSL libraries are libssl and libcrypto.
- For Windows platforms, the OpenSSL library is libeay32.

Library path environment variables must include paths to all but TAO_SSLIOP. A path to TAO_SSLIOP is only required at compile time for statically linked executables.

MPC projects for security aware applications can simply inherit from the security and ssliop base projects. For example, here is the project file for the ParticipatingApp example in $TAO_ROOT/orbsvcs/DevGuideExamples/Security/ParticipatingApp:

```
project(*Server): orbsvcsexe, portableserver, security, ssliop {
  requires += exceptions
  Source_Files {
    Messenger_i.cpp
    MessengerServer.cpp
  }
}

project(ParticipatingApp_Client): orbsvcsexe, security, ssliop {
  requires += exceptions
  exename   = MessengerClient

  Source_Files {
    MessengerC.cpp
    MessengerClient.cpp
  }
}
```

27.6 Security Unaware Application

Providing security for applications that do not take an active part in setting or enforcing security policy is largely a matter of configuring the environment, and configuring and installing TAO's SSLIOP protocol. The sections that follow explain how to do this:

- Describe how to set up the environment.

- Describe how to configure and load the SSLIOP protocol.
- Present relevant code examples.

27.6.1 Environment Setup

There are four separate environment variables that may be used by the SSL libraries depending upon circumstances at run time:

- `SSL_CERT_FILE`: gives the pathname of a file containing one or more certificates from certifying authorities.

  When this variable is set, the named file is the source of all certifying authority certificates. Refer to 27.6.1.1 for additional information.

  A default value for `SSL_CERT_FILE` is defined by the macro `ACE_DEFAULT_SSL_CERT_FILE` in `$ACE_ROOT/ace/SSL/sslconf.h`

- `SSL_CERT_DIR`: gives the pathname of a directory housing certificates from one or more certifying authorities.

  When this variable is set, the named directory is the source of all certifying authority certificates. Refer to 27.6.1.1 for additional information.

  A default value for `SSL_CERT_DIR` is defined by the macro `ACE_DEFAULT_SSL_CERT_DIR` in `$ACE_ROOT/ace/SSL/sslconf.h`.

- `SSL_EGD_FILE`: gives the pathname of a random data source generated by the entropy-gathering daemon (EGD).

  OpenSSL needs a source of random data. EGD is a perl script that produces random data and delivers it via a UNIX domain socket. EGD is widely used as a random data source, although it was not tested during the writing of this chapter.

- `SSL_RAND_FILE`: gives the pathname of a data file holding state data from SSL's pseudo random number generator.

  OpenSSL has its own pseudo-random number generator. When this mechanism is used as the random data source, state information is stored in the named file between invocations. OpenSSL's pseudo-random number source was not tested during the writing of this chapter.

All of the examples presented in this chapter employ only a single certifying authority and therefore use the `SSL_CERT_FILE` environment variable to refer

to the certifying authority's certificate. No values were assigned to any of the other environment variables.

27.6.1.1 Using Multiple Certificate Authorities

Authenticating a certificate when multiple certifying authorities are involved requires access to several certificates. For example, suppose a principal named Shannon obtains a certificate from a certifying authority named ACME Certificate Co. In turn, ACME obtained its certificate from Certificates Unlimited, Inc., which signs its own certificate and therefore acts as the root certificate authority (refer to Figure 27-12).

To verify Shannon's certificate, the server's authentication routines need to examine each certifying authority's certificate. The certificates may be concatenated into a single file, referred to by `SSL_CERT_FILE`, or they may be housed in a common directory, referred to by `SSL_CERT_DIR`. When a principal's certificate is authenticated and `SSL_CERT_FILE` is set, the named file is searched for certifying authority certificates. Otherwise, if `SSL_CERT_DIR` is set, the named directory is searched.

To facilitate certificate directory searches, each file is named based upon a hash code generated from the certificate holder's subject name. For example, the subject name:

```
C=US, ST=Missouri, L=St. Louis, O=Object Computing, Inc.,
 OU=TAO, CN=Certifying Authority/Email=ca@ociweb.com
```

yields a hash value of `53052543`. The string "`.0`" is appended yielding a file name of `53052543.0`. A file by that name, which may contain the certificate or may be a link to the certificate file, is placed in the certificate directory.

When authenticating a principal's certificate:

- The issuing authority's name is extracted from the certificate.
- The corresponding hash value is computed.
- The certificate is obtained from the certificate directory and authenticated.

The process continues until a trusted certificate is encountered or a required certificate cannot be found.

"Certificates Unlimited, Inc."

"ACME Certificate Co."

Shannon

Figure 27-12 Certificate Authentication Chain

27.6.2 Configuring and Loading SSLIOP

The use of SSLIOP is initialized by using the Service Configuration framework to load a SSLIOP factory. The SSLIOP factory initialization is tuned by way of the arguments listed in Table 27-2

Table 27-2 SSLIOP Factory Options

Option	Section	Description
`-SSLAcceptTimeout` *seconds*	27.10.1	Specify the amount of time that a SSLIOP acceptor will wait to complete the SSL handshake
`-SSLAuthenticate (none | client | server | server_and_client)`	27.10.2	Specify the authentication conducted as secure sessions are established between client and server.
`-SSLCAFile` *file*	27.10.3	Supply a specific file for Certificate Authority info.
`-SSLCAPath` *directory*	27.10.4	Supply a directory containing possibly several arbitrary CA files.
`-SSLCertificate` *file*	27.10.5	Supply the authentication certificate.
`-SSLCheckHost`	27.10.6	Cause the local endpoint to perform host name verification.
`-SSLCipherList` *list*	27.10.7	Supply the list or cipher types for SSL.
`-SSLDHParams` *file*	27.10.8	gives the pathname for Diffie-Hellman cipher parameters.
`-SSLNoProtection`	27.10.9	Enable support for plain IIOP as well as SSLIOP connections.
`-SSLPassword` *prompt*	27.10.10	Enable the use of password protected private keys by supplying the means to obtain a password.
`-SSLPrivateKey` *file*	27.10.11	Supply the private encryption key.
`-SSLrand` *file*	27.10.12	Specify the file containing the SSL random seed.
`-SSLServerCipherOrder`	27.10.13	Set SSL to use the server's cipher preference.
`-SSLVersionList` *list*	27.10.14	Constrains the list of cipher versions allowed.
`-verbose | -v`	27.10.15	Enables SSLIOP-specific debugging.

As with all pluggable protocols, the configured SSLIOP factory must be registered with the default or advanced resource factory (see 18.2.6, "Protocol

Factories") to be available for use by the ORB. Unlike other protocols, SSLIOP is derived from IIOP so the IIOP factory does not also need to be defined.

27.6.3 Security Unaware Application Example

To demonstrate an application that is protected via TAO's security service without the application's direct involvement, we configure the Messenger server and Messenger client from 3.3 so that they:

- Authenticate their peers.
- Provide message confidentiality.
- Provide message integrity.

The full source code and configuration files for this example can be found in `$TAO_ROOT/orbsvcs/DevGuideExamples/Security/SecurityUnawareApp`.

The examples that follow require:

- A certifying authority's certificate named `cacert.pem`.
- A certificate, signed by the certifying authority, and its corresponding private key, respectively named `servercert.pem` and `serverkey.pem`.
- A client certificate, signed by the certifying authority, and its corresponding private key respectively named `clientcert.pem` and `clientkey.pem`.
- The environment variable `SSL_CERT_FILE` containing the path to the certifying authority's certificate.

Configuring the server is easily accomplished with the service configuration directives found in Figure 27-13. These directives are placed in a service

```
dynamic SSLIOP_Factory Service_Object * \
   TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() "-SSLAuthenticate \
   SERVER_AND_CLIENT -SSLPrivateKey PEM:serverkey.pem \
   -SSLCertificate PEM:servercert.pem"
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

Figure 27-13 Security Unaware Server-side svc.conf

configuration file, here called `server.conf`, which is used to configure the Messenger server upon invocation as follows:

```
$ MessengerServer -ORBSvcConf server.conf
```

```
Enter PEM pass phrase:
IOR written to file Messenger.ior
```

In this example the pass phrase has not been stripped from the private key. Consequently, the user must type the pass phrase each time the server is started.

Invoking the Messenger client without configuring it to use SSLIOP causes an exception:

```
$ MessengerClient
Uncaught CORBA exception: NO_PERMISSION (IDL:omg.org/CORBA/NO_PERMISSION:1.0)
$
```

The exception indicates that the client does not have permission to invoke the requested operation. Configuring the client is also easily accomplished with the service configuration directives found in Figure 27-14.

```
dynamic SSLIOP_Factory Service_Object * \
   TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
   "-SSLAuthenticate SERVER_AND_CLIENT \
   -SSLPrivateKey PEM:clientkey.pem \
   -SSLCertificate PEM:clientcert.pem"
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

Figure 27-14 Security Unaware Client-side svc.conf

These directives are placed in a separate service configuration file, which we call `client.conf`, used to configure the Messenger client upon invocation:

```
$ MessengerClient -ORBSvcConf client.conf
Enter PEM pass phrase:
message was sent
$
```

The client now reports that the message was sent and the server prints the message:

```
Message from: Chief of Security
Subject:      New Directive
Message:      Implementing security policy now!
```

Failing to name the certifying authority's certificate via the `SSL_CERT_FILE` environment variable is a common mistake, and results in a different exception:

```
$ MessengerClient -ORBSvcConf client.conf
```

```
Enter PEM pass phrase:
ACE_SSL (784|1040) error code: 336134278 - error:14090086:SSL
routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify failed
Uncaught CORBA exception: TRANSIENT (IDL:omg.org/CORBA/TRANSIENT:1.0)
$
```

This exception is less helpful, indicating no more than a problem with a transient object reference. The error message from the ACE SSL wrappers is more helpful in that it indicates an error related to certificate authentication. In the case at hand, authentication of the server's certificate failed because the certifying authority's certificate was unknown to the client.

However, if the client application is concerned only with message confidentiality and integrity, authentication of the server can be disabled by modifying the client's service configuration as shown in Figure 27-15.

```
dynamic SSLIOP_Factory Service_Object * \
   TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
   "-SSLAuthenticate NONE -SSLPrivateKey PEM:clientkey.pem \
   -SSLCertificate PEM:clientcert.pem"
```

Figure 27-15 Client-side svc.conf for confidentiality and integrity only

In this configuration, the client no longer requires authentication of the server, so the absence of the certifying authority's certificate is of no consequence, at least so far as the client is concerned. The server's configuration still calls for authentication of peers so the server's environment must provide access to the certifying authority's certificate.

27.7 Security Policy Controlling Application

Applications can control the message protection level and the extent to which the ORB authenticates its peers. The level of message protection is controlled by the *Quality of Protection* policy. The extent to which an ORB authenticates its peers is controlled by the *Establish Trust* policy.

The following sections:

- Describe the Quality of Protection policy.
- Describe the Establish Trust policy.
- Present code examples that demonstrate the use of these policies.

27.7.1 Controlling Message Protection

The SecQOPPolicy permits control of the Quality of Protection, and carries a value of type Security::QOP. The Security module gives the following type definition:

```
module Security {
   // other definitions omitted
   enum QOP {
      SecQOPNoProtection,
      SecQOPIntegrity,
      SecQOPConfidentiality,
      SecQOPIntegrityAndConfidentiality
   };
};
```

Each value represents a different level of protection:

- SecQOPNoProtection: disables all message protection mechanisms. Consequently, messages are transmitted in plaintext.
- SecQOPIntegrity: enables mechanisms that deter tampering with messages in transit and support the detection of messages that have been tampered with in some way.
- SecQOPConfidentiality: enables mechanisms that prevent a message's disclosure to all but its intended recipients.
- SecQOPIntegrityAndConfidentiality: enables both integrity and confidentiality protection mechanisms.

SSL does not support message integrity and confidentiality separately. Thus, the only practical values for SecQOPPolicy are SecQOPNoProtection and SecQOPIntegrityAndConfidentiality. The other values have no effect.

27.7.2 Controlling Peer Authentication

The SecEstablishTrustPolicy permits control of the Establish Trust policy, and carries a value of type Security::EstablishTrust. The Security module gives the type definition for EstablishTrust as follows:

```
module Security {
   struct EstablishTrust {
      boolean trust_in_client;
      boolean trust_in_target;
   };
   // all other definitions omitted
```

```
};
```

`EstablishTrust::trust_in_client` specifies whether or not an ORB authenticates its peers when acting in the server role. `EstablishTrust::trust_in_target` specifies whether or not an ORB authenticates its peers when acting in the client role.

27.7.3 Security Policy Controlling Application Examples

We continue with the Messenger client and server to demonstrate security policies. Full source code and configuration files for this example can be found in `$TAO_ROOT/orbsvcs/DevGuideExamples/Security/PolicyControllingApp`.

We use the configurations shown in Figure 27-16 and Figure 27-17 to devise a server that authenticates its peers, without regard to its role, and accepts requests via IIOP as well as SSLIOP.

```
dynamic SSLIOP_Factory Service_Object* \
   TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
   "-SSLNoProtection -SSLAuthenticate SERVER_AND_CLIENT \
   -SSLPrivateKey PEM:serverkey.pem -SSLCertificate \
   PEM:servercert.pem"

static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

Figure 27-16 Policy Controlling Application Server-side svc.conf

The client, as configured, issues all requests via SSLIOP but does not authenticate its peers.

```
dynamic SSLIOP_Factory Service_Object* \
   TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
   "-SSLAuthenticate NONE -SSLPrivateKey PEM:clientkey.pem \
   -SSLCertificate PEM:clientcert.pem"
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

Figure 27-17 Policy Controlling Application Client-side svc.conf

Let us assume, however, that the Messenger client requires the Messenger server to be authenticated, after which it transmits messages in the clear. To accomplish this, message protection is disabled and peer authentication is enabled when operating as a client.

The client first obtains a reference to the Messenger server:

```
CORBA::Object_var obj = orb->string_to_object( "file://Messenger.ior" );
```

Next, the client constructs a `Security::SecQOPPolicy` to disable message protection:

```
Security::QOP qop = Security::SecQOPNoProtection;

CORBA::Any no_protection;
no_protection <<= qop;

CORBA::Policy_var policy =
  orb->create_policy (Security::SecQOPPolicy, no_protection);
```

Next, the client creates a `Security::SecEstablishTrustPolicy` to enable authentication of servers:

```
Security::EstablishTrust establish_trust;
establish_trust.trust_in_client = false;
establish_trust.trust_in_target = true;

CORBA::Any want_trust;
want_trust <<= establish_trust;

CORBA::Policy_var policy2 =
  orb->create_policy (Security::SecEstablishTrustPolicy, want_trust);
```

The client now assembles a policy list and uses it to create a new object reference that has the desired policies:

```
CORBA::PolicyList policy_list (2);
policy_list.length (2);
policy_list[0] = CORBA::Policy::_duplicate (policy.in ());
policy_list[1] = CORBA::Policy::_duplicate (policy2.in ());

CORBA::Object_var object =
  obj->_set_policy_overrides (policy_list, CORBA::SET_OVERRIDE);
```

Finally, the client narrows the new object reference to the appropriate type:

```
Messenger_var messenger = Messenger::_narrow( object.in() );
```

Here is the complete client for this example:

```
#include "MessengerC.h"
#include "orbsvcs/SecurityC.h"
```

```
int main( int argc, char* argv[] ) {
  try {
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv );
    CORBA::Object_var obj = orb->string_to_object( "file://Messenger.ior" );

    // downgrade to no message protection
    Security::QOP qop = Security::SecQOPNoProtection;
    CORBA::Any no_protection;
    no_protection <<= qop;

    CORBA::Policy_var policy =
      orb->create_policy (Security::SecQOPPolicy, no_protection);
    // upgrade to authenticate servers
    Security::EstablishTrust establish_trust;
    establish_trust.trust_in_client = false;
    establish_trust.trust_in_target = true;
    CORBA::Any want_trust;
    want_trust <<= establish_trust;
    CORBA::Policy_var policy2 =
      orb->create_policy (Security::SecEstablishTrustPolicy, want_trust);


    // prepare to create new object reference
    // having the desired policies
    CORBA::PolicyList policy_list (2);
    policy_list.length (2);
    policy_list[0] = CORBA::Policy::_duplicate (policy.in ());
    policy_list[1] = CORBA::Policy::_duplicate (policy2.in ());

    // create the new object reference and
    // narrow to appropriate type
    CORBA::Object_var object =
      obj->_set_policy_overrides (policy_list, CORBA::SET_OVERRIDE);
    Messenger_var messenger = Messenger::_narrow( object.in() );

    CORBA::String_var message =
      CORBA::string_dup( "Implementing security policy now!" );
    messenger->send_message(
      "Chief of Security", "New Directive", message.inout() );
  }
  catch (CORBA::Exception& ex) {
    ex._tao_print_exception("Client: main block");
  }
  return 0;
}
```

For the next example, we change the server and client configurations to those shown in Figure 27-18 and Figure 27-19, respectively.

```
dynamic SSLIOP_Factory Service_Object* \
   TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
   "-SSLAuthenticate SERVER_AND_CLIENT \
   -SSLPrivateKey PEM:serverkey.pem \
   -SSLCertificate PEM:servercert.pem"
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

Figure 27-18 Policy Controlling and Enforcing Server-side svc.conf

With these changed configurations, the client must upgrade the message protection policy, instead of downgrading as demonstrated by the previous example, because the server no longer accepts requests via IIOP. The server now enforces message protection on all requests.

```
dynamic SSLIOP_Factory Service_Object* \
   TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
   "-SSLNoProtection -SSLAuthenticate NONE \
   -SSLPrivateKey PEM:clientkey.pem \
   -SSLCertificate PEM:clientcert.pem"
static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

Figure 27-19 Policy Controlling and Enforcing Client-side svc.conf

Here is the relevant change to the client code:

```
Security::QOP qop = Security::SecQOPIntegrityAndConfidentiality;
CORBA::Any want_protection;
want_protection <<= qop;
CORBA::Policy_var policy =
   orb->create_policy (Security::SecQOPPolicy, want_protection);
```

The other code remains as is. Here is the complete example:

```
#include "MessengerC.h"
#include "orbsvcs/SecurityC.h"

int main( int argc, char* argv[] )
{
   try {
      CORBA::ORB_var orb = CORBA::ORB_init( argc, argv );
      CORBA::Object_var obj =
         orb->string_to_object( "file://Messenger.ior" );

      // upgrade to message protection
      Security::QOP qop = Security::SecQOPIntegrityAndConfidentiality;
      CORBA::Any want_protection;
      want_protection <<= qop;
      CORBA::Policy_var policy =
```

```
      orb->create_policy (Security::SecQOPPolicy, want_protection);

    // upgrade to authenticate servers
    Security::EstablishTrust establish_trust;
    establish_trust.trust_in_client = false;
    establish_trust.trust_in_target = true;
    CORBA::Any want_trust;
    want_trust <<= establish_trust;

    CORBA::Policy_var policy2 =
      orb->create_policy (Security::SecEstablishTrustPolicy, want_trust);

    // prepare to create new object reference
    // having the desired policies
    CORBA::PolicyList policy_list (2);
    policy_list.length (1);
    policy_list[0] = CORBA::Policy::_duplicate (policy.in ());
    policy_list.length (2);
    policy_list[1] = CORBA::Policy::_duplicate (policy2.in ());

    // create the new object reference and
    // narrow to appropriate type
    CORBA::Object_var object =
      obj->_set_policy_overrides (policy_list, CORBA::SET_OVERRIDE);
    Messenger_var messenger =  Messenger::_narrow( object.in() );
    CORBA::String_var message =
      CORBA::string_dup( "Implementing security policy now!" );
    messenger->send_message( "Chief of Security",
      "New Directive",
      message.inout()
    );
  }
  catch (CORBA::Exception& ex) {
    ex._tao_print_exception("Client: main block");
  }
  return 0;
}
```

27.8 Security Policy Enforcing Application

An application can take some responsibility for ensuring compliance with security policies during its operation. An application cannot take complete responsibility because, as was discussed previously, there are many contributing factors outside a software application’s control. To contribute towards compliance with security policies, an application needs access to security context information at run time.

TAO's implementation currently only offers a proprietary interface that provides access to security context information during secure operation invocations:

- `SSLIOP::Current` is a TAO-specific extension to the security specification. It provides access to the certificate and the certificate verification chain associated with a secure invocation's initiator. It also provides the means to determine whether or not the current invocation is taking place within a secure session.

The sections that follow describe this interfaces and present code examples demonstrating its use. Full source code and configuration files for this example can be found in `$TAO_ROOT/orbsvcs/DevGuideExamples/Security/ParticipatingApp`.

Note *The forthcoming SecurityLevel3 module also offers a Current which would provide similar information as to that provided in the now-obsolete SecurityLevel1 and SecurityLevel2 interfaces.*

27.8.1 SSLIOP::Current

The module `SSLIOP` defines the `SSLIOP::Current` interface and associated types. The relevant portions of this module are:

```
module SSLIOP {

   // portions omitted
   typedef sequence<octet> ASN_1_Cert;
   typedef sequence<ASN_1_Cert> SSL_Cert;

   local interface Current : CORBA::Current {
      exception NoContext {};

      ASN_1_Cert get_peer_certificate () raises (NoContext);
      SSL_Cert get_peer_certificate_chain () raises (NoContext);
      boolean no_context ();
   };
};
```

An `ASN_1_Cert` is a sequence of octets. Certificates are encoded using ASN1's distinguished encoding rules (DER). The OpenSSL library includes functions for converting certificates from this format to OpenSSL's internal format and routines for manipulating certificates once they are in the internal format.

The operation `get_peer_certificate()` returns the peer's certificate in ASN1 DER format. The operation `get_peer_certificate_chain()` returns the sequence of certificates used to validate the initiating peer's certificate; the chain does not include the originating principal's certificate. These operations are available only during a secure invocation and only at the target. Consequently, peer always refers to the operation's initiator, i.e., to the client.

The operation `no_context()` indicates whether or not the current operation was invoked via a secure session. Invoking the other operations outside the scope of a secure invocation results in a `SSLIOP::Current::NoContext` exception.

27.8.2 Security Policy Enforcing Application Examples

For the first example, the Messenger server will now indicate whether or not a message was delivered via a secure invocation. The new message format is:

```
SECURE message from: Chief of Security
Subject:             New Directive
Message:             Implementing security policy now!
```

We use `SSLIOP::Current::no_context()` to determine the nature of the invocation, secure or not secure. Several changes are necessary to provide this capability.

First, the `Messenger_i` servant needs a reference to `SSLIOP::Current` that is accessible during the invocation. For that purpose, we add an `SSLIOP::Current_var` attribute to the Messenger server object:

```
#include <orbsvcs/SSLIOPC.h>
#include "MessengerS.h"

class  Messenger_i : public virtual POA_Messenger {
public:
   Messenger_i (
      SSLIOP::Current_ptr ssliop_current
   );

   virtual ~Messenger_i (void);

   virtual CORBA::Boolean send_message (
      const char* user_name,
      const char* subject,
      char*& message
```

```
   );

protected:
   SSLIOP::Current_var ssliop_current_;
};
```

A reference to `SSLIOP::Current` is passed to the `Messenger_i` servant upon construction and stored in `ssliop_current_`:

```
#include "Messenger_i.h"
#include <iostream>

Messenger_i::Messenger_i (
   SSLIOP::Current_ptr ssliop_current
)
: ssliop_current_(SSLIOP::Current::_duplicate(ssliop_current))
{
}
```

Now the Messenger server's main function must obtain a reference to `SSLIOP::Current` to pass to the `Messenger_i` servant's constructor. A reference to `SSLIOP::Current` is obtained via the interface `CORBA::ORB::resolve_initial_references()`. "`SSLIOPCurrent`" is the name of the `SSLIOP::Current` object.

```
obj = orb->resolve_initial_references ("SSLIOPCurrent");
SSLIOP::Current_var ssliop_current =
   SSLIOP::Current::_narrow (obj.in ());
PortableServer::Servant_var<Messenger_i> messenger_servant =
  new Messenger_i(ssliop_current.in());
```

Finally, the message output function uses the boolean operation `SSLIOP::Current::no_context()` to select a message format:

```
CORBA::Boolean Messenger_i::send_message (
      const char* user_name,
      const char* subject,
      char*& message
   )
{
   if (ssliop_current_->no_context())
      std::cout << "Message from:        " << user_name << std::endl;
   else
      std::cout << "SECURE message from: " << user_name << std::endl;

   std::cout << "Subject:             " << subject << std::endl;
```

```
    std::cout << "Message:            " << message << std::endl;
    std::cout << std::endl;
    return true;
}
```

For the second example, we introduce a new operation called `shutdown()` that stops the Messenger server. However, we only allow authenticated users to shut down the Messenger service. Therefore, make sure that all invocations of `shutdown()` occur within a secure session by examining the return value of `SSLIOP::Current::no_context()`.

Here is the new interface definition:

```
interface Messenger {
    boolean send_message ( in    string user_name,
                           in    string subject,
                           inout string message );

    void shutdown ( in string user_name );
};
```

The ORB is shut down via the operation `CORBA::ORB::shutdown()`. The `Messenger_i` servant now needs a reference to the ORB as well as a reference to `SSLIOP::Current`.

Here is the updated `Messenger_i` servant header file:

```
#include <orbsvcs/SSLIOPC.h>
#include "MessengerS.h"

class  Messenger_i : public virtual POA_Messenger {
public:
    Messenger_i (
        CORBA::ORB_ptr orb,
        SSLIOP::Current_ptr ssliop_current
    );

    virtual ~Messenger_i (void);

    virtual CORBA::Boolean send_message (
        const char* user_name,
        const char* subject,
        char*& message
    );

    virtual void shutdown (
        const char* user_name
```

```
   );

protected:
   CORBA::ORB_var orb_;
   SSLIOP::Current_var ssliop_current_;
};
```

Now the implementation changes. First the constructor:

```
Messenger_i::Messenger_i (
   CORBA::ORB_ptr orb,
   SSLIOP::Current_ptr ssliop_current
)
: orb_(CORBA::ORB::_duplicate(orb)),
  ssliop_current_(SSLIOP::Current::_duplicate(ssliop_current))
{
}
```

And the `shutdown()` operation:

```
void Messenger_i::shutdown (
   const char* user_name
)
{
   if ( ! (ssliop_current_->no_context()) ) {
      std::cout << "Shutdown command from: " << user_name << std::endl;
      std::cout << "Status:                User authenticated." << std::endl;
      std::cout << "Action:                Sever shutdown in progress..."
                << std::endl;
      std::cout << std::endl;

      orb_->shutdown (0);
   }
   else {
      std::cout << "Shutdown command from: " << user_name << std::endl;
      std::cout << "Status:                User *NOT* authenticated."
                << std::endl;
      std::cout << "Action:                Ignored." << std::endl;
      std::cout << std::endl;
   }
}
```

After the servant calls `CORBA::ORB::shutdown()`, `CORBA::ORB::run()` returns in `main`. We announce the service is shut down and clean up before exiting. Here are the changes to the main function:

```
#include "Messenger_i.h"
```

```
#include <iostream>

int
main( int argc, char* argv[] ) {
   try {

   // Not shown:
   //   Init the ORB,
   //   get and activate the Root POA

   obj =
      orb->resolve_initial_references ("SSLIOPCurrent");

   SSLIOP::Current_var ssliop_current =
   SSLIOP::Current::_narrow (obj.in ());

   PotrableServer::Servant_var<Messenger_i> messenger_servant =
     new Messenger_i(orb.in(), ssliop_current.in() );

   // Not shown:
   //   activate the messenger servant,
   //   create and export a Messenger
   //   object reference

   orb->run();
   poa->destroy (true, true);
   orb->destroy ();

   std::cout << "Messenger Server is shut down!"
             << std::endl;
   std::cout << std::endl;

   }

   // Not shown:
   //   exception handling

   return 0;
}
```

The Messenger service still handles message requests submitted in the clear by unauthenticated clients, but it will not process a shutdown request unless it is submitted via a secure session:

```
Message from:        Chief of Security
Subject:             New Directive
Message:             Terminating Messenger service!

Shutdown command from: Chief of Security
```

```
Status:                User *NOT* authenticated.
Action:                Ignored.
```

Here is the complete implementation for `Messenger::shutdown()`:

```
void Messenger_i::shutdown (const char* user_name)
{
   if ( ! (ssliop_current_->no_context()) ) {
      // client has been authenticated

      std::cout << "Shutdown command from: " << user_name << std::endl;
      std::cout << "Status:                User authenticated." << std::endl;
      std::cout << "Action:                Sever shutdown in progress..."
                << std::endl;
      std::cout << std::endl;

      orb_->shutdown (0);
   }
   else {
      // requester has not been authenticated

      std::cout << "Shutdown command from: " << user_name << std::endl;
      std::cout << "Status:                User *NOT* authenticated."
                << std::endl;
      std::cout << "Action:                Ignored." << std::endl;
      std::cout << std::endl;
   }
}
```

Here is the complete Messenger server's main function:

```
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

int main( int argc, char* argv[] )
{
   try {

      CORBA::ORB_var orb = CORBA::ORB_init( argc, argv );
      CORBA::Object_var obj =
         orb->resolve_initial_references( "RootPOA" );
      PortableServer::POA_var poa =
         PortableServer::POA::_narrow( obj.in() );
      PortableServer::POAManager_var mgr =
         poa->the_POAManager();
      mgr->activate();

      obj = orb->resolve_initial_references ("SSLIOPCurrent");
```

```
        SSLIOP::Current_var ssliop_current =
          SSLIOP::Current::_narrow (obj.in ());

        PortableServer::Servant_var<Messenger_i> messenger_servant =
          new Messenger_i(orb.in(),
                          security_current.in(),
                          ssliop_current.in());

        PortableServer::ObjectId_var oid =
          poa->activate_object(messenger_servant.in());
        CORBA::Object_var messenger_obj =
          poa->id_to_reference( oid.in() );
        CORBA::String_var str =
          orb->object_to_string( messenger_obj.in() );
        std::ofstream iorFile("Messenger.ior");
        iorFile << str.in() << std::endl;
        iorFile.close();

        std::cout << "IOR written to file Messenger.ior" << std::endl;

        orb->run();
        poa->destroy (true, true);
        orb->destroy ();

        std::cout << "Messenger Server is shut down!"
                  << std::endl
                  << std::endl;
    }
    catch( const CORBA::Exception& ex ) {
        ex._tao_print_exception("Server Error: main block");
        return 1;
    }

    return 0;
}
```

27.9 Mixed Security Model Applications

Occasionally, an application may need to apply strict security for access to most objects, while exempting access to some. Or may need the inverse, applying weak security for most objects, while restricting access to some.

This can be achieved using the `SercurityLevel2::AccessDecision` object provided as part of the TAO security library. It is a local object, who's interface is described in the CORBA Security Service specification, version 1.8. The formal interface has a single operation, `access_allowed()` used by

the infrastructure to determine if a given request may be delivered to a particular object. TAO extends the interface to facilitate the configuration of the AccessDecision object.

From SecurityLevel2.idl, the interface for TAO's AccessDecision object is shown here:

```
module SecurityLevel2 {
   local interface AccessDecision {
     boolean access_allowed (
        in SecurityLevel2::CredentialList  cred_list,
        in Object                          target,
        in CORBA::Identifier               operation_name,
        in CORBA::Identifier               target_interface_name
     );
   };
};
```

The CORBA compliant interface presents two challenges to TAO. The first is general, there has to be a way to inform the AccessDecision object how do decide about specific invocations. The second is really a TAO-specific challenge, brought on by a performance optimization in the CDR marshalling code. TAO does not initialize the bytes of padding that result from buffer alignment. Thus two object references may point to the same object, but they cannot be compared using a simple byte-wise comparison function.

To address these challenges, a TAO-specific extension to the AccessDecision interface is provided.

```
module TAO {
   module SL2 {
     local interface AccesDecision : SecurityLevel2::AccessDecision {
        boolean access_allowed_ex (
           in ::CORBA::ORBid                  orb_id,
           in ::CORBA::OctetSeq               adapter_id,
           in ::CORBA::OctetSeq               object_id,
           in SecurityLevel2::CredentialList  cred_list,
           in CORBA::Identifier               operation_name
        );

        attribute boolean default_decision;

        void add_object(
           in ::CORBA::ORBid     orb_id,
           in ::CORBA::OctetSeq  adapter_id,
           in ::CORBA::OctetSeq  object_id,
           in boolean            allow_insecure_access
        );
        void remove_object(
```

```
            in ::CORBA::ORBid      orb_id,
            in ::CORBA::OctetSeq  adapter_id,
            in ::CORBA::OctetSeq  object_id
        );
      };
   };
};
```

The first challenge is addressed by providing accessor operations in the TAO specific extension. These allow an application to easily identify particular objects on which the AccessDecision is to decide, as well as a way to supply a default decision for any object not explicitly added.

The challenge of unambiguously identifying a target object is addressed by supplying the specific ORB, object adapter, and object identifiers. These can be safely compared against reference values.

27.9.1 Using the AccessDecision Object

Server applications may use the `AccessDecision` object's interface to add or remove objects, and set the default decision value. The SSLIOP invocation interceptor is responsible for calling the `access_allowed_ex()` operation. A reference to the `AccessDecision` object is obtained through the security manager interface. That interface returns the CORBA compliant base type object, which must then be narrowed to a TAO-specific object for initialization.

```
{
   CORBA::Object_var obj = orb->resolve_initial_references
   ("SecurityLevel2:SecurityManger");
   SecurityLevel2::SecurityManager_var sl2sm =
      SecurityLevel2::SecurityManager::_narrow(obj.in)
   SecurityLevel2::AccessDecision_var ad = sl2sm->access_decision()
   TAO::SL2::AccessDecision_var tao_ad =
      TAO::SL2::AccessDecision::_narrow(ad.in);
```

Once obtained, the server is able to supply specific object references which may allow more or less restrictive access. There are two configuration options with the TAO AccessDecision interface. Applications can set the default decision, which is initially set to deny all requests. Applications can also supply objects which are to be evaluted separate from the default.

The access being decided is less secure than what is configured for the endpoint. There are two scenarios to consider, either allow most to be accessed freely and require a higher level of security for some, or allow some to be accessed freely and require a higher level of security for the rest. To

achieve the first case, set the default decision attribute to true, and then add the specific objects to be secured individually. To achieve the second case allow the default decision to remain false, and then add the specific objects for open access.

Supplying objects for special consideration to the `AccessDecision` object is done via the `add_object()` operation. This requires the 3-tuple of ORB id, POA id, and object ID relative to the POA. Continuing with the example above, here we show adding some object to the `AccessDecision` object.

```
CORBA::String_var orbid = orb->id();
CORBA::OctetSeq_var poaid = my_poa->id();
PortableServer::ObjectId_var oid = my_poa->reference_to_id(my_ref.in())
tao_ad->add_object (orbid.in(), poaid.in(), oid.in(), true);
tao_ad->default_decision(false)
```

Once added, the object reference may be published via any ordinary means, and the application no longer has to consider the `AccessDecision` object, unless it wishes to later remove the object reference without terminating the server.

27.10 SSLIOP Factory Options

The remainder of this chapter describes the individual options interpreted by the default SSLIOP factory. These options are applied to the default SSLIOP factory by the service configurator as described in 27.6.2.

27.10.1 SSLAcceptTimeout *duration*

Description This option specifies the amount of time in seconds, as a floating point value, that SSLIOP will wait to complete the SSL handshake. For more information, see 27.3.3.1, “Handshake Protocol” on page 989.

The default time out duration is 10 seconds.

Impact The SSL handshake includes a TCP handshake, so the timeout value should take this the TCP handshake duration into account.

See Also 27.6.2, 27.6.3, 27.3.3.1

Example

```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLAcceptTimeout 15.0"
```

27.10.2 SSLAuthenticate *which*

Values for *which*

NONE (default)	Disable peer authentication.
SERVER	Client process will authenticate peers.
CLIENT	Server process will authenticate peers. The SSL protocol causes this setting to be equivalent to SERVER_AND_CLIENT.
SERVER_AND_CLIENT	Client and Server processes will authenticate peers.

Description This option specifies the level of peer authentication conducted as secure sessions are established between a client and a server.

Impact This option impacts the authentication required to participate in a secure session.

See Also 27.6.2, 27.6.3, 27.7.3

Example

```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLAuthenticate SERVER_AND_CLIENT"
```

27.10.3 SSLCAFile *FORMAT:filename*

Valuesfor *FORMAT*

ASN1	ASN.1, Abstract Syntax Notation
PEM	Privacy Enhanced Mail

Description Use this option to provide a file that contains an X.509 formatted trusted certificate chain.

The environment variable `SSL_CERT_FILE` is checked if no `-SSLCAFile` option is supplied.

Example

```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLCertificate PEM:cert.pem -SSLCAFile PEM:ca.pem"
```

27.10.4 SSLCAPath directory

Description Use this option to provide a directory that contains one or more files that include PEM encoded trusted certificate chains.

The environment variable `SSL_CERT_PATH` is checked if no `-SSLCAPath` option is supplied.

Note Older versions of OpenSSL do not correctly process this value.

See Also 27.3.2, 27.4.2

Example

```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLCertificate PEM:cert.pem -SSLCAPath /ssl/ca"
```

27.10.5 SSLCertificate *FORMAT:filename*

Valuesfor *FORMAT*

ASN1	ASN.1, Abstract Syntax Notation
PEM	Privacy Enhanced Mail

Description This option specifies the path to the certificate used by the principal and the format used to encode the data.

See Also 27.6.2, 27.6.3

Example

```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLCertificate PEM:client_cert.pem"
```

27.10.6 SSLCheckHost

Description This option is used to enable another level of verification. The actual host name of the peer is checked against specific host names or domain names listed in the peer’s certificate.

Impact The connection is terminated if the peer's host name cannot be determined, certificates were not exchanged, the peer's certificate does not include host or domain names, or if the version of the SSL implementation does not support the check host feature.

Example
```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLCheckhost -SSLAuthenticate SERVER"
```

27.10.7 SSLCipherList *list*

Description Supply a comma separated list of ciphers to the SSL implementation via a call to `SSL_CTX_set_cipher_list()`. Refer to documentation on that function for a description of acceptable cipher names.

Example
```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLCipherList DSS,SHA"
```

27.10.8 SSLDHparams *FORMAT:filename*

Valuesfor *FORMAT*

ASN1	ASN.1, Abstract Syntax Notation
PEM	Privacy Enhanced Mail

Description This option specifies the path to the Diffie-Hellman cipher parameters.

A Diffie-Hellman cipher (anonymous or otherwise) requires these parameters. Typically, a PEM-encoded DSA certificate includes a DH parameters section. However, because the Anonymous Diffie-Hellman (ADH) cipher uses no certificates, you *must* use this option (typically on the server side) for ADH.

Impact If you use the `-SSLDHparams` option together with `-SSLCertificate`, SSLIOP uses the parameters specified by this option.

Note SSLIOP only recognizes the PEM encoding, and gives an error if you specify ASN1

See Also 27.6.2, 27.6.3

Example
```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
```

```
"-SSLCertificate PEM:client_cert.pem \
 -SSLDHParams PEM:dhparams.pem"
```

27.10.9 SSLNoProtection

Description This option is used to enable the support of insecure IIOP connections along side SSL secured connections. It is equivalent to specifying `DEFAULT:eNULL` for the SSL cipher list.

When `-SSLNoProtection` is specified for server ORBs, the server will accept plain IIOP connections as well as SSLIOP connections. Without it, the server will reject plain IIOP traffic.

Clients configured with `-SSLNoProtection` will use IIOP whenever allowed, permitting it to use both clear and secure services. If a server supports both IIOP and SSLIOP, the configured client will use IIOP.

Specifying `-SSLNoProtection` allows insecure IIOP requests to be sent and received by ORBs.

See Also 27.6.2, 27.7.3

Example

```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLNoProtection"
```

27.10.10 SSLPassword *KIND:value*

Values for *KIND:value*

prompt:[*string*]	The value is a password entry prompt on the app’s terminal device. If value is omitted then “Enter Password” is used
file:*path*	The value is a path to a file containing only a password as a clear text string. There is no default file name.
env:[*var name*]	The value is an environment variable containing the password as a clear text string. The default variable name it `TAO_PEM_PASSWORD`.
string	A string not preceded by one of the above kind values is used as the password itself.

Description Enables the distribution of password protected private keys while providing deployment flexibility. A password protected key may be used without this

option, in which case the SSL library's default password entry mechanism is used.

Usage A prompt string containing spaces must be quoted. The quotes must surround the entire argument for `-SSLPassword`.

Any password kind beside the user prompt involves externalizing a clear text password.

Example

```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLPassword 'prompt:Enter client key password'
-SSLPrivateKey PEM:client_key.pem"
```

27.10.11 SSLPrivateKey *FORMAT:filename*

Valuesfor *FORMAT*

ASN1	ASN.1, Abstract Syntax Notation
PEM	Privacy Enhanced Mail

Description This option specifies the type of private key, either PEM or ASn1, and the pathname of the private key used by the principal.

See Also 27.6.2, 27.6.3

Example

```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLPrivateKey PEM:client_key.pem"
```

27.10.12 SSLRand *filename*

Description This option provides a path to the seed file used by SSL. Refer to your SSL library documentation for details on the contents of the file.

The environment variable `SSL_RAND_FILE` is checked if `-SSLRand` is not supplied.

27.10.13 SSLServerCipherOrder

Description Set the SSL CTX option `SSL_OP_CIPHER_SERVER_PREFERENCE` to have the server use its cipher list rather than the default behavior of selecting from the client's cipher list.

Example
```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLCipherList DSS,SHA -SSLServerCipherOrder"
```

27.10.14 SSLVersionList *list*

Description For newer versions of OpenSSL this option is an alternative to crafting a cipher list excluding compromised SSL or TLS versions, or including them as necessary for compatibility with older peers.

Usage By default, OpenSSL will include SSLv2, SSLv3, TLSv1, TLSv1.1 and TLSv1.2 in its set of ciphers. Provide this option with a comma separated subset list.

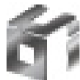
Example
```
dynamic SSLIOP_Factory Service_Object*
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() \
"-SSLVersionList sslv3,tlsv1.2"
```

27.10.15 verbose

Description This option enables debug output during loading of the SSLIOP factory.

Usage The default behavior is equivalent to *not* specifying this option, i.e., no debugging messages get printed.

Impact Specifying this option prints messages equivalent to passing `-ORBDebugLevel 1` on the command line. Note that you cannot use `-ORBDebugLevel` to get the same effect, because protocol factories get loaded before `-ORBDebugLevel` gets processed.



CHAPTER 28

Implementation Repository

28.1 Introduction

The TAO Implementation Repository (ImR) is a service which allows indirect invocation of CORBA operations. Instead of connecting to a server, clients initially connect to the ImR. The ImR will then optionally start the appropriate server if the server is not running, and then provide the address of the server to the client so that subsequent requests will go to this server.

Indirectly binding with the server in this manner can be useful in minimizing the changes needed to accommodate server or object migration, allow for automatic server start-up, and help in load balancing.

Chapter 3 demonstrated how a client uses an object reference to establish a connection with a server. Generally, when a server receives a request, it has to associate the request with one of its servants for processing. The process of a server opening a connection and associating the object reference with a servant is known as binding. TAO servers support both direct and indirect binding.

A variant of the Implementation Repository, discussed in 28.12.2, is available that provides a fault tolerant implementation which supports replication of

state between dual redundant servers, transparent and seamless failover for clients.

Note *The TAO ImR can also manage servers that use JacORB, an open source Java implementation of CORBA. See 28.14 for further information.*

28.1.1 Direct Binding

When using direct binding, the server address and an object key are embedded in an object's IOR, which in URL form might look something like:

```
corbaloc:iiop:1.2@host:port/object_key
```

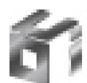
The first time a client invokes an operation on the object, the client ORB extracts the address from this IOR and uses it to connect to a server. The client ORB uses this connection to send requests to the server, and the server ORB uses the object key to locate a servant for the request.

28.1.2 Indirect Binding

When using indirect binding, the ImR address is substituted for the actual server address in the IOR. When the TAO ImR receives a client request it extracts the POA name from the object key, and uses it to find a matching registered server IOR. The ImR then responds with a `LOCATION_FORWARD` message which instructs the client to repeat the request using the new IOR. Subsequent client requests using the same object reference continue to use the established server connection, and the ImR is not involved.

Note *It is possible for a client to create another connection. For example, upon receiving a COMM_FAILURE exception, a client may decide to retry by invoking the operation again. The ImR will again forward the client to the registered server if possible.*

Throughout this chapter, a POA that uses a `PERSISTENT` lifespan policy and a `USER_ID` object id assignment policy is referred to as a persistent POA, and an object that is registered with a persistent POA is referred to as a persistent object. In TAO all persistent POAs are registered with the ImR upon POA creation if the `-ORBUseImR 1` option is specified at ORB initialization. The



`TAO_USE_IMR` environment variable controls the default value for this setting. Furthermore, any object references use indirect binding.

Keep in mind that there is a performance penalty for the first indirect request. If there are a large number of client requests the interaction with the ImR may become a bottleneck in the system.

Frequently this documentation uses the terms POA and server interchangeably, because the ImR treats each registered POA as a separate server. However, each server is capable of hosting multiple persistent POAs, each of which may contain multiple indirect object references.

The ImR supports the ability to launch server processes on demand. Without the ImR all servers must be running prior to any client request. This can result in performance problems, especially for large systems, since servers consume system resources even in their idle states.

The CORBA specification deliberately avoids standardizing the ImR behavior, mentioning only that indirect binding should be supported, and providing the `LOCATION_FORWARD` mechanism. This is sufficient to provide portability from the client perspective, but any server/ImR interaction described in this chapter is specific to TAO.

Note *More information about binding and the ImR can be found in Chapter 14 of Advanced CORBA Programming with C++. TAO's ImR implementation was originally based on the one described in that book.*

28.2 New for patched OCI TAO 2.2a

There are new capabilities available in the ImR that are not available in the base level of OCI TAO 2.2a.

Patch level 4 refined the behavior of the Implementation Repository when dealing with servers registered for manual start mode. A new locator command line option `-lockout` overrides the behavior when a server fails to start after the specified number of restart attempts. When -lockout is absent, the locator counts retries for a single engagement. When supplied, if any engagement runs out of restart attempts, user intervention is required to reset the server's restart counter. In either case, the counter is reset upon a successful start of the server.

Patch level 1 added new `tao_imr` utility commands that give more control over the life cycle of registered servers.

First is a command to identify groups of POAs as being members of the same server application. The new command is `link` ensures that the ImR will not start more than one instance of a server even if it received concurrent requests for different POAs. Linking POAs also means that a command line, working directory, etc. need only be entered once for all POAs, removing the possibility of inconsistent start up options.

Second is a command to send an explicit termination signal to a server process. This command is `kill` and takes an optional signal number. This is useful if a server is unresponsive to a shutdown command. Shutdown is a cooperative action, the ImR sends a CORBA request to the server to shut down that has to be acted on. If the server is in a state where it cannot process that request, it will not shut down. The kill command is implemented by the activator, so obviously it is only available when the ImR activator is running and was used to launch the server.

The behavior of the `list` command is slightly different in the patch 1 release. The difference is in how servers that are running but not completely ready to use are reported when `list -a` is used. Before patch 1, these servers were not listed as being alive. Now they are shown as being alive but with the qualifier `(maybe)` added to the server name. Server aliveness is a tri-state value. A server is definitely alive if it is ready to handle requests. It is definitely not alive if the process cannot be contacted. It is maybe alive if the server process may be contacted, but it isn't ready to handle requests.

28.3 The Operation of the ImR

ImR requires that an IOR it works with live beyond the server that created the IOR. Therefore, the IOR must be created using a POA with the `PERSISTENT` life span policy. If the server's ORB was initialized using `-ORBUseIMR 1` (or with `TAO_USE_IMR` set to 1), then the persistent POA registers itself, upon creation, with the ImR. If a server has more than one persistent POA, each is registered as a separate server within the ImR. For each persistent POA, the following information is registered:

- Server Name
- Address

- `ServerObject`

The Server Name takes one of two forms. If the server's ORB is initialized with the `-ORBServerID` option, it is of the form *server_id:poa_name*. If the server ID is not specified, it is simply the POA name. Either way the POA name must match that of the registered server.

The TAO-specific `ServerObject` is a simple object which is activated in the Root POA, and supports `shutdown()` and `ping()` operations.

The address is actually stored as the first part of the `ServerObject` IOR in URL format. (e.g., `corbaloc:iiop:1.2@127.0.0.1:8888/`)

The server must be able to find the ImR to register this information. If an Activator starts the server (see 28.5.1), then this happens automatically. Otherwise, the server can locate the ImR by either passing the `-ORBInitRef ImplRepoService=...` option, setting the `ImplRepoServiceIOR` environment variable, or using the IOR multicast feature. These methods are used to allow servers, Activators, and the ImR utility to find the ImR.

Additionally the ImR maintains the following information about each registered server:

- Activation mode (discussed in 28.6).
- Start-up command
- Environment variables
- Working directory
- Start retry limit

Most of the above information is registered using the `tao_imr` utility, or read from persistent storage when the ImR is started. The start-up command, working directory, and environment variables are optional. However, a start-up command is required for the ImR to start a server process. The ImR uses the Start Retry Limit to automatically reattempt starting a server if it fails to start the first time. How this limit is interpreted depends on the presence of the `-lockout` argument on the ImR locator command line. By default the retry limit is considered for each engagement of the ImR. An engagement being an explicit start command via the `tao_imr` utility, or an implicit start request resulting from a client locate request. When the limit is exceeded, a TRANSIENT exception is thrown back to the requestor and the engagement is cleared. The requestor, or another, may try again.

When `-lockout` is an ImR locator command line argument, exceeding the retry limit prevents any future attempts to start the target server until some user intervention. This may be using some other means of externally starting the server or using the `tao_imr` utility to update the server entry, resetting the retry count.

Any server started with the `-ORBUseIMR 1` command line argument will notify the ImR of its availability even if no record for that server exists in the ImR. In that case, the ImR creates an entry with `NORMAL` activation mode, and no start-up options. This can be modified later using the `tao_imr` utility if necessary.

The ImR server registry is indexed by persistent POA name, so no two servers using the same ImR may use the same persistent POA name. To differentiate instances of servers that have the same persistent POA, the server instances must use `-ORBServerId <serverid>`. In this case, the server is referenced using "serverid:poa name" in various `tao_imr` commands.

If a server dies unexpectedly while communicating with a client, then the client may get a `COMM_FAILURE` exception. If the client simply re-attempts the operation, then it is once again directed through the ImR, which results in a `TRANSIENT` exception unless start-up information is registered for the server.

Note *The* `ping()` *operation described in 28.5 is disabled if no start-up information is registered, because the ImR would only return a* `TRANSIENT` *exception anyway.*

28.3.1 Basic Indirection

When a server is already running, the steps involved in a client sending requests to a server using the ImR are as follows:

1. Client sends a request to ImR.
2. ImR looks up the server, and sees that it is running.
3. ImR sends `LOCATION_FORWARD` reply to Client.
4. Client sends request to Server.
5. Server sends reply to Client.

Note *The client will continue to use the new address for future invocations until a* `COMM_FAILURE` *or* `TRANSIENT` *exception occurs, after which the client will revert to the address of the ImR until forwarded again.*

28.4 Basic Indirection Example

The following example illustrates how to achieve indirect binding without the ability to start the servers on demand. The source code for this example can be found in `$TAO_ROOT/orbsvcs/DevGuideExamples/ImplRepo/Basic`.

28.4.1 Create the Server

We use a slightly modified version of our typical MessengerServer for this example. The only change we have to make is to use a persistent POA. The `Messenger_i.h` and `Messenger_i.cpp` files remain unchanged.

28.4.1.1 MessengerServer.cpp

```
#include "Messenger_i.h"
#include <iostream>
#include <fstream>

PortableServer::POA_ptr createPersistentPOA (
  PortableServer::POA_ptr root_poa, const char* poa_name)
{

  CORBA::PolicyList policies(2);
  policies.length(2);

  policies[0] = root_poa->create_lifespan_policy (
    PortableServer::PERSISTENT);
  policies[1] = root_poa->create_id_assignment_policy (
    PortableServer::USER_ID);

  PortableServer::POAManager_var mgr = root_poa->the_POAManager();
  PortableServer::POA_var poa = root_poa->create_POA (
    poa_name, mgr.in(), policies);

  policies[0]->destroy();
  policies[1]->destroy();

  return poa._retn();
}
```

```
void writeIORFile(const char* ior)
{
  std::ofstream out("Messenger.ior");
  out << ior << std::endl;
}

int main(int argc, char* argv[])
{
  try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var root_poa = PortableServer::POA::_narrow(obj.in());
    PortableServer::POAManager_var mgr = root_poa->the_POAManager();

    const char* poa_name = "MessengerService";

    PortableServer::POA_var poa = createPersistentPOA(root_poa.in(), poa_name);

    PortableServer::Servant_var<Messenger_i> servant = new Messenger_i;

   CORBA::ObjectId_var object_id = PortableServer::string_to_ObjectId("object");

    poa->activate_object_with_id(object_id.in(), servant.in());

    obj = poa->id_to_reference(object_id.in());
    CORBA::String_var ior = orb->object_to_string(obj.in());

    writeIORFile(ior.in());

    mgr->activate();

    std::cout << "Messenger server ready." << std::endl;

    orb->run();

    std::cout << "Messenger server shutting down." << std::endl;

    root_poa->destroy(true,true);
    orb->destroy();

    return 0;
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Server main() Caught Exception " << ex << std::endl;
  }
  return 1;
}
```

28.4.1.2 MessengerClient.cpp

```
#include "MessengerC.h"
#include <iostream>

int main(int argc, char* argv[])
{
  try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    CORBA::Object_var obj = orb->string_to_object("file://Messenger.ior");

    Messenger_var messenger = Messenger::_narrow(obj.in());
    if (is_nil(messenger.in())) {
      std::cerr << "Unable to get a Messenger reference." << std::endl;
    return 1;

    CORBA::String_var message = CORBA::string_dup("Hello!");
    messenger->send_message("TAO User", "TAO Test", message.inout());
    std::cout << "message was sent" << std::endl;
    std::cout << "Reply was : " << message.in() << std::endl;

    return 0;
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Client main() Caught Exception: " << ex << std::endl;
  }
  return 1;
}
```

28.4.2 Run the Example

Now that we have a modified version of our server we can start several command windows, and demonstrate how it works.

1. Start the `tao_imr_locator`

For simplicity, we set the `ImplRepoServiceIOR` environment variable, which allows applications, such as our server and the `tao_imr` utility, to find the ImR without the need to supply lengthy `-ORBInitRef` options on the command line. Alternatively we could start the ImR with `-m` to allow clients to find it using multicast service discovery.

```
$ ImplRepoServiceIOR=corbaloc::localhost:8888/ImR; export ImplRepoServiceIOR
$ cd $TAO_ROOT/orbsvcs/ImplRepo_Service
$ ./tao_imr_locator -ORBListenEndpoints iiop://:8888
Implementation Repository: Running
        Ping Interval : 10000ms
```

```
Startup Timeout : 60s
Persistence : Disabled
Multicast : Disabled
Debug : 1
Locked : False
```

2. Start the Server

We start the server, passing `-ORBUseIMR 1`, so that it registers itself with the ImR. Alternatively, we could have set the `TAO_USE_IMR` environment variable to make this the default.

```
$ MessengerServer -ORBUseIMR 1
Messenger server ready.
```

The ImR output shows that our server was automatically registered.

```
ImR: Server MessengerService is running at corbaloc:iiop:1.2@192.168.1.10:1323/
ImR: Auto adding NORMAL server:<MessengerService>
```

3. Run the Client

```
$MessengerClient
message was sent
Reply was : A reply.
```

The ImR output shows that it forwarded the client to our server.

```
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1323/…NUP…MessengerService…object>
```

(*Some parts of the IOR above replaced with ellipsis for brevity*)

The server output shows a successful invocation.

```
Message from: TAO User
Subject:      TAO Test
Message:      Hello!
```

28.5 Server Start-up

The exact conditions under which the ImR decides to start a server are determined by the Activation Mode described in 28.5.1, but typically the ImR

starts a server if it is not registered as running, or if the server cannot be pinged successfully. Once the server has been started, the ImR waits for the server to register its running information, which includes a `ServerObject` and a partial IOR. Multiple simultaneous client invocations are supported, and each blocks, waiting on the server to register this information. Each persistent POA in the server is treated as a separate server registration within the ImR unless they are linked via the `tao_imr link` command. POAs register with the ImR as soon as they are created.

Once the server registers its running information, the ImR wakes one of the waiting client operations, and uses it to ping the server to ensure that it is really running. The result of the ping determines whether the server is running, not running, or is in an indeterminate state.

- Running

  If the server is running, or if the start-up retry count has been exceeded, then all clients are awakened and forwarded to the server. By connecting the clients to the server if the retry count was exceeded the clients can get the appropriate error status directly from the server.

- Not Running

  If the server is not running, then the whole start-up process is repeated if a start-up retry count is configured for the server.

- Indeterminate

  If the status cannot be determined, the ping repeats a fixed number of times with an increasing delay between subsequent attempts.

To more efficiently handle multiple client requests, the `ping()` operation has a defined interval (see the -v command line option) that is passed to the ImR at start-up. If a ping has successfully completed within the specified interval, then the server is assumed to be running.

The `ping()` operation has a very short timeout configured (see the -g command line option), and a timeout is considered proof that the server is running. A well-written server should avoid activating the POA Manager until the server is actually ready to handle requests.

In some instances, a server may not register its running status with the ImR within the allowed start-up time. (See the `-t` command line option for the ImR.) This is treated as a start-up failure, and start-up may be retried as described previously.

If the start-up retry count is exceeded then a server becomes locked, and the ImR will not attempt to start that server again. Instead it immediately returns a `TRANSIENT` exception to any waiting clients. You can reset the start-up retry count using the `tao_imr` utility.

If a server dies unexpectedly while communicating with a client, then the client may get a COMM_FAILURE exception. If the client simply reattempts the operation, then it is once again directed through the ImR, which may allow the ImR to restart the server. However, if the server was pinged successfully within the configured ping interval, then the client may be redirected to the dead server, and receive a TRANSIENT.

28.5.1 Using An Activator

The ImR has the ability to start servers using a separate process called the ImR Activator. The Activator is a simple program that runs on the same machine as a server, and is capable of spawning processes. The ImR sends the command line, working directory, and environment variables to the Activator which uses this information to start the server. The Activator returns immediately, and does not maintain any kind of ownership of the launched processes.

Note *On UNIX and UNIX-like platforms the Activator can monitor the launched processes for cleanup purposes, and can optionally notify the ImR when processes die.*

The Activator automatically sets the `ImplRepoServiceIOR` and `TAO_USE_IMR` environment variables for all spawned processes. This means that servers started by the Activator do not need to pass `-ORBUseIMR 1`, nor do they have to specify an `-ORBInitRef ImplRepoService=...` option.

A server started by the Activator typically responds to the first ping with a `TRANSIENT` exception indicating that the POA is in a holding state, because the POA Manager has not been activated. This causes the ImR to retry the ping. This second ping typically times out, which is treated as a successful ping. This is because you do not get a second `TRANSIENT` exception unless `orb->run()` has been called, or another request is sent from the server.

The steps involved in a client being able to connect to a non-running server using an ImR and Activator are given below.

The first task is to activate the server using the Activator:

1. Client sends a request to ImR.
2. ImR looks up the server, and sees it is not running.
3. ImR sends a start request to ImR Activator.
4. Activator starts the Server.
5. Activator sends a start reply to ImR.
6. Server sends a running request to ImR.
7. ImR sends a `ping()` request to Server.
8. Server sends a `ping()` reply to ImR.

With the server waiting for requests the ImR can then inform the client to send requests to the server:

1. ImR sends a Location Forward reply to client.
2. ImR sends a running reply to Server.
3. Client sends a request to Server.
4. Server sends a reply to Client.

Note *Future versions of the ImR may change the POA implementation so that the server registers its running information when the POA Manager is activated instead of when the POA is created. This would eliminate steps #7 and #8 above, because registration of running information will be treated as a successful ping.*

28.5.2 Activator Example

The basic indirection example was enough to demonstrate the primary usage of the ImR, but often the ImR is also used to automatically start servers as needed. This example illustrates the use of the ImR and ImR Activator to start servers on demand. For this to work, an ImR Activator must be started and running on the same host as the server. We must also register a valid command line for our server using the `tao_imr` utility.

Our source code remains unchanged from the example in 28.4. However, the script to run the example is different and can be found in `$TAO_ROOT/orbsvcs/DevGuideExamples/ImplRepo/Activator`.

28.5.2.1 Run the Example

1. **Start the `tao_imr_locator`**

```
$ ImplRepoServiceIOR=corbaloc::localhost:8888/ImR; export ImplRepoServiceIOR
$ cd $TAO_ROOT/orbsvcs/ImplRepo_Service
$ ./tao_imr_locator -ORBListenEndpoints iiop://:8888
Implementation Repository: Running
        Ping Interval : 10000ms
        Startup Timeout : 60s
        Persistence : Disabled
        Multicast : Disabled
        Debug : 1
        Locked : False
```

2. Start an Activator

The activator must be started on the same host on which we want to run the server.

```
$ $TAO_ROOT/orbsvcs/ImplRepo_Service/tao_imr_activator
ImR Activator: Starting MYHOST
ImR Activator: Registered with ImR.
```

The ImR output shows that our activator registered correctly.

```
ImR: Activator registered for MYHOST.
```

3. Register the server

We assume `$ACE_ROOT/bin`, where the `tao_imr` utility is located, is in your `PATH`. We accept the defaults for the Activator name, and other start-up options. The server's command line does **not** require the `-ORBUseIMR 1` or `-ORBInitRef ImplRepoService=corbaloc::localhost:8888/ImR` options, because these will be supplied automatically by the Activator.

```
$ tao_imr add MessengerService -c "MessengerServer"
Successfully updated <MessengerService>.
```

The ImR output shows that we added the server.

```
ImR: Add/Update server <MessengerService>
```

4. Run the Client

In our example, the client relies on the existence of a file named `messenger.ior` that contains the Messenger object's IOR. You must either run the server once to generate the file, or use the file created in the basic indirection example.

```
$ MessengerClient
message was sent
Reply was : A reply.
```

The ImR output shows that it forwarded the client to our server.

```
ImR: Starting server <MessengerService>. Attempt 1/1.
ImR: Waiting for <MessengerService> to start...
ImR: Server MessengerService is running at corbaloc:iiop:1.2@192.168.1.10:1323/.
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1323/…NUP…MessengerService…object>
```

(*Some parts of the IOR above replaced with ellipsis for brevity.*)

The ImR Activator output shows that it successfully started the server.

```
ImR Activator: Successfully started <MessengerService>.
```

The server output shows a successful invocation.

```
Message from: TAO User
Subject:      TAO Test
Message:      Hello!
```

28.6 Activation Modes

The ImR supports registering servers with one of four different activation modes. These affect how a server is started, and have no meaning if a server is not startable. For a server to be startable, it must have a registered command line and Activator name, and the corresponding Activator must be registered and running.

Note *If a server registers itself automatically then no Activator or command line will be associated.*

You must configure the activation mode using the `tao_imr add` or `update` command. Alternatively, you may stop the ImR, edit the XML or Windows registry persistent data manually, then start the ImR to initiate the changes.

The ImR has the ability to start a server on demand, at ImR start-up, or in response to commands from the `tao_imr` utility.

Valid activation modes are:

- `normal`

  The common usage of the ImR is to have it automatically start any servers as needed for incoming client requests. This mode also allows servers to be manually started using the `tao_imr start` command.

- `auto_start`

  This behaves exactly like normal mode, except that the ImR attempts to start the server as the ImR itself is started. You can also use the `tao_imr autostart` command to manually start these servers.

- `manual`

  This prevents the server from starting automatically as the result of an incoming client request. You must use the `tao_imr start` command to start the server, or start the server manually using some other mechanism.

- `per_client`

  The name of this mode can be misleading, because the ImR does not actually keep track of which clients have attempted to use a server. Instead this simply means that a new server will be spawned for each incoming request to the ImR. Once a client has been forwarded by the ImR to an actual server, the client will continue to use this connection for future communications until a `TRANSIENT` or `COMM_FAILURE` exception occurs. In this case, the client will make a connection back to the ImR, and the process will repeat.

Note *It is possible for a client to make a second connection using the indirect object reference, and this will cause the ImR to launch another server. For this reason,* `per_client` *activation should be used with care.*

Note *See 28.7 for special steps that must be taken to use* `per_client` *activation with an IOR Table.*

28.7 Using the ImR and the IOR Table

The IOR Table described in Chapter 13 can also be used in servers that use the ImR. Recall that using the IOR Table costs an additional level of indirection with most servers. The first time a client accesses the server, it is forwarded by the IOR Table back to the server itself using the bound IOR. When combined with the ImR, the result is two levels of indirection. First the client is forwarded by the ImR to the server, and then the server forwards the client back to itself using the IOR Table.

Servers which use the ImR always change all object references associated with persistent POAs to point to the ImR (i.e., indirect binding). This needs to be done in case the server later terminates and needs to be restarted. In this case the IOR registered in the IOR Table points at the ImR, which causes another level of indirection. To avoid this extra cost, you can use a TAO specific function to create a direct bound object reference for use in the IOR Table.

```
TAO_Root_POA* tpoa = dynamic_cast<TAO_Root_POA*>(poa.in());
CORBA::Object_var obj = tpoa->id_to_reference_i(id, false);
CORBA::String_var ior = orb->object_to_string(obj.in());
```

You **must** use this code for `per_client` activation, or two servers will be started for each client invocation.

The `tao_imr` utility can be used to generate IORs for any registered server even if the server is not running. It simply constructs a `corbaloc` object URL using a simple object key passed as an argument. (e.g., `corbaloc:iiop:1.2@127.0.0.1:8888/MyPOA`). For this to work, you must register the simple key with the IOR Table in your server.

You can also bind multiple IORs to simple object keys for each persistent POA. Simply use a forward slash '/' to separate the POA name from the object key.

```
iorTable->bind("MessengerService/Messenger1", ior1);
iorTable->bind("MessengerService/Messenger2", ior2);
```

This is accessed using an object URL such as:

```
corbaloc:iiop:1.2@127.0.0.1:8888/MessengerService/Messenger1
```

This object URL can be generated using the following `tao_imr` command:

```
tao_imr ior MessengerService/Messenger1
```

For the best performance, you should avoid use of the IOR Table due to the extra level of indirection it introduces. To ensure that clients always access your server in the most efficient way possible, do not register anything with the IOR Table. This forces clients to use IORs, or object URLs with full object keys.

To ensure that your object references are always valid, you must be sure to start the server on a consistent endpoint. For example:

```
-ORBListenEndpoints iiop://:9999
```

To make the IORs more human-readable, you may also want to use the `-ORBObjRefStyle URL` option (see 17.13.52).

28.7.1 The Steps in Using an IOR Table

When a client sends a request the ImR that will become fulfilled by a server using an IOR Table that is already running, the following steps are taken to allow the client to send a request to the server:

1. Client sends a request to ImR.
2. ImR looks up the server, and sees that it is running.
3. ImR sends a `ping()` request to Server.
4. Server sends a `ping()` reply to ImR.
5. ImR sends a Location Forward reply to Client.
6. Client sends a request to Server.
7. Server sends a Location Forward reply to Client.
8. Client sends a request to Server.
9. Server sends a reply to Client.

28.8 ImR and IOR Table Example

We modify our first example to show how to support an IOR Table. We also show how to register more than one object in a persistent POA, and we expose

these objects using simple URLs. The source code for this example can be found in
`$TAO_ROOT/orbsvcs/DevGuideExamples/ImplRepo/IORTable`.

28.8.1 Create the Server

We modify our existing `MessengerServer.cpp` to support our new requirements.

28.8.1.1 MessengerServer.cpp

```
#include "Messenger_i.h"
#include <tao/PortableServer/POA.h>
#include <tao/IORTable/IORTable.h>
#include <iostream>
#include <fstream>

PortableServer::POA_ptr createPersistentPOA(
  PortableServer::POA_ptr root_poa, const char* poa_name)
{

  CORBA::PolicyList policies(2);
  policies.length(2);

  policies[0] = root_poa->create_lifespan_policy (
    PortableServer::PERSISTENT);
  policies[1] = root_poa->create_id_assignment_policy (
    PortableServer::USER_ID);

  PortableServer::POAManager_var mgr = root_poa->the_POAManager();
  PortableServer::POA_var poa = root_poa->create_POA (
    poa_name, mgr.in(), policies);

  policies[0]->destroy();
  policies[1]->destroy();

  return poa._retn();
}

void writeIORFile(const char* ior, const char* name)
{
  std::ofstream out(name);
  out << ior << std::endl;
}

int main (int argc, char *argv[])
{
  try {
```

```
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

  CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
  PortableServer::POA_var root_poa = PortableServer::POA::_narrow(obj.in());

  PortableServer::POAManager_var mgr = root_poa->the_POAManager();

  const char* poa_name = "MessengerService";

  PortableServer::POA_var poa = createPersistentPOA (root_poa.in(), poa_name);

  PortableServer::Servant_var<Messenger_i> servant1 = new Messenger_i;
  PortableServer::Servant_var<Messenger_i> servant2 = new Messenger_i;

  CORBA::ObjectId_var id1 = PortableServer::string_to_ObjectId("Messenger1");
  poa->activate_object_with_id(id1.in(), servant1.in());
  CORBA::ObjectId_var id2 = PortableServer::string_to_ObjectId("Messenger2");
  poa->activate_object_with_id(id2.in(), servant2.in());

  obj = poa->id_to_reference(id1.in());
  CORBA::String_var ior1 = orb->object_to_string(obj.in());
  obj = poa->id_to_reference(id2.in());
  CORBA::String_var ior2 = orb->object_to_string(obj.in());

  TAO_Root_POA* tpoa = dynamic_cast<TAO_Root_POA*>(poa.in());
  obj = tpoa->id_to_reference_i(id1.in(), false);
  CORBA::String_var direct_ior1 = orb->object_to_string(obj.in());

  obj = orb->resolve_initial_references("IORTable");
  IORTable::Table_var ior_table = IORTable::Table::_narrow(obj.in());
  ior_table->bind("MessengerService/Messenger1", direct_ior1);
  // Bind using an indirect reference.
  ior_table->bind("MessengerService/Messenger2", ior2);

  writeIORFile(ior1.in(), "Messenger1.ior");
  writeIORFile(ior2.in(), "Messenger2.ior");

  mgr->activate();

  std::cout << "Messenger server ready." << std::endl;

  orb->run();

  std::cout << "Messenger server shutting down." << std::endl;

  root_poa->destroy(true,true);
  orb->destroy();

  return 0;
}
```

```
  catch (CORBA::Exception& ex) {
    std::cerr << "Server main() Caught Exception" << ex << std::endl;
  }
  return 1;
}
```

28.8.1.2 MessengerClient.cpp

```
#include "MessengerC.h"
#include <ace/SString.h>
#include <iostream>

int main (int argc, char* argv[])
{
  try {
    if (argc <= 1) {
      std::cerr << "Error: Must specify the name of an IOR file." << std::endl;
      return 1;
    }
    ACE_CString ior = "file://";
    ior += argv[1];

    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    CORBA::Object_var obj = orb->string_to_object(ior.c_str());

    Messenger_var messenger = Messenger::_narrow(obj.in());
    if (CORBA::is_nil(messenger.in())) {
      std::cerr << "Unable to get a Messenger reference." << std::endl;
      return 1;
    }

    CORBA::String_var message = CORBA::string_dup("Hello!");
    messenger->send_message("TAO User", "TAO Test", message.inout());
    std::cout << "message was sent" << std::endl;
    std::cout << "Reply was : " << message.in() << std::endl;

    return 0;
  }
  catch (CORBA::Exception& ex) {
    std::cerr << "Client main() Caught Exception: " << ex << std::endl;
  }
  return 1;
}
```

28.8.2 Run the Example

Now that we have a modified version of our server we can start several command windows, and demonstrate how it works.

1. Start the `tao_imr_locator`

For simplicity, we set the `ImplRepoServiceIOR` environment variable, which allows applications, such as our server and the `tao_imr` utility, to find the ImR without the need to supply lengthy `-ORBInitRef` options on the command line.

```
$ ImplRepoServiceIOR=corbaloc::localhost:8888/ImR; export ImplRepoServiceIOR
$ cd $TAO_ROOT/orbsvcs/ImplRepo_Service
$ ./tao_imr_locator -ORBListenEndpoints iiop://:8888
Implementation Repository: Running
        Ping Interval : 10000ms
        Startup Timeout : 60s
        Persistence : Disabled
        Multicast : Disabled
        Debug : 1
        Locked : False
```

2. Start the Server

We start the server, passing `-ORBUseIMR 1`, so that it registers itself with the ImR.

```
$ MessengerServer -ORBUseImR 1
Messenger server ready.
```

The ImR output shows that our server was automatically registered:

```
ImR: Server MessengerService is running at corbaloc:iiop:1.2@192.168.1.10:1323/
ImR: Auto adding NORMAL server:<MessengerService>
```

3. Run Client 1

```
$ MessengerClient messenger1.ior
message was sent
Reply was : A reply.
```

The ImR output shows that it forwarded the client to our server:

```
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1323/…NUP…MessengerService…Messenger1>
```

(*Some parts of the IOR above were replaced with ellipsis for brevity.*)

The server output shows a successful invocation:

```
Message from: TAO User
Subject:      TAO Test
Message:      Hello!
```

4. Run Client 2

```
$ MessengerClient messenger2.ior
message was sent
Reply was : A reply.
```

The ImR output shows that it forwarded the client to our server:

```
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:8888/…NUP…MessengerService…Messenger2>
```

The server output is the same as above.

5. Create `corbaloc` object URLs

Here we demonstrate how to use the `tao_imr` utility to create an IOR, and also how to create one manually. We can create the IOR manually, because we started the ImR on a known endpoint.

```
$ tao_imr ior MessengerService/Messenger1 -f messenger3.ior
corbaloc:iiop:1.2@192.168.1.10:8888/MessengerService/Messenger1
$ echo corbaloc::localhost:8888/MessengerService/Messenger1 > messenger4.ior
```

6. Run Client 3

```
$ MessengerClient messenger3.ior
message was sent
Reply was : A reply.
```

The ImR output shows that this time we forward using a simple object key instead of a full-blown IOR. This means the server will have to forward the client again using its IOR Table.

```
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1272/MessengerService/Messenger1>
```

The server output is the same as above.

7. Run Client 4

```
$ MessengerClient messenger4.ior
message was sent
Reply was : A reply.
```

This time we can see that the ImR has to forward two invocations. This is because the server's IOR Table contains an indirect object reference.

```
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1272/MessengerService/Messenger2>
ImR: Forwarding invocation on <MessengerService> to
<corbaloc:iiop:1.2@192.168.1.10:1272/…NUP…MessengerService…Messenger2>
```

The server output is the same as above.

28.9 Advanced Examples

You may want to experiment with Activator example using the various Activation Modes, and other server start-up options. This should not require any source code changes. You can also experiment with the ImR persistence, start-up timeout, and ping interval settings.

You can find additional ImR tests and examples for TAO in `$TAO_ROOT/orbsvcs/examples/ImR` and `$TAO_ROOT/orbsvcs/examples/ImplRepo`.

28.10 Repository Persistence

The ImR can load and save its list of registered servers and Activators to persistent storage using one of three formats. The easiest to work with is an XML format that can be edited by hand. The XML file is rewritten in response to any change in registration information, and may therefore be inefficient for large, or very busy, repositories. A more efficient binary format is supported, but this can be difficult to work with, and is known to have problems on some platforms. The ImR can also save its registration information to the registry on Windows systems.

- XML

Starting the ImR with `-x repo.xml` creates a file containing an entry for every registered server and Activator. The schema for this XML file can be found at `$TAO_ROOT/orbsvcs/ImplRepo_Service/ImR.xsd`.

- Binary

Starting the ImR with `-p repo.bin` stores registered Activators and servers in a binary file.

Note *This option may have problems on some platforms when the memory mapped file used internally needs to be expanded.*

- Windows registry

Starting the ImR with `-r` stores registered servers and Activators at:

```
HKEY_LOCAL_MACHINE\SOFTWARE\TAO\ImplementationRepository
```

28.11 ImR Utility

TAO provides a command line tool called `tao_imr` that you can use to

- add or edit server information in the ImR,
- create IORs suitable for connecting to a server,
- view the status of registered servers,
- start or shutdown registered servers.

The full path to the `tao_imr` utility is `$ACE_ROOT/bin/tao_imr`.

28.11.1 Command Line Options

The general syntax for the ImR utility is

```
tao_imr command [options] [args]
```

Most of the commands take a server name as an argument. You can get help information on each command via the -h option. For example:

```
$ tao_imr start -h
```

```
Starts a server using its registered Activator.

Usage: tao_imr [options] start <name>
  where [options] are ORB options
  where <name> is the name of a registered POA.
  -h Displays this
```

The commands and options that can be passed to `tao_imr` are summarized in the table below, followed by a detailed description of each command.

Table 28-1 Command Line Options for `tao_imr`

Command	Option		Description
add / update	-a	**normal**	Start server on client invocation.
		manual	Start server with `tao_imr activate`.
		per_client	Start a new server for each client.
		auto_start	Start server when ImR is started or with `tao_imr auto_start`.
	-c *cmdline*		Command line string used to start server.
	-e var=value		Set environment variable at server start-up. Repeat this option to add more than one environment variable.
	-l *activator*		The activator to use for starting the server.
	-w *working_dir*		Specify the server's working directory.
	-r *count*		Set the startup/ping retry count to *count*.
autostart			Start all servers with activation mode of `auto_start`.
ior	-f *filename*		Write a corbaloc URL to standard output. Optionally place the newly-created IOR into filename.
kill	-s *signal*		kill the server by sending the specified signal. The default signal number is 9.

Table 28-1 Command Line Options for `tao_imr`

Command	Option	Description
`link`	`-p` *`peer_list`*	Provide a list of peer POAs to link to the named server.
`list`	`-v`	Display the names and status of registered servers. Optionally display registered start-up information.
	`-a`	Only show active servers by pinging the registered servers and excluding those servers that do not respond to the ping.
`remove`		Remove the server registration.
`shutdown`		Shutdown the server.
`shutdown-repo`	`-a`	Shutdown the ImR, and optionally all registered activators.
`start`		Start the server.

28.11.1.1 add/update

Synopsis

```
tao_imr add|update <server> [-a normal|manual|per_client|auto_start] [-c cmdline]
[-e var=value] [-w working dir] [-r count] [-l activator]
```

These commands are used to add a new server registration and to update an existing server registration respectively. They take exactly the same options. The only difference is that `add` cannot be used to change an existing server. This means you can use `update` to add new servers.

You may specify an activation mode of `auto_start`, `manual`, `normal`, or `per_client`. The default activation mode is `normal`.

Note *It is possible for a client to make a second connection using the indirect object reference, and this will cause the ImR to launch another server. For this reason,* `per_client` *activation should be used with care.*

The command line, working directory, environment variables, and retry count are all optional, but you must specify at least a command line if you want the

ImR to be able to start the server. The command line can be any command, and does not necessarily have to directly start the server. For example, it could run a script that causes the creation of a new persistent POA in an already running server process.

You can specify the `-e` option multiple times on a single command line to set multiple environment variables. The specified environment variables will replace any that currently exist when updating.

The working directory does not affect the command line. If the command is not in the path, then specifying the working directory does not allow the command to run. Therefore, you must also specify the directory to the cmdline argument.

The retry count is a feature to prevent further attempts to start servers that are not functioning correctly. If a server either cannot be started by the Activator, or does not register its running information with the ImR in a timely manor, then the ImR will attempt to start it again. The start-up timeout is a command line option to the ImR that is shared by all servers. An additional feature of the update command is that it always resets the start-up count for a server.

You can specify an activator for a server, and this defaults to the hostname of the machine on which the `tao_imr` utility is run. If you use the `-n name` feature of the Activator then you must use the same name here so that the ImR can find the correct Activator to start the server. Any auto-added servers will not have an Activator set, so the `tao_imr` update command must be used to set it if you want to make the server startable.

Note *In previous versions of the ImR, you could not change the Activator for a registered server. This restriction was due to an internal implementation detail that has since changed.*

28.11.1.2 autostart

Synopsis `tao_imr autostart <server>`

This command is used to start all servers registered with the `auto_start` activation mode. In other respects it works exactly the same as the start command.

28.11.1.3 ior

Synopsis

```
tao_imr ior <server> [-f filename]
```

This command can be used to construct a valid simple corbaloc URL for accessing a server registered with the ImR. This is only useful if the address of the ImR is unknown, because, for example, you are using multicast to find the ImR. If the address of the ImR is known, then it is easier to construct the URL manually using the form:

```
corbaloc:protocol:host_or_ip:port/ServerName
```

The `ior` command does not actually contact the ImR to lookup the address of the server. Instead it uses the first available protocol specified for the ImR connection.

28.11.1.4 kill

Synopsis

```
tao_imr kill <server> [-s <signal>]
```

Kill the identified server by having its activator send the specified signal, or a signal 9 if none is supplied.

This is useful if a server is unresponsive to a shutdown command. Shutdown is a cooperative action, the ImR sends a CORBA request to the server to shut down that has to be acted on. If the server is in a state where it cannot process that request, it will not shut down. The kill command is implemented by the activator, so obviously it is only available when the ImR activator is running and was used to launch the server.

28.11.1.5 link

Synopsis

```
tao_imr link <server> [-p <peers>] ...
```

Use link to create an association between two or more POAs in a server.

Servers may have more than one POA managing objects that are registered with the ImR. This poses a risk of having multiple instances of the service started on concurrent requests. Since start up is a multi-step process, one client could request an object on POA1 while another client requests an object on POA2. The result would be two server instances running, each with a POA1 and POA2, and thus the contact information for the first instance would be overwritten by the second. And if that first server instance shut down, it would

communicate to the ImR, causing the ImR to believe no instance of the server is running.

To avoid this situation, the `link` command is used to associate POA2 with POA1 so that a request for an object on either POA will trigger a launch of the same server process. A subsequent request for the other POA will simply wait for the server to complete its startup without launching another

To use the `link` command, first `add` or `update` a server entry for one of the POAs. It doesn't matter which, the POAs are considered peers in the ImR regardless how they are related in the server itself. This first POA is considered the owner of the server registration. Second, issue the link command. The peer POAs may be added individually or in a comma separated list. For example a server application named myapp contains three POAs, named myserver1, secondserv, and anotherserv. The following steps show how to link them

```
tao_imr add myserver1 -c "./myapp -ORBUseIMR 1 ..."
tao_imr link myserver1 -p secondserv,anotherserv
```

Or

```
tao_imr link myserver1 -p secondserv -p anotherserv
```

Or

```
tao_imr link myserver1 -p secondserv
tao_imr link myserver1-p anotherserv
```

These variations of the link command are equivalent.

POAs may be removed from the group by using the remove command, except for the owner. If it is remove, all of the linked POAs are removed as well. When listed, linked POAs show the same start up command options.

28.11.1.6 list

Synopsis `tao_imr list [-v] [-a]`

Use this command to list servers registered in the ImR.

You will probably want to use the verbose option `-v` most of the time. All information registered using the add/update commands is displayed as well as the current running status of the server and whether the server is locked due to exceeding its retry count. By status is meant the endpoint the server is assumed to be running, or, if the activator is being used and it detected that the server is not running, then "Not currently running" is shown. If the server is locked, you can unlock it using the `update` command. For example:

```
tao_imr update myserver
```

In some cases the locator or activator may be shutdown and restarted. While being shutdown registered servers may have been terminated. In these cases the ImR is not aware of inactive servers. To ensure that the of servers includes only the active ones, use the `-a` option. When this option is used a check is made for each server if there has been a successful ping within the ping interval discussed in 28.12.1. If there was a successful ping within this interval the server is included in the list. If not, an attempt is made to ping the server. If this ping was successful then the server is included in the list, shown as being fully active. If the ping failed, but a connection to the server was completed, the server is included in the list, but with a "maybe" status shown.

28.11.1.7 remove

Synopsis `tao_imr remove <server>`

This command simply removes all information about the server from the ImR. If the server is running it is not shut down by this command. If the server is removed without being shut down first, then a `NOT_FOUND` exception will be caught by the server when it tries to unregister from the ImR. This exception is ignored, but an error message is displayed.

28.11.1.8 shutdown

Synopsis `tao_imr shutdown <server>`

This command shuts down a running server by using the `ServerObject` that every server internally registers with the ImR at start-up. The `orb->shutdown(0)` operation is called in the server, typically causing the server to return from `orb->run()` and shut down gracefully. Note that this means servers with multiple persistent POAs can be shut down using any one of the POA names. If this behavior is not desired, then you should use separate ORBs in the server.

28.11.1.9 shutdown-repo

Synopsis `tao_imr shutdown-repo [-activators]`

This command shuts down the ImR cleanly. This can also be done by using `Ctrl-C`, or sending a `SIGINT` or `SIGTERM` to the ImR and/or Activator processes.

The `-a` option specifies that you also want to attempt to shut down any registered Activators. If any Activator cannot be shut down, it is ignored, and no error is returned.

28.11.1.10 start

Synopsis
```
tao_imr start <server>
```

This command is used to ensure that the specified server is activated, starting the server process, if necessary, using the Activator registered for the server. If the server is not already running, this requires that

1. The server has already been registered using add or update.
2. The registration information includes a command line and activator name.
3. A matching Activator is registered and running.

If any of the above conditions are not met, or if there is some problem activating the server, then an error message is returned.

Note *This command was previously named* activate*. That command name still works but displays a warning that it is deprecated.*

Note *It is possible for the utility to report success when the ImR exceeds its ping retry count. In this case the server was not activated, and list –v will show that the server is locked.*

28.11.2 Examples

For simplicity, we assume the `ImplRepoServiceIOR` environment variable has been set to the appropriate IOR. This environment variable allows applications, such as our server and the `tao_imr` utility, to find the ImR without the need to supply lengthy `-ORBInitRef` options on the command line. Alternatively, we could start the ImR with `-m` to allow clients to find it using multicast service discovery.

28.11.2.1 list existing servers

```
$ tao_imr list
No servers found.
```

28.11.2.2 add a new server

```
$ tao_imr add mysrv -l myhost -c mysrv -w $SERVERS/mysrv -e env1=1 -e env2=2 -a
per_client -r 1
Successfully registered server <mysrv>
```

28.11.2.3 list again

```
$ tao_imr list
<mysrv>

$tao_imr list -v
Server <mysrv>
  Activator: myhost
  Command Line: mysrv
  Working Directory: $SERVERS/mysrv
  Activation Mode: PER_CLIENT
  Number of retries: 1
  Environment Variable: env1=1
  Environment Variable: env2=2
  No running info available for PER_CLIENT mode
```

28.11.2.4 update the server

```
$ tao_imr update mysrv -l newhost -a normal
Successfully registered <mysrv>.
```

28.11.2.5 start the server

```
$ tao_imr start mysrv
Cannot activate server <TestMessengerService>, reason: <Cannot start server.>

$ tao_imr update mysrv -c "mysrv -ORBUseIMR 1"
Successfully registered <mysrv>

$ tao_imr start mysrv
Successfully activated server <mysrv>
```

28.11.2.6 shutdown the server

```
$ tao_imr shutdown mysrv
Successfully shut down server <mysrv>
```

28.11.2.7 create an ior

```
$ tao_imr ior mysrv
corbaloc:iiop:1.2@10.20.200.123:3223/mysrv
```

28.11.2.8 remove the server

```
$ tao_imr remove mysrv
Successfully removed server <mysrv>
```

28.12 tao_imr_locator

`tao_imr_locator` is the primary application that we generally refer to as the ImR. It is responsible for maintaining a repository of server and activator information, and using it to support indirect binding of CORBA objects.

If the ImR is not started with multicast service discovery support, then you must provide some mechanism for other processes to find the ImR when they use `resolve_initial_references()`. This requires the servers, Activators, and `tao_imr` to be started with an –ORBInitRef option, such as:

```
-ORBInitRef ImplRepoService=corbaloc::host:port/ImR
-ORBInitRef ImplRepoService=corbaloc::host:port/ImplRepoService
-ORBInitRef ImplRepoService=file://ImR.ior
-ORBDefaultInitRef corbaloc::host:port
```

Alternatively, you can set the `ImplRepoServiceIOR` environment variable:

```
$ export ImplRepoServiceIOR=corbaloc::host:port/ImR
```

For most of the above to work, the ImR must be started at a known endpoint.

```
-ORBListenEndpoints iiop://:8888
-ORBListenEndpoints iiop://host:port
```

The full path to the ImR is:

```
$TAO_ROOT/orbsvcs/ImplRepo_Service/tao_imr_locator
```

28.12.1 Command Line Options

Synopsis

```
tao_imr_locator [-h|-?] [-d 0..5] [-l] [-m] [--lockout] [-o filename] [-p
filename] [-r] [-x filename] [-i] [-g timeout] [-v interval] [-t timeout]
tao_imr_locator [-c cmd] [-d 0|1|2] [-m] [-o file]
```

```
[-r|-p file|-x file] [-s] [-t secs] [-v secs]
```

Table 28-2 tao_imr_locator Command Line Options

Option	Description
`-h or -?`	Display help/usage.
`-d level`	Specify IMR specific debugging level, 0 to 5. Default is `0`.
`-l`	Lock the database to prevent `tao_imr add`, `update`, and `remove`.
`-m`	Enable support for IOR multicast to find the ImR.
`-o filename`	Write the ImR Administration IOR to `filename`.
`-p filename`	Use binary persistence.
`-r`	Use Windows registry persistence.
`-x filename`	Use XML persistence.
`-i`	Enable active pinging
`-g timeout`	Ping operation timeout in milliseconds. Default is 1 second.
`-v interval`	Ping interval in milliseconds. Default is 10 seconds.
`-t timeout`	Start-up timeout in seconds. Default is 60 seconds.
`-c command`	`install` or `remove` the Windows service.
`-e`	Erase all persistence information at start-up.
`--lockout`	Prevent restart attempts until restart count is explicitly reset
`-u or --UnregisterIfAddressReused`	This option causes the ImR to automatically remove a server from the implementation repository when another server is registered with the same endpoint. This option should not be used if more than one persistent POA in the same ORB is being used via the ImR.
`--primary`	Take on the role of primary service in a redundant service pair.

Table 28-2 tao_imr_locator Command Line Options

Option	Description
`--backup`	Take on the role of backup service in a redundant service pair.
`--directory` *`dirname`*	Use a shared file system directory with name `dirname` as the backing store for redundant tao_imr_locator service pair.

The debug level varies from 0 to 5 with progressively more detail logged for each level. This value is independent of the `ORBDebugLevel`, although the ORB's logging mechanism is used.

The `-l` option prevents the ImR repository information from being modified by the `tao_imr add` or `update` commands. However, server auto-registration, Activator registration, and server running status are all still persisted to the repository.

The `-m` option provides a convenient way for all servers and Activators to find the ImR without the need to pass IORs, use `corbaloc` object URLs, or use the `-ORBInitRef` option. Instead, a multicast endpoint is established when the ImR starts, and servers and Activators automatically find the correct IOR using multicast. The multicast port may be specified using:

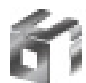
```
-ORBMulticastDiscoveryEndpoint <port>
```

If this option is not specified, then the environment variable `ImplRepoServicePort` is used. The default port number is `10018`.

The output file for the `-o` option is a simple stringified IOR stored in a text file, suitable for use as a `file://` object URL.

The `-v` option allows the ImR to more efficiently work with servers. If the ImR has successfully pinged a server within the specified number of milliseconds, then it assumes that the server is still running. Setting the ping interval to zero disables the ping feature completely, and servers are assumed to be running if they register a running status with the ImR.

Note *In the current implementation for server registration, the server notifies the ImR of its running status when the persistent POA is created. As such, the ImR may sometimes forward a client to a server that is not yet ready to handle requests, thereby causing the client to receive a TRANSIENT exception indicating that the server's POA is in the holding state. The ImR's ping*



feature can be used to allow the ImR to ensure the server is ready to handle requests before forwarding the client to the server. If the ping feature is disabled, the application should be prepared to handle the TRANSIENT exception (e.g., by retrying the request).

The `-t` option allows the user to set a timeout value, in seconds, by which the ImR will wait before giving up on starting a server. If the ImR does not receive running information from a server within the specified timeout interval, it will abort the server start-up. Depending on the server's retry count setting, the ImR may attempt to launch the server again. Once the retry count has been exceeded, the ImR will not attempt to start the server again until an administrator uses `tao_imr update` to reset the server's start-up counter. The start-up timeout value can be disabled by specifying `-t 0` when starting the ImR. However, disabling the start-up timeout is not recommended as the ImR could become blocked permanently waiting for a server to start. The default start-up timeout value is 60 seconds.

The `-UnregisterIfAddressReused` option is intended to solve a specific issue with server restarts when a group of servers, each with one persistent POA, are sharing a common set of ports via the `portspan` endpoint option. Because these servers can swap ports on a restart, the ImR needs to be able to automatically clear them out when a new server takes the port. The `-ORBForwardInvocationOnObjectNotExist` ORB initialization option should be used in the clients of this scenario.

In addition to the normal start-up options, the `-c install` or `-c remove` options can be used to install or remove the ImR as a Windows service. When installing, all other command line options are saved in the Windows registry under:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TAOImR
```

These options are used when the service is started.

28.12.2 Fault Tolerant Implementation Repository

The `tao_imr_locator` can be ran as a fault tolerant service by using the following options:

- The `--primary` option tells the tao_imr_locator that it will be the primary service in a redundant service pair.

- The `--backup` option tells the tao_imr_locator that it will be the backup service in a redundant service pair.
- The `--directory dirname` option tells the tao_imr_locator the shared file system directory to use as the backing store for the redundant service pair.
- When `--backup` is used, the `-o filename` option tells the tao_imr_locator to output the redundant service pair ImR service IOR file, which it can only do after successfully starting the primary and backup ImR service instances. The IOR file contains the combined profiles of the primary and backup tao_imr_locators. Clients must use the IOR file to use the Fault Tolerant ImplRepo Service.

The primary and backup tao_imr_locators instances should have the same ORBEndPoint protocol list so that any client can send a request to either primary or backup regardless of protocol (IIOP,UIOP,etc...).

When the primary server is started it will write out the replication IOR to a file the shared directory. The backup server will then read the IOR file so that it can construct the multi-profile IOR to write to the filename passed to the `-o` option.

The tao_imr_locator primary and backup options cannot be passed on the command line along with `-ORBObjRefStyle URL`, since that style will cause the backup profile to not be available in the IOR.

When starting up the Implementation Repository as a fault tolerant service initially, the primary `tao_imr_locator` must be started first, followed by the backup `tao_imr_locator`. In a case where both the primary and backup are shut down, then either the backup or primary may be restarted independently, reusing the ImR Locator persistent state and IOR which are set up as part of the initial startup process. If you intend to start up the primary or backup on a different host or port, then you will need to restart the process as described above (by starting the primary first). If there is no change in the primary or backup, the then the IORs can continue to be used by any of the existing tao_imr_locator clients.

The previous IOR file will only remain valid if the `ORBEndPoint` list remains the same for both instances. As long as both the primary and backup `tao_imr_locator`s are not shutdown at the same time the fault tolerant

ImR will remain available. If both servers are shut down and one of them is restarted, the server will be available using the originally generated IOR file created with the `-o` option.

28.12.3 Examples

Start the ImR with default options:

```
$tao_imr_locator.exe
Implementation Repository: Running
        Ping Interval : 10000ms
        Startup Timeout : 60s
        Persistence : Disabled
        Multicast : Disabled
        Debug : 1
        Locked : False
```

Start with alternative options:

```
$tao_imr_locator.exe -v 500 -t 5 -x repo.xml -m -d 2 -l
Implementation Repository: Running
        Ping Interval : 500ms
        Startup Timeout : 5s
        Persistence : repo.xml
        Multicast : Enabled
        Debug : 2
        Locked : True
```

Install as a Windows service with several non-default options

```
$tao_imr_locator -v 1000 -t 30 -p repo.bin -ORBListenEndpoints iiop://:8888 -c
install
```

Start a server

```
$myserver -ORBUseIMR 1 -ORBInitRef ImplRepoService=file://imr.ior
```

Start with fault tolerant options

1. Start the primary server
   ```
   $tao_imr_locator --primary -ORBEndpoint localhost:8888 --directory
   imr_shared_dir
   ```
2. Start the backup server
   ```
   $tao_imr_locator --backup -ORBEndpoint localhost:8888 --directory
   imr_shared_dir -o ImR.ior
   ```

28.13 tao_imr_activator

The Activator is an extremely simple process-starting service. It accepts start-up information from the ImR, and attempts to launch a process.

Note *On UNIX and UNIX-like platforms the Activator can detect when spawned processes terminate, and can optionally notify the ImR when this happens.*

Once started, the server registers itself with the ImR directly. Each persistent POA within the server registers itself as a separate server within the ImR.

The start-up information passed to the Activator does not necessarily have to directly start a server. For example, it could run a script that causes the creation of a new persistent POA in an already running server process.

At start-up, the ImR Activator tries to register itself with an ImR. If an ImR cannot be found, the Activator will not be able to notify the ImR when it is shut down or when spawned processes are terminated.

The full path to the `tao_imr_activator` is `$TAO_ROOT/orbsvcs/ImplRepo_Service/tao_imr_activator`.

28.13.1 Command Line Options

Synopsis

```
tao_imr_activator [-c cmd] [-d 0|1|2] [-e buflen] [-o file] [-l] [-n name]
               [-m maxenv]
```

Table 28-3 tao_imr_activator Command Line Options

Option	Description
`-h or -?`	Display help/usage.
`-d` *`level`*	Specify IMR specific debugging level, `0-2`. Default is `1`.
`-o` *`filename`*	Write the Activator IOR to `filename`.
`-n` *`name`*	The name of the Activator. Default is the name of the host on which the Activator is running.

Table 28-3 tao_imr_activator Command Line Options

Option	Description
`-l`	Notify the ImR when spawned processes die. (Not supported on all platforms.)
`-c` *`command`*	`install`, `remove`, or `install_no_imr` the Windows service.
`-e` *`buflen`*	Sets the length in bytes of the environment buffer for activated servers. The default is 16 KB.
`-m` *`maxenv`*	Sets the maximum number of environment variables for activated servers. The default is 512.

There are currently three valid settings for the debug level:

- 0

  Most output is disabled.

- 1

  Very basic information is displayed, with usually just one output per interesting operation.

- 2

  More messages are displayed, and more details are displayed for existing messages.

In practice, debug level 1 is probably the best choice for day-to-day usage, while level 2 is most useful when trying to resolve problems.

The output file for the `-o` option is a simple stringified IOR stored in a text file, suitable for use as a `file://` object URL. This option is not very useful, because typically the ImR is the only client for the Activator.

You can change the `name` of an Activator instance, which allows running more than one Activator on a single machine. This probably is not of much practical value unless you were to create your own Activator class that behaves differently from the default simple process launching service. For example, you might create an activator that spawns servers as threads within its own process.

The -e and -m options are both configuration parameters that control size limitations of the environment passed to spawned processes. The -e option

sets a limit for the size of the buffer (in bytes) that is used to hold the environment variables. The -m option sets a limit for the number of environment variables that can be passed. The default limits are 512 variables and 16 KB of buffer space.

In addition to the normal start-up options, the `-c install`, `-c remove`, and `-c install_no_imr` options can be used to install or remove the Activator as an Windows service. When installing, all other command line options are saved in the Windows registry under:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TAOImR
```

These options are used when the service is started. The `install_no_imr` option should be used when installing the Activator on a separate machine with no ImR, otherwise the installation will ensure that the ImR is always started before the Activator by setting a dependency between the two services.

Note *Only a single Activator can be installed as a Windows service due to the way the start-up options are stored in the registry.*

28.13.2 Examples

Start the ImR Activator with default options:

```
>tao_imr_activator.exe
ImR Activator: Starting MYHOST
... Multicast error message ommitted
ImR Activator: Not registered with ImR.
```

Start the ImR Activator with debug level 2, store the Activator’s stringified IOR in a file, and register with an ImR that was started with multicast discovery enabled:

```
>tao_imr_activator.exe -d 2 -o activator.ior
ImR Activator: Starting MYHOST
ImR Activator: Registered with ImR.
```

Install the ImR Activator as a Windows service on a machine with no ImR:

```
>tao_imr_activator.exe -d 2 -o activator.ior -c install_no_imr
```

28.14 JacORB Interoperability

In an environment where both TAO and JacORB servers coexist, the TAO ImR can also be used to manage the JacORB servers, thus simplifying the task of managing the servers. To take advantage of this feature, make sure you have JacORB 3.3 or later. Consult the "TAO Implementation Repository" chapter in the JacORB's Programming Guide document for more details about integration of JacORB with TAO ImR. You can visit `<http://www.ociweb.com/product-suite/support-jacorb>` to learn more about JacORB, and `<http://www.jacorb.org/download.html>` for obtaining JacORB source code, release notes, programming guide, and other downloads.

Demos and integration test suites for JacORB integration with TAO ImR can be found in $JACORB_HOME/demo/tao_imr and $JACORB_HOME/test/orbreinvoke/tao_imr respectively.

28.14.1 JacORB Servers and the TAO ImR

The `-ORBUseImR` and `TAO_USE_IMR` environment variable as discussed in 28.3 are for TAO based servers only and are not applied to JacORB based servers.

For communicating with a TAO ImR, a JacORB server must turn on the property `jacorb.use_tao_imr` and turn off the property `jacorb.use_imr`. In additional, the JacORB server must set the property `jacorb.implname`.

Note *The use of IOR tables discussed in 28.7 does not apply to JacORB servers.*

28.14.2 Using the TAO ImR Utilility with JacORB Servers

When using the tao_imr utility discussed in 28.11, most of its commands take a server name as an argument. For TAO servers, the "server name" is the name of the qualified POA name; however for JacORB servers, the "server name" has the format `JACORB:<implname>/<qualified POA name>`,
where `<implname>` is the value set by the property `jacorb.implname`.
For example,

```
tao_imr add JACORB:EchoServer1/Message-POA {…other parameters…}
```

would register a JacORB server with the "implname" of "EchoServer1" and the qualified POA name of "Message-POA".

A server's "implname" must be unique in a TAO ImR repository, thus, you can not have an "implname" of "EchoServer1" and a registered POA name of "EchoServer1" which would confuse the matching logic of the TAO ImR. However, you can specify a same qualified POA name for multiple "implname" instances as long as they are uniquely identified. For example, you can add an entry for "EchoServer2" as following:

```
tao_imr add JACORB:EchoServer2/Message-POA {…other parameters…}
```

Note *When using the tao_imr ior command as discussed in 28.11.1.3, the form of the URL is different than that for a TAO server. For a JacORB server, the form is* `corbaloc:protocol:host_or_ip:port/implname/poaName`

Part 5

Appendices

Appendix A

Configuring ACE/TAO Builds

Your build of ACE/TAO can be configured or customized in a number of ways:

- Optional features can be explicitly configured to build. These features are typically disabled because they have external dependencies on other software libraries or products. An example is Secure Sockets Layer (SSL) support. When TAO is configured to build with SSL support, it knows that the OpenSSL library is present and it should compile in support for using this transport layer.
- Normally enabled TAO features can be disabled for specific builds. This is often done for memory footprint reasons or maybe to enforce project policies against using that feature. An example is Portable Interceptors support. By default, TAO supports the interceptor interfaces, but the build can be customized to disable this feature which reduces the footprint and reduces processing overhead.
- The build process itself can be customized in a number of ways, typically through compiler options. Examples of these types of customizations include enabling/disabling support for generating debugging symbols, multithreading, and function inlining.

There are three mechanisms for specifying the above configurations in your build:

- MPC features can be configured in the `default.features` file. This is a user-defined file that is located in `$ACE_ROOT/bin/MakeProjectCreator/config/`.
- GNU Make build flags can be set in the `platform_macros.GNU` file (and via other related mechanisms). This mechanism is not available with other build systems such as Microsoft Visual Studio.
- C++ macros can be set in the `config.h` file.

Some options must be set via a specific mechanism and others can be specified at multiple levels. For example, setting ssl=1 in the `default.features` file instructs MPC to include SSL-dependent projects in the build files generated.

This appendix discusses these different mechanisms and options for configuring builds of ACE and TAO. It discusses how to specify MPC features, GNU Make build flags, and C++ macros as well as some of the options available from each mechanism.

A.1 System Requirements

Since ACE and TAO are used to develop software, you will need a working C++ compiler. The GNU g++ compiler is supported for most platforms, and as of this writing, g++ version 4.2 is recommended. Also, the platform vendor's compiler is usually supported, as are certain third-party compilers. In all cases other than a Windows platform, the GNU Make program must be used. The vendor-provided linker is generally used, but the GNU linker can also be used. In very rare cases, the GNU assembler must be used. To run MPC or any of the automated test scripts, you must have Perl 5.6 or later. A complete list of the tools used to build ACE and TAO on various platforms, including version numbers and patch levels, is available in the release notes. The release notes are included with the ACE and TAO source code distribution in the file `ACE_wrappers/OCIReleaseNotes.html`.

How fast a processor and how much memory you need are somewhat a matter of preference and your patience for waiting on files to compile. Generally, you will want at least a 1.5 GHz processor and at least 1 GB RAM.

The amount of disk space you need will depend heavily on the machine, the compiler, and the build flags you choose. A production system should fit in 100 MB or less of disk space, after all the intermediary files are removed. You may need much less for a minimal build. Conversely, a full build with all debugging information and other features enabled, plus intermediary and temporary files could require over 5.0 GB of disk space.

A.2 Generating Makefiles and Project Files

MakeProjectCreator (MPC) is used to generate build files for use with various tools including GNU Make, Microsoft Visual C++, Borland Make, and Microsoft NMake. MPC eliminates the need to maintain separate build files for these various build environments, and allows easy customization of build flags on all platforms. OCI releases and DOC group full and micro release kits all contain GNU makefiles and Visual C++ project files that were generated with MPC. You are free to use these files to build ACE/TAO, but many users generate their own custom build files using MPC. For more information on how to use MPC, see `<http://www.ociweb.com/products/mpc>`.

A.2.1 Specifying Features to Build

ACE and TAO use the MPC *features* mechanism to define which areas of functionality are enabled and disabled. They also determine what targets the generated build files construct and the details of how those components are built. When running MPC with ACE and TAO (using `$ACE_ROOT/bin/mwc.pl` or `$ACE_ROOT/bin/mpc.pl`), feature-related files are kept in the `$ACE_ROOT/bin/MakeProjectCreator/config` directory. The global feature file, `global.features`, defines the default values for features of ACE and TAO. You can specify any build-specific feature settings by creating a `default.features` file in this same directory. The features set in `default.features` override the values from the global features file. Any features that are not specified in either file are assumed to be enabled.

For example, to tell ACE and TAO to generate build files that compile SSL-related code, including the ACE SSL library and TAO SSLIOP pluggable protocol, place the following in your `default.features` file:

```
ssl=1
```

When using GNU makefiles, you also need to set the SSL makefile build flag to make sure that the libraries are actually built.

The table below summarizes the features that can be specified in the `default.features` file. Note, that disabling a feature may also disable other features, tests, and examples that depend on this feature. Conversely, even though a feature may be specifically enabled in `global.features` and the table below, the actual features may not be enabled in the build system due to other features it is dependent upon. For example, the `ace_flreactor` and `tao_flresource` are listed as enabled below, but they both depend on `fl` which is disabled. Enabling `fl` and generating build files would then physically enable all three features and cause the `ACE_FlReactor` and `TAO_FlResource` libraries to get built. This is typical of the pattern for ACE/TAO features that depend on external products. One feature (like `fl`) announces the availability of the external product, other features control the building of specific ACE/TAO functionality that depends on the external product. Some of the product dependencies are listed below.

To build as intended, most of these features must be combined with the corresponding build flag discussed in the next section.

Table 1-1 MPC features

Feature	Description	Default
`ace_codecs`	Enables building of the ACE encoding/decoding mechanism. Supports Base64 transfer encoding.	enabled
`ace_filecache`	Enables building of the ACE cached virtual file system.	enabled
`ace_flreactor`	Enable support for using a reactor that integrates event handling with the Fast Light Toolkit. Depends on `fl`.	enabled
`ace_for_tao`	Enables building only the subset of ACE features required for TAO.	disabled
`ace_foxreactor`	Enables building of Fox reactor. Depends on `fox`.	enabled
`ace_other`	Enables building of the ACE naming service and NT service-related functionality.	enabled
`ace_qtreactor`	Enables building of Qt reactor. Depends on `qt`.	enabled
`ace_svcconf`	Enables building of the ACE service configurator functionality.	enabled

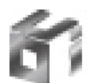


Table 1-1 MPC features

Feature	Description	Default
ace_tkreactor	Enables building of Tk reactor. Depends on tk.	enabled
ace_token	Enables building of the ACE token service.	enabled
ace_uuid	Enables building of the ACE UUID class.	enabled
ace_xtreactor	Enables building of Xt reactor. Depends on xt	enabled
acexml	Enables building ACE XML support. Required for TAO Implementation Repository, Notification Service persistence, and CIAO.	enabled
ami	Enables building TAO Asynchronous Method Invocation support.	enabled
athena	Athena widget libraries are available	disabled
boost	Specifies that the boost library is present. Required for building CIAO's CIDL compiler.	disabled
bzip2	bzip2 library is available	disabled
cidl	Enables building of CIAO's CIDL compiler.	disabled
corba_e_compact	Enable building of the CORBA/e compact profile subset.	disabled
corba_e_micro	Enable building of the CORBA/e micro profile subset.	disabled
corba_messaging	Enables building of TAO's CORBA messaging specification support. This includes AMI and the CORBA policy framework.	enabled
dummy	Dummy feature that prevents certain obsolete components from building. Should never be enabled.	disabled
dummy_label	Dummy feature that prevents certain obsolete components from building. Should never be enabled.	disabled
ec_typed_events	Enables support for typed event channels in the COS Event Service.	enabled
exceptions	Enables use of native C++ exceptions and the standard CORBA C++ error reporting mechanism.	enabled
fl	Fast Light Toolkit libraries are available	disabled
fox	Fox libraries are available	disabled
gen_ostream	This feature causes the IDL compiler to generate ostream operators for IDL-defined types (-Gos)	disabled
gl	OpenGL libraries are available. Depends on fl.	enabled

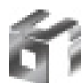


Table 1-1 MPC features

Feature	Description	Default
`interceptors`	Enables support for Portable Interceptors in TAO. Security, fault tolerance, and load balancing features all depend on interceptors.	enabled
`ipv6`	Enables support for IPv6 in ACE and TAO.	disabled
`java`	Specifies that java SDK is present. Required for certain interoperability tests.	disabled
`link_codecfactory`	This feature causes TAO to always link with the `TAO_CodecFactory` library.	disabled
`lzo2`	lzo2 compression library is available.	disabled
`mcpp`	mcpp portable preprocessor is available (for use with TAO IDL compiler)	disabled
`mfc`	Configures ACE/TAO build for use with the Microsoft Foundation Class library. This feature is only for use with Microsoft Visual C++.	disabled
`minimum_corba`	Enabling this feature disables a number of services, tests, and examples that cannot work when TAO is built in a minimum CORBA configuration.	disabled
`motif`	Motif libraries are available	disabled
`negotiate_codesets`	Enabling this feature links TAO clients and servers with the TAO Codeset library, giving them the ability to negotiate codesets with other processes.	disabled
`openssl`	Specifies that the OpenSSL library is available.	enabled
`optimize_collocated_invocations`	When disabled, this feature suppresses generation of collocation optimization stubs by the TAO IDL compiler (for a smaller memory footprint)	enabled
`qos`	Enables build support for the ACE QoS library. Required for the TAO AV Streaming service.	disabled
`qt`	Qt library is available.	disabled
`repo`	Enabling this feature disables some Load Balancing and IDL tests	disabled
`rmcast`	Enables build support for the ACE Reliable Multicast library.	enabled
`rpc`	Specifies that Remote Procedure Call (RPC) is present. Required for RPC-related performance tests.	disabled
`rt_corba`	Enables building of TAO’s Real-Time CORBA specification support.	enabled
`rwho`	Enables building of the Distributed rwho utility.	enabled

Table 1-1 MPC features

Feature	Description	Default
`ssl`	Specifies that Secure Sockets Layer (SSL) is present. Required for ACE SSL and TAO SSLIOP libraries.	disabled
`stl`	The C++ standard template library is available.	disabled
`tao_flresource`	Enables build of Fast Light resource factory (with support for Fast Light Toolkit Reactor). Depends on `fl`.	enabled
`tao_foxresource`	Enables build of Fox resource factory (with support for Fox Reactor). Depends on `fox`.	enabled
`tao_no_iiop`	Causes the TAO library to be built without IIOP support (for small footprint systems that use other protocols.	disabled
`tao_qtresource`	Enables build of Qt resource factory (with support for Qt Reactor). Depends on `Qt`.	enabled
`tao_tkresource`	Enables build of Tk resource factory (with support for Tk Reactor). Depends on `tk`.	enabled
`tao_xtresource`	Enables build of Xt resource factory (with support for Xt Reactor). Depends on `xt`.	enabled
`threads`	Enables multithreading support and components dependent on it.	enabled
`tk`	Tk library is available.	disabled
`transport_current`	Enables building of the TAO's Transport Current feature.	enabled
`uses_wchar`	Enabling this feature disables various examples and utilities that do no support wide character builds.	disabled
`vc8_avoid_dominance_warning`	Suppresses Visual C++ 8.0 dominance warnings.	enabled
`vc8_avoid_unimpl_exception_spec_warning`	Suppresses Visual C++ 8.0 exception specification warnings.	enabled
`vc8_stl_deprecated_warnings`	Suppresses Visual C++ 8.0 C run-time security warnings.	enabled
`vc_avoid_crt_secure_warnings`	Suppresses Visual C++ C run-time security warnings.	enabled
`vc_avoid_this_ini_initializer_warnings`	Suppresses Visual C++ this initializer warnings.	enabled
`vc_scl_secure_warnings`	Suppresses Visual C++ SCL security warnings.	enabled
`vcl`	Borland's Visual Component Library is available.	disabled

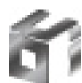


Table 1-1 MPC features

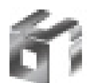
Feature	Description	Default
`versioned_namespace`	Builds ACE with the version number embedded in the ACE namespace.	disabled
`wfmo`	Disabling this feature disables examples that use the Wait For Multiple Objects (WFMO) reactor.	enabled
`wince`	Enables the Front-end for ACE CE (FaCE), a simple front-end framework for testing and debugging non-graphical application on Windows CE.	disabled
`winregistry`	Enables examples dependent on use of the Windows Registry.	enabled
`wxWindows`	Specifies that wxWindows library is present. Required for TAO's wxNamingViewer utility and the ACE Configuration Viewer.	disabled
`x11`	The x11 library is available. Depends on `xt`.	enabled
`xt`	Xt library is available.	disabled
`zlib`	Specifies that the Zlib compression library is present and should be used.	disabled
`zzip`	Specifies that the ZZip compression library is present and should be used.	disabled

A.3 GNU Make Build Flags

When MPC generates GNU makefiles using the default `gnuace` project type, the makefiles support a number of build flags that are used to customize your build. Some of these are identically named as the MPC features discussed in the previous section and simply cause compilation of the projects and/or functionality that their corresponding feature caused to be included in the makefiles.

You can specify any of the GNU Make flags in the `$ACE_ROOT/include/makeinclude/platform_macros.GNU` file or using the `MAKEFLAGS` environment variable as well as on the command line when you invoke make. Because many of the flags require that they be used consistently across your application, the `platform_macros.GNU` file is usually the preferred place to put these flags.

When not using the ACE/TAO GNU makefile system to build your application, it is your responsibility to ensure that compatible environment



variables and compiler options are used in building your application. Many of the build flags effects can also be specified via preprocessor macro definitions in the `$ACE_ROOT/ace/config.h` file.

The tables below lists the GNU Make flags that affect not only the way the ACE and TAO libraries and executables are built, but also the way your applications are built. As indicated in the table's right-most column, when you build your applications, there are certain flags whose settings *must match* the settings that were used when ACE and TAO were built. For example, *inlining*: If ACE and TAO were built to inline functions, then applications must also be built with this setting, or link errors result. Other flags only affect the target being built at that time. For example, *debug*: If ACE and TAO were built with debug enabled, the applications do not have to be built with debug enabled, and vice-versa.

Each of these tables lists the GNU Make flags, a brief description of the flag, the default setting of the flag (for most UNIX and UNIX-like platforms) for building ACE and TAO, and the way applications must use the flag with respect to how ACE and TAO were built. Almost all of the GNU Make flags are enabled by setting them to one (`1`), and disabled by setting them to zero (`0`). They are divided into separate tables based on the different varieties of build flags that ACE and TAO use.

Table 1-2 shows the make flag options that control the details of the ACE, TAO, and application builds. Later tables focus on make flags that control whether specific features of ACE and TAO are built.

Table 1-2 GNU Make Flags - Compilation Control

Flag	Description	Default	Must Match?[1]
`debug`	Enable support for debugging	`1`	N
`fast`	Use `-fast` compiler option (SunCC only)	`0`	N
`inline`	Enable C++ function inlining	`1`	Y
`no_hidden_visibility`	Disables use of hidden visibilities in limiting the export of symbols from libraries.	`0 (GCC>=V4)` `1 (GCC<V4)`	N
`optimize`	Turn on compiler optimization	`1`	N
`pipes`	Add `-pipe` compiler option for pipe support	`1`	Y

Table 1-2 GNU Make Flags - Compilation Control

Flag	Description	Default	Must Match?[1]
`profile`	Enable profiling	`0`	N
`purify`	Third party product support	`0`	N
`quantify`	Third party product support	`0`	N
`repo`	Use GNU template repository	`0`	N
`shared_libs`	Build shared libraries	`1`	E
`shared_libs_only`	Build shared libraries *only*	`0`	E
`split`	Splits ACE source files before compilation	`0`	B
`static_libs`	Build archive libraries	`0`	E
`static_libs_only`	Build archive libraries *only*	`0`	E
`static_link`	Link only static libraries to executables (uses GNU linker's `-static` flag)	`0`	N
`templates` *`type`*	Specifies type of template instantiation (`automatic`, `used`, `explicit`)	`automatic`	Y
`threads`	Enable threads (if OS supports)	`1`	Y
`versioned_so`	Add versioning to library names	`1`	Y

1. The values for **Must Match?** have the following meanings:
   Y The application must use the same value as the build.
   N The application may use any value it chooses.
   E The application can only use this flag if it was set at ACE/TAO build time.
   B This flag is only effective at ACE/TAO build time.

Table 1-3 shows the make flags that indicate whether external dependencies (usually third party libraries) are present and whether the dependent parts of ACE and TAO should be built.

Table 1-3 GNU Make Flags - External Product Dependencies

Flag	Description	Default	Must Match?[1]
`fl`	Build fl (Fast Light) components (reactor and resource factory)	`0`	Y
`fox`	Build Fox components (reactor and resource factory)	`0`	Y
`qt`	Build Qt components (reactor and resource factory)	`0`	Y

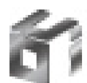


Table 1-3 GNU Make Flags - External Product Dependencies

Flag	Description	Default	Must Match?[1]
sctp	Build ACE SCTP and TAO SCIOP support	0	B
ssl	Build with SSL support in ACE and TAO	0	B
stlport	Build with STLport support	0	B
tk	Build tk components (reactor and resource factory)	0	Y
wfmo	Build with Wait For Multiple Objects reactor support	0	B
winregistry	Build with windows registry support	0	B
xt	Build Xt components (reactor and resource factory)	0	Y
zlib	Build with zlib (compression) components	1	B

1. The values for **Must Match?** have the following meanings:
   Y The application must use the same value as the build.
   N The application may use any value it chooses.
   E The application can only use this flag if it was set at ACE/TAO build time.
   B This flag is only effective at ACE/TAO build time.

Table 1-4 shows the make flags that control the building of specific ACE and TAO features.

Table 1-4 GNU Make Flags - ACE/TAO Feature Control

Flag	Description	Default	Must Match?[1]
ace_for_tao	Builds a the minimal subset of ACE necessary for TAO.	0	Y
ami	Enable Asynchronous Method Invocation (AMI)	1	Y
corba_messaging	Enable CORBA Messaging	1	Y
ec_typed_events	Enables support for typed event channels in the COS Event Service.	1	Y
interceptors	Enable Portable Interceptors	1	Y
minimum_corba	Build with minimum CORBA support	0	Y
probe	Enable ACE_Timeprobes	0	B
rt_corba	Enable Real-time CORBA support	1	Y

Table 1-4 GNU Make Flags - ACE/TAO Feature Control

Flag	Description	Default	Must Match?[1]
`rwho`	Build the distributed rwho utility	1	B

1. The values for **Must Match?** have the following meanings:
   Y The application must use the same value as the build.
   N The application may use any value it chooses.
   E The application can only use this flag if it was set at ACE/TAO build time.
   B This flag is only effective at ACE/TAO build time.

A.4 Using the Build Flags

In the remainder of this section, we describe each flag listed in A.3, including how to specify it using a GNU Make flag, a preprocessor macro, or both.

The flags are described below. *They are shown in the form necessary for changing the default.*

ace_for_tao

Builds the minimal subset of ACE necessary for TAO.

GNU Make Flag	Preprocessor Macro
`ace_for_tao=1`	None

This option is useful for minimizing the footprint of TAO applications.

ami

Enable Asynchronous Method Invocation (AMI).

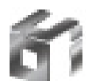
GNU Make Flag	Preprocessor Macro
`ami=0`	`#define TAO_HAS_AMI 0`

If this flag is defined via a preprocessor macro in `config.h`, you must also set the `TAO_HAS_AMI_CALLBACK` and `TAO_HAS_AMI_POLLER` macros with respect to your particular needs. See 6.2 for more information on using AMI.

corba_messaging

Build with full CORBA Messaging support.

GNU Make Flag	Preprocessor Macro
`corba_messaging=0`	`#define TAO_HAS_CORBA_MESSAGING 0`



This flag is defined by default, unless `minimum_corba` is enabled. See Chapter 6 for more information on CORBA Messaging.

debug

Causes the compiler to generate debugging information (e.g., by passing the `-g` option for some compilers), and builds an in-memory database of ACE objects.

GNU Make Flag	Preprocessor Macro
`debug=0`	`#define ACE_NDEBUG`

ec_typed_events

Controls whether typed event channel support is built for the Cos event service.

GNU Make Flag	Preprocessor Macro
`ec_typed_events=1`	`#define TAO_HAS_TYPED_EVENT_CHANNEL 1`

This option allows application developers to minimize the size of the Cos Event service libraries when the typed event channels are not being used.

fast

Enable the `-fast` option. This only applies if you are using a Sun Workshop C++ compiler that implements this option.

GNU Make Flag	Preprocessor Macro
`fast=1`	None

Note that the `fast` flag is mutually exclusive of the `debug` flag.

fl

Build ACE/TAO components dependent on the FLTK (The Fast Light Tool Kit). This includes the ACE Fl reactor and TAO Fl resource libraries.

GNU Make Flag	Preprocessor Macro
`fl=1`	None

See 18.7.6 for more information on using the Xt reactor with TAO.

fox

Build ACE/TAO components dependent on the Fox library. This includes the ACE Fox reactor and TAO Fox resource libraries.

GNU Make Flag	Preprocessor Macro
fox=1	None

See 18.7.6 for more information on using the Fox reactor with TAO.

inline

Enable or disable C++ function inlining.

GNU Make Flag	Preprocessor Macro
inline=0	#define ACE_NO_INLINE

If inlining is enabled when the ACE and TAO libraries are built, `inline` hints are passed to the compiler for many simple functions, and the resulting libraries may contain some inlined code. If inlining is disabled when the ACE and TAO libraries are built, no `inline` hints are passed to the compiler, and the libraries will contain independent function entries for these simple functions. If inlining is enabled when your application is built, inlined ACE and TAO functions may be inserted into your code.

interceptors

Build with support for Portable Interceptors.

GNU Make Flag	Preprocessor Macro
interceptors=0	#define TAO_HAS_INTERCEPTORS 0

Enabling this flag builds in support for Portable Interceptors as defined in the CORBA specification. This flag is defined by default, unless `minimum_corba` is enabled. For example, if ACE and TAO are built with this flag enabled, the types and interfaces defined as part of the `PortableInterceptor` module will be included in TAO. See `$TAO_ROOT/tao/PortableInterceptor.pidl` for the specific type and interface definitions that are included. See Chapter 9 for more information on Portable Interceptors.

minimum_corba

Build a minimally-supported CORBA system.

GNU Make Flag	Preprocessor Macro
`minimum_corba=1`	`#define TAO_HAS_MINIMUM_CORBA 1`

optimize

Generate time-optimized code. Note that some compilers do not allow both debug and optimize to be specified at the same time.

GNU Make Flag	Preprocessor Macro
`optimize=0`	None

pipes

Add `-pipe` compiler option for pipe support.

GNU Make Flag	Preprocessor Macro
`pipes=0`	None

For use with gcc only, this option may provide increased compilation speed, at the cost of increased memory usage during compilation.

probe

Build ACE with time probes.

GNU Make Flag	Preprocessor Macro
`probe=1`	`#define ACE_COMPILE_TIMEPROBES`

profile

Enable the use of the GNU `gprof` tool.

GNU Make Flag	Preprocessor Macro
`profile=1`	None

purify

Process the object code so that memory integrity checking can be performed using IBM Rational PURIFY®. Using the `purify` flag causes the `purify` command to precede the linker command during the linking phase of building

your application. The `purify` command must be found in your `PATH` environment variable.

GNU Make Flag	Preprocessor Macro
`purify=1`	None

qt

Build ACE/TAO components dependent on the Qt library from Trolltech. This includes the ACE Qt reactor and TAO Qt resource libraries.

GNU Make Flag	Preprocessor Macro
`qt=1`	`None`

See 18.3 for more information on using the Qt reactor with TAO.

quantify

Process the object code so that performance profiling can be obtained using IBM Rational QUANTIFY®. Using the `quantify` flag causes the `quantify` command to precede the linker command during the linking phase of building your application. The `quantify` command must be found in your `PATH` environment variable

GNU Make Flag	Preprocessor Macro
`quantify=1`	None

repo

Use the GNU template repository.

GNU Make Flag	Preprocessor Macro
`repo=1`	None

rt_corba

Build with full Real-Time CORBA support.

GNU Make Flag	Preprocessor Macro
`rt_corba=0`	`#define TAO_HAS_RT_CORBA 0`

This flag is defined by default, unless `minimum_corba` is enabled or `corba_messaging` is disabled. See Chapter 8 for more information on Real-Time CORBA.

rwho

Build the distributed rwho utility.

GNU Make Flag	Preprocessor Macro
`rwho=0`	None

sctp

Build ACE SCTP and TAO SCIOP support.

GNU Make Flag	Preprocessor Macro
`sctp=lksctp`	`#define ACE_HAS_SCTP 1`

This flag should be set to name the SCTP implementation to use, either `lksctp` or `openss7`. Since both of these SCTP implementations are linux kernel options, the SCTP-related features are currently only supported on linux.

shared_libs

Shared libraries are built by default. If the `shared_libs_only` flag is used, this flag is also set. If the `static_libs_only` flag is used, this flag is ignored.

GNU Make Flag	Preprocessor Macro
`shared_libs=0`	None

shared_libs_only

Pseudo-flag that prevents static libraries from being built whether or not the `static_libs` flag is used.

GNU Make Flag	Preprocessor Macro
`shared_libs_only=1`	None

split

Splits ACE source files before compilation.

GNU Make Flag	Preprocessor Macro
`split=1`	None

Splitting the ACE source files allows processes linked with static libraries to only include the parts of ACE they require and minimizes the memory footprint of the application.

ssl

Build with SSL support in ACE and TAO.

GNU Make Flag	Preprocessor Macro
`ssl=1`	None

Enables building the SSL-related libraries in ACE and TAO. Requires `$SSL_ROOT` environment variable to point to the OpenSSL installation directory.

static_libs

Static libraries are not built by default. If the `static_libs_only` flag is used, this flag is also set. If the `shared_libs_only` flag is used, this flag is ignored.

GNU Make Flag	Preprocessor Macro
`static_libs=1`	None

static_libs_only

Pseudo-flag that prevents shared libraries from being built whether or not the `shared_libs` flag is used.

GNU Make Flag	Preprocessor Macro
`static_libs_only=1`	None

static_link

Link only static libraries to executables (uses GNU linker's `-static` flag).

GNU Make Flag	Preprocessor Macro
`static_link=1`	None

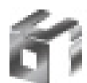
This flag is only supported when using the GNU linker.

stlport

Build with STLport support.

GNU Make Flag	Preprocessor Macro
`stlport=1`	`#define ACE_HAS_STLPORT 1`

Allow components that use the STLport library to be built.



templates

Specifies type of template instantiation.

GNU Make Flag	Preprocessor Macro
`templates=used`	None

This flag takes a value of either `automatic` (the default), `explicit`, or `used`. Explicit template instantiation is not currently supported in TAO as all supported platforms are currently working with automatic. The used value is only for use with the OSF cxx C++ compiler.

threads

Build the ACE and TAO libraries to use threads (only applies if the operating system supports multithreaded programming). We recommend not modifying this flag from its default setting.

GNU Make Flag	Preprocessor Macro
`threads=0`	See text.

The `threads` flag can also be specified in the `config.h` file, but that is more complicated as there are several variables that must be set in a way that is specific to the operating system on which ACE and TAO are being built. You are advised not to try to change these variables. Instead, you should use the GNU Make flag above.

tk

Build ACE/TAO components dependent on Tk. This includes the ACE Tk reactor and TAO Tk resource libraries.

GNU Make Flag	Preprocessor Macro
`tk=1`	`None`

See 18.7.6 for more information on using the Xt reactor with TAO.

versioned_so

Add versioning to library names.

GNU Make Flag	Preprocessor Macro
`versioned_so=0`	None

If set to zero, this flag does not append version information to the library name. If set to one, this flag causes each shared library to be appended with

the contents of the SOVERSION variable. If SOVERSION is not set, then the version number defaults to .major.minor.beta for DOC group releases and .major.minor.patch_level for OCI versions.

wfmo

Build with Wait For Multiple Objects reactor support.

GNU Make Flag	Preprocessor Macro
`wfmo=1`	None

The WFMO reactor is only supported on Microsoft Windows and is usually enabled there automatically.

winregistry

Build with windows registry support.

GNU Make Flag	Preprocessor Macro
`winregistry=1`	None

The `winregistry` option is only available on Windows and is usually enabled there automatically.

xt

Build support for Xt-related components. This implicitly enables build of the ACE for the Xt reactor and TAO Xt resource libraries.

GNU Make Flag	Preprocessor Macro
`xt=1`	None

See 18.4 for more information on using the Xt reactor with TAO.

zlib

Enables building the zlib-dependent features of TAO.

GNU Make Flag	Preprocessor Macro
`zlib=1`	None

This enables building of the TAO zlib compression library allowing for its use with the ZIOP Protocol. See 14.14 for ZIOP details.

Appendix B

Choosing How To Build ACE and TAO

There are several ways to build ACE and TAO, depending upon your operating system, compiler, and preferences. This appendix will help you choose the type of build you need to perform.

The instructions in this and subsequent appendices apply only if you want to build the ACE and TAO libraries and executables directly from the source code distribution. Alternatively, for many platforms, you can install pre-built binary versions of the libraries and executables directly from the OCI CD-ROMs. See 2.2 and the instructions that come with the CD for more information.

Building ACE and TAO on UNIX with GNU Make

For UNIX and UNIX-like platforms, it is possible to build TAO similarly to previous releases, using provided configuration files and GNU Makefiles generated by MPC. This method is described in Appendix C.

Building ACE and TAO on Windows with Visual C++

You can build ACE and TAO on Windows (NT, 2000, XP) using Visual C++. See Appendix D for details.

Using ACE and TAO with VxWorks

You can build and use ACE and TAO on the VxWorks operating system. See Appendix E for details.

Using ACE and TAO with LynxOS

You can build and use ACE and TAO on the LynxOS operating system. See Appendix G for details.

Appendix C

Building ACE and TAO on UNIX

This appendix discusses how to build the ACE and TAO libraries and executables after installing the source code. Alternatively, you can install pre-built binary versions of the libraries and executables directly from the OCI CD-ROMs. See the instructions that come with the CD for more information.

C.1 Building ACE and TAO on a UNIX System

ACE and TAO can be used on many versions of UNIX and UNIX-like platforms, including Linux, Solaris, AIX, HP-UX, Tru64, IRIX, and others.

Before you build the ACE and TAO libraries, you should review C.2 to learn about reducing the memory footprint of the ACE and TAO libraries. The following steps must be performed in order to build TAO. They will be discussed at length in the pages that follow.

1. Create a build tree. (Optional, but recommended.)
2. Set environment variables.
3. Configure the source code for your platform.
4. Determine what features to build.

5. Generate makefiles. (Optional, but recommended.)
6. Build ACE and TAO.
7. Verify the build.

C.1.1 Create a Build Tree

MPC includes a Perl script called `clone_build_tree.pl` that you can use to create multiple build trees from a single, shared source tree. The builds are a duplicate of `ACE_wrappers`, and are placed in `ACE_wrappers/build/`*`name`* where *`name`* is the name you supply to identify the build. Each build can use a different compiler and different settings.

The script must be run from the `ACE_wrappers` directory. You can name your build anything you like. For example:

```
MPC/clone_build_tree.pl default
```

will create an `ACE_wrappers/build/default` directory with a complete duplicate of the directory structure of `ACE_wrappers`, with symbolic links to source code and other necessary files. The build directory will not contain any makefiles as it is assumed you will use MPC to generate them.

If you are modifying ACE or TAO, and have added or removed files in the original tree, then you should run `clone_build_tree.pl` again without any arguments. This will update all existing builds.

You may choose not to create a build tree. In this case you can put your makefiles directly in `ACE_wrappers`.

C.1.2 Set Environment Variables

The environment variables for building ACE and TAO are `ACE_ROOT` and `TAO_ROOT`. Assuming you installed the source directories under `/usr/local`, and created a new build tree as outlined in C.1.1, the following `sh` commands set these environment variables.

```
ACE_ROOT=/usr/local/ACE_wrappers/build/default; export ACE_ROOT
TAO_ROOT=$ACE_ROOT/TAO; export TAO_ROOT
```

For the ACE and TAO executables to link, the ACE and TAO libraries must be in the library path. In TAO, the output library path is easily configured using MPC. By default, libraries are placed in `$ACE_ROOT/lib`. The

following sh command sets the search path for libraries, assuming MPC was used to generate your makefiles:

```
LD_LIBRARY_PATH=$ACE_ROOT/lib:$LD_LIBRARY_PATH; export LD_LIBRARY_PATH
```

Note *You must define* ACE_ROOT *when building the ACE and TAO libraries.* TAO_ROOT *defaults to* $ACE_ROOT/TAO. *The same settings must also be defined to use ACE and TAO. See C.1.3 for information on setting the* platform_macros.GNU *file.*

C.1.3 Configure the Source Code for Your Platform

To achieve ACE and TAO's portability on a wide variety of platforms with a minimum impact to the source code, platform dependencies for the source code and the makefiles are constrained to three files.

- $ACE_ROOT/ace/config.h
- $ACE_ROOT/include/makeinclude/platform_macros.GNU
- $ACE_ROOT/bin/MakeProjectCreator/config/default.features

These files serve as documentation of which settings were used to build ACE and TAO, and also to ensure that your own projects use consistent settings where necessary.

The config.h file defines C++ preprocessor macros that control operating system and C++ compiler characteristics and system library coverage. Most of these macros are of the form ACE_HAS_feature or ACE_LACKS_feature. Currently, we recommend also putting any TAO_HAS_feature options in this file, though in the future those may be moved to another file.

Here are some examples of preprocessor macros that control how ACE and TAO are built:

```
ACE_LACKS_SIGNED_CHAR
ACE_HAS_IP_MULTICAST
TAO_HAS_MINIMUM_CORBA
```

The platform_macros.GNU file defines macros used by GNU Make to build ACE and TAO and applications, such as exceptions, inline, debug, and optimize. For details about available options for use in platform_macros.GNU, see Appendix A.

Here are some examples of build flags that affect how ACE and TAO and applications are built:

```
debug=0
optimize=1
fast=1
inline=0
```

Creating the Platform Configuration Files

The three configuration files described above do not exist in the ACE and TAO source code distribution; you have to create them.

Your `config.h` file should `#include` a `config-*.h` file that is specific to your platform, such as `config-hpux11.00.h`, `config-linux.h`, or `config-sunos5.9.h`. Then, you can add your own configuration options before the `#include`.

As an example, we will create configuration files to build ACE and TAO for Solaris 9 using the Sun ONE Studio 8 Compiler. We also enable `qos` and `ssl` features.

To support Solaris 9, we create the following `$ACE_ROOT/ace/config.h` file:

```
// Add any configuration macros here.
#include "ace/config-sunos5.9.h"
```

Your `platform_macros.GNU` file should include the correct file for the platform and compiler, such as `platform_hpux_aCC.GNU`, `platform_linux.GNU`, or `platform_sunos5_sunc++.GNU`. Then, you can add your own configuration options before the `include`.

For example, to support the Sun ONE Studio 8 Compiler, we create the following `$ACE_ROOT/include/makeinclude/platform_macros.GNU` file:

```
# Add any configuration options such as debug, release, and exceptions here.
include $(ACE_ROOT)/include/makeinclude/platform_sunos5_sunc++.GNU
```

Note *Several platform-specific* `config*.h` *and* `platform*.GNU` *files are provided as part of the ACE and TAO source code distribution. These cover all of the OCI supported platforms, as well as other platforms. If files are not*

included for a particular platform, operating system, and compiler combination in which you are interested, you may be able to create one by starting with one of the provided files and modifying it slightly. OCI can help you port ACE and TAO to a new platform.

C.1.4 Determine what features to build

Before generating makefiles, you may want to go over the features listed in A.2.1 to see if any are enabled/disabled that you wish to be disabled/enabled. If so you will need to create a `default.features` file containing your custom feature settings.

C.1.5 Generate Makefiles

We are now ready to use the MPC tool to generate the makefiles for our build.

```
cd $ACE_ROOT
bin/mwc.pl -type gnuace -recurse
```

This command will generate makefiles for any `.mwc` workspace files found in our build tree, as well as generating makefiles for any `.mpc` project files. By generating makefiles for workspaces, we will be able to build related projects with a single command.

Note *See the MPC documentation at* `<http://www.ociweb.com/products/MPC>` *for more information on using MPC.*

C.1.6 Build ACE and TAO

This step briefly describes how to build ACE and TAO. For more detailed instructions, see `$ACE_ROOT/ACE-INSTALL.html` and `$TAO_ROOT/TAO-INSTALL.html`.

The makefiles require version 3.79.1 or later of GNU Make. If you do not have GNU Make version 3.79.1, you can obtain it at no cost via `<http://www.theaceorb.com/references/>`.

Assuming MPC was used to generate makefiles as described above, it is possible to build a complete ACE and TAO development installation by running a single `make` command from `$TAO_ROOT`:

```
cd $TAO_ROOT
make
```

This will build everything you need to use TAO. It will not build any examples, tests, or performance tests.

Note *You must use GNU Make to build ACE+TAO on UNIX and UNIX-like platforms. The* `make` *command on your system may actually invoke the native operating system's* `make` *program instead of GNU Make. If you are not sure, type* `make --version`*. If the* `make` *command on your system is not GNU Make, check with your system administrator to see if GNU Make is available. On some systems, you may be able to use the* `gmake` *command.*

C.1.7 Verify Your Build

Once you have a complete TAO build, you may want to run a few tests to verify that it is working correctly. The TAO source code distribution includes some basic tests and performance tests that you can use.

- `$TAO_ROOT/tests/Hello` contains a basic "Hello, world!" CORBA client and server application. If this test does not work, then your TAO build or your run time environment is seriously impaired.
- `$TAO_ROOT/DevGuideExamples` and `$TAO_ROOT/orbsvcs/DevGuideExamples` directories contains all of the source code for the examples in this Developer's Guide, along with MPC files and run scripts.
- `$TAO_ROOT/tests/Param_Test` contains an application that exercises the ORB's basic parameter passing conventions for many OMG IDL data types.
- `$TAO_ROOT/performance-tests/Cubit/TAO/IDL_Cubit` contains tests for certain optimizations, such as collocation. You can also use the IDL Cubit server and client to test passing various command line options to applications.

See the `README` file in each of the above directories for more information on building and running these tests. Each of the above directories also contains MPC files to use for building the test, and a Perl script named `run_test.pl` for semi-automated running of the test.

C.2 Customizing ACE and TAO Builds

In certain situations (e.g., embedded environments) you may want to minimize the memory footprint required by the ACE and TAO libraries. There are several build parameters you can customize to help reduce the size and memory footprint of these libraries. You can also disable MPC features to prevent generation of makefiles for related projects.

Note *To customize your build, you must set these parameters before building ACE and TAO.*

C.2.1 Minimizing the Size of the TAO Library

TAO provides support for the *minimumCORBA* specification from the OMG. The *minimumCORBA* specification omits the following features from the CORBA specification:

- Dynamic Skeleton Interface (DSI)
- Dynamic Invocation Interface (DII)
- Dynamic Any
- Interceptors
- Interface Repository
- Advanced POA features
- CORBA/COM interworking

To select the minimumCORBA subset of TAO, you can define the `TAO_HAS_MINIMUM_CORBA` preprocessor macro with a value of `1` in `$ACE_ROOT/ace/config.h` or include `minimum_corba=1` in `platform_macros.GNU`. You must also add `minimum_corba=1` to your `default.features` file before generating your makefiles. Building only the minimumCORBA subset typically results in about a 26% reduction in the size of the TAO library (`$ACE_ROOT/lib/libTAO.*`). However, it may not be appropriate for your application due to the omission of the features listed above.

You can choose the minimumCORBA configuration of TAO, but retain the advanced POA features, such as Servant Managers, Default Servant, and

Adapter Activators, by defining `TAO_HAS_MINIMUM_POA = 0` in `$ACE_ROOT/ace/config.h` before building TAO.

Here is an example of a `config.h` file that could be used on Linux to build TAO with minimumCORBA support, yet retain the advanced POA features:

```
#define TAO_HAS_MINIMUM_CORBA=1
#define TAO_HAS_MINIMUM_POA=0
#include "ace/config-linux.h"
```

C.2.2 Selectively Building TAO Services

All TAO service libraries (`$ACE_ROOT/lib/libTAO_*.*`) are built by default. These libraries include all of TAO's currently-supported services (see Chapter 21 for more about TAO's services). You can reduce the number of services that are built (and thereby reduce the overall build time and disk space consumption) by excluding unused services. In previous versions of TAO, this was controlled by setting the `TAO_ORBSVCS` variable to contain only the names of the services you want to build. This is no longer supported. However, you can change directory to `$TAO_ROOT/orbsvcs/orbsvcs` and build individual projects such as `CosNaming` or `RTEvent` to build only the services you want to use. For example:

```
$ cd $TAO_ROOT/orbsvcs/orbsvcs
$ make CosNaming RTEvent
```

You can view the available projects by looking at the list of GNU make files. In a given directory, you will see a make file for each project, such as `GNUmakefile.CosNaming` and `GNUmakefile.CosNaming_Skel`. Simply pass the set of project names you want to build on the make command line.

Another way to build a subset of TAO services is to create your own workspace for use with MPC. You could make a copy of the existing `$TAO_ROOT/TAO_ACE.mwc` workspace file and modify it to build only the services that you want. This approach has the added advantages of portability and repeatability.

The following is an example of a custom workspace (`.mwc`) file located in `$TAO_ROOT` that contains everything necessary to build the `CosNaming` and `RTEvent` libraries along with the `tao_cosnaming` executable.

```
// $TAO_ROOT/CosNaming_RTEvent.mwc
```

```
workspace {
  ../ace
  ../apps/gperf/src
  tao
  TAO_IDL
  orbsvcs/orbsvcs/Svc_Utils.mpc
  orbsvcs/orbsvcs/CosNaming.mpc
  orbsvcs/orbsvcs/RTEvent.mpc
  orbsvcs/Naming_Service
}
```

Appendix D

Building ACE and TAO Using Visual C++

This appendix discusses how and why you might want to build the ACE and TAO libraries and executables from the source code for the Windows platform using Visual C++.

D.1 Building ACE and TAO

Building ACE and TAO with Visual C++ requires several steps:

1. Create a build tree. (Optional, but recommended.)
2. Set environment variables.
3. Configure the source code for your platform.
4. Determine what features to build.
5. Generate Project Files. (Optional, but recommended.)
6. Build ACE and TAO.
7. Verify the build.

D.1.1 Create a Build Tree

As discussed in C.1.1, this optional step of working with build trees allows you to create multiple builds from a single set of source code.

Note *This features requires use of NTFS, because it relies on linked files.*

MPC includes a Perl script called `clone_build_tree.pl` that can create multiple build trees from a shared source tree. The builds are a duplicate of `ACE_wrappers`, and are placed in `ACE_wrappers/build/<name>`. Each build can use a different compiler and settings.

The script must be run from `ACE_wrappers`, and you can name the build anything you like. For example:

```
clone_build_tree.pl default
```

will create a directory named `ACE_wrappers/build/default` with a complete duplicate of the directory structure of `ACE_wrappers`, with links to source code and other necessary files. The build directory will not contain any project files, as it is assumed you will use MPC to generate them.

If you are modifying ACE or TAO, then you should run `clone_build_tree.pl` again without any arguments. This will update all existing builds. Unlike UNIX and UNIX-like platforms, which support symbolic links, Windows uses hard links that can be broken by many tools. For example, editing a file with some editors appears to actually remove the existing file and create a new one in its place. This is not a problem with `Notepad` or Visual Studio. When `clone_build_tree.pl` updates an existing build, it will overwrite existing files with the version from the build directory if it is newer, and the original will be renamed with the addition of a `.bak` extension. For example, if you edit `ACE_wrappers\build\default\VERSION`, then `clone_build_tree.pl` will rename `ACE_wrappers\VERSION` to `ACE_wrappers\VERSION.bak`, and copy the one from the build directory.

You may choose not to create a build tree. In this case, you can use the default projects, or generate new ones directly in `ACE_wrappers`.

D.1.2 Set Up Your Environment

You should create ACE_ROOT and TAO_ROOT environment variables, and update your PATH. This is not strictly necessary on Windows, but it makes using TAO more convenient.

For example:

```
set ACE_ROOT=c:\ACE_wrappers\build\default
set TAO_ROOT=%ACE_ROOT%\TAO
set PATH=%PATH%;%ACE_ROOT%\bin;%ACE_ROOT%\lib
```

TAO requires two entries in your PATH. The first is the location for generated executables, and the second is the location for DLLs.

D.1.3 Configure the Source Code for Your Platform

Before you are able to build the libraries and executables, you must configure the source for the environment. For all Win32 platforms, create a new file called `%ACE_ROOT%\ace\config.h` and add the following lines:

```
#include "ace/config-win32.h"
```

You can modify this file to suit your custom build requirements. Here are some additional settings you may need to add to your `config.h` file:

- Open Socket Limit
  - By default, Windows' sockets library allows only 64 simultaneous open sockets. You can add the following definition to increase this limit, however ACE already increases it to 1024 by default:

```
#define FD_SETSIZE 256
```

D.1.4 Determine what features to build

Before generating project files, you may want to go over the features listed in A.2.1 to see if any are enabled/disabled that you wish to be disabled/enabled. If so you will need to create a `default.features` file containing your custom feature settings.

D.1.5 Generate Project files

We can now use MPC to generate the project for our build:

```
cd %ACE_ROOT%
mwc.pl -type vc12 TAO_ACE.mwc
```

This command will generate an `.sln` file for ACE and TAO including all of the optional libraries, services, and utilities.

The example above shows a build environment type selection for Visual C++ 12, which ships with Visual Studio 2013. Build environment types are identified by the compiler identifier, not the Visual Studio vintage. MWC will create workspace solutions for several Visual C++ versions shown here.

Table D-1 Visual C++ Type values for mwc.pl and mpc.pl

-type value	Usage
vc6	Visual Studio 6.0, for OCITAO 1.6a and older
vc7	Visual Studio .NET (2002), not supported for any TAO
vc71	Visual Studio .NET 2003, for OCITAO 2.0a and older
vc8	Visual Studio 2005, for OCITAO 2.0a and older
vc9	Visual Studio 2008, for OCITAO 2.2a and older
vc10	Visual Studio 2010
vc11	Visual Studio 2012
vc12	Visual Studio 2013
vc14	Visual Studio 2015
nmake	command line build, compatible with all the above compilers

To create a separate `.sln` for the regression test suite or the TAO examples use `mwc.pl` without naming an mwc file name. In this case `mwc.pl` generates a workspace that contains a separate project for each of the mpc files found in the directory tree. Individual example or tests projects may be contained in separate solution files by running `mwc.pl` in individual example or test directories..

Table D-2 Notable top-level directories

Directory	Contains
`%TAO_ROOT%\examples`	Examples of use of the various TAO core features and optional libraries.
`%TAO_ROOT%\tests`	Regression tests for TAO core functionality.
`%TAO_ROOT%\DevGuideExamples`	All of the examples presented in this guide.

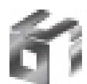


Table D-2 Notable top-level directories

Directory	Contains
`%TAO_ROOT%\orbsvcs\examples`	Examples of use of the various TAO ORB services such as naming or notification.
`%TAO_ROOT%\orbsvcs\tests`	Regression tests for the ORB services.

D.1.6 Build the Libraries

All TAO libraries and executables can be built from single Visual C++ workspace found in `%TAO_ROOT%\TAO_ACE.sln`.

You may use the `Batch Build` command in Visual Studio IDE to build the libraries and configurations in which you are interested. Alternatively, you may find that it is easier to build just the configurations in which you are interested from the command line, as follows:

```
devenv TAO_ACE.sln /build debug /out build.log
```

The Express Editions of Visual Studio lack the `devenv` tool, but can build from the command line using `msbuild` or `vcbuild`. For example, the following `msbuild` command builds all projects in the `TAO_ACE.sln` solution:

```
msbuild /p:Configuration=Debug TAO_ACE.sln
```

An easy way to build the most commonly used TAO projects is to build the `Naming_Service` project. Its dependencies build ACE, TAO, the TAO IDL compiler, and some other common projects. You can build this project from the command line as follows:

```
devenv TAO_ACE.sln /build debug /project Naming_Service /out build.log
```

or with the msbuild

```
msbuild /t:Naming_Service /p:Configuration=Debug TAO_ACE.sln
```

The advantage of using this method over using `Batch Build` is that you do not build libraries and executables that you do not need, and you do not need to know which projects are required to build a particular service.

D.1.7 Verify Your Build

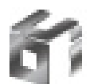
Once you have a complete TAO build, you may want to run a few tests to verify that it is working correctly. The TAO source code distribution includes some basic tests and performance tests that you can use.

- `%TAO_ROOT%\tests\Hello` contains a basic “Hello, world!” CORBA client and server application. If this test does not work, then your TAO build or your run time environment is seriously impaired.
- `%TAO_ROOT%\DevGuideExamples` and `%TAO_ROOT%\orbsvcs\DevGuideExamples` contain all of the source code for the examples in this Developer’s Guide, along with MPC files and run scripts.
- `%TAO_ROOT%\tests\Param_Test\` contains an application that exercises the ORB’s basic parameter passing conventions for several OMG IDL data types.
- `%TAO_ROOT%\performance-tests\Cubit\TAO\IDL_Cubit\` contains tests for certain optimizations, such as collocation. You can also use the IDL Cubit server and client to test passing various command line options to applications.

See the `README` file in each of the above directories for more information on building and running these tests. Each of the above directories also contains MPC files to use for building the test, and a Perl script named `run_test.pl` for semi-automated running of the test.

D.2 Build Notes

- MPC is used to generate all projects for ACE and TAO. Using MPC ensures that all project types are kept up to date and makes it easier to maintain and add projects. Using MPC also ensures consistent settings across all projects.
- If you receive errors while building the static libraries for TAO, it may be because the IDL generated files do not yet exist. The easiest way to generate these files is to just build twice. On the first pass, Visual C++ will run the custom build command on the IDL files, which generates code from the IDL files. On the second pass, Visual C++ will be able to build without errors. Do not set dependencies between the various static



library projects; doing so has the unwanted side-effect of including the complete contents of the dependent library within the library for the project, resulting in some very large libraries.

- If you join the ranks of contributors to ACE and TAO, you will probably find yourself rebuilding certain files often. If this is the case, you may want to enable incremental linking, and/or minimal build in the settings for the project you are working on.
- You can build any project or workspace using Visual C++ from the command line. This is most easily shown with a few examples:
  - To build the Debug configuration of all the projects in the `TAO_ACE` solution, use the following command for Visual Studio:

    ```
    devenv TAO_ACE.sln /build Debug
    ```

    or using the `msbuild`:

    ```
    msbuild /t:Build /p:Configuration=Debug TAO_ACE.sln
    ```

  - To clean all of the projects you can use:

    ```
    devenv TAO_ACE.sln /clean
    ```

    or with `msbuild`:

    ```
    msbuild /t:Clean /p:Configuration=Debug TAO_ACE.sln
    ```

- When you use the `clean` feature of Visual C++, it leaves many temporary files behind. To make sure all the unneeded files are cleaned, you can search for the following wildcards using the Windows Search utility on the `%ACE_ROOT%` folder, then delete the found files:

  ```
  *.pch;*.lib;*.dll;*.sbr;*.plg;*.ncb;*.opt;*.clw;*.idb;*.obj;*.exe;*.ilk;*.
  bsc;*.exp;*.pdb;*.suo
  ```

- You must restart Visual C++ after making changes to environment variables. You may run into this problem the first time you build `TAO_ACE.sln` on a machine if you had Visual C++ running before setting `ACE_ROOT`, `TAO_ROOT`, and `PATH`.

Appendix E

Using ACE and TAO with VxWorks

This appendix describes how to build and use ACE and TAO with the VxWorks (version 6.x) operating system from Wind River Systems, Inc. VxWorks is a real-time operating system that is used in many embedded and real-time systems.

TAO can also be built for use with VxWorks 5.5.x. See `$ACE_ROOT/ACE_INSTALL.html` for details.

Cross-compilation is the only way to build ACE and TAO for VxWorks. Here, we will describe building ACE and TAO on a Windows host and on a UNIX host, such as Linux, for a PowerPC target.

VxWorks versions prior to 6.x placed application code in the same address space as the kernel. Starting with VxWorks 6.0, VxWorks also supports Real-Time Processes (RTP). Applications can still be built as kernel space modules, but can also be built as RTPs. TAO supports both modes of building for VxWorks.

E.1 Kernel and System Configuration

The VxWorks kernel is highly configurable and may require some modification for optimal performance with ACE and TAO. A good starting point for this configuration is the Development Profile from Wind River's Workbench. In addition, enable the following components from the kernel configuration tool:

```
INCLUDE_CORE_NFS_CLIENT
INCLUDE_CPLUS_IOSTREAMS
INCLUDE_GETADDRINFO_SYSCTL
INCLUDE_IFCONFIG
INCLUDE_IPCOM_USE_AUTH
INCLUDE_IPDNSC
INCLUDE_IPIFCONFIG_CMD
INCLUDE_IPNETSTAT_CMD
INCLUDE_IPNSLOOKUP_CMD
INCLUDE_IPPING_CMD
INCLUDE_IPTELNETS
INCLUDE_IPWRAP_GETADDRINFO
INCLUDE_IPWRAP_IFCONFIG
INCLUDE_IPWRAP_NETSTAT
INCLUDE_IPWRAP_PING
INCLUDE_NETSTAT
INCLUDE_NET_HOST_SHOW
INCLUDE_NFS3_CLIENT
INCLUDE_PING
INCLUDE_POSIX_PTHREAD_SCHEDULER
INCLUDE_PX_SCHED_DEF_POLICIES
INCLUDE_Q_PRI_BMAP_CREATE_DELETE
INCLUDE_RPC
INCLUDE_SHELL_BANNER
INCLUDE_USE_NATIVE_SHELL
```

For ACE and TAO to run properly under VxWorks, the `NUM_FILES` kernel configuration value must be changed. This value defaults to 50 but should be changed to at least 64. The value increase is required due to the possible high number sockets that can be created under normal operation of TAO.

Optionally, if DNS is enabled (as it is in the example above), you must modify the DNS server and domain in the kernel configuration to reflect your DNS information. If DNS is not enabled, the command line argument `-ORBDottedDecimalAddresses 1` must be used with TAO clients and servers to tell the ORB not to use host names under the IIOP protocol. Alternatively, `TAO_USE_DOTTED_DECIMAL_ADDRESSES` can be defined

when compiling TAO to accomplish the same thing without requiring the command line option. (See E.2.)

E.2 Environment Setup

ACE, ace_gperf and tao_idl must be statically built for your host prior to setting up the environment to build for VxWorks. We describe how to do this for both UNIX (in E.2.1) and Windows (in E.2.2) build hosts.

Note *It is possible to build a shared version of* ACE, ace_gperf *and* tao_idl, *but statically linking can help you avoid problems with library mismatches and finding the shared library at run time.*

E.2.1 UNIX

If your host is Solaris or Linux, follow the directions laid out in Appendix C until you reach C.1.3, “Configure the Source Code for Your Platform.”

Next, create the following files for your host machine:

```
$ACE_ROOT/ace/config.h
$ACE_ROOT/include/makeinclude/platform_macros.GNU
```

For example, if your build host machine is running Linux, you would create platform_macros.GNU in $ACE_ROOT/include/makeinclude with the following contents:

```
debug=0
optimize=1
static_libs_only=1
include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU
```

Your config.h file can be set up once and can take into account both the host and target platforms based on preprocessor macros. The following config.h file will suffice for Solaris, Linux, and Windows hosts, as well as cross-compiling for VxWorks:

```
#if defined (sun)
# include "ace/config-sunos5.10.h"
```

```
#elif defined (linux)
# include "ace/config-linux.h"

#elif defined (_MSC_VER)
# define ACE_DISABLE_WIN32_ERROR_WINDOWS
# define ACE_DISABLE_WIN32_INCREASE_PRIORITY
# include "ace/config-win32.h"

#elif defined (ACE_VXWORKS)
# include "ace/config-vxworks.h"
#endif
```

Once the above configuration is complete, build a static version of `ACE`, `ace_gperf`, and `tao_idl`. To do this, first generate a minimal set of makefiles with MPC.

```
cd $ACE_ROOT/TAO
../bin/mwc.pl -type gnuace TAO_ACE.mwc
```

Now, build the ACE library by running GNU Make in `$ACE_ROOT/ace` with the `ACE` target:

```
cd $ACE_ROOT/ace
make ACE
```

The `ace_gperf` and `tao_idl` executables can be built similarly by running GNU Make with no parameters in the `$ACE_ROOT/apps/gperf/src` and `$ACE_ROOT/TAO/TAO_IDL` directories.

```
cd $ACE_ROOT/apps/gperf/src
make
cd $ACE_ROOT/TAO/TAO_IDL
make
```

Now, you can reconfigure your source tree to build for your target. First, clean up libraries and object files left from the static build for your host:

```
cd $ACE_ROOT/ace
make realclean
```

Next, modify your `platform_macros.GNU` file in `$ACE_ROOT/include/makeinclude` as follows:

```
debug=0
optimize=1
```

```
static_libs_only=1
CPU=PPC32
include $(ACE_ROOT)/include/makeinclude/platform_vxworks.GNU
```

This example builds for RTP. To build for kernel mode applications, add `rtp=0` to your `platform_macros.GNU`. See the VxWorks documentation for the appropriate value of the CPU variable for your target processor.

Start a VxWorks development shell by running the `wrenv.sh` script that is part of the VxWorks installation. This script sets up a number of environment variables necessary for building with VxWorks. For example:

```
cd /opt/VxWorks
./wrenv.sh -p vxworks-6.7
```

At this point, you can follow the instructions in C.1.5, “Generate Makefiles” and finish with the remaining instructions in Appendix C.

E.2.2 Windows

If your host is Windows, follow the directions in Appendix D until you reach D.1.3, “Configure the Source Code for Your Platform”.

Next, create the following file for your host machine:

```
%ACE_ROOT%\ace\config.h
```

Your `config.h` file can be set up once and can take into account both the host and target platforms based on preprocessor macros. The following `config.h` file will suffice for Solaris, Linux, and Windows hosts, as well as cross-compiling for VxWorks:

```
#if defined (sun)
# include "ace/config-sunos5.10.h"

#elif defined (linux)
# include "ace/config-linux.h"

#elif defined (_MSC_VER)
# define ACE_DISABLE_WIN32_ERROR_WINDOWS
# define ACE_DISABLE_WIN32_INCREASE_PRIORITY
# include "ace/config-win32.h"

#elif defined (ACE_VXWORKS)
# include "ace/config-vxworks.h"
```

```
#endif
```

Once the above configuration is complete, build a static version of ACE, ace_gperf and tao_idl. To do this, first generate project and workspace/solution files using MPC.

For example, if you are using Visual C++ version 9:

```
cd %ACE_ROOT%\TAO
..\bin\mwc.pl -static -type vc9 TAO_ACE.mwc
```

Next, build the ACE library, ace_gperf and tao_idl by opening the TAO_ACE workspace or solution file (e.g., TAO_ACE.sln) in the %ACE_ROOT%\TAO directory (with Visual C++), selecting the active build configuration of "**Release**", and building the TAO_IDL_EXE project.

Next, create a platform_macros.GNU file in %ACE_ROOT%\include\makeinclude with the following contents:

```
debug=0
optimize=1
static_libs_only=1
CPU=PPC32
include $(ACE_ROOT)/include/makeinclude/platform_vxworks.GNU
```

This example builds for RTP. To build for kernel mode applications, add rtp=0 to your platform_macros.GNU. See the VxWorks documentation for the appropriate value of the CPU variable for your target processor.

Start a Visual Studio command prompt using the Start Menu shortcut installed by Microsoft Visual Studio. From within this window, start a VxWorks development shell by using the wrenv.exe. For example:

```
C:\VxWorks\wrenv.exe -p vxworks-6.7
```

Note *Running within a Visual Studio command prompt is necessary in order for TAO's IDL compiler to properly utilize the Visual C++ preprocessor.*

At this point, you can follow the directions in C.1.5, "Generate Makefiles" and finish with the remaining instructions in Appendix C.

Appendix F

Using ACE and TAO with Android

This appendix describes how to build and use ACE and TAO for Android operating system from Google, Inc. Android is an embedded operating system based on Linux. Application development is largely done using the Java language, but ACE and TAO provide the basis for products such as Open DDS which provides Java wrappers.

The ACE and TAO build described in this appendix are static libraries for use with the Native Development Kit (NDK). As of this writing NDK versions r5c and r6 are supported.

Cross-compilation is the only way to build ACE and TAO for Android. Here, we will describe building ACE and TAO on a Linux host an ARM based target.

The general sequence of steps to follow are:

- Download and configure Android SDK and NDK packages.
- Download and configure ACE/TAO for multiple builds.
- Set the required environment variables.
- Build the required host tools.

- Build the deployable libraries
- Run tests

F.1 Android Development Kits

Building for Android requires both the Software Development Kit (SDK) as well as the NDK. Obtain each from `http://developer.android.com`. The NDK should be at least release r5c, although the latest release will be something newer than that. Download and expand each archive into its own directory. The environment variables `$NDK` and `$SDK` should be set to point to each installed root directory.

For ACE and TAO development, a stand-alone tool chain should be used. This is necessary as ACE and TAO require RTTI and exceptions. Use the script `$NDK/build/tools/make-standalone-toolchain.sh` to do this. The target platform must be "android-9" and the architecture should be either "arm" or "x86." For example:

```
$cd $NDK
$./build/tools/make-standalone-toolchain.sh --arch arm --platform android-9 \
  --install-dir=arm_toolchain
```

This will construct a stand alone tool chain for the arm architecture in a directory called "arm_toolchain" which should be set as the value for the environment variable `$NDK_TOOLS`. `$NDK_TOOLS/bin` must be added to the `$PATH`.

Optionally, set the architecture value supplied to the tool chain maker script to a variable called `$ANDROID_ARCH`. This default to "arm" if not set.

The last environment variable to be set is `$SYSROOT`, which is defaulted to "`$NDK/platform/android-9/arch-$ANDROID_ARCH`."

F.2 Setup ACE/TAO Workspaces

`ACE`, ace_`gperf` and `tao_idl` must be statically built for your host prior to setting up the environment to build for Android. As of the writing of this appendix, building ACE and TAO for the Android target is supported on

Linux only, although there are NDK packages available for building on Windows and MacOS as well.

Cross compiled targets require two build workspaces. Follow the directions laid out in Appendix C until you reach C.1.3, “Configure the Source Code for Your Platform.” While those instructions indicate setting up a build workspace as an optional step, it is required for cross compiled targets since two workspaces are required, one for the host tools and the other for the target.

F.3 Build The Host Tools

Now that you have set up the two workspace directories, one for the host and one for the target. You should now set `$ACE_ROOT` to point to your host build workspace, and `$TAO_ROOT` point to the associated TAO directory.

Create the following files for your host machine:

```
$ACE_ROOT/ace/config.h
$ACE_ROOT/include/makeinclude/platform_macros.GNU
```

For example, since your build host machine is running Linux, you would create `platform_macros.GNU` in `$ACE_ROOT/include/makeinclude` with the following contents:

```
debug=0
optimize=1
include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU
```

Your `config.h` file can be set up once and can take into account both the host and target platforms based on preprocessor macros. The following `config.h` file will suffice for Linux:

```
# include "ace/config-linux.h"
```

Once the above configuration is complete, build a static version of `ACE`, `ace_gperf`, and `tao_idl`. To do this, first generate a minimal set of makefiles with MPC.

```
cd $TAO_ROOT
$ACE_ROOT/bin/mwc.pl -type gnuace TAO_ACE.mwc
```

You now have all the makefiles necessary for building a full ACE/TAO suite for the Linux host environment. This can come in handy for testing by providing a second peer to be a client or server as needed for inter-host tests with your Android applications. In this case, from the $TAO_ROOT directory you simply run GNU Make.

```
cd $TAO_ROOT
make
```

However, this is not necessary if you are building exclusively for Android and do not need a local ACE/TAO build for the host platform. In this case, you still need to build the tools necessary to run the IDL compiler. First, build the ACE library by running GNU Make in $ACE_ROOT/ace with the ACE target:

```
cd $ACE_ROOT/ace
make ACE
```

The ace_gperf and tao_idl executables can be built similarly by running GNU Make with no parameters in the $ACE_ROOT/apps/gperf/src and $ACE_ROOT/TAO/TAO_IDL directories.

```
cd $ACE_ROOT/apps/gperf/src
make
cd $TAO_ROOT/TAO_IDL
make
```

Once built, verify that the environment variable $HOST_ROOT is set to the host's $ACE_ROOT directory.

F.4 Build The Target Libraries

Now reset your $ACE_ROOT and $TAO_ROOT to point to the target workspace directories you prepared earlier.

As you did for the host environment, it is necessary to set up the config.h and platform_macros.GNU files.

The platform_macros.GNU file for the android target is very simple, relying on previously set environment variables for configuration. The platform details are in platform_android_linux.GNU which must be included. Apart from that, you can set debugging and optimization options. As of this

writing, support for other options, such as SSL has not been provided. Here is an example of the platform macros file for building Android-targeted libraries.

```
debug=0
optimize=1
include $(ACE_ROOT)/include/makeinclude/platform_android_linux.GNU
```

To recap, the environment variables required along with this file are:

- `$ANDROID_ARCH` - either `arm` or `x86`. Note that `x86` is only available in r6 and greater, and is still experimental. `ANDROID_ARCH` will be set to `arm` by default.
- `$NDK` — the path to the android native development kit.
- `$SDK` — the path to the android software development kit.
- `$SYSROOT` — the platform/architecture directory. This defaults to `$NDK/platforms/android-9/arch-$ANDROID_ARCH`.
- `$HOST_ROOT` — the path to the `TAO_IDL` & `gperf` tools, which is the host's `$ACE_ROOT`.
- `$NDK_TOOLS` — the path to the standalone tool chain created as shown above.
- `$PATH` — must include `$NDK_TOOLS/bin`.

For the ace/config.h file, yours should include the supplied Android-specific config file:

```
#include "ace/config-android.h"
```

Now the target build workspace is configured. As you did with the host build workspace, you need to generate GNU Make compatible makefiles. Switch to your new TAO directory and run MWC. Once done, you can run GNU Make. You should be able to observe that the C++ compiler being used is the one from the NDK Toolchain rather than the native `gcc` or `g++`.

```
cd $TAO_ROOT
$ACE_ROOT/bin/mwc.pl -type gnuace TAO_ACE.mwc
make
```

Appendix G

Using ACE and TAO with LynxOS

This appendix describes how to build and use ACE and TAO with the LynxOS (Version 5.0) operating system from LynuxWorks, Inc. ACE and TAO are built with a cross-compiler for LynxOS.

G.1 Cross-Compilation

G.1.1 Configuring and Building the Source Code

Setting up for cross-compilation is slightly different than configuring for native compilation. The main difference is that you must statically build `ACE`, `ace_gperf` and `tao_idl` for your host prior to configuring to build for your target.

Note *It is possible to build a shared version of* `ACE`, `ace_gperf` *and* `tao_idl`*, but statically linking can help you avoid problems with library mismatches and with finding the shared library at run time.*

Follow the directions in Appendix C until you reach C.1.3, “Configure the Source Code for Your Platform”.

At this point, create the following files for your host machine:

```
$ACE_ROOT/ace/config.h
$ACE_ROOT/include/makeinclude/platform_macros.GNU
```

For example, if your build host machine is running Linux, and you are using the GNU compiler, you would create `platform_macros.GNU` in `$ACE_ROOT/include/makeinclude` with the following contents:

```
debug=0
optimize=1
static_libs_only=1
include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU
```

Your `$ACE_ROOT/ace/config.h` file can be set up just once for all platforms using preprocessor macros. The following `config.h` file is sufficient for Linux and Windows hosts, as well as for cross-compiling for LynxOS:

```
#if defined (linux)
# include "ace/config-linux.h"

#elif defined (_MSC_VER)
# include "ace/config-win32.h"

#elif defined (__Lynx__)
# include "ace/config-lynxos.h"
#endif
```

Once the above configuration is complete, build a static version of `ACE`, `ace_gperf`, and `tao_idl`. To do this, first generate a minimal set of makefiles with MPC.

```
cd $ACE_ROOT/TAO
../bin/mwc.pl -type gnuace TAO_ACE.mwc
```

Now, build the ACE library by running GNU Make in `$ACE_ROOT/ace` with the `ACE` target:

```
cd $ACE_ROOT/ace
make ACE
```

The `ace_gperf` and `tao_idl` executables can be built similarly by running GNU Make with no parameters in the `$ACE_ROOT/apps/gperf/src` and `$ACE_ROOT/TAO/TAO_IDL` directories.

```
cd $ACE_ROOT/apps/gperf/src
make
cd $ACE_ROOT/TAO/TAO_IDL
make
```

Now, you can reconfigure your source tree to build for your target. First, clean up libraries and object files left from the static build for your host:

```
cd $ACE_ROOT/ace
make realclean
```

Next, modify your `platform_macros.GNU` file in `$ACE_ROOT/include/makeinclude`. An additional variable, `VERSION`, must be set when you cross-compile ACE and TAO for LynxOS. For example, your `platform_macros.GNU` file would appear as follows:

```
VERSION = 5.0.0
include $(ACE_ROOT)/include/makeinclude/platform_lynxos.GNU
```

The following steps are necessary to prepare the LynxOS cross-compilation environment. See the LynxOS 5.0 Installation Guide, Chapter 2, “Using the License Management Server”, for details.

1. Start the FlexLM license server if it is not already running as a daemon.
2. Set LynxOS environment variables (including `PATH`) using the provided script:

   ```
   cd /usr/LynxOS/5.0.0
   . SETUP.bash x86
   ```

   (or replace x86 with ppc)

3. Start the LynxOS session manager by running `lwsmgr` which is now on the `PATH`. When the build is complete you can stop the session manager by running `lwsmgr -s`.

At this point, you can follow the instructions in C.1.5, “Generate Makefiles” and finish with the remaining instructions in Appendix C.

Appendix H

Testing ACE and TAO on VxWorks and LynxOS

Once you have built ACE and TAO for VxWorks or LynxOS, it is a good idea to build the tests for ACE, TAO, and orbsvcs. These tests can be useful in determining possible run-time problems that will affect your application. You can use individual tests that focus on particular features of ACE, TAO, and the orbsvcs to narrow down any particular problems you might have with your build or your environment. Not all tests will pass on both VxWorks and LynxOS.

H.1 Building the Tests

Run MPC with the following commands to generate GNU makefiles:

```
cd $ACE_ROOT/tests
$ACE_ROOT/bin/mwc.pl -type gnuace -recurse

cd $ACE_ROOT/TAO/tests
$ACE_ROOT/bin/mwc.pl -type gnuace -recurse

cd $ACE_ROOT/TAO/orbsvcs/tests
$ACE_ROOT/bin/mwc.pl -type gnuace -recurse
```

Run GNU Make in the following directories to build the tests:

```
$ACE_ROOT/tests
$ACE_ROOT/TAO/tests
$ACE_ROOT/TAO/orbsvcs/tests
```

This can be done very easily with the following commands:

```
cd $ACE_ROOT
make -C tests
make -C TAO/tests
make -C TAO/orbsvcs/tests
```

H.2 Running the Tests

H.2.1 ACE Tests

A script is provided to non-interactively run the ACE tests on UNIX, Windows, and LynxOS. However, there is currently no script to run the tests on VxWorks. On VxWorks, you will have to run the tests manually.

The automated test script runs the ACE tests that apply to the platform being tested. Output from each test is stored in a separate file in the `$ACE_ROOT/tests/log` directory. These files can be analyzed later for test bugs or other run-time problems.

- To run the ACE tests on LynxOS, change to the `$ACE_ROOT/tests` directory and run the perl script, `run_test.pl`.
- To run the ACE tests for VxWorks kernel mode, you must manually load the test you want to run using `ld` from VxWorks shell and run it by calling the `spa` function with `ace_main` as the first argument.
- To run the ACE tests for VxWorks RTP mode, you must run each test using `rtpSpawn` in the C Shell or `rtp exec` in the cmd shell.

The following tests in `$ACE_ROOT/tests` provide pretty good coverage of ACE features:

```
Basic_Types_Test
Cached_Allocator_Test
Collection_Test
```

```
Date_Time_Test
Dynamic_Priority_Test
High_Res_Timer_Test
INET_Addr_Test
MT_SOCK_Test
OS_Test
Object_Manager_Test
Priority_Task_Test
Reactor_Notify_Test
Reactors_Test
SOCK_Test
Task_Test
Thread_Mutex_Test
ARGV_Test
MT_Reactor_Timer_Test
```

H.2.2 TAO Tests

The TAO tests are less automated than the ACE tests. Therefore, the tests must be selectively run and the output checked by hand. Here are some of the more important tests to run, all found in `$TAO_ROOT/tests`:

```
Hello
IORManipulation
InterOp-Naming
Leader_Followers
MT_Client
MT_Server
NestedUpcall
  MT_Client_Test
  Simple
  Triangle_Test
ORB_init
OctetSeq
POA
  Current
  Default_Servant
  Etherealization
  Excessive_Object_Deactivations
  Identity
  MT_Servant_Locator
  Nested_Non_Servant_Upcalls
  Non_Servant_Upcalls
  Object_Reactivation
  Persistent_ID
  POA_Destruction
  Policies
  Single_Threaded_POA
```

```
  wait_for_completion
Param_Test
Timeout
```

Most TAO tests have a perl script, `run_test.pl`, that makes them a little easier to run on LynxOS. To run an individual test, change to the specific test directory and execute the `run_test.pl` script.

Testing on VxWorks can be performed in numerous ways. Since most tests consist of one or more client processes and one or more server processes, they cannot be run on VxWorks kernel mode. Even in RTP mode, many of these tests remain difficult to run. We will discuss three possible ways to run these tests:

- Run the client on the target and the server or servers on the host machine.

  Both machines may need access to input and output files. So remember, unless you have enabled NFS in the VxWorks kernel or have a target local file system, the file system that the tasks see is that of the machine on which the target server is running.

- Run the server on the target and the client or clients on the host machine.

  This is similar to the above method.

- When using kernel mode, it may be possible to combine the client or clients and server or servers into one module and run on the target.

  This will require writing a main function to spawn threads for each client and server.

Look at each test's `run_test.pl` script to see the order in which the test's programs should be run and the command line options they require. Many TAO tests depend upon the Naming Service, which can be run on the host or another machine. Many of the `orbsvcs` tests rely on multicast discovery of services; you will need to skip these tests if you do not have multicast routing enabled. Each test sends its output to the terminal.

H.2.3 ORB Services Tests

As with the TAO tests, the `orbsvcs` tests are less automated than the ACE tests. Most tests have a perl script, `run_test.pl`, just as the TAO tests. The following tests, found in `$TAO_ROOT/orbsvcs/tests`, are built by default. Of course, the tests you run will depend upon which services you have chosen to build.

```
Simple_Naming
EC_Multiple
EC_Throughput
EC_Mcast
EC_Custom_Marshal
Property
Time
Event
  Basic
  Performance
CosEvent
  Basic
ImplRepo
Trading
```

Note *The ImplRepo test depends upon the ability to spawn processes, thus it does not apply to VxWorks in kernel mode.*

Appendix I

CORBA Compliance

I.1 Introduction

This appendix contains detailed information about TAO's compliance with various OMG CORBA specifications. The information in this appendix should not be considered as a legally binding statement. It reflects the intent of the developers of TAO as based on their interpretation of the various OMG specifications. Presently, no generally recognized acceptance tests exist for the OMG specifications listed in this appendix.

Whereas compliance with OMG specifications is a design goal of TAO, it is also a continuing pursuit. As a user of TAO, you can help in this pursuit by contacting us if you feel TAO's implementation misinterprets any part of the cited OMG specifications, or even better, by contributing ideas and source code for how TAO's compliance could be improved.

Note *Throughout this document, "TAO" and "TAO 2.2a" refer to "OCI's Distribution of TAO, Version 2.2a."*

TAO is designed to be compliant with several OMG specifications. In this appendix, we address TAO's compliance with the following OMG specifications:

- CORBA 3.1 (Core, Interoperability, Interworking, and Quality of Service)
- CORBA for embedded (CORBA/e)
- Real-Time CORBA
- CORBA C++ Language Mapping
- Naming Service
- Notification Service
- Security Service

The following sections list the exact features of each of the above specifications that TAO supports as well as describing where TAO deviates from each specification.

I.2 CORBA 3.1

TAO is mainly compliant with the OMG CORBA 3.1 specification. This specification is defined in three separate parts. Part 1: CORBA Interfaces (OMG Document formal/08-01-04) and Part 2: CORBA Interoperability (OMG Document formal/08-01-06) are discussed in this section. Part 3: CORBA Component Model (OMG Document formal/08-01-08) covers the CORBA Component Model (CCM). CIAO, TAO's implementation of CCM, and its conformance to Part 3 are not discussed here.

Some areas of TAO, which have not implemented the new CORBA 3.1 remain compliant with the CORBA 3.0.3 specification (OMG Document formal/04-03-12).

Prior to CORBA 3.1, The CORBA Core specification divided compliance into four separate areas:

- CORBA Core
- CORBA Interoperability
- CORBA Interworking
- CORBA Quality of Service

The minimum required for a CORBA-compliant system was adherence to the specifications in CORBA Core and one language mapping. See I.5 for information about TAO's C++ language mapping compliance.

The CORBA 3.1 specifications are no longer organized along these lines and do not discuss compliance based on these categories. Because these categories remain useful, we continue to use them to organize our discussion of TAO's compliance in the sections below.

I.2.1 CORBA Core

TAO supports the CORBA 3.1 Core specification with the following exceptions:

- The IDL keyword `import` is not supported.
- Use of request contexts is not supported.
- IDL fixed data types are not supported.
- Domain Managers are not supported.
- Support for value types is nearly complete; exceptions are listed in 10.8.

I.2.2 CORBA Interoperability

An ORB is considered to be interoperability-compliant when it meets the following requirements:

- In the CORBA Core part of the specification, standard APIs are provided by an ORB to enable the construction of request-level inter-ORB bridges. APIs are defined by the Dynamic Invocation Interface (DII) (CORBA 3.1, Part 1, Chapter 11), the Dynamic Skeleton Interface (DSI) (CORBA 3.1, Part 1, Chapter 12), and by the object identity operations described in the Interface Repository chapter (CORBA 3.1, Part 1, Chapter 14).
- The Internet Inter-ORB Protocol (IIOP) (CORBA 3.1, Part 2, Chapter 9) defines a transfer syntax and message formats — described independently as the General Inter-ORB Protocol (GIOP) — and defines how to transfer messages via TCP/IP connections. The IIOP can be supported natively or via a half-bridge.

TAO fully supports the DII and DSI APIs. TAO also fully supports the object identity operations, such as `CORBA::Object::_is_equivalent()` and `CORBA::Object::_is_a()`. TAO implements IIOP as a pluggable protocol,

and is interoperable with ORBs that support IIOP version 1.0, 1.1, or 1.2, including bi-directional GIOP/IIOP (CORBA 3.1, Part 2, Chapter 9).

TAO supports the dynamic management of `any` values as described in the CORBA specification (CORBA 3.1, Part 1, Chapter 13) with the exception of the `DynamicAny::DynAnyFactory` interface which is missing the `create_dyn_any_without_truncation` and `create_multiple_dyn_anys` operations.

CORBA 3.1 also defines IIOP versions 1.3 and 1.4. IIOP version 1.3 allows the existing IIOP 1.2 messages to be used with component-related types. IIOP 1.4 improves support for certain wide character types. While TAO does not support IIOP 1.3 and 1.4, it supports the intended component-related functionality of IIOP 1.3 with its IIOP 1.2 implementation.

I.2.3 CORBA Interworking

The main purpose of the Interworking architecture was to specify support for two-way communication between CORBA objects and COM objects. Most of this material has been removed from the CORBA 3.1 specifications. TAO has never supported the CORBA Interworking architecture.

CORBA Interworking also defines the Portable Interceptor facilities (CORBA 3.1, Part 1, Chapter 16). TAO implements the majority of the interfaces and features described in this chapter, with the following exceptions:

- IOR interceptors are not completely implemented
    - The `ObjectReferenceFactory` interface does not support the `equals()` and `make_profiles()` operations.

I.2.4 CORBA Quality of Service

Quality of Service (QoS) is a general concept that is used to specify the behavior of a service. Programming service behavior by means of QoS settings offers the advantage that the application developer need indicate only *what* is wanted rather than *how* this QoS should be achieved. Generally speaking, QoS is comprised of several QoS policies. Each QoS policy is an independent description that associates a name with a value. Describing QoS by means of a list of independent QoS policies gives rise to greater flexibility in CORBA applications.

TAO supports the Quality of Service portions of the CORBA 3.1 specification as follows:

- TAO fully supports the policy management framework.
- TAO implements a subset of the Messaging QoS policies, such as synchronization scope (`SyncScope`) for `oneway` requests and relative round-trip time-outs (`RelativeRoundtripTimeout`), and extends the Messaging QoS policies with the addition of policies for connection time-out and `oneway` buffering constraints (see Chapter 6).
- TAO implements the Asynchronous Method Invocation (AMI) callback model, but not the polling model (see Chapter 6).
- TAO does not support Time-Independent Invocation (TII).

I.3 CORBA for Embedded

TAO implements the *CORBA for Embedded* (CORBA/e) specification defined in OMG Document formal/08-11-06. This specification defines two profiles (or subsets) of CORBA that are designed for systems with limited resources. It attempts to satisfy the resource constraints of such systems while preserving portability, interoperability, and full IDL support. The CORBA/e Compact Profile, formerly known as *minimum CORBA*, is targeted to embedded systems running with constrained resources. The CORBA/e Micro Profile is targeted for systems with severely constrained resources. Support for the CORBA/e Compact and Micro Profiles are disabled by default when TAO is built. The preprocessor macros `CORBA_E_COMPACT` and `CORBA_E_MICRO` are used to enable/disable support for these profiles when building TAO.

TAO supports the use of the `TAO_HAS_MINIMUM_CORBA` preprocessor macro. Its effects are similar to the CORBA/e Compact Profile described below. The main difference is that the minimum CORBA profile also disables support for AMI. The minimum CORBA profile also allows for selectively re-enabling a number of features. AMI support can be enabled by defining `TAO_HAS_AMI` and the advanced POA features can be enabled by defining `TAO_HAS_MINIMUM_POA` (as 0).

I.3.1 CORBA/e Compact Profile

The following features are disabled in TAO when the *CORBA/e Compact Profile* support is enabled:

- Dynamic Invocation Interface (DII).
- Dynamic Skeleton Interface (DSI).
- DynamicAny.
- Interface Repository.
- Implementation Repository.
- Remote Policies.
- Interceptors.
- The following interfaces, exceptions, and types in module `CORBA`:

```
Context and ContextList
Request and RequestSeq
ServerRequest
ConstructionPolicy
NamedValue, NVList, and NameValuePair
WrongTransaction
AnySeq
DynAny, DynSequence, DynStruct, etc.
FieldName
ORB::InconsistentTypeCode
```

- The following operations on `CORBA::Object`:

```
_non_existent()
_get_implementation()
_get_interface()
_get_component()
_create_request()
_request()
```

- `PortableServer::ForwardRequest`.
- Support for pluggable protocols other than IIOP.
- Value Boxes.
- Abstract Interfaces

In addition, the following advanced POA features in the `PortableServer` module are disabled by default when the *CORBA/e Compact Profile* support is enabled:

- The following CORBA policies:

```
ThreadPolicy
ImplicitActivationPolicy
ServantRetentionPolicy
RequestProcessingPolicy
```

- Servant Managers (`ServantActivator` and `ServantLocator`).
- Default Servant.
- Adapter Activators.
- The following POA Manager operations and associated POA Manager states:

```
hold_requests()
discard_requests()
deactivate()
```

- The following `UserException` types in interface `PortableServer::POA`:

```
AdapterInactive
NoServant
```

I.3.2 CORBA/e Micro Profile

The following additional features are disabled in TAO when the *CORBA/e Micro Profile* support is enabled (in addition to those listed in the Compact profile):

- Value Types
- Any
- Type Codes
- Policies
- Creating POAs (only a single Root POA is supported)

I.4 Real-Time CORBA

TAO implements the Real-Time CORBA version 1.2 specification (OMG Document formal/05-01-04). Real-Time CORBA is an optional set of extensions to CORBA tailored to equip ORBs to be used as a component of a real-time system. The goals of the specification are to support developers in meeting real-time requirements by facilitating the end-to-end predictability of activities in the system and by providing support for the management of resources.

The Real-Time CORBA specification defines one mandatory compliance point, as defined in Appendix A of the Real-Time CORBA 1.2 specification. The Real-Time CORBA Dynamic Scheduling, defined in Chapter 3 of the Real-Time CORBA 1.2 specification, is a separate and optional compliance point. TAO supports Real-Time CORBA Dynamic Scheduling.

Support for Real-Time CORBA is enabled by default when TAO is built. The preprocessor macro `TAO_HAS_RT_CORBA` is used to enable/disable Real-Time CORBA support when TAO is built. Real-Time CORBA support is dependent upon CORBA Messaging support.

TAO supports all of the features of the mandatory portion of the Real-Time CORBA 1.2 specification with the following exceptions:

- Priority transforms are not supported.
- Threadpool request buffering is not supported.
- Thread borrowing among threadpool lanes is not supported.

TAO extends the Real-Time CORBA 1.2 specification in the following ways:

- Addition of a Priority Mapping Manager.
- Addition of named mutexes.
- Use of a reactor-per-lane threadpool model.
- RT CORBA protocol configuration in TAO supports the selection and configuration of certain TAO pluggable protocols.
- `TCPProtocolProperties` are extended to allow the application to enable network priorities via Diffserv code points (DSCP).

See Chapter 8 for more information about using the features of the Real-Time CORBA specification with TAO.

I.5 C++ Language Mapping

TAO is compliant with the CORBA C++ Language Mapping version 1.2 specification (OMG Document formal/08-01-09) except for those elements of the CORBA Core specification that are not supported as described in I.2.1 (i.e., fixed data types).

TAO no longer supports the alternative mappings for modules and exceptions (as described in Section 4.46 of the V1.2 C++ Mapping).

I.6 Naming Service

TAO implements the Naming Service version 1.2 specification (OMG Document formal/02-09-02). In addition to the `IOR`, `corbaloc`, and `corbaname` ObjectURL formats, TAO supports the following ObjectURL schemes:

- `file`
- `iiop`
- `mcast`
- `http`

and optionally supports the following object URL schemes for pluggable protocols implemented by TAO (though the `corbaloc` ObjectURL scheme is preferred):

- `uiop`
- `shmiop`
- `diop`
- `uipmc`
- `sciop`
- `htiop`
- `ziop`

The `iiop`, `uiop`, `shmiop`, `diop`, `uipmc`, `sciop`, `htiop`, and `ziop` schemes are based on pluggable protocol implementations that are supplied with TAO. TAO supports the addition of custom pluggable transport protocols and their associated object URL schemes. See Chapter 14 for more information on pluggable protocols.

I.7 Notification Service

TAO implements the Notification Service version 1.1 specification (OMG Document formal/04-10-13) with the following exceptions:

- Only the push model is supported; the pull model is not supported.
- Mapping filters are not supported.
- Only structured and untyped events are supported; typed events are not supported.
- Filtering of untyped events is only partially supported.
- Filtering supports both the Trader Constraint Language (TCL) and the Extended Trader Constraint Language (ETCL).
- All standard Quality of Service properties are supported except for:
    - StartTime
    - StopTime
    - StartTimeSupported
    - StopTimeSupported
- Some of the standard QoS properties are not applicable at all levels defined by the specification (see 25.4.6).
- The Event Type Repository (which is an optional compliance point) is not supported.

See Chapter 25 for more information on using TAO's Notification Service. See also `$TAO_ROOT/docs/releasenotes/notify.html`.

I.8 Security Service

The implementation of CORBA Security that is in the version of TAO from the DOC Group that forms the basis for OCI's TAO 2.2a is in a state of transition. Previous versions of TAO included a partial implementation of CORBA Security and SSLIOP. The implementation in TAO 2.2a is based on newer versions of the CORBA Security specification. Unfortunately, the implementation is incomplete, so some features are not available. Further, some features that were available in previous releases may not be available as

the implementation currently exists. Specifically, *Policy Enforcing* applications will very likely not work properly, though all *Security Unaware* and many *Policy Changing* applications will work.

See Chapter 27 for more information on using TAO's implementation of CORBA Security.

References

Bolton, Fintan. 2002. *Pure CORBA: A Code-Intensive Premium Reference*. Sams Publishing.

Deshpande, Mayur, Douglas C. Schmidt, Carlos O'Ryan, and Darrell Brunsch. 2002. *Design and Performance of Asynchronous Method Handling for CORBA*. Department of Electrical and Computer Engineering, University of California, Irvine.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Harrison, Timothy H., David L. Levine, and Douglas C. Schmidt. 1997. "The Design and Performance of a Real-time CORBA Event Service." In *Proceedings of OOPSLA '97* (Atlanta, GA, October 1997). Association for Computing Machinery.

Henning, Michi and Steve Vinoski. 1999. *Advanced CORBA Programming with C++*. Addison-Wesley.

Huston, Stephen D., James CE Johnson, and Umar Syyid. 2004. *The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming*. Addison-Wesley.

Kuhns, Fred, Douglas C. Schmidt, Carlos O'Ryan, Ossama Othman, and Bruce Trask. 2000. *Implementing Pluggable Protocols for TAO*. St. Louis, MO: Center for Distributed Object Computing, Washington University.

Lewis, Bil, Daniel J. Berg. 1996. *Threads Primer: A Guide to Multithreaded Programming*. Sun Microsystems, Inc.

Martin, Robert C., Dirk Riehle, Frank Buschmann. 1998. *Pattern Languages of Program Design 3: Software Patterns Series.* Addison-Wesley Longman.

The Object Management Group.[1] 2008. *Common Object Request Broker Architecture Specification*, *Version 3.1 Part 1: CORBA Interfaces*, January 2008. OMG Document formal/08-01-04.

The Object Management Group. 2008. *Common Object Request Broker Architecture Specification*, *Version 3.1 Part 2: CORBA Interoperability*, January 2008. OMG Document formal/08-01-06.

The Object Management Group. 2008. *Common Object Request Broker Architecture Specification*, *Version 3.1 Part 3: CORBA Component Model*, January 2008. OMG Document formal/08-01-08.

The Object Management Group. 2009. *C++ Language Mapping Specification*, Version 1.2, January 2008. OMG Document formal/08-01-09.

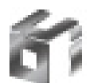
The Object Management Group. 2004. *Event Service Specification*, Version 1.2, October 2004. OMG Document formal/04-10-02.

The Object Management Group. 2004. *Extensible Transport Framework (ETF)*, Final Adopted Specification, March 2004. OMG Document ptc/04-03-03.

The Object Management Group. 2002. *Naming Service Specification*, Version 1.2, September 2002. OMG Document formal/02-09-02.

The Object Management Group. 2004. *Notification Service Specification*, Version 1.1, October 2004. OMG Document formal/04-10-13.

1. See page xxxi in the Preface for instructions on obtaining OMG documents.



The Object Management Group. 2002. *Persistent State Service Specification*, Version 2.0, September 2002. OMG Document formal/02-09-06.

The Object Management Group. 2005. *Real-Time CORBA Specification*, Version 1.2, January 2005. OMG Document formal/05-01-04.

Pyarali, Irfan, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishall Kachroo, and Aniruddha Gokhale. 1999. "Applying Optimization Principle Patterns to Real-time ORBs." In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)*. San Diego, CA: May 1999.

Ruh, William, Thomas Herron, Paul Klinker. 1999. *IIOP Complete: Understanding CORBA and Middleware Interoperability.* Addison-Wesley Longman.

Schmidt, Douglas C. 1995. "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching." In *Pattern Languages of Program Design* (James O. Coplien and Douglas C. Schmidt, editors). Addison-Wesley.

Schmidt, Douglas C. and Steve Vinoski. 1998. "An Introduction to CORBA Messaging." In *C++ Report*, November/December 1998. SIGS Publications.

Schmidt, Douglas C. and Steve Vinoski. 1999. "Programming Asynchronous Method Invocations with CORBA Messaging." In *C++ Report*, February 1999. SIGS Publications.

Schmidt, Douglas C. and Steve Vinoski. 2001. "Object Interconnections: Real-time CORBA, Part 1: Motivation and Overview." In *C/C++ Users Journal C++ Experts Forum*, December 2001. CMP Media.

Schmidt, Douglas C. and Steve Vinoski. 2003. "Object Interconnections: CORBA Metaprogramming Mechanisms, Part 1: Portable Interceptors Concepts and Components." In *C/C++ Users Journal C++ Experts Forum*, March 2003. CMP Media.

Schmidt, Douglas C., Michael Stal, Hans Rohnert, and Frank Buschmann. 2000. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* (POSA2). John Wiley & Sons.

Schmidt, Douglas C., Stephen D. Huston. 2001. *C++ Network Programming, Volume 1: Mastering Complexity Using ACE and Patterns* (C++NPv1). Addison-Wesley Longman.

Schmidt, Douglas C., Stephen D. Huston. 2003. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks* (C++NPv2). Addison-Wesley Longman.

Stevens, W. Richard. 1998. *UNIX Network Programming*, Second Edition, Volumes 1 and 2. Prentice Hall.

Index

Symbols

_add_ref() operation 264, 309
_get_policy() operation 207
_id() operation 62, 64–65, 67, 92, 252, 339, 699–700
_make_resources() operation 439
_narrow() operation 313
_refcount_value() operation 264
_remove_ref() operation 264, 309
_set_policy_overrides() operation 105
_unchecked_narrow() operation 285–286, 340
_validate_connection() operation 108, 151, 167

A

abstract factory pattern 16
abstract keyword
  value types 272
abstract value type 269, 272, 274
acceptor 462, 549
acceptor pattern 16
access identity 967
AccessDecision object 1026
ACE 592
  service configurator 12, 433–434, 436–448, 450–458, 463, 541, 550, 574, 595, 604, 617, 620, 623
    _make_resources() operation 439
    ACE_Dynamic_Service class 456
    ACE_FACTORY_DECLARE macro 453
    ACE_FACTORY_DEFINE macro 453
    ACE_SERVICE_ALLOCATOR 451
    ACE_Service_Config class
      close() operation 443
      open() operation 441–442, 463, 479–480, 541–542
      process_directive() operation 444
      reconfigure() operation 444, 458
    ACE_Service_Object class 438, 445, 447, 449, 451
    ACE_STATIC_SVC_DECLARE macro 453
    ACE_STATIC_SVC_DECLARE_EXPORT macro 453
    ACE_STATIC_SVC_DEFINE macro 453
    ACE_STATIC_SVC_REQUIRE macro 454
    ACE_SVC_NAME macro 452, 454
    ACE_SVC_OBJ_T macro 452
    Client_Strategy_Factory configuration 437
    commanding 443
    control options 439
    DELETE_OBJ macro 452
    DELETE_THIS macro 452
    destructor 450
    dynamic directive 438, 447, 450, 454
    example
      dynamic service 454
      static service 455
    factories 434

finalizer 450
fini() operation 450, 452
framework 440
helper macros 452
info() operation 456
init() operation 450
instance() operation 457
loading service objects 440
obtaining services 456
options 437
ORB initialization options 52, 55, 439, 441, 459–460, 473, 949
ORBServiceConfigLoggerKey 440
ORBSkipServiceConfigOpen 440
ORBSvcConf 440
ORBSvcConfDirective 440
process_directive() operation 458
remove directive 444, 448, 450
Resource_Factory configuration 437–438, 447
resume directive 444–445, 448, 456
Server_Strategy_Factory configuration 437
service finalization 450
service initialization 450
service manager 434, 457
service objects 445, 448
service state 456
static directive 437, 447, 450–451, 457, 595, 604
suspend directive 444–445, 448, 456
XML 434, 445
service manager 444–445

ACE classes

ACE_Addr 779
ACE_Allocator 551
ACE_Data_Block 553–554
ACE_Dynamic_Service 456–457
ACE_Dynamic_Service_Base 456
ACE_Event_Handler 375–376, 449
ACE_Log_Msg 443
ACE_MEM_Acceptor 354
ACE_MEM_Connector 355
ACE_Message_Block 553, 556, 578
ACE_QtReactor 568
ACE_Reactor 441, 449–450, 551–552

ACE_Select_Reactor 411, 584
ACE_Service_Config 440–444, 458, 463, 479–480, 537, 541–542
close() operation 443
open() operation 441–442, 463, 479–480, 541–542
process_directive() operation 444
reconfigure() operation 444, 458
ACE_Service_Object 438, 445, 447–449, 451, 454, 551, 596–597, 618
ACE_Service_Object ACE_Service_Object class 452, 454–455
ACE_Service_Type 452
ACE_Shared_Object 449
ACE_Singleton 282
ACE_SOCK_Acceptor 354
ACE_SOCK_Connector 355
ACE_Static_Svc_Descriptor 451, 453
ACE_Strategy_Acceptor 354
ACE_Strategy_Connector 355
ACE_Task_Base 395, 665
ACE_TP_Reactor 411
ACE_XtReactor 569–570
ACE macros
__ACE_INLINE__ 41
ACE_COMPILE_TIMEPROBES 1097
ACE_DECLARE_NEW_CORBA_ENV 570
ACE_DEFAULT_LOGGER_KEY 442, 537
ACE_DEFAULT_MAX_SOCKET_BUFSIZ 536, 539
ACE_ENDTRY 569, 571
ACE_FACTORY_DECLARE 453–455, 562
ACE_FACTORY_DEFINE 453–455, 562
ACE_HAS_XML_SVC_CONF 445
ACE_MAX_DGRAM_SIZE 335
ACE_NDEBUG 1095
ACE_NO_INLINE 1096
ACE_SERVICE_ALLOCATOR 451
ACE_STATIC_SVC_DECLARE 453
ACE_STATIC_SVC_DECLARE_EXPORT 453, 455, 562
ACE_STATIC_SVC_DEFINE 453, 455, 562
ACE_STATIC_SVC_REQUIRE 454–455, 562
ACE_SVC_NAME 452, 454–455
ACE_SVC_OBJ_T 452, 455
ACE_TCHAR 452
ACE_TRY 569–570
ACE_TRY_CHECK 569–570

DELETE_OBJ 452, 455
DELETE_THIS 452, 455
ACE service configurator
See ACE, service configurator
ACE_Addr class 779
ACE_Allocator class 551
ACE_Data_Block class 553–554
ACE_Dynamic_Service class 456–457
ACE_Dynamic_Service_Base class 456
ACE_Event_Handler class 375–376, 449
ace_for_tao build flag 1094
ACE_HAS_XML_SVC_CONF macro 445
ACE_Log_Msg class 443
ACE_MEM_Acceptor class 354
ACE_MEM_Connector class 355
ACE_Message_Block class 553, 556, 578
ACE_QtReactor class 568
ACE_Reactor class 441, 449–450, 551–552
ACE_ROOT environment variable 26, 40, 45, 52, 1106
ACE_Select_Reactor class 411, 584
ACE_Service_Config class 440–444, 458, 463, 479–480, 537, 541–542
ACE_Service_Object class 438, 445, 447–449, 451, 454, 551, 596–597, 618
ACE_Service_Type class 452
ACE_Shared_Object class 449
ACE_Singleton class 282
ACE_SOCK_Acceptor class 354
ACE_SOCK_Connector class 355
ACE_Static_Svc_Descriptor class 451, 453
ACE_Strategy_Acceptor class 354
ACE_Strategy_Connector class 355
ACE_Task_Base class 395, 665
ACE_TP_Reactor class 411
ACE_XtReactor class 569–570
activate_object_with_id_and_priority() operation 211
activate_object_with_id() operation 92, 339
activate_object_with_priority() operation 210
activate_object() operation 30
activate() operation 30, 235, 397, 452, 598, 649, 666–667, 880, 904, 1025
activation 92
activation-per-AMI-call strategy 91–92
active demultiplexing 11, 16, 413, 427, 476, 601, 607–608, 610, 613–614
active object map 92, 302, 600, 602–603, 606, 614

parameters 602
active object pattern 16
adapter pattern 16
adapter_manager_state_changed() operation 245
adapter_state_changed() operation 245
ADAPTIVE Communication Environment (ACE)
See ACE
add_client_request_interceptor() operation 230
add_constraints() operation 890
add_filter() operation 890
add_ior_component() operation 246
add_ior_interceptor() operation 245, 247
add_server_request_interceptor() operation 234
Advanced CORBA Programming Using TAO course xxxv
Advanced CORBA Programming with C++ book xxvii, xxxi, xxxv, 8, 25, 60–61, 79, 105, 140, 278, 553, 645–647, 695, 833, 1039, 1157
advanced resource factory 328, 409, 550, 552–555, 558, 565, 571–573, 590
options 588
alert protocol 992
AMH
See asynchronous method handling (AMH)
amh_response_handler_allocator() operation 565
AMI
See asynchronous method invocation (AMI)
ami build flag 1093–1094
ami_response_handler_allocator() operation 565
ANY_EVENT type 872, 878
asymmetric encryption 985
asymmetric protocol 341–342
asynchronous invocations 173
asynchronous method handling (AMH) 72, 125–147, 363, 564, 1157
advantages 126
disadvantages 127
example 128, 131
response handler 128, 130
servant 128
skeleton 128
using with
collocation 141
CORBA::Current objects 140
portable interceptors 139
reference counted servants 141

asynchronous method invocation (AMI) 75, 78, 126, 278, 361, 426, 564, 1159
- activation-per-AMI-call strategy 91–92
- associating replies with requests 91
- callback 76–77, 95–96, 130
- drawbacks 78
- example 95
- exception replies 83
- ExceptionHolder class 80–81, 83
- IDL compiler 78
- local interfaces 300
- processing sendc_ operation 93
- reply handler 82, 84
  - exception 82, 90
  - non-exception 82, 86
- request delivery 102
- sendc_ prefix 77, 79, 93
- servant-per-AMI-call strategy 91–92
- server-differentiated-reply strategy 91–92

audio/video streaming service 14, 637
- ORB service libraries 637

auditing 968
- identity 967

AV streams
- *See audio/video streaming service*

B

BAD_INV_ORDER exception 72, 371–372, 375
BAD_PARAM exception 71, 248
BAD_QOS exception 848
basic filter builder
- real-time event service (RTES) 757

basic object adapter (BOA) 11
begin_scheduling_segment() operation 174, 176–177, 212, 214
Berg, Daniel 1158
bi-directional GIOP 341
- BidirectionalPolicyValue 116
- *See also general inter-ORB protocol (GIOP), bi-directional*

bind() operation 312

binding
direct 1038
indirect 1038
bindings 652, 670, 980–981, 986
explicit 11, 151, 166
indirect 312, 1053
bit-mask filters 759
real-time event service (RTES) 759
blocking strategy 578
BlockingPolicy property 844–845, 916
BOA
See basic object adapter (BOA)
Bolton, Fintan xxvii, xxxi, 1157
Borland 32, 1085
boxed value types 274–275
bridge pattern 16
Brunsch, Darrell 1157
BUFFER_MESSAGE_COUNT 113–114
BUFFER_TIMEOUT 113
buffering
oneways 112
requests 152
BufferingConstraint policy 112–114
build flags 281, 287
ace_for_tao 1094
ami 1093–1094
corba_messaging 201, 1093–1094, 1098
debug 1091, 1095
ec_typed_events 1095
fast 1091, 1095
fl 1092, 1095
fox 1092, 1096
inline 41, 45–46, 1091, 1096
interceptors 1093, 1096
minimum_corba 201, 220, 1093, 1095–1098
optimize 1091, 1097
pipes 1091, 1097
probe 1093, 1097
profile 1092, 1097
purify 1092, 1097–1098
qt 1092, 1098
quantify 967, 1092, 1098

repo 1098
rt_corba 201, 1093, 1098
rwho 1094, 1099
sctp 1093, 1099
shared_libs 1092, 1099
shared_libs_only 1092, 1099
split 1092, 1099
ssl 1002–1003, 1093, 1100
static_libs 1092, 1100
static_libs_only 1092, 1100
static_link 1092, 1100
stlport 1092–1093, 1100
templates 1101
threads 1092, 1101
tk 1093, 1101
tk_reactor 573, 592
versioned_so 1092, 1101
wfmo 1093, 1102
winregistry 1093, 1102
xt 1093, 1102
zlib 1102

building 19
$ACE_ROOT/ace/config.h 41, 446, 568, 570, 1091
$ACE_ROOT/bin/generate_export_file.pl 52, 453
$ACE_ROOT/bin/svcconf-convert.pl 446
$ACE_ROOT/include/makeinclude/platform_macros.GNU 1090
ACE and TAO 19, 1103
from source code distribution 1103
on UNIX 1103
UNIX 1105–1112
with Visual C++ 1103, 1115
build libraries 1119
configure source code 1117
setting up environment 1117
verify build 1120
cross compilation
MakeProjectCreator (MPC) 25, 1106, 1109
messaging 1094, 1098
security libraries 1001
on UNIX 1002
on Windows 1003

Buschmann, Frank 17, 393, 423, 1158–1159

C

C++ language mapping 5, 13, 1152
CORBA compliance 1152
See also interface definition language (IDL), C++ mapping
C++ Network Programming book
Volume 1 (C++NPv1) 17, 358, 365, 1159
Volume 2 (C++NPv2) 17, 370, 386, 398, 409, 411, 434, 1160
cache_maximum() operation 557–558
caching
cache_maximum() operation 557–558
create_cached_connection_lock() operation 557
create_purging_strategy() operation 557
locked_transport_cache() operation 557
management 557
max_muxed_connections() operation 557–558
purge_percentage() operation 557
cancel() operation 175
CDR
See common data representation (CDR)
CEC_Factory configuration 715, 722–727, 729–731, 733–740
CECConsumerControl option 717, 722–725, 731, 738
CECConsumerControlPeriod option 717, 723, 731, 737–738
CECConsumerControlRoundtripTimeout option 724
CECConsumerControlTimeout option 717–718, 724
CECConsumerOperationTimeout option 718, 725
CECDispatching option 715–716, 719, 726–727
CECDispatchingThreads option 715–716, 726–727
CECProxyConsumerCollection option 719, 728–729
CECProxyConsumerLock option 717, 730
CECProxyDisconnectRetries option 718
CECProxySupplierCollection option 719, 732–733
CECProxySupplierLock option 717, 734
CECReactivePullingPeriod option 720, 735
CECSupplierControl option 717, 736–737, 739
CECSupplierControlPeriod option 717, 737
CECSupplierControlRoundtripTimeout option 738
CECSupplierControlTimeout option 718
CECSupplierOperationTimeout option 718, 739
CECUseORBId option 720–721, 740
certificates 986–987, 994

authority 987, 995
commands summary 1001
creating requests 996
issuing 998
multiple authorities 1007
self-signed 987
signing requests 998

character set
definition 563

ciphertext 984

client interceptors 221–222, 227, 239

client role 359

client strategy factory 412–417, 421–422, 436–437, 462, 617–631
cached connection strategy 575
Client_Strategy_Factory configuration 389, 414–415, 421–422, 424, 426, 437–438, 617, 625–630
connect strategy 623, 625
interface definition 618
multiplexing strategies 619
options 624
ORBClientConnectionHandler 622
ORBConnectionHandlerCleanup 622, 624–625
ORBConnectStrategy 414, 421, 624–626
ORBProfileLock 389, 619, 626–627
ORBReplyDispatcherTableSize 627
ORBTransportMuxStrategy 391, 415–416, 421, 620, 628
ORBTransportMuxStrategyLock 620, 629
ORBWaitStrategy 387, 389, 414, 416–417, 421–422, 424, 426, 622, 629–630
profile locking 618, 626
TAO_Wait_Strategy class 621, 623
transport multiplexing 619
wait strategy 620, 630

Client_Strategy_Factory configuration 389, 412, 414–415, 421–422, 424, 426–427

ClientInterceptor class 227

client-propagated priority model 205, 207

ClientRequestInfo interface 223
get_effective_component() operation 245

CLOSE_WAIT socket state 425

code set 563
configuration 563–564
definition 563

code set translator 564
codec (coder/decoder) 237
  codec_factory() operation 237, 241
  CodecFactory reference 237–238
codepoints
  diffserv 121, 191
Collocated-only inter-ORB protocol (COIOP) 322
collocation 10
  asynchronous method handling (AMH) 141
  event channel 702
  notification channel
    example 903
  objects 361, 465, 476, 704
  optimization 427–429
  real-time collocation resolver 205, 428, 487
COMM_FAILURE exception 61, 65, 70
common data representation (CDR) 320–321, 435, 556
  conversion allocators 553
common secure interoperability (CSI) packages 962, 971, 974–975
  CSI 0 971, 974
  CSI 1 962, 971, 974
  CSI 2 962, 971, 974
compliance
  See CORBA compliance
components_established() operation 245
concurrency xxxv, 4, 12–13, 16, 433, 595, 597
  behavior 607, 611
  server 597
  strategy 597
concurrency control service 4, 14, 638
  ORB service libraries 638
concurrency models 335, 340, 385–395, 462
  reactive 385–389, 394
  thread-per-connection 335, 340, 385, 389–392, 599, 612, 706
  thread-pool 385, 392–397, 411, 427
  wait-on-leader-follower 630–631
config.h file 41, 446, 567–568, 570, 586, 706, 716, 771, 786, 818, 1084, 1091, 1094, 1101, 1107–1108, 1111–1112, 1117, 1125, 1127, 1131, 1136
configuration
  source code
    See also building
conjunction groups 760

connect strategies 413–414, 623
blocking 414
leader-follower 413
reactive 414
connect_push_consumer() operation 705, 750, 756, 771
connect_push_supplier() operation 697, 746, 771
connect_sequence_push_consumer() operation 902
connect_sequence_push_supplier() operation 898
connect_structured_push_consumer() operation 879
connect_structured_push_supplier() operation 873
connecting and disconnecting
real-time event service (RTES) 750–751
connection handler 549
connection timeout 108
ConnectionReliability property 840–841, 845, 848–849, 864
connector 549
connector pattern 16
consumer and supplier control options 717, 789
CECConsumerControl 717
CECConsumerControlPeriod 717
CECConsumerControlTimeout 717–718
CECConsumerOperationTimeout 718
CECSupplierControl 717
CECSupplierControlPeriod 717
CECSupplierControlTimeout 718
CECSupplierOperationTimeout 718
ECConsumerControl 789, 793, 820–825
ECConsumerControlPeriod 789, 794
ECConsumerControlTimeout 789, 795
ECConsumerValidateConnection 796
ECObserver 802
ECSupplierControl 789, 811
ECSupplierControlPeriod 789, 812
ECSupplierControlTimeout 790, 813
consumer proxies 742
ConsumerAdmin interface 828, 836, 877–878, 884, 892, 901
context_data field 228
context_id field 228
CORBA
architecture 8, 152
audio/video streaming service 14
client role 359

compliance xxviii, 7, 13, 15, 1145–1155
  C++ language mapping 1152
  CORBA Component Model (CCM) 1155
  CORBA for Embedded 1149
  core 1147
  DII COE 1155
  interoperability 1147
  interworking 1148
  naming service 1153
  notification service 1154
  quality of service (QoS) 1148
  real-time CORBA 1151
  security service 1154
  value types 274
concurrency control service 14
CORBA::Any 237, 695
CORBA::Boolean 252
CORBA::OctetSeq 242
core 1147
  specification 76
data distribution service (DDS) 14
environment parameter
  exceptions 60
event service 14
gateway 774
interface repository 14
interoperability 1147
interworking 1148
life cycle service 14
load balancing service 14
log service 14
messaging
  See messaging
minimum 1097
naming service 14
notification service 14
object 10, 70, 475, 540, 645, 704
ORBPolicyManager interface 168
PolicyCurrent interface 168
PolicyList interface 163–164
property service 14
quality of service (QoS) 1148

real-time 72, 350, 385, 391, 394, 1159
See also real-time CORBA
security service 14
server role 359, 361
services 14
system exceptions 61
time service 14
trading service 14
VoidData native type 176

CORBA Programming with C++ course xxxiv
corba_messaging build flag 201, 1093–1094, 1098
CORBA/e Compact Profile 1149–1150
CORBA/e Micro Profile 1149
corbaloc
object URL
example 655
corbaloc object URL 311–312, 640–641, 654–658, 661, 670
example 656
iiop protocol 655
rir protocol 655–656
corbaname object URL 579, 657–660, 663, 1153
example 659
CosEvent_Service program 695, 702
command line options 713–714
create_cached_connection_lock() operation 557–558
create_channel() operation 870, 904
create_corba_object_lock() operation 566
create_filter() operation 889
create_flushing_strategy() operation 556
create_for_unmarshal() operation 266
create_lf_strategy() operation 553
create_mutex() operation 161
create_named_mutex() operation 198
create_POA() operation 105, 153, 157, 163
create_policy() operation 103–104, 107–108, 111, 114, 117, 1015–1018
create_priority_banded_connection_policy() operation 167
create_proxy() operation 278, 282–283, 285, 288, 293
create_purging_strategy() operation 557
create_reference_with_id_and_priority() operation 211
create_reference_with_priority() operation 211
create_resource_manager() operation 178, 181
cross compilation

See building, cross compilation
CSD
See Custom Servant Dispatching (CSD)
CSI
See common secure interoperability (CSI) packages
CSI ECMA protocol 971, 975
Current interface 92, 159, 208, 249, 301, 1019
custom factory 583
custom interoperable object reference (IOR) parsers 560
Custom Servant Dispatching (CSD) 50, 363, 403
customer support
See support
customizing
access to services 636
ACE and TAO builds
See building
event structure 755
filters 764

D

data distribution service (DDS) 14
data-centric publish-subscribe (DCPS)
instance 50
marshaling 50
Datagram inter-ORB protocol (DIOP) 169
datagram inter-ORB protocol (DIOP) 12, 169, 322, 333–335, 517, 522, 583
DIOP_Factory configuration 334, 583
endpoints 522
DCE common inter-ORB protocol (DCE-CIOP) 971, 974
deadline timeouts 765, 767
debug build flag 1091, 1095
debugging
ACE_NDEBUG macro 1095
ORBDebug option 460, 462, 464, 480
ORBDebugLevel option 462, 464, 480–481
TAO_ORB_DEBUG environment variable 480
default proxy factory 283, 285
default resource factory 328, 550–551, 553–556, 558–559, 566, 572
default_filter_factory() operation 889

default.features file 1084
DefaultValueRefCountBase class 264
demarshaling 51, 70, 92, 413, 427, 476–477
demultiplexing 595
  active 11, 16, 599, 601
  dynamic hash 599–600
  linear search 599–600
  strategies 599
Deshpande, Mayur 1157
design patterns
  See also patterns
Design Patterns: Elements of Reusable Object-Oriented Software book 17, 1157
destroy_mutex() operation 161
destroy() operation 30, 64–65, 305, 364
DevGuideExamples directory xxx
differentiated services 121, 191
  codepoints (DSCP) 121, 191
  diffserv field (DF) 121, 191
DII
  See dynamic invocation interface (DII)
DIIPollable interface 302
DIOP
  See datagram inter-ORB protocol (DIOP)
DIOP_Factory configuration 334
direct binding 1038
direct mapping 187–188
directives 436, 444
  dynamic 324, 389, 438, 450, 454
    components 438
  resume 445, 456
  static 389, 437, 450, 550, 595, 617, 715, 785, 818
  suspend 445, 456
DiscardPolicy property 839, 842, 844–846, 916
disconnect_push_consumer() operation 699, 705, 711, 748, 771
disconnect_push_supplier() operation 697–698, 705, 746, 771
disconnect_sequence_push_consumer() operation 899–900
disconnect_sequence_push_supplier() operation 894
disconnect_structured_push_consumer() operation 874–876
disconnect_structured_push_supplier() operation 867
disjunction group 759
dispatching

events 716
module 743
real-time event service (RTES) 743
options
CECDispatching 715–716, 719, 726–727
CECDispatchingThreads 715–716, 726–727
ECDispatching 786, 791, 797–800
ECDispatchingThreadFlags 786, 800
ECDispatchingThreads 786, 797, 799
ECQueueFullServiceObject 787
strategies
event channel 716
multithreaded 716, 786
priority 786
reactive 716, 786
distributable thread 151, 155, 174–177, 179
distribution model
pull 694
push 694
DNS
See domain name servers (DNS)
do() operation 176, 212
DOC Group xxvi, xxxii, 3, 1154
domain name servers (DNS) 325, 482, 489, 509–510
DSI
See dynamic skeleton interface (DSI)
dynamic hash 600, 606, 609–610, 613–614
dynamic invocation interface (DII) 11, 13, 76, 300
dynamic scheduling 15, 149, 151, 174, 199
dynamic service example 454
dynamic skeleton interface (DSI) 11, 13
Dynamic Thread Pool 401
Dynamic Thread pool 398
dynamic threads 162
DynamicAny module 301

E

eager resource usage strategy 551, 567
earliest deadline first (EDF) scheduling 182

EC_Factory configuration 785–786, 795, 813
EC_QueueFullSimpleActions configuration 809
ec_typed_events build flag 1095
ECConsumerControl option 795, 813
ECConsumerControlPeriod option 813
ECConsumerControlTimeout option 795
ECConsumerValidateConnection option 788
ECDispatching option 786, 791, 797–800
ECDispatchingThreadFlags option 786, 800
ECDispatchingThreads option 786, 797, 799
ECFiltering option 757, 762, 785, 787–788, 801
ECObserver option 771, 788, 802
ECProxyConsumerLock option 789, 803
ECProxyPushConsumerCollection option 790, 804
ECProxyPushSupplierCollection option 790, 806
ECProxySupplierLock option 789, 808
ECQueueFullServiceObject option 787, 809
ECScheduling option 788, 801, 810
ECSupplierFilter option 788
ECSupplierFiltering option 787, 814
ECTimeout option 787, 815
ECTPCDebug option 788, 816
ECUseORBId option 788, 817
EDF_Scheduling module 182
enable_network_priority attribute 169, 191
end_scheduling_segment() operation 177, 212
endpoint-specific options 523
end-to-end priority propagation 15, 151, 166
environment specific inter-ORB protocol (ESIOP) 319–320
  DCE 319
environment variables 26, 1106
  ACE_ROOT 26, 40, 45, 52, 1106
  ImplRepoServiceIOR 1041, 1045, 1048, 1058, 1068, 1070
  ImplRepoServicePort 507, 643, 1072
  InterfaceRepoServicePort 643, 948
  InterfaceRepositoryIOR 707, 948, 951, 959
  LD_LIBRARY_PATH 26, 439
  MAKEFLAGS 755
  NameServiceIOR 655, 696, 744
  NameServicePort 527, 643, 653
  on UNIX 26
  on Windows 26

OPENSSL_CONF 994, 1001
PATH 439, 447, 482, 509–510, 521–522, 532, 541, 607, 1001, 1098
SSL_CERT_DIR 1006–1007
SSL_CERT_FILE 1006–1007
SSL_EGD_FILE 1006
SSL_RAND_FILE 1006
SSL_ROOT 1002–1003
TAO_IDL_PREPROCESSOR 42, 44
See also IDL compiler
TAO_IDL_PREPROCESSOR_ARGS 44
See also IDL compiler
TAO_ORB_DEBUG 480
TAO_ORBENDPOINT 462, 489, 518
TAO_ORBSVCS 1112
TAO_ROOT 26, 40, 42, 44–45, 1106, 1120
TAO_USE_IMR 1039–1040, 1046, 1048
TradingServicePort 542, 643

error handling 59–74
errno 72
error number codes 72
TAO error number codes 72

ESIOP
See environment specific inter-ORB protocol (ESIOP)

establish_components() operation 243–244

ETCL
See extended trader constraint language (ETCL)

etherealize 477

event channel 745
attributes 770
consumer_poa 704
consumer_reconnect 704–705
disconnect_callbacks 705
supplier_poa 704
supplier_reconnect 704–705
collocating 702
consumer and supplier control options 717
consumer proxy 697
creation and destruction 695, 702
disconnecting consumer 705, 722
disconnecting supplier 697, 705, 736
example 768
implementation 695

locating 702
locking options 716
managing own servants 702
object 696
options
CECConsumerControl 717, 722–725, 731, 738
CECConsumerControlPeriod 717, 723, 731, 737–738
CECConsumerControlRoundtripTimeout 724
CECConsumerControlTimeout 717–718, 724
CECConsumerOperationTimeout 718, 725
CECDispatching 715–716, 719, 726–727
CECDispatchingThreads 715–716, 726–727
CECProxyConsumerCollection 719, 728–729
CECProxyConsumerLock 717, 730
CECProxyDisconnectRetries 718
CECProxySupplierCollection 719, 732–733
CECProxySupplierLock 717, 734
CECReactivePullingPeriod 720, 735
CECSupplierControl 717, 736–737, 739
CECSupplierControlPeriod 717, 737
CECSupplierControlRoundtripTimeout 738
CECSupplierControlTimeout 718
CECSupplierOperationTimeout 718, 739
CECUseORBId 720–721, 740
ECConsumerControl 795, 813
ECConsumerControlPeriod 813
ECConsumerControlTimeout 795
ECConsumerValidateConnection 788
ECDispatching 786, 791, 797–800
ECDispatchingThreadFlags 786, 800
ECDispatchingThreads 786, 797, 799
ECFiltering 757, 762, 785, 787–788, 801
ECObserver 771, 788, 802
ECProxyConsumerLock 789, 803
ECProxyPushConsumerCollection 790, 804
ECProxyPushSupplierCollection 790, 806
ECProxySupplierLock 789, 808
ECQueueFullServiceObject 787, 809
ECScheduling 788, 801, 810
ECSupplierFilter 788
ECSupplierFiltering 787, 814
ECTimeout 787, 815

ECTPCDebug 788, 816
ECUseORBId 788, 817
proxy 696
queue full service object
EC_QueueFullSimpleActions 809
resource factory 715, 784
CEC_Factory 715, 722–727, 729–731, 733–740
EC_Factory 785, 795, 813
options 721, 792
setting attributes 704, 770
supplier proxy 701
typed 695
creating 707, 714
destroying 714
example 706
untyped 695
event channel attributes 770
event correlation
real-time event service (RTES) 743
event filtering 693, 832
adding to consumer 891
adding to supplier 889
notification service 832
event service xxix, xxxv, 14, 693–740, 742, 746, 830, 837
and naming service 714
compatability with notification service 909
connecting to the channel 700
CosEvent_Service program 695, 702
creating an event channel 695
creating servants 702
decoupled supplier consumer 693
dispatching 716
event filtering 693
Event_Service program 745
example 695, 703
ORB service libraries 638
overview 694
proxy collection options 718, 790
pull model support 706
push
consumer interface 699
distribution model 694

supplier interface 698
pushing events 696
red-black tree 719
starting CosEvent_Service server 696
structured event types 693
supplier 693
typed
and interface repository 707
typed event channel 695, 706
creating 707, 714
destroying 714
untyped event channel 695

event structure
notification service 829–830
real-time event service 753
real-time event service (RTES) 753

event supplier
notification service 868

Event_Service program 745, 783
command line options 783

EventBatch data type 832

EventChannel interface 746, 769, 772, 776, 783, 828, 834, 841, 871, 877, 892, 896, 901, 904

EventHeader structure
members 753
real-time event service (RTES) 753

EventReliability property 841, 845, 851–852, 864, 941

EventSourceID 746, 753

EventType 746, 753–754

eventtype keyword
value types 261, 275

example xxix, 27, 95–101, 225
asynchronous method handling (AMH) 128, 131
asynchronous method invocation (AMI) 95
building 34
collocated notification channel 903
corbaloc object URL 656
corbaname object URL 659
DevGuideExamples directory xxx
event channel
local 768
event service collocation 703

getting started 25
IDL compiler 40
implementation repository
    activator 1049
    basic indirection 1043
    ImplRepo_Service program 1075
    ImR_Activator program 1078
    IOR table 1054
    tao_imr utility 1068
interface repository 951
IORTable 312
local interface 306
local interfaces 306
locality constrained 306
logging 289
messaging 95
Messenger interface 27
Messenger_i implementation class 28
MessengerClient 27, 30
MessengerServer 27, 29
multiple protocol endpoint 524
multithreading 387, 392, 395
naming service persistence 676
naming sevice 648
notification service 865
portable interceptors 245
    client-side recursion 250
    simple authentication interceptor 225
    using the codec 238
real-time CORBA 156, 209, 217
real-time event service (RTES) 744
    local event channel 768
running 34
security 1010
send_message() operation 30
servant manager 302
ServantLocator class 302
service configurator 455
setting up your environment 26
smart proxies 290–297
TAO_Naming classes 665, 676
typed event channel 706–707

value type 262
ExceptionHolder class 80–81, 83
exceptions 60–68, 70, 72, 74, 838, 1107–1108
BAD_INV_ORDER 72, 371–372, 375
BAD_PARAM 71, 248
BAD_QOS 848
catching 62, 65
COMM_FAILURE 61, 65, 70
completed() operation 63
CORBA 61
INCOMPATIBLE_SCHEDULING_DISCIPLINES 181
INV_OBJREF 61, 70
location code 69
MARSHAL 257, 300, 475
minor codes 68
minor() operation 63, 68
MutexNotFound 198
NO_IMPLEMENT 301
OBJECT_NOT_EXIST 61, 304, 372
real-time CORBA
INCOMPATIBLE_SCHEDULING_DISCIPLINES 181
MutexNotFound 198
system 61
TIMEOUT 70
TRANSIENT 61, 65, 69, 71, 532, 601–602, 609, 613, 1012
UNKNOWN 70
user 66
WrongPolicy 211
explicit binding 11, 151, 166
extended trader constraint language (ETCL) 832–833, 889, 891

F

factory 703
advanced resource 328, 571
client strategy 412–417, 421–422, 436–437, 462, 617–624
initialization 597
resource 324, 417, 421, 462, 549–553, 557, 559–560, 563–564, 568–572, 574–575, 581–583, 586, 588
server strategy 71, 386, 389–392, 435, 437–438, 462, 595–599, 602–615

value type 261–263, 265–266, 268
factory keyword
value types 262
fast build flag 1091, 1095
fault tolerant implementation repository 1073
Fault Tolerant Naming Service 678
feature packages
common secure interoperability (CSI) 962
federating event channels
IP multicast 781
mechanism 782
real-time event service (RTES) 772
selection mechanism 782
UDP 777
FIFO
See first in first out (FIFO) strategy
filtering
adding to consumer 891
adding to supplier 889
by event type 757
by source ID 758
construction 764
correlation 755
disjunction groups 759
example 759
InterFilterGroupOperator parameter 834, 871–872, 878
notification service 832, 889, 891
real-time event service 758
fini() operation 450, 452
first in first out (FIFO) strategy 551, 557
fixed priority scheduling 174, 181
fl build flag 1092, 1095
flushing strategies 412–413, 416–417, 556
blocking 417
leader-follower 416
reactive 416
footprint 436, 476, 571, 609, 814, 1105, 1111
minimizing library size 1111
reduction 5, 91, 300, 421, 436, 476, 571, 609, 628, 814
minimum_corba build flag 201, 220, 1093, 1095–1098
forward declaration 947
fox build flag 1092, 1096

FP_Scheduling module 181
FP_Scheduling::FP_Scheduler interface 181
FP_Scheduling::SegmentSchedulingParameterPolicy interface 181
Free Software Foundation
GNU Public License (GPL) xxxvii, xxxix
frequently asked questions (FAQ) xxxii

G

Gamma, Erich 17, 1157
Gang of Four (GoF)
See Design Patterns: Elements of Reusable Object-Oriented Software book
general inter-ORB protocol (GIOP) 12, 265, 319–321, 324, 328, 330, 335, 337
bi-directional 13, 15, 76, 115–118, 139
BidirectionalPolicyValue 116
BiDirPolicy 118
security 118
version 1.2 116
generic security service (GSS) procotol 975
get_client_policy() operation 301
get_component() operation 301
get_domain_managers() operation 301
get_effective_component() operation 245, 248
get_event_channel() operation 871, 877
get_interface() operation 301
get_parser_names() operation 560
get_peer_certificate_chain() operation 1020
get_peer_certificate() operation 1020
get_policy_overrides() operation 301
get_policy() operation 301, 973
get_protocol_factories() operation 559
get_proxy() operation 283–284
get_qos() operation 843, 847
get_slot() operation 250
getting started 25–35
GIOP
See general inter-ORB protocol (GIOP)
GNU Make xxx, 32–34, 201, 1084–1085, 1090–1102, 1109, 1126, 1132, 1136–1137, 1140

build flags
- ace_for_tao 1094
- ami xxix, 1093–1094
- corba_messaging 201, 1093–1094, 1098
- debug 1091, 1095
- ec_typed_events 1095
- fast 1091, 1095
- fl 1095
- fl_ 1092
- fox 1092, 1096
- inline 41, 45–46, 1091, 1096
- interceptors xxix, 5, 300, 477, 970, 973, 1093, 1096
- minimum_corba 1093, 1095, 1097–1098
- no_hidden_visibility 1091
- optimize 1091, 1097
- pipes 1091, 1097
- probe 1093, 1097
- profile 1092, 1097
- purify 1092, 1097–1098
- qt 1092, 1098
- quantify 967, 1092, 1098
- repo 1092, 1098
- rt_corba 201, 1093, 1098
- rwho 1094, 1099
- sctp 1093, 1099
- shared_libs 1092, 1099–1100
- shared_libs_only 1092, 1099
- smart_proxies 281, 287
- split 1092, 1099
- ssl 1002, 1093, 1100
- static_libs 1092, 1099–1100
- static_libs_only 1092, 1099–1100
- static_link 1092, 1100
- stlport 1092–1093, 1100
- templates 1101
- threads 1092, 1101
- tk 1093, 1101
- tk_reactor 573, 592
- versioned_so 1092, 1101
- wfmo 1093, 1102
- winregistry 1093, 1102
- xt 1093, 1102

zlib 1102
gprof program 1097
GNU Public License (GPL) xxxvii, xxxix
gnuace build type 33
GoF
See Design Patterns: Elements of Reusable Object-Oriented Software book
Gokhale, Aniruddha 1159
gprof program 1097

H

half-sync/half-async pattern 16
handle_events() operation 410–411, 422
handle_input() operation 411
handle_timeout() operation 375
Harris, Timothy 1157
hash() operation 301
hashing
perfect 11, 16
Helm, Richard 17, 1157
Henning, Michi xxvii, xxxi, 1157
Herron, Thomas 320, 1159
hostname_in_ior option 327, 344, 346, 517
HTBP
See HTTP tunneling bi-directional protocol (HTBP)
HTIOP
See HTTP tunneling inter-ORB protocol (HTIOP)
HTTP
See hypertext transfer protocol (HTTP)
HTTP tunneling bi-directional protocol (HTBP) 341
HTTP tunneling inter-ORB protocol (HTIOP) 12, 322, 341
Huston, Stephen D. 17, 365, 1158–1160
hypertext transfer protocol (HTTP) 12

I

id_to_reference() operation 30
IDL
See interface definition language (IDL)

IDL compiler 10, 28, 39–57, 78
A option 43
Ce option 57
ci option 45
cs option 45
Cw option 56
D option 43
d option 56
E option 43
GA option 49
GC option 49
Gd option 52
Gdcps option 50
GH option 49, 130
GI option 28, 47, 291, 307
GIa option 47
GIb option 47
GIc option 47
GId option 47
GIe option 47
GIh option 47
GIs option 47
Gp option 52
Gsp option 49, 281, 283, 285–287, 289, 291
GT option 49
Gt option 49
Guc option 49
H option 51
hc option 45
hI option 46
hs option 45
hT option 46
I option 43
ic option 49
in option 49
o option 45
operation lookup strategy 50
options 42–57
back-end processing 52
export_include 53
export_macro 53
obv_opt_accessor 53

pch_include 53
post_include 53
pre_include 53
skel_export_include 53
skel_export_macro 53
stub_export_include 53
stub_export_macro 53
code generation 48
code suppression 54
collocation strategy 51
output and reporting 55
output files 40–41, 45
pragma ID 44
pragma ident 44
pragma prefix 44
pragma version 44
preprocessing 42
Sa option 54
Sd option 55
sI option 46
si option 45
Sm option 55
smart proxies 278, 286
Sp option 55
ss option 45
St option 55
sT option 46
starter code 46
t option 56
U option 43
V option 56
v option 56
w option 56
Wb option 52
Wp option 43
Yp option 43

IETF
See Internet Engineering Task Force (IETF)

IFR_Service program 947–948, 951, 958–959
multithreaded 948

IIOP
See internet inter-ORB protocol (IIOP)

IIOP Complete: Understanding CORBA and Middleware Interoperability book 1159
implementation repository 312, 468, 506, 1071, 1076
  and IOR table 1053
  directives 1062
  example
    activator 1049
    basic indirection 1043
    ImplRepo_Service program 1075
    ImR_Activator program 1078
    IOR table 1054
    tao_imr utility 1068
  fault tolerant 1073
  ImplRepo_Service program 485, 507, 544
  ImplRepoServiceIOR environment variable 1041, 1045, 1048, 1058, 1068, 1070
  ImplRepoServicePort 507, 643, 1072
  ImR_Activator program 544, 1050
  JacORB interoperability 1079
  options
    ORBImplRepoServicePort 468, 506–507, 643
  server start-up 1046
  tao_imr utility 1061–1062
    add/update commands 1063
    autostart command 1064
    ior command 1065
    kill command 1065
    link command 1065
    list command 1066
    remove command 1067
    shutdown command 1067
    shutdown-repo command 1067
    start command 1068
  TAO_USE_IMR environment variable 1039–1040, 1046, 1048
ImplRepoServiceIOR environment variable 1041, 1045, 1048, 1058, 1068, 1070
ImplRepoServicePort environment variable 507, 643, 1072
INCOMPATIBLE_SCHEDULING_DISCIPLINES exception 181
indirect binding 312, 1038, 1053
info() operation 456, 458
init_protocol_factories() operation 559
init() operation 450, 664
inline build flag 41, 45–46, 1091, 1096

input_cdr_allocator_type_locked() operation 554
input_cdr_buffer_allocator() operation 554
input_cdr_dblock_allocator() operation 554
input_cdr_msgblock_allocator() operation 554
installation
ACE and TAO 1103
instance
data-centric publish-subscribe (DCPS) 50
instance() operation 284, 441, 457
Institute for Software Integrated Systems (ISIS) 3
interceptors
See portable interceptors
interceptors build flag 220, 1093, 1096
interface definition language (IDL) xxvi, xxxiv, xxxix–xl, 10, 27, 84
back-end options 52
C++ mapping 5, 39, 60
collocation strategy options 51
compiler
See IDL compiler
compiler front end (CFE) xxxix–xl
definitions 95
environment variables 44
preprocessing options 42
pseudo (PIDL) 108, 113, 155, 221–223, 243–244, 249, 255, 1096
starter implementation files 46
stubs and skeletons 10
suppression options 54
tao_idl program 28, 39–40, 42–47, 51, 53–54, 96
interface repository xxix, 14, 62, 945–947, 952, 958–959, 1111
command line options 947
compliance 1147, 1150
example 951
forward declaration 947
IFR_Service program 947–948, 951, 958–959
implementation 947
InterfaceDef interface 946
InterfaceRepoServicePort 643, 948
Repository object 946
RepositoryId 224, 946–947
tao_ifr program 947, 949–950, 958–959
options 949–950
InterfaceRepoServicePort environment variable 643, 948

InterfaceRepositoryIOR environment variable 707, 948, 951, 959
interfaces
CORBA::Current 92, 159, 208, 249, 301, 1019
CORBA::DIIPollable 302
CORBA::DynamicAny 301
CORBA::InterfaceDef 946
CORBA::LocalObject 303, 307, 309
CORBA::LocalObject class 245
CORBA::Policy 302
CORBA::PolicyCurrent 92, 104, 302, 508
CORBA::PolicyList 103–105, 107–109, 111, 114, 117, 157, 163–164, 168, 173, 193–194, 206, 209, 305, 382, 1015–1016, 1018
CORBA::PolicyManager 104, 107, 111, 114, 193, 302
CORBA::Pollable 302
CORBA::PollableSet 302
CosEventComm::PushConsumer 699
699
CosEventComm::PushSupplier 698
CosNaming::NamingContextExt 660, 662
CosNotifyChannelAdmin::ConsumerAdmin 828, 836, 877–878, 884, 892, 901
CosNotifyChannelAdmin::EventChannel 834, 841, 871, 877, 892, 896, 901, 904
CosNotifyChannelAdmin::ObtainInfoMode 887–888
CosNotifyChannelAdmin::ProxyConsumer 833, 872, 887, 897
CosNotifyChannelAdmin::ProxySupplier 833, 878–879, 887, 901–902
CosNotifyChannelAdmin::SequenceProxyPushConsumer 829, 832, 870, 872–874, 894, 897, 899, 902
CosNotifyChannelAdmin::SequenceProxyPushSupplier 829, 867, 873, 878–879, 893, 902
FP_Scheduling::FP_Scheduler 181
FP_Scheduling::SegmentSchedulingParameterPolicy 181
IORTable::Locator 314
IORTable::Table 312
ORBInitializer 225, 235, 245, 251
ORBInitInfo 225
PortableInterceptor::IORInfo 243–246
PortableInterceptor::PICurrent 223, 249–253, 257
PortableServer::ServantLocator 302–303, 305
RTCORBA::Current 176
RTCORBA::Mutex 178
RTCORBA::NetworkPriorityMapping 195–197
RTCORBA::RTORB 153
RtecEventChannelAdmin::EventChannel 746, 769, 772, 776, 783

RtecEventComm::PushConsumer 748–749
RtecEventComm::PushSupplier 746
RTScheduling::Current 155, 174, 176–177, 199–200, 212
RTScheduling::DistributableThread 174
RTScheduling::ResourceManager 178, 181
RTScheduling::Scheduler 178–180, 199
RTScheduling::ThreadAction 175, 212, 214
InterFilterGroupOperator parameter 834, 871–872, 878
Internet Engineering Task Force (IETF) 121, 191, 344
internet inter-ORB protocol (IIOP) 12–13, 319–328, 331, 334–336, 338, 349–350, 619
endpoints 518
example 523
IIOP_Factory configuration 324, 583
lite 328, 584
options 326
portspan option 326, 336, 517–518, 523
interoperable object reference (IOR) 489, 531, 540, 560, 605, 607, 618, 648, 770
custom parsers 560
formats 511, 560
interceptors 243, 257
profile locking 618
inter-process communications (IPC) xxxv, 170, 319, 322, 328, 330, 521, 532, 583
local 12, 517
interval timeouts
real-time event service (RTES) 765–766
Introduction to CORBA course xxxiv
invocation
asynchronous 173
dynamic 13
oneway 15, 76–77, 109–110, 112–114, 159, 173, 278, 333, 335, 337, 339–340, 361, 375, 420, 522
reliable oneway 173
static 10, 13
IOR
See interoperable object reference (IOR)
IOR table 312
IORInfo interface 243–246
IORTable
example 312

IORTable::Locator interface 314
IORTable::Table interface 312
is_a() operation 301, 313
is_equivalent() operation 301
is_nil() operation 31, 371
ISIS
See Institute for Software Integrated Systems (ISIS)

J

JacORB xxxiii, 1038
TAO interoperability 1079
Java Reflection API 945
jitter 4
Johnson, James CE 1158
Johnson, Ralph 17, 1157

K

Kachroo, Vishall 1159
Kerberos 971, 975, 984
kernel and system configuration
VxWorks 1124
Klinker, Paul 320, 1159
Kuhns, Fred 1158

L

latency 4
lazy resource usage strategy 551, 567
LD_LIBRARY_PATH environment variable 26, 439
leader/followers
pattern 393
strategy 417, 556, 578
least frequently used (LFU) strategy 551, 557, 572, 589
least laxity first (LLF) scheduling 183
least recently used (LRU) strategy 551, 557–558, 572, 589

Levine, David 1157
Lewis, Bil 1158
LFU
See least frequently used (LFU) strategy
libraries
minimizing size 1111
licensing xxv–xxvi, xxxvii, xxxix–xl
Free Software Foundation
GNU Public License (GPL) xxxvii, xxxix
terms xxv
life cycle service 14, 638
ORB service libraries 638
Linux xxxiv, 25, 1002
Linux kernel stream control transmission protocol (lksctp) 345
LLF_Scheduling module 183
load balancing service 14, 280, 489, 619, 638
ORB service libraries 638
local interfaces 260, 299–301, 303, 307, 309
_add_ref() operation 309
_remove_ref() operation 309
asynchronous method invocation (AMI) 300
C++ mapping 300
CORBA::LocalObject 300–301, 303
dynamic invocation interface 300
example 306
object 299
ORB mediation 300
postinvoke() operation 303
preinvoke() operation 303
reference counting 309
servant activation 302
servant locator 302
locality constrained 300–301, 303, 307, 309
example 306
LocalObject interface 315
locate() operation 314
location codes 69
LOCATION_FORWARD 69, 222, 314, 619, 626
lock() operation 178
locked_transport_cache() operation 557–558
locking
lock() operation 160

option
real-time event service (RTES) 788
strategies 433
log service 14
ORB service libraries 638
logical AND groups
real-time event service (RTES) 761
LRU
See least recently used (LRU) strategy
LynuxWorks, Inc. 1135
LynxOS iv, 1104, 1135, 1139–1140, 1142
cross compilation 1135
using ACE and TAO 1135

M

magic number 352, 583
MAIN_THREAD_MODEL policy 378, 381, 384
MAKEFLAGS environment variable 755
MakeProjectCreator (MPC) xlii, 25, 1085, 1106, 1109
build type
gnuace 33
nmake 32, 1085
vc6 34
Make Project Creator (mpc) 25, 31–32, 35
Make Workspace Creator (mwc) 33–34
mwc.pl program 33–34
options
idlflags 42
rt_client base project 201
rt_server base project 201
managing connections
real-time event service (RTES) 750
mapping
C++ language mapping 5
direct 187–188
mapping() operation 190
MARSHAL exception 257, 300, 475
marshaling 10, 51, 70, 92, 237, 320–321, 351, 413, 427, 465, 474–476, 553, 754
data-centric publish-subscribe (DCPS) 50

Martin, Robert C. 1158
match_prefix() operation 561
max_muxed_connections() operation 557–558
Max_Utility_Scheduling module 184
maximize accrued utility (MAU) scheduling 184
maxPriority constant 158
mcast option 483
messaging 75, 125, 320
  _validate_connection() operation 108
  activation-per-AMI-call strategy 91–92
  associating replies with requests 91
  asynchronous method invocation (AMI) 1159
  asynchronous requests 112
  bi-directional general inter-ORB protocol (GIOP) 115
  BidirectionalPolicyValue 116
  BUFFER_MESSAGE_COUNT 113–114
  BUFFER_TIMEOUT 113
  buffered oneway request 112
  BufferingConstraint policy 112–114
  callback solution 77
  client-side policy management 104
  connection timeout 107–108
    ConnectionTimeout policy 108
  controlling the delivery of AMI-based requests 102
  corba_messaging build flag 201
  create_policy() operation 103–104, 107–108, 111, 114, 117
  creating a reply-handler class 84
  drawbacks to using AMI 78
  exceptions
    ExceptionHolder class 80–81, 83
    replies 83
    reply-handler functions 90
  introduction 1159
  oneway requests 112
  policies
    creating 103
    destroying 105
    management of 103
  quality of service (QoS) 106
  reply
    handler 82, 84
    timeouts 106

request
    delivery 102
    timeouts 106
secure 969
sendc_ prefix 77, 79
server-differentiated-reply strategy 91–92
server-side policy management 105
specification 149
SYNC_DELAYED_BUFFERING 113
SYNC_NONE 113
SyncScope policy 102, 109–110, 112–114
TAO.pidl 113

Microsoft Foundation Classes (MFC) 670
MIF_Scheduling module 185
minimizing size 1097
    libraries 1111
    minimum_corba build flag 201, 220, 1093, 1095–1098, 1111
minimum_corba build flag 1093, 1097
minor() operation 64
minPriority constant 158
MIOP
    See multicast inter-ORB protocol (MIOP)
MMAPFilePrefix option 332
MMAPFileSize option 332
most important first (MIF) scheduling 184
MPC
    See MakeProjectCreator (MPC)
msg_wfmo reactor 592
multicast 468, 481, 483, 506–507, 527, 651, 672, 773, 781, 948, 958
    naming service 651, 654
    service discovery 483
multicast inter-ORB protocol (MIOP) 12, 322, 337, 339–340
multithreading 357
    client thread 360
    connect strategies 359
        blocking 414
        leader-follower 413
        reactive 414
    example 387, 392, 395
    flushing strategies 359, 412–413, 416
    guideline 367, 372, 382
    main thread model 380

multithreaded select reactor 410
ORB-controlled model 378
polling loop 368
request invocation 360
request processing 361
select reactor 409
server thread 360
shutting server down 374
single thread model 379
thread-pool reactor 411
transport multiplexing strategies 415
wait strategies 359, 417
wait-on-leader-follower 422
wait-on-leader-follower-no-upcall 424
wait-on-reactor 422
wait-on-read 418
MutexNotFound exception 198
mwc.pl program 33–34

N

name() operation 232, 245
NameServiceIOR environment variable 655, 696, 744
NameServicePort environment variable 527, 643, 653
naming service 14, 63, 312, 456, 635–636, 640, 645, 647–660, 662–667, 670–672, 678–680, 714, 784
client source code 650
command line options 672
compliance 1153
corbaloc object URL 655
corbaname object URL 657
discriminating between multiple services 653
example 648
TAO_Naming classes 665, 676
executable 646
fault tolerant 678
load balancing 678
persistence 681
multicast 651, 654
NameServiceIOR environment variable 655, 696, 744

NameServicePort environment variable 527, 643, 653
NamingContextExt interface 660, 662
NamingViewer utility 670
object group 678
object URLs 655
options 672
ORBNameServicePort 468, 526–528, 643, 653
ORB service libraries 638
persistence 673
replication 678
resolve_initial_references() operation 647
resolving 647
root naming context 696
TAO_Naming_Client class 663–664
TAO_Naming_Server class 664, 666
tao_nsadd utility 667, 669
tao_nsdel utility 667, 669
tao_nslist utility 667
TAO-specific classes 663
utilities 667
NamingViewer 670
tao_nsadd 669
tao_nsdel 669
tao_nslist 667
Naming_Service program 647
NamingContextExt interface 660, 662–663
NamingViewer utility 670
negating the logic of filters
real-time event service (RTES) 761
nested upcalls 16, 420, 422–425
nesting groups
real-time event service (RTES) 762
NetworkPriorityMapping interface 195–197
NetworkPriorityMappingManager interface 196–198
new_for_consumers() operation 877, 901
new_for_suppliers() operation 871, 896
NMake 32, 1085
nmake build type 32, 1085
no operation (NOOP) strategy 551, 557
no_context() operation 1020
no_hidden_visibility build flag 1091
NO_IMPLEMENT exception 301

non_existent() operation 301
NON_RETAIN policy 302, 305
NOOP
See no operation (NOOP) strategy
notification service 14, 312, 827–829, 832–833, 835–838, 842, 847, 864–866, 869–871, 889, 891, 893, 909, 916, 918, 1154
845
adding subscriptions 884
administration properties 841
architecture 828
collocated 903
compatability with event service 909
conflict resolution 847
connecting
consumers 880
suppliers 881
ConnectionReliability property 840–841, 845, 848–849, 864
destroying the notification channel 882
developing structured event supplier 895
disconnecting
consumers 880
suppliers 881
event consumer 900
event filtering 832, 889, 891
consumer 891
supplier 889
event structure 829–830
event supplier 868, 895
EventReliability property 841, 845, 851–852, 864, 941
example 865
features 829
implementing intefaces
sequence push consumer 899
sequence push supplier 894
structured push consumer 875
structured push supplier 867
limits
MaxConsumers 842, 845, 892
MaximumBatchSize 832, 840, 845
MaximumEventsPerConsumer 840, 845, 916
MaxQueueLength 842, 845, 892
MaxSuppliers 842, 845, 892

managing connections 880
MaxConsumers property 842, 845, 892
MaxQueueLength property 842, 845, 892
MaxSuppliers property 842, 845, 892
negotiating quality of service (QoS) 847
notification administration properties 841
    MaxConsumers 842, 845, 892
    MaxQueueLength 842, 845, 892
    MaxSuppliers 842, 845, 892
    RejectNewEvents 839, 842, 844, 846
notify manager
    resource factory options 911, 941
Notify_Service program 866–867, 870, 903, 909–910
obtain_subscription_types() operation 836
offer_change() operation 835
offers and subscriptions 882, 887
ORB service libraries 638
PacingInterval property 832, 840, 845
quality of service (QoS)
    get_qos() operation 847
    properties 892
        accessing and modifying 845
        ConnectionReliability 840–841, 845, 848–849, 864
        DiscardPolicy 839, 842, 844–846, 916
            AnyOrder 839
            DeadlineOrder 840
            FifoOrder 839
            LifoOrder 839
            PriorityOrder 840
        EventReliability 841, 845, 851–852, 864, 941
        MaximumBatchSize 832, 840, 845
        MaximumEventsPerConsumer 840, 845, 916
        OrderPolicy 839, 845–846, 893, 916
            AnyOrder 839
            DeadlineOrder 839
            FifoOrder 839
            PiorityOrder 839
        PacingInterval 832, 840, 845
        PriorityOrder 839, 845
        setting in a structured event header 846
        StopTime 838, 846
        Timeout 838, 846

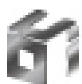
support 837
quality of service (QoS) properties 838, 845, 892
resuming consumer connections 881
sequence push consumer 899
sequence push supplier 894
setting quality of service (QoS) 846
structured event 830–831
consumer 876
supplier 868
type 830
structured push consumer 875
offer_change() operation 875
push_structured_event() operation 875
structured push supplier 867
subscriptions 835, 882, 887
adding to consumer 884
subscription_change() operation 836
suspending
consumer connections 881
supplier connections 881
TAO-specific properties
BlockingPolicy 844–845, 916
ThreadPool 843, 846, 853, 855, 908, 913
ThreadPoolLanes 843, 846, 853–854, 905, 913
transmitting an EventBatch 893
unsupported quality of service (QoS) exceptions 847
Notify_Service program 866–867, 870, 903, 909–910
command line options 909–910
starting 866
null filters
real-time event service (RTES) 764

O

O'Ryan, Carlos 1157–1159
object adapter
see portable object adapter (POA)
Object Computing, Inc. (OCI) i, xxvii, xxxi–xxxiv, 3–4, 20, 1154
object factories 434–435
object group 678

Object Management Group (OMG) specifications
C++ Language Mapping (formal/08-01-09) 13, 39, 299, 309, 1152
Compressed GIOP submission (mars/08-12-20) 347
CORBA 2.6 (formal/01-12-35) 81
CORBA 3.0.3 (formal/04-03-12) 1146
CORBA 3.1
Part 1 (formal/08-01-04) 13, 75, 220, 256, 259, 299, 362, 514–515, 523, 531, 537, 640, 945, 1146, 1158
Part 2 (formal/08-01-06) 13, 76, 118, 320, 656, 962, 1146, 1158
Part 3 (formal/08-01-08) 13, 1146, 1158
CORBA Component Model (CCM) (formal/02-06-65) 1158
CORBA for Embedded (CORBA/e) (formal/08-11-06) 1149
Event Service 1.2 (formal/04-10-02) 693, 827, 1158
Extensible Transport Framework (ETF) (ptc/04-03-03) 1158
Naming Service 1.2 (formal/02-09-02) 648, 1153, 1158
Naming Service 1.3 (formal/04-10-03) 645
Notification Service 1.1 (formal/04-10-13) 827, 1154, 1158
Persistent State Service (formal/02-09-06) 1159
Real-Time CORBA 1.0 (ptc/99-06-02) 149
Real-Time CORBA 1.1 (formal/02-08-02) 149
Real-Time CORBA 1.2 (formal/05-01-04) 13, 149, 1151, 1159
Real-Time CORBA 2.0 (dynamic scheduling) (formal/03-11-01) 149
Security Service (original version) (security/00-12-02) 962
Security Service 1.8 (formal/02-03-11) 964
object reference
See interoperable object reference (IOR)
object URL 655–660
corbaloc 311–312, 640–641, 654–658, 661, 670
corbaname 560, 579, 657–660, 663
OBJECT_NOT_EXIST exception 61, 304, 372
object_to_string() operation 30
objects by value (OBV)
See value types
observers
custom gateways 772
real-time event service (RTES) 771
obtain_notification_push_consumer() operation 872, 897
obtain_notification_push_supplier() operation 878, 901
obtain_offered_types() operation 837, 884, 887–888
obtain_subscription_types() operation 836, 887–888
ObtainInfoMode interface 887–888
OBV

See value types
OCIReleaseNotes.html file xxviii
OctetSeq 238–239, 242
offer_change() operation 835–836, 875–876, 882–884, 887–888, 899–900
OMG
See Object Management Group (OMG) specifications
oneway invocation 15, 76–77, 109–110, 112–114, 159, 173, 278, 333, 335, 337, 339–340, 361, 375, 420, 522
open source software xxv
open_named_mutex() operation 198
OpenSSL 322, 964, 983, 991, 994–996, 998, 1000–1003, 1005–1006, 1019
OPENSSL_CONF environment variable 994, 1001
operating systems
Linux xxxiv, 25, 1002
LynxOS 1135
QNX 328
UNIX 12, 33, 40, 43, 56, 322, 328–330, 332, 334, 338, 438, 459, 461, 479, 489, 512–513, 519, 521, 530, 958, 1002, 1091
VxWorks 479, 664
Windows 25–26, 28, 33–34, 56, 453, 670, 1002–1003, 1084
operation lookup
binary search 51
dynamic search 51
linear search 51
perfect hash 51
operations
_add_ref() 264, 309
_get_policy() 207
_id() 62, 64–65, 67, 92, 252, 339, 699–700
_narrow() 313
_refcount_value() 264
_remove_ref() 264, 309
_rep_id() 64–65, 67
_set_policy_overrides() 105
_unchecked_narrow() 285–286, 340
_validate_connection() 108, 167
activate_object_with_id_and_priority() 211
activate_object_with_id() 92, 339
activate_object_with_priority() 210
activate_object() 30
activate() 30, 235, 397, 598, 649, 666–667, 880, 904, 1025
adapter_manager_state_changed() 245



adapter_state_changed() 245
add_client_request_interceptor() 230
add_constraints() 890
add_filter() 890
add_ior_component() 246
add_ior_interceptor() 245, 247
add_server_request_interceptor() 234
amh_response_handler_allocator() 565
ami_response_handler_allocator() 565
begin_scheduling_segment() 174, 176–177, 212, 214
bind() 312
cache_maximum() 557–558
cancel() 175
codec_factory() 237, 241
components_established() 245
connect_push_consumer() 705, 750, 756, 771
connect_push_supplier() 697, 705, 746, 771
connect_sequence_push_consumer() 902
connect_sequence_push_supplier() 898
connect_structured_push_consumer() 879
connect_structured_push_supplier() 873
create_cached_connection_lock() 557–558
create_channel() 870, 904
create_corba_object_lock() 566
create_filter() 889
create_flushing_strategy() 556
create_for_unmarshal() 266
create_lf_strategy() 553
create_mutex() 161
create_named_mutex() 198
create_POA() 105, 153, 157, 163
create_policy() 103–104, 107–108, 111, 114, 117, 1015–1018
create_priority_banded_connection_policy() 167
create_proxy() 278, 282–283, 285, 288, 293
create_purging_strategy() 557
create_reference_with_id_and_priority() 211
create_reference_with_priority() 211
create_resource_manager() 178, 181
default_filter_factory() 889
destroy_mutex() 161
destroy() 30, 364
disconnect_push_consumer() 699, 705, 711, 748, 771

disconnect_push_supplier() 697–698, 705, 746, 771
disconnect_sequence_push_consumer() 899–900
disconnect_sequence_push_supplier() 894
disconnect_structured_push_consumer() 874–876
disconnect_structured_push_supplier() 867
do() 176, 212
end_scheduling_segment() 177, 212
establish_components() 243–244
fini() 450, 452
get_client_policy() 301
get_component() 301
get_domain_managers() 301
get_effective_component() 245, 248
get_event_channel() 871, 877
get_interface() 301
get_parser_names() 560
get_peer_certificate_chain() 1020
get_peer_certificate() 1020
get_policy_overrides() 301
get_policy() 301, 973
get_protocol_factories() 559
get_proxy() 283–284
get_qos() 843, 847
get_slot() 250
handle_events() 410–411, 422
handle_input() 411
handle_timeout() 375
hash() 301
id_to_reference() 30
info() 456, 458
init_protocol_factories() 559
init() 450, 664
input_cdr_allocator_type_locked() 554
input_cdr_buffer_allocator() 554
input_cdr_dblock_allocator() 554
input_cdr_msgblock_allocator() 554
instance() 284, 441, 457
is_a() 301, 313
is_equivalent() 301
is_nil() 31, 371
locate() 314
lock() 160, 178

locked_transport_cache() 557–558
mapping() 190
match_prefix() 561
max_muxed_connections() 557–558
name() 232, 245
new_for_consumers() 877, 901
new_for_suppliers() 871, 896
no_context() 1020
non_existent() 301
object_to_string() 30
obtain_notification_push_consumer() 872, 897
obtain_notification_push_supplier() 878, 901
obtain_offered_types() 837, 884, 887–888
obtain_subscription_types() 836, 887–888
offer_change() 835–836, 875–876, 882–884, 887–888, 899–900
open_named_mutex() 198
ORB_Init()
    orb_identifier parameter 460–461
ORB_init() 30–31, 71, 155, 213, 225, 230, 324–325, 459–462, 473, 654–655, 703
output_cdr_buffer_allocator() 554
output_cdr_dblock_allocator() 554
output_cdr_msgblock_allocator() 554
parse_string() 561
perform_work() 89, 93–94, 364, 366–371, 374, 376, 381, 384, 392, 420, 423
ping() 1041–1042
post_init() 225, 229–230, 234, 245, 247
postinvoke() 303–304
pre_init() 225, 229, 245
preinvoke() 303
process_directive() 444
pthread_create() 397
purge_percentage() 557–558
push_structured_event() 873, 875, 880
push_structured_events() 898–899, 903
push() 697, 743, 747–748, 750, 760
rebind() 312, 714
receive_exception() 222
receive_other() 222
receive_reply() 222
receive_request_service_context() 222
receive_request() 200, 223
reclaim_reactor() 552

reconfigure() 444, 458
register_new_factory() 266
register_proxy_factory() 282–283
register_value_factory() 261
release() 309, 371
repository_id() 301
resolve_initial_references() 30, 155, 160, 179, 190, 196, 252, 300, 312, 482, 508, 640, 647, 655–656, 664, 980, 983, 1021, 1070
resolve() 696
resource_usage_strategy() 567
resume_connection() 881
RTScheduling::Scheduler::create_resource_manager() 178, 181
RTScheduling::Scheduler::set_scheduling_parameter() 181
run() 30, 93, 362, 367, 373, 381, 393, 423, 665, 1067
select() 352, 410–411, 592
send_exception() 223
send_other() 223
send_poll() 221
send_reply() 223
send_request() 222
set_locator() 314–315
set_policy_overrides() 105, 301, 972, 980, 983
set_qos() 843
set_scheduling_parameter() 181
set_slot() 250
shutdown() 364, 371–376, 1022–1023, 1025, 1041, 1067
spawn() 174, 176–177, 212
string_dup() 31
string_to_object() 31, 251, 314, 339, 485–486, 532, 560–561, 579, 641, 647, 655–657, 659
subscription_change() 836, 867–868, 879, 884–885, 887–888, 894, 902
suspend_connection() 881
svc() 215, 397, 665
TaggedComponent() 247
target_is_a() 232
the_POAManager() 30
to_CORBA() 159, 186, 195, 197
to_native() 159, 186
to_network() 195, 197
to_url() 661
try_lock() 160, 179
unbind() 312

unlock() 160–161, 179
unregister_proxy_factory() 282, 289
update_scheduling_segment() 177
use_locked_data_blocks() 554
validate_connection() 301
WaitForMultipleObjects() 352, 593
work_pending() 89, 364, 366–368, 370–371, 374
optimize build flag 1091, 1097
options
BUFFER_MESSAGE_COUNT 113–114
BUFFER_TIMEOUT 113
CECConsumerControl 717, 722–725, 731, 738
CECConsumerControlPeriod 717, 723, 731, 737–738
CECConsumerControlRoundtripTimeout 724
CECConsumerControlTimeout 717–718, 724
CECConsumerOperationTimeout 718, 725
CECDispatching 715–716, 719, 726–727
CECDispatchingThreads 715–716, 726–727
CECProxyConsumerCollection 719, 728–729
CECProxyConsumerLock 717, 730
CECProxyDisconnectRetries 718
CECProxySupplierCollection 719, 732–733
CECProxySupplierLock 717, 734
CECReactivePullingPeriod 720, 735
CECSupplierControl 717, 736–737, 739
CECSupplierControlPeriod 717, 737
CECSupplierControlRoundtripTimeout 738
CECSupplierControlTimeout 718
CECSupplierOperationTimeout 718, 739
CECUseORBId 720–721, 740
ECConsumerControl 795, 813
ECConsumerControlPeriod 813
ECConsumerControlTimeout 795
ECConsumerValidateConnection 788
ECDispatching 786, 791, 797–800
ECDispatchingThreadFlags 786, 800
ECDispatchingThreads 786, 797, 799
ECFiltering 757, 762, 785, 787–788, 801
ECObserver 771, 788, 802
ECProxyConsumerLock 789, 803
ECProxySupplierLock 789, 808
ECQueueFullServiceObject 787, 809

ECScheduling 788, 801, 810
ECSupplierFilter 788
ECSupplierFiltering 787, 814
ECTimeout 787, 815
ECTPCDebug 788, 816
ECUseORBId 788, 817
ORBSchedPolicy 202–203
ORBScopePolicy 202, 204
RTORBDynamicThreadIdleTimeout 204
RTORBDynamicThreadRunTime 204
SO_DONTROUTE 467, 488
SO_KEEPALIVE 467, 513
SO_LINGER 515
SO_REUSEADDR 327
SSLCertificate 1034
SSLNoProtection 1033
SSLPrivateKey 1029, 1034
TCP_NODELAY 530
tp 592

ORB 435, 462, 596–597
configuring real-time CORBA 202
connection management 466
controlling service configurator 463
core 12, 16
debugging 464
endsystem 9
event handling 362, 364–370
initialization 696
initialization options 459–548
interface definition 460
Java xxxiii
mediation 300
optimizing request processing 465
options
ORBAcceptErrorDelay 466, 473
ORBAMICollocation 429, 465, 474
ORBCDRTradeoff 465, 475, 574
ORBCollocation 428, 460, 465, 475–476
ORBCollocationStrategy 52, 55, 428–429, 460, 465, 476, 478
ORBConnectIPV6Only 479
ORBDaemon 439, 463, 479, 524–525
ORBDebug 460, 462, 464, 480

ORBDebugLevel 462, 464, 480–481
ORBDefaultInitRef 460, 468, 481–487, 508, 527, 641, 656–657, 660
ORBDisableRTCollocation 205, 465, 487–488
ORBDontRoute 489
ORBDottedDecimalAddresses 325, 466, 489–490
ORBEndpoint 462, 466, 489–490, 515, 518, 654
ORBEnforcePreferredInterfaces 491
ORBForwardInvocationOnObjectNotExist 471, 492–496
ORBForwardIOnceOnCommFailure 495
ORBForwardOnceOnInvObjref 496
ORBForwardOnceOnTransient 495
ORBForwardOnCommFailureLimit 497
ORBForwardOnObjectNotExistLimit 499–500
ORBForwardOnObjrefLimit 498
ORBFTSendFullGroupTC 502
ORBGestalt 503
ORBId 460–462, 720, 788
ORBIgnoreDefaultSvcConfFile 504
ORBImplRepoServicePort 468, 506–507, 643
ORBIMREndpointsInIOR 472, 504, 506
ORBInitRef 71, 252, 254, 460, 468, 481–486, 507–512, 527–528, 532, 544, 560, 579, 640–641, 653–657, 660, 1045, 1048, 1058, 1068, 1070
ORBIPHopLimit 512
ORBIPMulticastLoop 513
ORBKeepAlive 513
ORBLaneEndpoint 165, 466, 513
ORBLaneListenEndpoints 165, 466, 513–514
ORBLingerTimeout 515–516
ORBListenEndpoints 71, 165, 313, 325, 329–330, 333–334, 339, 344, 346, 349, 460, 462, 466, 484–486, 489–490, 513, 515–523, 653–654, 656, 677–678, 1054
ORBMaxMessageSize 525
ORBMulticastDiscoveryEndpoint 642, 653
ORBNameServicePort 468, 526–528, 643, 653
ORBNegotiateCodesets 465, 528–529
ORBNodelay 465, 530
ORBNoProprietaryActivation 460, 530
ORBNoServerSideNameLookups 466, 531
ORBObjRefStyle 464, 531, 533, 1054
ORBParallelConnectDelay 466, 533, 546
ORBPreferIPV6Interfaces 534
ORBPreferredInterfaces 491, 534–535

ORBRcvSock 460, 465, 536
ORBServerId 460, 468, 472, 536–537
ORBServiceConfigLoggerKey 463, 537
ORBSingleReadOptimization 538–539
ORBSkipServiceConfigOpen 464
ORBSndSock 465, 539–540
ORBStdProfileComponents 465, 540–541
ORBSvcConf 440, 464, 541–542, 715, 786, 818, 1010–1011
ORBSvcConfDirective 440, 464, 542
ORBTradingServicePort 468, 542–543, 643
ORBUseIMR 472, 543–544, 1046, 1048, 1058
ORBUseLocalMemoryPool 472, 545
ORBUseParallelConnects 533, 546
ORBUseSharedProfile 546–547
ORBVerboseLogging 547–548
protocol selection 466
protocols 559
real-time 155, 159, 195, 202
service libraries 489, 637–638
audio/video streaming 14, 637
concurrency control 14, 638
data distribution 14
event 14, 638
interface repository 14
life cycle 14, 638
load balancing 14, 638
log 14, 638
naming 14, 638
notification 14, 638
property 14, 638
real-time event 15, 639
real-time scheduling 639
security 14, 639
time 14, 639
trading 14, 638
shutdown 374–376
wait_for_completion parameter 364, 371–375
ORB_CTRL_MODEL policy 378, 381–382, 384
ORB_init() operation 30–31, 71, 155, 213, 225, 230, 239, 324–325, 459–462, 473, 654–655, 703
ORBAcceptErrorDelay option 466, 473
ORBActiveHintInIDs option 603–605

ORBActiveHintInPOANames option 602, 605–606
ORBActiveObjectMapSize option 603, 606
ORBAllowReactivationOfSystemIDs option 603, 605, 607
ORBAMHResponseHandlerAllocator option 565, 588
ORBAMICollocation option 429, 465, 474
ORBAMIResponseHandlerAllocator option 565, 588
ORBCDRTradeoff option 465, 475, 574
ORBCharCodesetTranslator option 564, 574
ORBClientConnectionHandler option 622
ORBCollocation option 428, 460, 465, 475–476
ORBCollocationStrategy option 52, 55, 428–429, 460, 465, 476–478
ORBConcurrency option 386, 389–390, 392, 598, 607–608
ORBConnectionCacheLock option 555, 558–559, 575, 590
ORBConnectionCacheMax option 559, 575–576
ORBConnectionCachePurgePercentage option 559, 576, 590
ORBConnectionHandlerCleanup option 622, 624–625
ORBConnectionPurgingStrategy option 558, 572, 589–590
ORBConnectIPV6Only option 468, 479
ORBConnectorCacheLock option 575
ORBConnectStrategy option 414, 421, 624–626
ORBCorbaObjectLock option 566
ORBDaemon option 439, 463, 479, 524–525
ORBDebug option 460, 462, 464, 480
ORBDebugLevel option 462, 464, 480–481
ORBDefaultInitRef option 460, 468, 481–487, 508, 527, 641, 656–657, 660
ORBDefaultSyncScope option 622, 626
ORBDefaultSyncStrategy option 626
ORBDisableRTCollocation option 205, 465, 487–488
ORBDontRoute option 467, 489
ORBDottedDecimalAddresses option 325, 466, 489–490
ORBDropRepliesDuringShutdown option 577–578
ORBDynamicThreadPoolName 490
ORBEndpoint 490
ORBEndpoint option 462, 466, 489–490, 513, 515, 518, 654
ORBEnforcePreferredInterfaces option 466, 491
ORBFlushingStrategy option 417, 421, 556, 578–579
  blocking strategy 578
  leader_follower strategy 578
  reactive strategy 578
ORBForwardInvocationOnObjectNotExist option 471, 492–496
ORBForwardIOnceOnInvObjref option 496
ORBForwardIOnceOnObjectNotExist option 494

ORBForwardIOnceOnTransient option 495
ORBForwardOnceOnCommFailure option 471, 494
ORBForwardOnceOnInvObjref option 471, 496
ORBForwardOnceOnObjectNotExist option 471, 493–494
ORBForwardOnceOnTransient option 471
ORBForwardOnceOnTransport option 495
ORBForwardOnCommFailureLimit option 497
ORBForwardOnObjectNotExistLimit option 499–500
ORBForwardOnObjrefLimit option 498
ORBFTSendFullGroupTC option 472, 502
ORBGestalt option 503
ORBId option 460–462, 472, 504, 720, 788
ORBIgnoreDefaultSvcConfFile 504
ORBIgnoreDefaultSvcConfFile option 463
ORBIIOPClientPortBase option 466, 504
ORBIIOPClientPortSpan option 466
ORBIIOPClientPortSpan span 505
ORBImplRepoServicePort option 468, 506–507, 643
ORBIMREndpointsInIOR enabled 506
ORBIMREndpointsInIOR option 472, 504, 506
ORBInitializer interface 225, 229, 235, 245, 251
ORBInitInfo interface 225
ORBInitRef option 460, 468, 481–486, 507–512, 527–528, 532, 544, 1045, 1048, 1058, 1068, 1070
DLL Style object reference 510
HTTP style object reference 511
ORBInputCDRAllocator option 389, 554–555, 572, 575, 590–591
ORBIORParser option 563, 579
ORBIPHopLimit option 467, 512
ORBIPMulticastLoop option 467, 513
ORBKeepAlive option 467, 513
ORBLaneEndpoint option 165, 466, 513
ORBLaneListenEndpoints option 165, 466, 513–514
ORBLingerTimeout option 467, 515–516
ORBListenEndpoints option 71, 165, 313, 325, 329–330, 333–334, 339, 344, 346, 349, 460, 462, 466, 484–486, 489–490, 513, 515–523, 653–654, 656, 677–678, 1054
ORBLogFile option 464, 524
ORBMaxMessageSize option 472, 525
ORBMulticastDiscoveryEndpoint option 468, 526, 642, 653
ORBMuxedConnectionMax option 559, 580
ORBNameServicePort option 468, 526–528, 643, 653
ORBNativeCharCodeset option 564

ORBNativeWCharCodeset option 564
ORBNegotiateCodesets option 465, 528–529
ORBNodelay option 465, 530
ORBNoProprietaryActivation option 460, 468, 472, 530
ORBNoServerSideNameLookups option 466, 531
ORBObjectKeyTableLock option 582
ORBObjRefStyle option 464, 531, 533, 1054
ORBOutputCDRAllocator option 555, 582
ORBParallelConnectDelay option 466, 533, 546
ORBPersistentIDPolicyDemuxStrategy option 602, 609
ORBPOALock option 389, 609
ORBPOAMapSize option 602, 609–610
ORBPolicyManager option 104, 107
ORBPreferIPV6Interfaces option 534
ORBPreferredInterfaces option 467, 491, 534–535
ORBPriorityMapping option 190, 202
ORBProfileLock option 389, 619, 626–627
ORBProtocolFactory option 323–324, 328–329, 331–332, 334, 338, 517, 560, 583–584, 1010–1011, 1014, 1017
- custom factory 583
- DIOP_Factory 583
- IIOP_Factory 583
- SHMIOP_Factory 583
- SSLIOP_Factory 583
- UIOP_Factory 583

ORBRcvSock option 460, 465, 536
ORBReactorMaskSignals option 584
ORBReactorThreadQueue option 573, 591–592
ORBReactorType option 387, 389, 409–412, 542, 553, 573, 592–593
ORBReplyDispatcherTableSize option 627
ORBResourceUsage option 567–568, 585–586
ORBSchedPolicy option 202–203
ORBScopePolicy option 202, 204
ORBServerId option 460, 468, 472, 536
ORBServiceConfigLoggerKey option 440, 463, 537
ORBSingleReadOptimization option 465, 538–539
ORBSkipServiceConfigOpen option 440, 464
ORBSndSock option 465, 539–540
ORBStdProfileComponents option 465, 540–541
ORBSvcConf option 440, 464, 541–542, 715, 786, 818, 1010–1011
ORBSvcConfDirective option 440, 464, 542
ORBSystemIDPolicyDemuxStrategy option 603, 610

ORBThreadFlags option 598, 611
ORBThreadPerConnectionTimeout option 391–392, 599, 612
ORBTradingServicePort option 468, 542–543, 643
ORBTransientIDPolicyDemuxStrategy option 602, 613
ORBTransportMuxStrategy option 391, 415–416, 421, 620, 628
ORBTransportMuxStrategyLock option 620, 629
ORBUniqueIDPolicyReverseDemuxStrategy option 603, 613–614
ORBUseIMR option 472, 504, 506, 543–544, 1046, 1048, 1058
ORBUseLocalMemoryPool option 472, 545
ORBUseParallelConnects option 466, 533, 546
ORBUserIDPolicyDemuxStrategy option 604, 614–615
ORBUseSharedProfile option 467, 546–547
ORBVerboseLogging option 464, 547–548
ORBWaitStrategy option 387, 389, 414, 416–417, 421–422, 424, 426, 622, 629–630
ORBWCharCodesetTranslator option 564, 587
ORBZeroCopyWrite option 555, 587
OrderPolicy property 839, 845–846, 893, 916
Othman, Ossama 1158
output_cdr_buffer_allocator() operation 554
output_cdr_dblock_allocator() operation 554
output_cdr_msgblock_allocator() operation 554

P

PacingInterval property 832, 840, 845
parse_string() operation 561
PATH environment variable 439, 1001, 1098, 1106–1107, 1117
Pattern Languages of Program Design 3: Software Patterns Series book 1158
Pattern-Oriented Software Architecture:Patterns for Concurrent and Networked Objects (POSA2) book 17, 393, 411, 423, 434, 591, 1159
patterns 16
abstract factory 16
acceptor 16
active object 16, 598
adapter 16
asynchronous completion token (ACT) 92
bridge 16
connector 16

half-sync/half-async 16
leader/followers 393
optimization principle 1159
proxy 16
reactor 16, 549, 1159
service configurator 16, 562
strategy 16
thread-specific storage (TSS) 16
perfect hashing 11, 16, 51, 599
See also gperf
perform_work() operation 89, 93–94, 364, 366–371, 374, 376, 381, 384, 392, 420, 423
per-hop-behavior (PHB) 121, 191
PHB
See per-hop behavior (PHB)
PICurrent interface 223, 249–253, 257
ping() operation 1041–1042
pipes build flag 1091, 1097
platform support xxviii, 5
platform_macros.GNU file 529, 755, 1002–1003, 1084–1086, 1090, 1107–1108, 1111, 1125–1126, 1128, 1131–1132, 1136–1137
platforms
supported xxxii
pluggable protocols xxix, 5, 12, 16, 70, 319–320, 322–323, 331, 339, 351–356, 463, 483, 511, 517, 583, 596, 620, 1150, 1152–1153, 1158
acceptor 354, 549
address definition 324, 329, 333–334, 336, 338
building 323
combining protocols 341
configuring SHMIOP factory 332
connection
handler 355, 549
connector 354, 549
declaring 324, 328
endpoint 355
framework 353
guidelines 330
identifier tags 352
loading 323, 331, 333, 336–337
profile 355
protocol factory 549
restrictions 335, 340

TAO_Acceptor class 354
TAO_Connector class 354
TAO_Endpoint class 355
transport 321
transport behavior 352
POA
See portable object adapter (POA)
poa_policies attribute 181
policies
BidirectionalPolicyValue 116
BufferingConstraint 112–114
ConnectionTimeout 108
MAIN_THREAD_MODEL 378, 381, 384
NON_RETAIN 302, 305
ORB_CTRL_MODEL 378, 381–382, 384
PriorityModelPolicy 166
PrivateConnectionPolicy 168
PROCESS 204
RequestProcessingPolicy 305
RETAIN 302
SCHED_FIFO 203
SCHED_OTHER 203
SCHED_RR 203
scope_policy 204
ServantRetentionPolicy 302, 305
SERVER_DECLARED 209
SINGLE_THREAD_MODEL 378, 381–383
SyncScope 102, 109–110, 112–114
SYSTEM 204
ThreadPolicy 378, 381–382, 384
USE_SERVANT_MANAGER 305
Policy interface 302
PolicyCurrent interface 92, 104, 302, 508
PolicyList interface 103–105, 107–109, 111, 114, 117, 157, 194, 206, 209, 382, 1015–1016, 1018
PolicyManager interface 104, 107, 111, 114, 193, 302
Pollable interface 302
PollableSet interface 302
portability 16
portable interceptors xxix, 5, 174, 179, 199–200, 219–220, 249, 256–257, 1093, 1096, 1159
adapter_manager_state_changed() operation 245

adapter_state_changed() operation 245
asynchronous method handling (AMH) 139
client interceptors 221–222, 227, 239
ClientInterceptor class 227
ClientRequestInfo interface 223
    get_effective_component() 245
codec (coder/decoder) 237
codec_factory() operation 237, 241
CodecFactory reference 237–238
coders 257
components_established() operation 245
context_data field 228
context_id field 228
decoders 257
decoding the tag 247
destroy() operation 245
example 225, 245
    client-side recursion 250
    simple authentication interceptor 225
    using the codec 238
extracting tagged information 245
get_slot() operation 250
installing the interceptor 229, 234
interceptors build flag 477, 970, 973
IOR interceptors 243–244, 246–247, 257
IORInfo.pidl 244
IORInterceptor.pidl 243
marshaling 237
name() operation 232, 245
ORBInitializer interface 229
PortableInterceptor::Current interface 249
PortableInterceptor::IORInfo interface 244
PortableInterceptor::PICurrent interface 223, 249–253, 257
PortableServer module 235
post_init() operation 225
pre_init() operation 225
real-time CORBA 174, 179, 199–200
receive_exception() operation 222
receive_other() operation 222
receive_reply() operation 222
receive_request_service_context() operation 222
receive_request() operation 223

registering interceptors 224
IOR 245
request interceptors 220
request parameters 223
RequestInfo.pidl 223
send_exception() operation 223
send_other() operation 223
send_poll() operation 221
send_reply() operation 223
send_request() operation 222
server interception points 222
server request interceptors 222, 231–232, 241
ServerRequestInterceptor.pidl 222
server-side interceptor 231
service context 174, 237
set_slot() operation 250
tagged components 243
target_is_a() operation 232
portable object adapter (POA) 11, 14
active object map 92, 302, 600, 602–603, 606, 614
manager 477, 596, 704, 750, 769
map 600
map options 601
policy
MULITPLE_ID 613
PERSISTENT 608–609
SYSTEM_ID 606
TRANSIENT 609
UNIQUE_ID 613
USER_ID 604, 606
portable priorities 15, 151
PortableServer module 275, 339
interfaces 302
PortableServer::ServantActivator interface 302
PortableServer::ServantLocator interface 302–303, 305
portspan option 326, 336, 517–518, 523
post_init() operation 225, 229–230, 234, 245, 247
postinvoke() operation 303–304
pre_init() operation 225, 229, 245
prefix filter builder
real-time event service (RTES) 757
preinvoke() operation 303

priority
end-to-end 15, 151, 166
model 208, 211
client-propagated 207
server-declared 211
timers
real-time event service (RTES) 743
priority inversion 4, 16, 151, 166, 204, 427–428, 476, 514, 538–539, 627, 728–729, 732–733, 797, 804–807, 815
priority mapping 160, 185, 189
continuous 185–186
custom network 197
direct 187
linear 188
maxPriority 158
minPriority 158
network 191–194, 196–197
NetworkPriorityMapping interface 195–197
NetworkPriorityMappingManager interface 196–198
ORBPriorityMapping option 190, 202
ORBRTPriorityRange option 155
PriorityMapping interface 159, 190
PriorityMappingManager interface 190
RTORBNetworkPriorityMapping option 196, 203
RTORBPriorityMapping option 203
priority model
client-propagated 15, 205
server-declared 15
priority propagation 166
end-to-end 15, 151, 166
priority-banded connections 167
PriorityBands 167
PriorityMapping interface 159–160, 190
PriorityMappingManager interface 190
PriorityModelPolicy policy 166, 207
private keyword
value types 262, 264
PrivateConnectionPolicy policy 168
probe build flag 1093, 1097
process_directive() operation 444, 458
profile build flag 1092, 1097
profile locking 618

properties
BlockingPolicy 844–845, 916
ConnectionReliability 840–841, 845, 848–849, 864
DiscardPolicy 839, 842, 844–846, 916
EventReliability 841, 845, 851–852, 864, 941
MaxConsumers 842, 845, 892
MaxEventsPerConsumer 840, 845, 916
MaximumBatchSize 832, 840, 845
MaxQueueLength 842, 845, 892
MaxSuppliers 842, 845, 892
OrderPolicy 839, 845–846, 893, 916
PacingInterval 832, 840, 845
Priority 839, 845
RejectNewEvents 839, 842, 844, 846
ThreadPool 843, 846, 853, 855, 908, 913
ThreadPoolLanes 843, 846, 853–854, 905, 913
Timeout 838, 846
property service 14
ORB service libraries 638
protocol
asymmetric 341–342
ProtocolList 171
protocols 522
Collocated-only inter-ORB protocol (COIOP) 322
Compression IIOP (ZIOP) 322
corbaloc rir 656
CSI ECMA 971, 975
datagram inter-ORB protocol (DIOP) 12, 322, 333–335, 517, 522, 583
DCE Common Inter-ORP protocol (DCE-CIOP) 971, 974
environment specific inter-ORB protocol (ESIOP) 319–320
factories 356, 559, 583
general inter-ORB protocol (GIOP) 265, 319–321, 324, 328, 330, 335, 337, 522
generic security service (GSS) 975
HTTP tunneling bi-directional protocol (HTBP) 341
HTTP tunneling inter-ORB protocol (HTIOP) 12, 322, 341
hypertext transfer protocol (HTTP) 12
internet inter-ORB protocol (IIOP) 319–328, 331, 334–336, 338, 349–350
lite 328, 584
options 326
multicast inter-ORB protocol (MIOP) 12, 322, 337, 339–340
pluggable 5, 12, 16
properties 151, 169

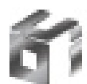
provided with TAO 321
resolve initial reference 483
SCTP inter-ORB protocol (SCIOP) 12, 322, 344
secure inter-ORB protocol (SECIOP) 971, 974–975
secure socket layer inter-ORB protocol (SSLIOP) 12, 322, 335–336, 517, 522, 961, 982–983, 988, 1001, 1003, 1005–1006, 1009–1012, 1014, 1017, 1019–1026, 1029–1035, 1154
secure sockets layer (SSL) 12, 322, 335, 522, 963, 975, 983
shared memory inter-ORB protocol (SHMIOP) 12, 322, 330–333, 341, 349–350, 352–355, 482, 509, 511–512, 516–517, 520–521, 523–524, 552, 583, 622, 656, 661
stream control transmission protocol (SCTP) 12, 344
transmission control protocol/internet protocol (TCP/IP) 9, 169, 319–321, 324, 331, 352, 489, 510, 559, 583, 783, 988
UNIX inter-ORB protocol (UIOP) 12, 169, 172–173, 322, 328–330, 353, 482, 509–512, 516–517, 521–524, 552, 583–584
unreliable IP multicast protocol (UIPMC) 322–323, 337, 339–340
user datagram protocol (UDP) 12, 321–322, 333, 335, 337, 352, 522, 583, 773, 777–778, 780–781, 783

protocols properties 169
proxy collection options
CECProxyConsumerCollection 719, 728–729
CECProxySupplierCollection 719, 732–733
ECProxyPushConsumerCollection 790
ECProxyPushSupplierCollection 790
real-time event service (RTES) 790

proxy factory adapter 282
proxy pattern 16
ProxyConsumer interface 833, 872, 887, 897
ProxyPushSupplier interface 701
ProxySupplier interface 833, 878–879, 887, 901–902
pseudo-IDL (PIDL) 108, 113, 155, 221–223, 243–244, 249, 255, 299, 1096
pthread_create() operation 397
public keyword
value types 262, 264

public-key cryptography 985
publish and subscribe paradigm 694
consumers 694
suppliers 694

pull model support 694, 706
Pure CORBA book xxvii, xxxi, 259, 275, 646–647, 1157
purge_percentage() operation 557–558



purging strategies 557
first in first out (FIFO) 551, 557
least frequently used (LFU) 551
least recently used (LRU) 551
no operation (NOOP) 551, 557
purify build flag 1092, 1097–1098
push consumer interface
real-time event service 748
push_structured_event() operation 873, 875, 880
push_structured_events() operation 898–899, 903
push() operation 697, 743, 747–748, 750, 760
PushConsumer interface 748–749
disconnect_push_consumer() operation 699, 748
push() operation 748
PushSupplier interface 698, 746
Pyarali, Irfan 1159

Q

QNX operating system 328
QoP
See quality of protection (QoP)
qt build flag 1092, 1098
Qt GUI toolkit 550, 568, 572
quality of protection (QoP) policy 1012
quality of service (QoS) 75, 102–104, 112, 115, 151, 153, 693, 746–747, 749–752, 756–768, 772, 828–830, 832, 837–838, 841–843, 845–848, 870–871, 877, 892–893
asynchronous requests 112
buffered oneway 112
connection timeouts 107
notification service 837
object reference level 104
oneway 112
ORB level 104
ORBPolicyManager interface 104, 107
policies 102
policy framework 153
properties
notification service 892
RELATIVE_RT_TIMEOUT_POLICY_TYPE 106

reliable oneway calls 109
request and reply timeouts 106
support 837
SyncScope 109, 112
SYNC_DELAYED_BUFFERING 110
SYNC_NONE 109
SYNC_WITH_SERVER 110
SYNC_WITH_TARGET 110
SYNC_WITH_TRANSPORT 110
thread level 104
time independent invocation (TII) 75
quantify build flag 967, 1092, 1098

R

rate-monotonic scheduling
See real-time CORBA, fixed priority scheduling
Rational Software
Purify 1097
Quantify 1098
reactive
concurrency model 385–388, 394
strategy 578
reactors 16–412, 549, 1159
ACE_FlReactor class 552
ACE_FoxReactor class 552
ACE_QtReactor class 552, 568
ACE_Reactor class 441, 449–450, 551–552
ACE_TkReactor class 552
ACE_XtReactor class 552, 569–570
fl build flag 1092, 1095
fox build flag 1092, 1096
management 552
msg_wfmo option 592
qt build flag 1092, 1098
reclaim_reactor() operation 552
select_mt option 410–411, 573, 592
select_st option 387, 389, 409–410, 553, 573, 592
thread pool 163
tk build flag 1093, 1101

tk_reactor option 573, 592
wfmo option 370, 573, 592
xt build flag 1093, 1102
real-time collocation resolver 205, 428, 487
real-time CORBA 11, 15, 72, 149–164, 166–167, 169–173, 185, 189–198, 201–212, 350, 385, 391, 394, 1151, 1159
_get_policy() operation 207
architecture 152–153
asynchronous invocations 173
attributes
poa_policies 181
scheduling_discipline_name 181
scheduling_policies 181
client-propagated priority model 15, 205
communication resources 151
configuring 202
ORB 202
corba_messaging build flag 201
create_mutex() operation 161
create_named_mutex() operation 198
create_priority_banded_connection_policy () operation 167
create_reference_with_id_and_priority() operation 211
create_reference_with_priority() operation 211
datagram inter-ORB protocol (DIOP) 169
destroy_mutex() operation 161
differentiated services (diffserv) 121, 191
codepoints (DSCP) 121, 191
diffserv field (DF) 121, 191
distributable thread 151, 155, 174–177, 179
dynamic scheduling 15, 149, 151, 174, 199
dynamic threads 162
earliest deadline first (EDF) scheduling 174, 182
EDF_Scheduling module 182
enable_network_priority attribute 169, 191
enabling support in TAO 201
end-to-end priority propagation 151, 166
example 156, 209, 217
explicit binding 151, 166
extensions 15, 153
features 173
fixed priority scheduling 174, 181
FP_Scheduling module 181

FP_Scheduling::FP_Scheduler interface 181
FP_Scheduling::SegmentSchedulingParameterPolicy interface 181
general inter-ORB protocol (GIOP) 169
implementation 185
INCOMPATIBLE_SCHEDULING_DISCIPLINES exception 181
least laxity first (LLF) scheduling 183
LLF_Scheduling module 183
lock() operation 160
Max_Utility_Scheduling module 184
maximize accrued utility (MAU) scheduling 184
maxPriority 158
MIF_Scheduling module 185
minimum_corba build flag 201
minPriority 158
modules 154
most important first (MIF) scheduling 184
mutex
    named 198
Mutex interface 178
MutexNotFound exception 198
named mutexes 198
NetworkPriorityMapping interface 195–197
NetworkPriorityMappingManager interface 196–198
object binding 166
open_named_mutex() operation 198
options
    ORBSchedPolicy 202–203
    ORBScopePolicy 202, 204
    RTORBDynamicThreadIdleTimeout 204
    RTORBDynamicThreadRunTime 204
ORB 155, 159, 195, 202
    configuration 202
ORBPriorityMapping option 190, 202
ORBRTPriorityRange option 155
ORBSchedPolicy option 203
ORBScopePolicy option 204
overview 150
policies 152
portable interceptors 174, 179, 199–200
portable object adapter (POA) 153, 156
portable priorities 15, 151
priority 160

priority mapping 153, 157, 160, 185–186, 189
continuous 185–186, 203
direct 185, 187, 202–203
linear 185, 188, 202
maxPriority 158
minPriority 158
network 191–194, 196–197
NetworkPriorityMapping interface 195–197
NetworkPriorityMappingManager interface 196–198
ORBPriorityMapping option 190, 202
ORBRTPriorityRange option 155
PriorityMapping interface 159, 190
PriorityMappingManager interface 190
RTORBPriorityMapping option 203
priority models 208, 211
client-propagated 205, 207
server-declared 208–209, 211
priority propagation 166
priority-banded connections 167
PriorityBands 167
PriorityMapping interface 159, 190
PriorityMappingManager interface 190
PriorityModelPolicy policy 166, 207
private connections 168
PrivateConnectionPolicy policy 168
PROCESS policy 204
processor resources 151
protocol properties 151, 169
ProtocolList 171
quality of service (QoS) 151
policy framework 153
rate-monotonic scheduling
See real-time CORBA, fixed priority scheduling
real-time operating system (RTOS) 157
reliable oneway invocation 173
request buffering 152
resources 151
rt_corba build flag 201
RT_ORB_Loader 190, 202
RTCORBA module 154, 391, 394
RTCORBA::Current interface 92, 153, 159, 176
RTCORBA::Mutex interface 153, 160

RTCORBA::PriorityMapping interface 160
RTCORBA::RTORB interface 153
RTCORBA.pidl 155
RTORBDynamicThreadIdleTimeout option 204
RTORBDynamicThreadRunTime option 204
RTORBNetworkPriorityMapping interface 196
RTORBNetworkPriorityMapping option 203
RTORBPriorityMapping option 203
RTPortableServer extension 11, 153–157, 201, 210–211
RTPortableServer.pidl 155
RTScheduler.pidl 155, 179
RTScheduling extension 155
RTScheduling::Current interface 155, 174, 176–177, 199–200, 212
RTScheduling::DistributableThread interface 174
RTScheduling::ResourceManager::lock() operation 178
RTScheduling::ResourceManager::try_lock() operation 179
RTScheduling::ResourceManager::unlock() operation 179
RTScheduling::Scheduler interface 179, 199
RTScheduling::ThreadAction interface 175, 212, 214
SCHED_FIFO policy 203
SCHED_OTHER policy 203
SCHED_RR policy 203
schedulers 151
scheduling 173
    ResourceManager 178
scheduling segment 174, 176–179, 182, 199, 212, 215
scope_policy 204
SCTP inter-ORB protocol (SCIOP) 169
SERVER_DECLARED policy 209
server-declared priority model 15, 208
ServiceContexts 166
shared memory inter-ORB protocol (SHMIOP) 169
standard synchronizers 151
static scheduling 149, 151, 174
static threads 162
support 201
SYSTEM policy 204
TAO_HAS_CORBA_MESSAGING macro 201
TAO_HAS_MINIMUM_CORBA macro 201
TAO_HAS_RT_CORBA macro 201, 1098
TCPProtocolProperties 169, 191
thread action objects 175

thread borrowing 162
thread lanes 161
thread pools 16, 151, 161, 163
ThreadpoolId 153, 162
ThreadpoolLane 153
ThreadpoolLanes 164
ThreadpoolPolicy 163
timeouts 173
to_CORBA () operation 159, 186, 195
to_native () operation 159, 186
to_network () operation 195
transmission control protocol/internet protocol (TCP/IP) 169
try_lock() operation 160
UNIX inter-ORB protocol (UIOP) 169
unlock() operation 161
utility function 184
real-time event service (RTES) xxix, xxxv, 15, 693, 741–747, 749, 753, 755, 762, 764–765, 773–817, 1157
architecture 742
attributes 770
consumer_poa 770
consumer_reconnect 770
disconnect_callbacks 770
scheduler 770
supplier_poa 770
supplier_reconnect 770
bit-mask filters 759
connecting 746, 749–751
constructing filters 756
consumer proxy 742–743, 747, 749–750, 881
CORBA gateway 774
deadline timeouts 765
disconnecting 750–751
dispatching model 743
event
correlation 743, 760
data 754
dispatching 743
filtering 743
header 753, 774
structure 753
type 747, 749, 766

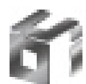
event channel
dispatching strategy 716
resource factory 784
options 792
ECConsumerControl 789, 793, 820–825
ECConsumerControlPeriod 789, 794
ECConsumerControlTimeout 789, 795
ECConsumerValidateConnection 796
ECDispatching 797
ECDispatchingThreadFlags 800
ECDispatchingThreads 799
ECFiltering 801
ECObserver 802
ECProxyConsumerLock 803
ECProxyPushConsumerCollection 804
ECProxyPushSupplierCollection 806
ECProxySupplierLock 808
ECQueueFullServiceObject 809
ECScheduling 810
ECSupplierControl 789, 811
ECSupplierControlPeriod 789, 812
ECSupplierControlTimeout 790, 813
ECSupplierFiltering 814
ECTimeout 815
ECTPCDebug 816
ECUseORBId 817
servants 768
event channel attributes 770
event structure 753
customizing 755
options
TAO_LACKS_EVENT_CHANNEL_ANY 755
TAO_LACKS_EVENT_CHANNEL_OCTET_SEQUENCE 755
TAO_LACKS_EVENT_CHANNEL_TIMESTAMPS 755
Event_Service program
command line options 783
EventData 754
EventHeader 753
EventType special values
ACE_ES_EVENT_ANY 754
ACE_ES_EVENT_DEADLINE_TIMEOUT 754
ACE_ES_EVENT_INTERVAL_TIMEOUT 754

ACE_ES_EVENT_SHUTDOWN 754
ACE_ES_EVENT_UNDEFINED 754
example 744
local event channel 768
feature control options 787
ECConsumerValidateConnection 788
ECFiltering 787
ECObserver 788
ECProxySupplierLock 788
ECScheduling 788
ECSupplierFilter 788
ECSupplierFiltering 787
ECTPCDebug 788
federating event channels 772, 782
IP multicast 781
UDP 777
filtering
and correlation 755
building 757
combinations 758
complex filters 756
conjunction groups 760
construction 764
custom filters 764
disjunction groups 759
event type 757
logical AND groups 761
negating the logic of filters 761
nesting groups 762
null filters 764
rejection filters using bit masks 763
source ID 758
interval timeouts 765
locking options 788
ECProxyConsumerLock 789
ECProxySupplierLock 789
managing connections 750
observers 771
operations
resume_connection() 752
suspend_connection() 752
ORB service libraries 639

prefix filter builder 757
priority timers 743
proxy collection options 790
push consumer interface 748
reconnecting
consumers 751
suppliers 751
rejection filters 763
resource factory
consumer and supplier control options 789
resuming consumer connections 752
specifying and constructing filters 756
subscription and filtering 743
supplier proxies 743, 751
suspending consumer connections 752
timeout 765
events 743
real-time scheduling 173, 212–217
real-time scheduling service
ORB service libraries 639
rebind() operation 312, 714
receive_exception() operation 222
receive_other() operation 222
receive_reply() operation 222
receive_request_service_context() operation 222
receive_request() operation 200, 223
reclaim_reactor() operation 552
reconfigure() operation 444, 458
record protocol 990
red-black tree 719
reference counting 309
register_new_factory() operation 266
register_proxy_factory() operation 282–283
register_value_factory() operation 261
registering interceptors 224
IOR 245
regular value types 275
rejection filters using bit masks 763
RejectNewEvents property 839, 842, 844, 846
relationship between ACE and TAO 16
RELATIVE_RT_TIMEOUT_POLICY_TYPE 103, 106–107
release() operation 309, 371

reliable oneway invocation 109, 173
SyncScope policy 109
remove directive 444
reply handler 82, 84
exception 90
non-exception 86
operations 82
repo build flag 1092, 1098
repository_id() operation 301
RepositoryId 224, 946–947
request
buffering 152
demultiplexing
active 11, 16
interceptors 220
invocation 360
processing 361
request invocation 360–361
request processing 361–384
RequestProcessingPolicy policy 305
resolve_initial_references() operation 30, 155, 160, 179, 190, 196, 237, 252, 300, 312, 482, 508, 640, 647, 655–656, 664, 980, 983, 1021, 1070
resolve_str() operation 661
resolve() operation 696
resource factory 323, 331, 389, 412, 416, 437, 462, 549–553, 557, 559–560, 563–564, 568–572, 574–576, 578–593, 715, 786
ACE_Data_Block 553
ACE_FlReactor 552
ACE_FoxReactor 552
ACE_Message_Block 553
ACE_QtReactor 552
ACE_QtReactor class 568
ACE_Reactor 552
ACE_TkReactor 552
ACE_XtReactor 552
advanced 328, 550, 552–555, 558, 565, 571–572, 590
advanced resource factory options 571, 588
Advanced_Resource_Factory configuration 328, 389, 410, 412
allocator configuration 564
amh_response_handler_allocator() operation 565
ami_response_handler_allocator() operation 565
cache management strategies 556

cache_maximum() operation 557–558
CDR conversion allocators 549, 553
Client_Strategy_Factory configuration 437–438, 617, 625–630
codeset identifiers and translators 563
common data representation (CDR) 553
connection cache management strategies 556
create_cached_connection_lock() operation 557–558
create_corba_object_lock() operation 566
create_flushing_strategy() operation 556
create_lf_strategy() operation 553
create_purging_strategy() operation 557
custom factory 583
custom interoperable object reference (IOR) parsers 560
default 328, 550–551, 553–556, 558–559, 566, 572
event channel 715–721, 769, 784–790
Fl 550
flushing strategy 556
Fox 550
general inter-ORB protocol (GIOP) 553, 583
get_parser_names() operation 560
get_protocol_factories() operation 559
init_protocol_factories() operation 559
input_cdr_allocator_type_locked() operation 554
input_cdr_buffer_allocator() operation 554, 566
input_cdr_dblock_allocator() operation 554, 566
input_cdr_msgblock_allocator() operation 554, 566
interface definition 551
internet inter-ORB protocol (IIOP) 552
locked_transport_cache() operation 557–558
max_muxed_connections() operation 557–558
options 574
    notify manager 911
    ORBAMHResponseHandlerAllocator 565, 588
    ORBAMIResponseHandlerAllocator 565, 588
    ORBCharCodesetTranslator 564
    ORBConnectionCacheLock 555, 558–559, 575, 590
    ORBConnectionCacheMax 559, 575
    ORBConnectionCachePurgePercentage 559, 576, 590
    ORBConnectionPurgingStrategy 558, 572, 589–590
    ORBCorbaObjectLock 566
    ORBDropRepliesDuringShutdown 577–578
    ORBFlushingStrategy 417, 421, 556, 578–579



ORBInputCDRAllocator 389, 554–555, 572, 575, 590–591
ORBIORParser 563, 579
ORBMuxedConnectionMax 559, 580
ORBNativeCharCodeset 564
ORBNativeWCharCodeset 564
ORBObjectKeyTableLock 582
ORBOutputCDRAllocator 555
ORBPriorityMapping 202
ORBProtocolFactory 323–324, 328–329, 331–332, 334, 338, 517, 560, 583–584, 1010–1011, 1014, 1017
ORBReactorMaskSignals 584
ORBReactorThreadQueue 573, 591–592
ORBReactorType 387, 389, 409–412, 542, 553, 573, 592–593
fl 592
msg_wfmo 592
select_mt 592
select_st 592
tk_reactor 592
tp 592
wfmo 592
ORBResourceUsage 567–568, 585–586
ORBSchedPolicy 203
ORBWCharCodesetTranslator 564
ORBZeroCopyWrite 555
output_cdr_buffer_allocator() operation 554
output_cdr_dblock_allocator() operation 554
output_cdr_msgblock_allocator() operation 554
protocol factory 324, 338, 549, 559, 583, 1010–1011, 1014, 1017
purge_percentage() operation 557–558
purging strategies 556
first in first out 557
least frequently used 557
least recently used 557–558
no operation (NOOP) 557
Qt GUI toolkit 550, 568, 572
reactor management 549, 552
reclaim_reactor() operation 552
resource usage strategy 566–567, 585
eager 551, 567
lazy 551, 567
Resource_Factory configuration 550, 569–570, 574–576, 578–582, 584–587
resource_usage_strategy() operation 567

select_st option 553
TAO_CONNECTION_CACHE_MAXIMUM macro 576
TAO_Flushing_Strategy class 556
TAO_LF_Strategy class 553
TAO_QtResource_Factory class 568
TAO_Resource_Factory class 551
TAO_Strategies library 571
TAO_Transport_Cache_Manager class 558
TAO_XT_Resource_Factory class 570
Tk 550
UNIX inter-ORB protocol (UIOP) 552, 583
use_locked_data_blocks() operation 554
WaitForMultipleObjects() operation 593
X windows toolkit 568–569
Xt 550, 572
XtAppContext 570
resource usage strategy 566–567, 585
eager 551, 567
lazy 551, 567
Resource_Factory configuration 550, 569–570, 574–576, 578–582, 584–587
resource_usage_strategy() operation 567
resume directive 445
resume_connection() operation 881
RETAIN policy 302
reuse_addr option 327
Riehle, Dirk 1158
Rohnert, Hans 17, 393, 423, 1159
root naming context 696
RT CORBA
See real-time CORBA
RT_Collocation_Resolver class 205, 487
rt_corba build flag 201, 1093, 1098
RT_ORB_Loader 190
options
ORBPriorityMapping 202
ORBSchedPolicy 202–203
ORBScopePolicy 202, 204
RTORBDynamicThreadIdleTimeout 204
RTORBDynamicThreadRunTime 204
RTORBNetworkPriorityMapping 203
RT_ORB_Loader, configuration 196, 202

RTCORBA module 72, 149–164, 166–167, 169–173, 185, 189–198, 201–212, 391, 394
RTCORBA::Mutex interface 160, 178
RTCORBA::Priority interface 160
RTORBDynamicThreadIdleTimeout option 204
RTORBDynamicThreadRunTime option 204
RTORBNetworkPriorityMapping option 196, 203
RTPortableServer extension 11, 153–157, 201, 210–211
RTPortableServer module 157
RTScheduling extension 154–155
RTScheduling::Current interface 177
RTScheduling::ResourceManager interface 178, 181
RTScheduling::Scheduler interface 178–180, 199
Ruh, William 320, 1159
run() operation 367, 381, 423, 1067
rwho build flag 1094, 1099

S

SCHED_FIFO policy 161, 202–203
SCHED_OTHER policy 203
SCHED_RR policy 161, 203
schedulers 151
scheduling
  real-time 212–217
scheduling segment 174, 176–179, 182, 199, 212, 215
scheduling service 190, 745, 747, 771, 784, 788, 801, 810
scheduling_discipline_name attribute 181
scheduling_policies attribute 181
Schmidt, Douglas C. xxiii, xxxvii–xxxix, 3, 6, 17, 93, 153, 220, 365, 393, 423, 1157–1160
scope_policy policy 204
SCTP
  See SCTP inter-ORB protocol (SCIOP)
  See stream control transmission protocol (SCTP)
sctp build flag 1093, 1099
SCTP inter-ORB protocol (SCIOP) 12, 169, 322, 344
SECIOP
  See secure inter-ORB protocol (SECIOP)
secret-key cryptography 984

secure inter-ORB protocol (SECIOP) 971, 974–975
  interoperability 971, 974–975
secure socket layer inter-ORB protocol (SSLIOP) 12, 322, 335–336, 961, 982–983, 988, 1001, 1003, 1005–1006, 1009–1012, 1014, 1017, 1019–1026, 1029–1035, 1154
  endpoints 522
  factory 336
  options 1029
  SecurityLevel1 module 1019
  SSLCertificate option 1032, 1034
  SSLIOP_Factory, configuring 583
  SSLIOP::Current interface 1019
  SSLNoProtection option 1033, 1035
  SSLPrivateKey option 1029, 1034
secure sockets layer (SSL) protocol 12, 322, 335, 522, 963, 965, 975, 983–993
  architecture 987
  example 992
  SSLCertificate option 1034
security 961, 965
  access identity 967
  administration 969
  alert protocol 992
  application
    control 969
    enforcement 969
  architecture 975, 977, 981
  asymmetric encryption 985
  audit identity 967
  auditing 968
  binding 987
    client and target 980
  building 1001
    on UNIX 1002
    on Windows 1003
    security-aware applications 1004
  certificate 986–987, 994
    authority 987, 995
    commands summary 1001
    issuing 998
    multiple authorities 1007
    request 996, 998
  change cipher specification protocol 991

ciphertext 984
common secure interoperability (CSI) packages 962, 971, 974–975
  CSI 0 971, 974
  CSI 1 962, 971, 974
  CSI 2 962, 971, 974
context information 980
controlling
  message protection 1013
  peer authentication 1013
creating
  certificate authority 995
  certificate requests 996
credential 969
CSI ECMA protocol 971, 975
DCE Common Inter-ORP protocol (DCE-CIOP) 971, 974
delegation of attributes 967
digital signatures 986
distinguished name 987
environment setup 975, 994, 1006
establish trust policy 1012
example 992, 1010
  building libraries 1014
  security policy enforcement 1020
  SSL session 992
feature packages 970
  common secure interoperability (CSI) 974
  main functionality 972
  optional 973
  SECIOP + DCE-CIOP interoperability 974
  SECIOP interoperability 974
  security mechanism 974
    CSI ECMA protocol 975
    generic security service (GSS) protocol 975
    secure sockets layer (SSL) protocol 975
    simple public key mechanism (SPKM) protocol 974
  security replaceability 973
    ORB services 973
    security services 973
generic security service (GSS) 975
handshake protocol 989
identity
  attributes 967

binding 986
certificates 986
implementation architecture 975
initiating principal 967
Kerberos 971, 975, 984
key
pair 985
removing pass phrases 1000
server 984
Level 1 972
Level 2 972
libraries
building 1014
testing 1004
message
authentication codes 986
confidentiality 983
controlling protection 1013
integrity 986
protection 1013
messaging 969
module 118, 219, 225, 246, 321, 323, 330, 335, 351, 522, 540
non-repudiation 968, 970, 973
OpenSSL 322, 964, 983, 991, 994–996, 998, 1000–1003, 1005–1006, 1019
optional packages 973
options
SSLCertificate 1034
SSLNoProtection 1033
SSLPrivateKey 1029, 1034
originating principal 967
peer authentication 1013
controlling 1013
plaintext 984
policy
controlling 961, 964, 969, 1012, 1155
example 1014
enforcing 961, 969, 1018, 1155
example 1020
establish trust 1012
principal 966
identification and authentication 966
privacy enhanced mail (PEM) 994

private key 985
privilege attributes 967
public key 985
public-key cryptography 985
quality of protection (QoP) 1012
record protocol 990
reference model 966
removing key pass phrases 1000
secret-key cryptography 984
secure inter-ORB protocol (SECIOP) 971, 974–975
secure socket layer inter-ORB protocol (SSLIOP) 961, 1009, 1154
secure sockets layer (SSL) protocol 12, 322, 335, 522, 963, 965, 975, 983–993
architecture 987
Security module 1013
SecurityAdmin module 970, 972
SecurityLevel1 module 962, 970, 972, 982, 1019
Current interface 1019
SecurityLevel2 module 962, 970, 972, 982, 1019
SecurityLevel3 module 962, 1019
self-signed certificate 987
signing certificate requests 998
simple delegation 968
symmetric encryption 984
ticket 984
transparent protection 968
transport layer security (TLS) 983
unaware 961, 964, 1155
application 1005
example 1010
building executables 1004
Security Model Mixed 1026
Security module 972, 1013, 1015–1018
security service 14
SecurityAdmin module 970, 972
SecurityLevel1 module 962, 970, 972, 982, 1019
SecurityLevel2 module 962, 970, 972, 982, 1019
SecurityLevel3 module 962, 1019
select reactor
single-threaded 409
select_mt option 410–411, 573, 592
select_st option 387, 389, 409–410, 553, 573, 592
select() operation 352, 410–411, 592

send_exception() operation 223
send_other() operation 223
send_poll() operation 221
send_reply() operation 223
send_request() operation 222
sendc_ prefix 77, 79, 93
SEQUENCE_EVENT type 872, 878, 897, 901
SequenceProxyPushConsumer interface 829, 897
SequenceProxyPushSupplier interface 902
SequencePushConsumer interface 832, 894, 899, 902
SequencePushSupplier interface 832, 894, 898
servant 10, 94, 96
activation
USE_SERVANT_MANAGER 305
locator 302
manager 302, 477
NON_RETAIN 305
RequestProcessingPolicy 305
ServantRetentionPolicy 305
ServantLocator interface 302–303, 305
servant-per-AMI-call strategy 91–92
ServantRetentionPolicy policy 302, 305
server concurrency 597
server interceptors 241
interception points 222
request 222
server role 359, 361
server strategy factory 71, 386, 389–392, 435, 437, 462, 595–599, 602–604, 612–615
active object map 602
default options 604
demultiplexing strategies 599
interface definition 596
options
ORBActiveHintInIDs 603–605
ORBActiveHintInPOANames 602, 605–606
ORBActiveObjectMapSize 603, 606
ORBConcurrency 386, 389–390, 392, 598, 607–608
ORBPOALock 389, 609
ORBPOAMapSize 602, 609–610
ORBSystemIDPolicyDemuxStrategy 603, 610
ORBThreadFlags 598, 611

ORBThreadPerConnectionTimeout 391–392, 599, 612
ORBUniqueIDPolicyReverseDemuxStrategy 603, 613–614
ORBUserIDPolicyDemuxStrategy 604, 614–615
POA map 601
Server_Strategy_Factory configuration 386, 389–390, 392, 437–438, 595, 605–615
SERVER_DECLARED policy 166, 209, 211–212
server-differentiated-reply strategy 91–92
ServerRequestInfo interface 223
service configurator 12, 433–434, 436–448, 450–458, 463, 562, 571, 574, 721
configuring TAO clients and servers 434
control options 439
creation 451
directives 437–438, 444
DTD 446
initialization and finalization 450
interface definition 449
manager 457
objects 448
specializing factories 436
state 456
static service example 455
svc.conf file 329, 338, 392, 436, 441, 443, 541, 786, 818, 1014, 1017
See also ACE, service configurator
service configurator file 388
service manager 444–445
ServiceContexts 166
services 635–640
audio/video streaming 14, 637
concurrency control 4, 14, 638
Cos prefix 636
customizing access 636
data distribution 14
event xxix, xxxv, 14, 638, 693–695, 741–749, 755, 830, 837
implementation repository 312, 910, 1071
ImR_Activator 1076
interface repository xxix, 14, 62, 945–948, 951–952, 958–959
life cycle 14, 638
load balancing 14, 280, 489, 619, 638
log 14
naming 14, 312, 484, 636, 640, 645, 647–660, 662–667, 670–672, 678–680, 744, 769, 870, 877, 896, 910

NamingViewer utility 670
notification xxxv, 14, 312, 693, 827–829, 832–833, 837–838, 842, 847, 864–866, 869–871, 889, 891, 893, 909, 916, 918–941
ORB service libraries 280, 638
overview of TAO 635
property 14
real-time event service (RTES) xxix, xxxv, 15, 693, 741–747, 749, 753, 755, 762, 764–765, 773
scheduling 190, 745, 747, 771, 784, 788, 801, 810
security 14, 639
selectively building TAO services 1112
TAO ORB libraries 637
tao_nsadd utility 667, 669
tao_nsdel utility 667, 669
tao_nslist utility 667
time 14, 106, 183
trading 14, 468, 482, 484–487, 508, 542–543, 635, 704
set_locator() operation 314–315
set_policy_overrides() operation 105, 301, 972, 980, 983
set_qos() operation 843
set_scheduling_parameter() operation 181
set_slot() operation 250
shared memory inter-ORB protocol (SHMIOP) 12, 169, 322, 330–333, 341, 349–350, 352–355, 482, 509, 511–512, 516–517, 520–521, 523–524, 552, 583, 622, 661
endpoint 520
example 523
factory 331
SHMIOP_Factory configuration 332, 583
shared_libs build flag 1092, 1099–1100
shared_libs_only build flag 1092, 1099
SHMIOP
See shared memory inter-ORB protocol (SHMIOP)
shutdown() operation 364, 371–376, 1022–1023, 1025, 1041, 1067
shutting down 374
SII
See static invocation interface (SII)
single thread model 379
SINGLE_THREAD_MODEL policy 378, 381–383
smart proxies xxix, 6, 49, 277–278, 280–281, 286–288, 290–291, 296–298
_unchecked_narrow() operation 285
asynchronous method invocation (AMI) 278

base_proxy_ variable 284
batch processing 280
choosing between target objects 280
client-side caching 279
create_proxy() operation 278, 282–283, 285–286, 288, 293
creating 285
default proxy factory 283, 285
example 289–297
logging 289
executing a sequence of operations 280
factory object 288
framework 280
get_proxy() operation 283–284
IDL compiler 278
instance() operation 284
logical target object 280
oneway 278
overridden member functions 284
proxy factory adapter 282, 285
register_proxy_factory() operation 282–283
smart proxy base class 283
smart proxy class 278, 287, 293
smart proxy factory class 285, 288, 292
smart_proxies build flag 281, 287
TAO_Default_Proxy_Factory class 281–282
TAO_Proxy_Factory_Adapter class 281
TAO_Smart_Proxy_Base class 281, 283–284, 294
tao/SmartProxies/Smart_Proxies.h 281, 287
unregister_proxy_factory() operation 282, 289
use cases 279
writing and using 286
smart_proxies build flag 281, 287
SO_DONTROUTE socket option 467, 488
SO_KEEPALIVE socket option 467, 513
SO_LINGER socket option 515
SO_REUSEADDR socket option 327
sockets
CLOSE_WAIT state 425
spawn() operation 174, 176–177, 212
specializing factories 436
split build flag 1092, 1099
SSI

See static skeleton interface (SSI)
SSL
See secure sockets layer (SSL) protocol
ssl build flag 1002–1003, 1093, 1100
SSL_CERT_DIR environment variable 1006–1007
SSL_CERT_FILE environment variable 1006–1007
SSL_EGD_FILE environment variable 1006
ssl_port option 517
SSL_RAND_FILE environment variable 1006
SSL_ROOT environment variable 1002–1003
SSLIOP
See secure socket layer inter-ORB protocol (SSLIOP)
Stal, Michael 17, 393, 423, 1159
standard synchronizers 151
state member
value type 259
static
threads 162
static invocation interface (SII) 10, 13
static scheduling 149, 151, 174
static skeleton interface (SSI) 10, 13
static_libs build flag 1092, 1099–1100
static_libs_only build flag 1092, 1099–1100
static_link build flag 1092, 1100
Stevens, W. Richard 1160
STLport 1093, 1100
stlport build flag 1092–1093, 1100
strategies 435
activation-per-AMI-call 91–92
blocking 578
connect 625
first in first out (FIFO) 557
leader_follower 417
leader/followers 556, 578
least frequently used (LFU) 557, 572, 589
least recently used (LRU) 557–558, 572, 589
no operation (NOOP) 557
reactive 578
servant-per-AMI-call 91–92
server-differentiated-reply 91–92
strategy pattern 16
stream control transmission protocol (SCTP) 12, 344

string_dup() operation 31
string_to_object() operation 31, 251, 314, 339, 485–486, 532, 560–561, 579, 641, 647, 655–657, 659
structured event types 693
STRUCTURED_EVENT type 872, 878–879
StructuredProxyPushConsumer interface 870, 872–874
StructuredProxyPushSupplier interface 829, 878–879, 893
StructuredPushConsumer interface 831, 875, 879, 907, 909
StructuredPushSupplier interface 831, 867, 873
stubs and skeletons 10, 28, 39
subscription and filtering
 real-time event service (RTES) 743
subscription_change() operation 836, 867–868, 879, 884–885, 887–888, 894, 902
supplier proxies 743
 real-time event service (RTES) 743
SupplierAdmin interface 828, 836, 871–872, 882, 889–890, 896–897
support
 commercial xxxiii
 Object Computing, Inc. (OCI) xxxiii
suspend directive 445
suspend_connection() operation 881
suspending and resuming consumer connections
 real-time event service (RTES) 752
svc.conf file 332, 338, 436, 441, 443, 1014, 1017
svc() operation 215, 397, 665
SYNC_DELAYED_BUFFERING 113
SYNC_NONE 109–110, 113–114
SYNC_WITH_TARGET 111
SYNC_WITH_TRANSPORT 110
synchronizers
 standard 151
SyncScope policy 102, 109–110, 112–114
 SYNC_DELAYED_BUFFERING 110, 113
 SYNC_NONE 109–110, 113–114
 SYNC_WITH_SERVER 110
 SYNC_WITH_TARGET 110
 SYNC_WITH_TRANSPORT 110–111
System Requirements 1084
Syyid, Umar 1158

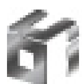
T

tagged components 243, 245, 540
TaggedComponent() operation 247
TAO xxv, 3, 551
architecture 8
design goals 4
development history 6
high performance 15
minor codes 68
ORB service libraries 280
real-time CORBA support 15, 201
real-time event service (RTES) 15
relationship to ACE 16
source code distribution 1120
supported platforms xxxii
tao_catior utility 248
TAO_ROOT environment variable 42, 44–45
version xxxii
TAO classes
RT_Collocation_Resolver 205, 487
TAO_Acceptor 71, 353–354, 557
TAO_Acceptor_Registry 71, 551
TAO_CEC_EventChannel 695, 702
TAO_CEC_EventChannel_Attributes 704
TAO_CEC_Factory
interface definition 716
TAO_Connect_Strategy 618, 623
TAO_Connection_Handler 353, 355
TAO_Connection_Purging_Strategy 551, 557
TAO_Connector 70, 72, 353–354, 557
TAO_Connector_Registry 70, 72, 551
TAO_Default_Proxy_Factory 281–282
TAO_EC_Event_Channel_Attributes 770–771
TAO_EC_Factory
dispatching 786
TAO_Endpoint 353, 355
TAO_Flushing_Strategy 551, 556
TAO_IOR_Parser 561, 579
TAO_LF_Strategy 553
TAO_Local_RefCounted_Object 299, 309

TAO_MProfile 70
TAO_Naming_Client 663–664
TAO_Naming_Server 664, 666
TAO_Notify_EventChannelFactory_i 904
TAO_ORB_Core 71, 462, 596–597, 618, 623
TAO_Priority_Mapping_Manager 190
TAO_Profile 353, 355
TAO_Protocol_Factory 353, 356
TAO_ProtocolFactorySet 552
TAO_Proxy_Factory_Adapter 281
TAO_QtResource_Factory 568–569
TAO_Resource_Factory 551
TAO_Server_Strategy_Factory 596–597
TAO_Smart_Proxy_Base 281, 283–284, 294
TAO_Transport 353, 618, 620–621
TAO_Transport_Cache_Manager 558
TAO_Wait_Strategy 618, 621
TAO_XT_Resource_Factory 570–571

TAO configuration xxviii–xxix, 5, 433
client strategy factory configuration 414–415, 421–422, 424, 426, 437–438, 617, 625–630
directives 436
dynamic 324, 389, 438, 450, 454
dynamic compontents 438
resume 445, 456
static 389, 437, 450, 550, 595, 617, 715, 785, 818
suspend 445, 456
event channel servants
real-time event service (RTES) 768
object factory 434–435
resource factory configuration 324, 338, 417, 421, 437–438, 447–448, 542, 550, 569–570, 574–576, 578–582, 584–587, 1010–1011, 1014, 1017
server strategy factory configuration 386, 389–390, 392, 435, 437–438, 595, 605–615
source code
on UNIX 1107
specializing factories 436
static directive 437

TAO libraries
ORB service 489, 619
TAO_BiDirGIOP 117
TAO_CosConcurrency 638



TAO_CosEvent 638
TAO_CosLifeCycle 638
TAO_CosLoadBalancing 638
TAO_CosNaming 54, 638, 646
TAO_CosNaming_Serv 646
TAO_CosNaming_Skel 646
TAO_CosNotification 638
TAO_CosProperty 638
TAO_CosTime 639
TAO_CosTrading 638
TAO_DsEventLogAdmin 638
TAO_DsLogAdmin 638
TAO_DsNotifyLogAdmin 638
TAO_Messaging 101, 115
TAO_PortableGroup 323, 337–338
TAO_RTCORBA 72, 190, 196, 201
TAO_RTEvent 639, 755
TAO_RTEventLogAdmin 638
TAO_RTPortableServer 201
TAO_RTSched 639
TAO_RTSchedEvent 639
TAO_Security 639
TAO_SmartProxies 286, 289
TAO_Strategies 323, 328–329, 331–334, 389, 410, 412, 571, 588–593
TAO_Svc_Utils 639
TAO macros
CORBA_E_COMPACT 1149, 1151
CORBA_E_MICRO 1149
TAO_CONNECTION_CACHE_MAXIMUM 576
TAO_CONNECTOR_REGISTRY_NO_USABLE_PROTOCOL 70
TAO_DEFAULT_IMPLREPO_SERVER_REQUEST_PORT 507
TAO_DEFAULT_NAME_SERVER_REQUEST_PORT 527, 642
TAO_DEFAULT_SERVER_ACTIVE_OBJECT_MAP_SIZE 599, 606
TAO_DEFAULT_SERVER_POA_MAP_SIZE 609
TAO_DEFAULT_TRADING_SERVER_REQUEST_PORT 543
TAO_HAS_AMI 1094
TAO_HAS_AMI_CALLBACK 1094
TAO_HAS_AMI_POLLER 1094
TAO_HAS_CORBA_MESSAGING 201, 1094
TAO_HAS_INTERCEPTORS 1096
TAO_HAS_MINIMUM_CORBA 201, 1097, 1107, 1111
TAO_HAS_MINIMUM_POA 1112

TAO_HAS_RT_CORBA 201, 1098
TAO_INVOCATION_LOCATION_FORWARD_MINOR_CODE 69
TAO_INVOCATION_RECV_REQUEST_MINOR_CODE 70
TAO_INVOCATION_SEND_REQUEST_MINOR_CODE 69
TAO_MPROFILE_CREATION_ERROR 70
TAO_POA_DISCARDING 69
TAO_POA_HOLDING 69
TAO_STD_PROFILE_COMPONENTS 541
TAO_THREAD_PER_CONNECTION_TIMEOUT 612
TAO_TIMEOUT_CONNECT_MINOR_CODE 70
TAO_TIMEOUT_RECV_MINOR_CODE 70
TAO_TIMEOUT_SEND_MINOR_CODE 70
TAO_UNHANDLED_SERVER_CXX_EXCEPTION 70
TAO_USE_LAZY_RESOURCE_USAGE_STRATEGY 567, 586
TAO_Acceptor_Registry class 551
TAO_BiDirGIOP library 117
tao_catior utility 248
TAO_CEC_EventChannel class 695
TAO_Connection_Purging_Strategy class 551, 557
TAO_Connector_Registry class 551
TAO_DEFAULT_SERVER_ACTIVE_OBJECT_MAP_SIZE macro 606
TAO_DEFAULT_SERVER_POA_MAP_SIZE macro 609
TAO_DEFAULT_THREAD_PER_CONNECTION_TIMEOUT macro 599
TAO_EC_Event_Channel_Attributes class 770–771
TAO_Flushing_Strategy class 551, 556
TAO_HAS_MINIMUM_CORBA macro 1111
TAO_HAS_MINIMUM_POA macro 1112
tao_idl program 28, 39–40, 42–47, 51, 53–54, 96, 307
TAO_IDL_PREPROCESSOR environment variable 42, 44
TAO_IDL_PREPROCESSOR_ARGS environment variable 44
tao_ifr program 947, 949–950, 958–959
options 949–950
tao_imr program 1062
TAO_IOR_Parser class 561
TAO_Naming_Client class 663–664
TAO_Naming_Server class 664, 666
TAO_Notify_EventChannelFactory_i class 904
tao_nsadd utility 667, 669
tao_nsdel utility 667, 669
tao_nslist utility 667

TAO_ORB_Core class 462
TAO_ORB_DEBUG environment variable 480
TAO_ORBENDPOINT environment variable 462, 489, 518
TAO_ProtocolFactorySet class 552
TAO_Resource_Factory class 551
TAO_ROOT environment variable 26, 42, 44–45
TAO_THREAD_PER_CONNECTION_TIMEOUT macro 612
TAO_Transport_Cache_Manager class 558
TAO_USE_IMR environment variable 1039–1040, 1046, 1048
TAO_USE_LAZY_RESOURCE_USAGE_STRATEGY macro 567, 586
target_is_a() operation 232
TCP_NODELAY option 530
TCP/IP
  See transmission control protocol/internet protocol (TCP/IP)
TCPProtocolProperties 169, 172, 191–194
template option 46
templates build flag 1101
testing 1139
  building tests 1139
  running tests 1140
    ACE tests 1140
    ORB Services tests 1142
    TAO tests 1141
The ACE Programmer's Guide (APG) book 17, 370, 386, 398, 409, 434, 1158
the_POAManager() operation 30
the_priority attribute 160, 208
THR_BOUND 598, 611
THR_DAEMON 611
THR_DETACHED 598, 611
THR_JOINABLE 396–397
THR_NEW_LWP 396–397, 598, 611
THR_SUSPENDED 611
thread action objects 175
thread borrowing 162
thread pools 151, 153–154, 161–163, 165, 204
  concurrency model 392–395, 411, 427
  reactor 385, 392–395, 409–411, 424, 426
thread-per-connection concurrency model 335, 340, 385, 389–392, 599, 612, 706
ThreadPolicy policy 378, 381–382, 384

thread-pool concurrency model 385
ThreadPool property 843, 846, 853, 855, 908, 913
thread-pool reactor 163, 592
ThreadpoolId 153, 162–164
ThreadpoolLane 153
ThreadpoolLanes 164
ThreadPoolLanes property 843, 846, 853–854, 905, 913
ThreadpoolPolicy 163
threads
 dynamic 162
 lanes 161
 static 162
 thread specific storage (TSS) 16
threads build flag 1092, 1101
Threads Primer: A Guide to Multithreaded Programming book 358, 1158
time independent invocation (TII) 75
time service 14, 106, 183
TimeBase module 106, 183
Timeout property 838, 846
timeouts 173
 real-time event service (RTES) 765
TimeT type 106
tk build flag 1093, 1101
tk_reactor build flag 573, 592
TLS
 See transport layer security (TLS)
to_CORBA() operation 159, 186, 195, 197
to_native() operation 159, 186
to_network() operation 195, 197
to_string() operation 661
to_url() operation 661
tools
 GNU Make 33–34
 MakeProjectCreator (MPC) 31
 Visual C++ 31, 33–34
tp option 592
trading service 14, 468, 482, 484–487, 508, 542–543, 635, 704
 options
  ORBTradingServicePort 468, 542–543, 643
 TradingServicePort 542, 643
TradingServicePort environment variable 542, 643

training xxxiv, xxxvi
Advanced CORBA Programming Using TAO xxxv
CORBA Programming with C++ xxxiv
Introduction to CORBA xxxiv, 964
Object Oriented Design Patterns and Frameworks xxxv
Using the ACE C++ Framework xxxv
TRANSIENT exception 61, 65, 69, 71, 532, 601–602, 609, 613, 1012
transmission control protocol/internet protocol (TCP/IP) 9, 169, 319–321, 324, 331, 352, 489, 510, 559, 583, 783, 988, 1147
transport 356
multiplexing strategies 415–416
transport layer security (TLS) 983
Trask, Bruce 1158
truncatable keyword
value types 271, 275
try_lock() operation 160, 179
type codes
wchar 563–564, 586–587
typed event channel 695

U

UDP
See user datagram protocol (UDP)
UIOP
See UNIX inter-ORB protocol (UIOP)
UIOP_Factory configuration 328–329, 583–584
UIPMC
See unreliable IP multicast protocol (UIPMC)
unbind() operation 312
University of California, Irvine
See DOC Group
UNIX 12, 33, 40, 43, 56, 328, 332, 459, 461, 489, 512–513, 519, 521, 530, 583, 1002, 1091
building ACE and TAO 1105
configuring source code 1107
customizing ACE and TAO builds 1111
set environment variables 1106
verifying build 1110
See also building

UNIX domain sockets 12
UNIX inter-ORB protocol (UIOP) 12, 169, 322, 328–330, 332, 334, 338, 482, 509, 511–512, 516, 521–522, 524, 532, 584
endpoint 521
examples 524
UIOP_Factory configuration 328–329, 583–584
UNIX Network Programming book 1160
unlock() operation 160–161, 179
unmarshaling 590
unregister_proxy_factory() operation 282, 289
unreliable IP multicast protocol (UIPMC) 322–323, 337, 339–340
untyped event channel 695
update_scheduling_segment() operation 177
use_locked_data_blocks() operation 554
USE_SERVANT_MANAGER policy 305
user datagram protocol (UDP) 12, 321–322, 333, 335, 337, 352, 522, 583, 773, 777–778, 780–781, 783
Using the ACE C++ Framework course xxxv
utilities
naming service 667
tao_nsadd 669
tao_nsdel 669
tao_nslist 667
tao_catior 248
tao_imr 1062
utility function 184

V

validate_connection() operation 301
value box 274–275
value types 259–275
abstract keyword 272
abstract value type 269, 272, 274
class 263
compliance 274
DefaultValueRefCountBase class 264
defining in IDL 261
eventtype keyword 261, 275
example 262

factory 261–263, 265–266, 268
implementing 262
private keyword 262, 264
public keyword 262, 264
regular value types 275
state members 259
truncatable keyword 271, 275
uses 260
value box 274–275
value type class 263
value type factory 265
ValueFactoryBase class 265
valuetype keyword 259, 261
ValueFactoryBase class 265
valuetype keyword
value types 259, 261
Vanderbilt University xix, xxxvii, 3
vc6 build type 34
versioned_so build flag 1092, 1101
Vinoski, Steve xxvii, xxxi, 153, 220, 1157, 1159
Visual C++ xxx, 31, 33–34, 1085, 1115
build libraries 1119
configure source code 1117
setting up environment 1117
verify build 1120
Vlissides, John 17, 1157
VxWorks iv, 479, 664, 1104, 1123–1125, 1129, 1139–1140, 1142
environment 1125
kernel and system configuration 1124
using ACE and TAO 1123, 1129

W

wait strategies 417–427, 620
wait-on-leader-follower 418, 422–424
wait-on-leader-follower-no-upcall 418, 424–427
wait-on-reactor 418, 422
wait-on-read 414–422
wait_for_completion parameter 364, 371–375
WaitForMultipleObjects() operation 352, 593

wait-on-leader-follower concurrency model 630–631
wait-on-leader-follower wait strategy 418, 422–424
wait-on-leader-follower-no-upcall wait strategY 424
wait-on-leader-follower-no-upcall wait strategy 418, 424–426
wait-on-reactor wait strategy 418, 422
wait-on-read wait strategy 414–421
Wang, Nanbor 1159
Washington University, St. Louis, Missouri xxxvii, 3
See also DOC Group
wchar type codes 563–564, 586–587
wfmo build flag 1093, 1102
wfmo reactor 370, 573, 592
Windows 25, 33, 453
winregistry build flag 1093, 1102
work_pending() operation 89, 364, 366–368, 370–371, 374
WrongPolicy exception 211
wxWindows 670

X

X windows toolkit 568–569
XML service configurator 434, 445–447
syntax 447
xt build flag 1093, 1102
XtAppContext 570

Z

ZIOP 322
zlib build flag 1102