Building, Configuring, and Running the Example Application for SETT Article "Bridging XMPP and DDS Messaging Frameworks"

The following steps are described for the purpose of locating, building/installing, and running the sample code related to the Software Engineering Tech Trends (SETT) article entitled <title>.

Prerequisites

1. Perl 5.10 or later
2. OpenDDS 3.0.1 or later (www.opendds.org)
3. TAO 1.6a or later (www.theaceorb.org)
4. Make Project Creator (MPC) (included with TAO)
5. OpenFire 3.7.1 or later (http://www.igniterealtime.org/projects/openfire/)
6. Boost 1.42 or later (www.boost.org)
7. Swiften (http://swift.im/swiften)
    a. Swiften requires Python and OpenSSL (see installation below)
8. An XMPP-compliant chat client (Pidgin is used in the example)
9. Example Message Engine implementation source < SETT code location>
10. A C++ compiler (see the Compiler section at www.opendds.org/building.html)

Perl

Perl is used for various steps in the installation of OpenDDS, TAO and MPC as well as for building the SETT article example code. For Windows we recommend the installation from Activestate located here: http://www.activestate.com/activeperl/downloads . Perl is present in most all *nix distributions, but ensure that you have a version that is 5.10 or later by typing: `perl -v`

OpenDDS, TAO, and MPC Installation

If you do not have either OpenDDS or TAO, the installation of OpenDDS, TAO, and MPC can be combined. As of OpenDDS 2.2, the installation is a standard **`./configure`** style installation approach which detects the existence of a current TAO installation. If TAO is not found, it is downloaded and built along with OpenDDS. If you have a TAO installation of 1.6a or above, ensure the environment variables ACE_ROOT and TAO_ROOT are set to their proper locations for your build platform and proceed with the OpenDDS installation. MPC is included with the TAO distribution, so no additional action is required for MPC.

1. Download the latest release of OpenDDS (http://download.ociweb.com/OpenDDS/)
2. Unpack the download into a directory of choice
    a. For Windows assume (c:\dev)
    b. After unpacking the result will be: c:\dev\DDS

3. Follow the instructions in the c:\dev\DDS\INSTALL file for your respective platform to run the configure step and execute the build

OpenFire XMPP Server

This service provides the multi-user chat environment for our example. There are three user accounts and one chat room that need to be configured in the service.

1. Download and execute the installer from the link above for your platform. You can follow the installation steps at the following link: http://www.igniterealtime.org/builds/openfire/docs/latest/documentation/install-guide.html During the installation:
    a. Choose to use the default built-in database
    b. Set the password for the admin account
    c. Assume we have the final installation at `c:\Program Files\Openfire`
2. Create three user accounts. (`messageengine, bob, lou`)
    a. Navigate to `c:\Program Files\Openfire\bin`
    b. Execute `openfire.exe`
    c. The executable launches a dialog box. Click the "Launch Admin" button
    d. A browser window is started with the admin login screen. Login with user `admin` and the password that you set in the installation steps above. You will now be in the administrative console.
    e. Select the "`Users/Groups`" tab and click "`Create New User`":
        Username: `messagengine`
        Name: `Message Engine`
        Email: `(blank)`
        Password: `connect`
        Confirm Password: `connect`
        Is Administrator: `(unchecked)`
        Click "`Create User`"
    f. Repeat Step e for user `bob` and `lou`

3. Create the chat room
    a. Select the "Group Chat" tab and click "Create New Room":

        Room ID: `em`
        Room Name: `Enterprise Messages`
        Description: `Room for managing devices.`
        Topic: `(blank)`
        Maximum Room Occupants: `(default)`
        Broadcast Presence for: `(default)`
        Password Required to Enter: `(blank)`
        Confirm Password: `(blank)`
        Show Real JIDs of Occupants to: `(default)`
        Click "`Save Changes`"

4. Configuration of OpenFire XMPP server is complete

Swiften

As of this writing the Swiften library for XMPP is only available in binary distribution form for *nix platforms using an APT repository installation. For Windows and other *nix variants a local build is required. This example uses a snapshot of the Swift Git repository at http://swift.im/git/swift . If you have a Git client available to you then you can clone the repository to your local machine with the following command:

```
git clone git://swift.im/swift
```

For our example we've assumed that the above command was executed at a terminal prompt at `c:\dev\tools` resulting in `c:\dev\tools\swift`. If you do not have a Git client available to you, we have provided the snapshot of the Swift source needed for this example as a zip file name swift.zip located along with the example source code in this distribution. When reaching the repository you will see a Tag listing of the various distributions as shown below. The swift-1.0 distribution is needed for this exercise. You can click on either the .zip or .bz2 file depending on your platform.

The assumption for this exercise is that we have unpacked the .zip file into c:\dev\tools\swift on a Windows platform.

Swiften depends on **Python 2.7.x**, **OpenSSL**, and **scons** to build the Swiften.lib library.

1. Download and run the installer for Python 2.7.x from here: http://www.python.org/getit
2. Be sure that Python is set on your path
3. Download and install OpenSSL from here: http://www.slproweb.com/products/Win32OpenSSL.html
    a. The best candidate distribution for Windows on this page is: **Win32 OpenSSL v1.0.1**
    b. For our example, the installation is at c:\OpenSSL-Win32
4. Download and install **scons** from here: http://www.scons.org
5. Proceed to build the Swiften.lib library
    a. For Windows follow the instructions at the link below, however, there is an update to those instructions that must be added. In the instructions you are directed to create a file 'config.py' where you insert the path to the 'openssl' installation. In addition to that value, you will need to add another line item as follows

        ```
        set_iterator_debug_level="false"
        ```

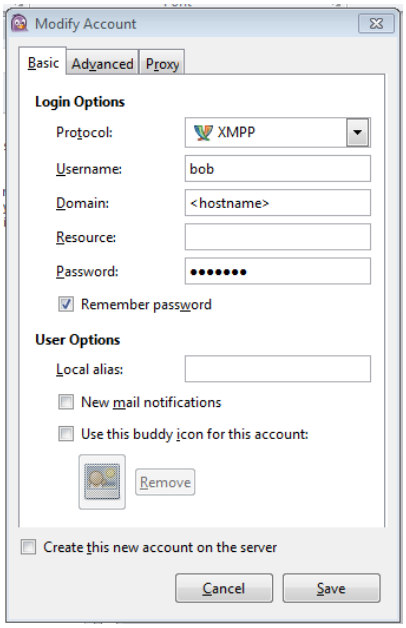
        This value if left to the default of true causes a conflict with the other software built for this example. Here is the link to the full instructions: http://swift.im/git/swift-contrib/tree/Documentation/BuildingOnWindows.txt

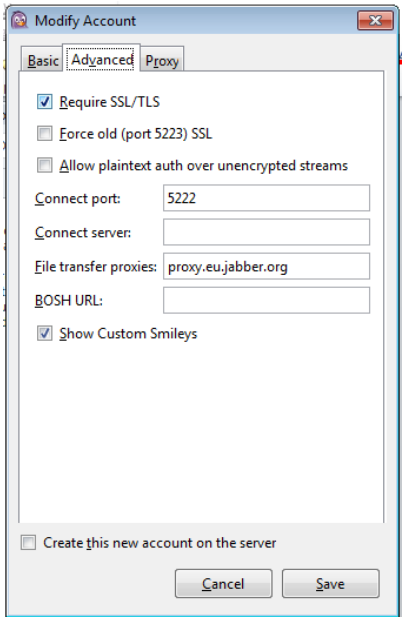
b. For *nix follow the instructions here: http://swift.im/git/swift-contrib/tree/Documentation/BuildingOnUnix.txt
c. NOTE: For this example we do not use the Qt portion of Swift, so the option of building the Qt utilities for Swift is not necessary

Installing and Configuring Pidgin

If you don't already have an XMPP-compliant chat client Pidgin is well suited for this purpose. It has a straightforward installer for most platforms at the following location: http://www.pidgin.im/download/

When we installed and configured OpenFire earlier in this document, we created three accounts: `bob`, `louie`, and `messageengine`. When running the example you will need to start more than one chat session to represent bob and `louie`, but the MessageEngine executable will handle its own XMPP chat client session to OpenFire and participate in the chat with the human users. When creating the account for the user `bob` and `louie` , set the communication protocol to XMPP, the Domain to the hostname that OpenFire is running on and set the password to '`connect`' as configured earlier with OpenFire. You should turn on SSL/TLS for the connection and use port 5222 to connect to OpenFire. Here is what these settings look like in Pidgin for the user `bob`.





Repeat the above for the user `louie`.

Installing and Building the SETT Article <title> Example Code

The example code for this article is a simple compressed file structure provided as a download from <download location here>. Let's assume that after downloading this software that it has been unpacked in c:\dev leaving us with the path c:\dev\messageengine or the equivalent on *nix platforms.

The structure of the unpacked content is as follows:

```
messageengine
   *bin - The final built executables
   *doc - This file and the SETT article
   *lib - Shared libraries built from the project
   *src - Start of source tree
      *ddsutil - Source for common OpenDDS library used by Message Engine and Devices
      *device  - Source for a Device
      *engine  - Source for Message Engine
      *idl     - Message definition exchanged between Message Engine and Devices
```

Building the Example Source

There are a number significant dependencies for this source code, so a key to the successful build of the example code is to setup your environment variables cleanly. We have provided a script to assist in setting the correct variables, however, you need to potentially edit the file to point the variables to the proper location on your platform.

1. `cd c:\dev\messageengine`
2. Open `setup.cmd`
3. Read the opening banner to understand the assumptions of where the variable are point now
4. Edit the necessary variables to match your platform
5. Execute `setup.cmd`

Generate the Project Files

This example uses the Make Project Creator (MPC) build utilities to create build system files for various compilers on multiple platforms. At the top level directory of the project (c:\dev\messageengine) there is a file named messageengine.mwc . In each subdirectory under src there is a .mpc file as well. These files will be read by MPC for the build file generation. For this example we will be using Microsoft Visual C++ 2010 as our compiler. Execute the following to generate the build files:

1. `cd c:\dev\messageengine`
2. Type `mwc.pl -type vc10 messageengine.mwc`
3. For *nix Type `mwc.pl -type gnuace messageengine.mwc`

The above will generate the proper Visual Studio 2010 solution and project files at the top level as well as the individual directories under `src` . If you have a different version of Visual Studio (e.g. version 7,8, or 9) then substitute your version in the above command with

```
mwc.pl -type vc<version> messageengine.mwc
```

Building the Project

For Windows:

1. `cd c:\dev\messageengine`
2. Execute `setup.cmd` if you closed the prior session
3. Type the following: `messageengine.sln`

Visual Studio should now be open with the `messageengine.sln` solution active. Step 3 above opens the Visual Studio product with the proper environment variables set. To build the full solution

1. Right-mouse click on "Solution 'messageengine' " at the top
2. Select "Build"

The solution will step through the four included projects in the proper order of dependency. When complete you will have executables and supporting shared libraries in the bin and lib directories of your project tree on the file system as follows:

C:\dev\messageengine\bin\MessageEngine.exe

C:\dev\messageengine\bin\Device.exe

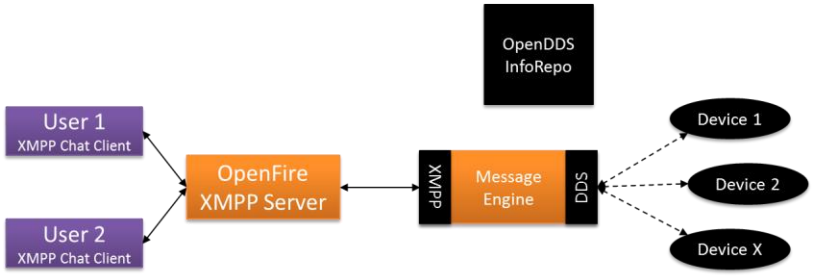
C:\dev\messageengine\lib\MessageEngine_Idld.lib

C:\dev\messageengine\lib\MessageEngine_DdsUtild.lib

The above results are the basis for the running the project as described in the next section.

Running the Example Application

As shown in the SETT article, the following figure represents the major components that need to be started for the example session. The session can be run with one or more devices started, so it is only necessary to start one device.



Here are the steps in the order that they should be executed:

1. Start the OpenFire XMPP Server
2. Start the OpenDDS InfoRepo
3. Start the Message Engine

4. Start one or more Devices
5. Join the Mult-User chat session with an XMPP chat client under the account bob or louie or both

Start the OpenFire server

When we installed the server we made the assumption that it was installed under `c:\Program Files\Openfire`. If the server is not still running from the installation and configuration steps, then navigate to `c:\Program Files\Openfire\bin` and execute `openfire.exe`. Once the console dialog shows it has started then minimize the dialog and proceed with the remaining components.

Start the OpenDDS InfoRepo

The InfoRepo process allows all participants in an OpenDDS application to locate one another via built-in discovery logic, so in our example the Message Engine and each of the started Devices will discover one another via this repository process.

1. Open a terminal session
2. `cd c:\dev\messageengine`
3. Execute: `setup.cmd` to ensure the proper environment variables are set
4. Start the InfoRepo with: `DCPSInfoRepo –ORBListenEndpoints iiop://:12345`

The command line parameter for the executable `DCPSInfoRepo` instructs the process to listen for connections from participants on local port 12345.

Start the Message Engine

The Message Engine executable was placed in the bin directory of our project root (c:\dev\messageengine\bin). You will need a separate terminal session for this process.

1. Start a new terminal session
2. Navigate to the project root: cd c:\dev\messageengine
3. Set the proper environment: setup.cmd
4. Navigate to the bin directory: cd bin

Now we are ready to execute the process. The MessageEngine has several command line parameters to improve its reuse beyond this example. At the bin directory prompt type MessageEngine -? To get a listing of the usage format as shown here.

```
c:\dev\messageengine\bin>MessageEngine -?
-u userid   : User ID used by gateway to login to XMPP server
-p password : Gateway password for XMPP server (default is 'default')
-r roomname : JabberID for the chat room (room@conference.domain.com)
-verbose    : turn on verbose logging to console
-trace      : turn on XMPP tracing
-DCPSConfigFile filename : config file used by OpenDDS participants.
```

The first two parameters are the user id and password that the Message Engine uses to connect to the OpenFire XMPP server. The user ID is in the form of a Jabber ID which includes the domain of the user account along with the user name (e.g. username@mydomain). So when we created the accounts in OpenFire we created the '`messageengine`' account with a password of '`connect`'. The roomname parameter also uses a Jabber ID format. When we created the chat room in OpenFire we gave the chat room an ID of 'em' . For multi-user chat rooms the standard approach for most facilities is to add a resource context to the Jabber ID domain. In the case of OpenFire, it defaults to adding the term 'conference' in front of the remainder of the domain. So for our example session the –r parameter looks like the following: `-r em@conference.<yourdomain>` where the domain is set to the domain of the OpenFire server host.

The `verbose` and `trace` parameters are for convenience should you wish to see message content being passed from component to component during the session. The `trace` parameter is specific to the Swift XMPP content between the Message Engine and the OpenFire XMPP server. It is in XML format and is quite verbose, so it is not recommended if not necessary.

The final parameter (`-DCPSConfigFile` *`filename`*) points to a configuration file that facilitates the use of the DCPSInfoRepo service and also allows for configuring options like timeouts, transport types, and logging levels for communications between processes that use OpenDDS. This file is used for our example to point the Message Engine to the DCPSInfoRepo that is listening on port 12345 as discussed earlier. This configuration file is found in the project room bin directory (c:\dev\messageengine\bin) with a file name of '`me.ini`' . If you change the listen port to something other than 12345 when you start the DCPSInfoRepo, then edit this file and replace the port number as necessary.

Given the above descriptions for starting the Message Engine process the command line for our example session should be executed from the bin directory like this:

```
C:\dev\messageengine\bin>  MessageEngine.exe –u messageengine@<yourdomain> -p connect
–r em@conference.yourdomain –DCPSConfigFile me.ini
```

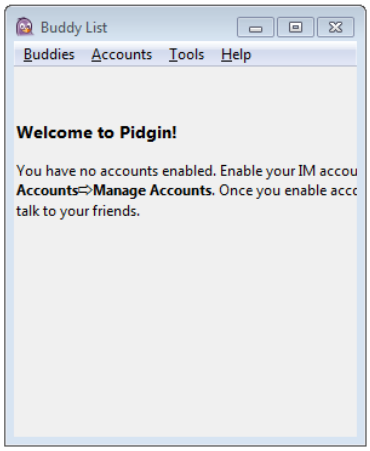
Once you've started the Message Engine, you will get a message stating that it has connected to the OpenFire XMPP server as follows: `Connected to XMPP server…`

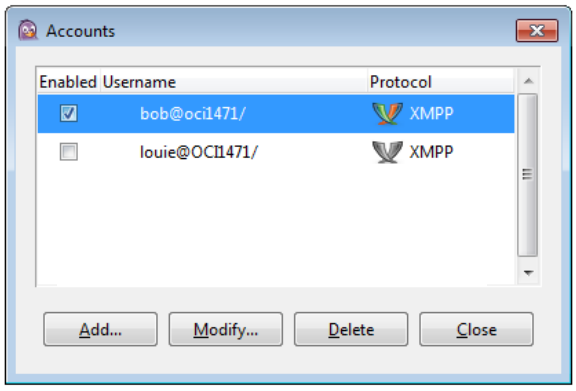
You may now minimize this terminal window and proceed to the next step.

Start a Pidgin Session

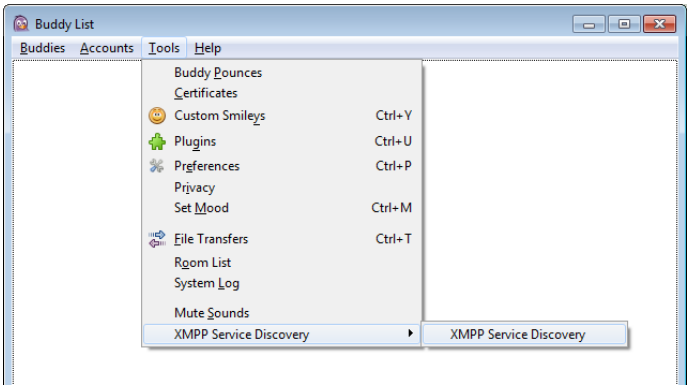
At this point in the process you have the DCPSInfoRepo, OpenFire, and the Message Engine started. Before we start up any Devices we want to be connected to the chat room with a chat client so we can see the messages that each device begins to publish at startup.

We will use the account 'bob' that we configured for Pidgin earlier to attach to the group chat room. Open Pidgin and you will see an opportunity to activate the 'bob' account that you created earlier.

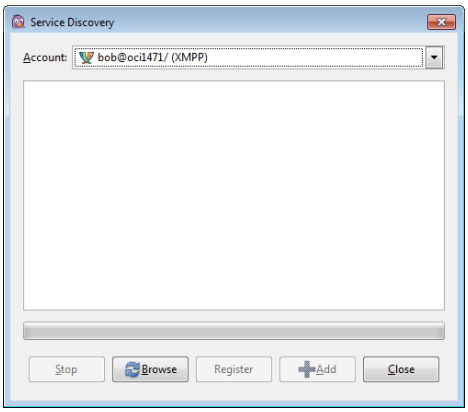




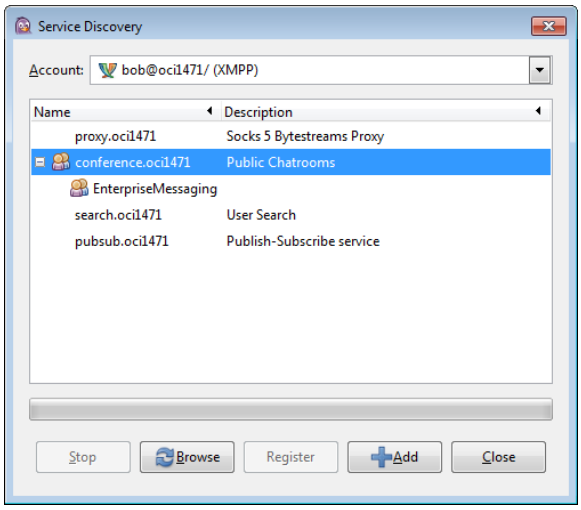
Next select **Accounts->Manage Accounts** from the menu and you will see a dialog box similar to the one on the right above. You can activate both 'bob' and 'louie' if you wish at this time. Click Close and we will now join the Enterprise Messages chat room and save the settings so it will be available to us from this point forward. From the Buddy List select Tools->XMPP Service Discovery-> XMPP Service Discovery



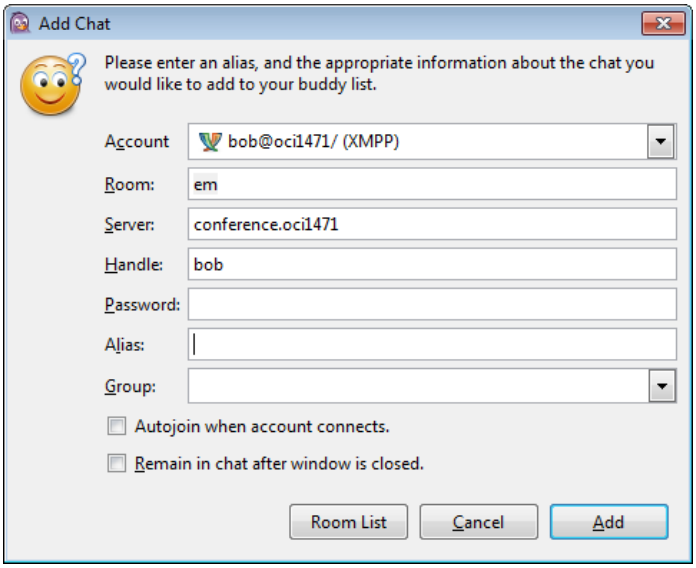
You will be presented with another dialog box that let's you choose the active user account and look for available XMPP services (e.g. group chat rooms).



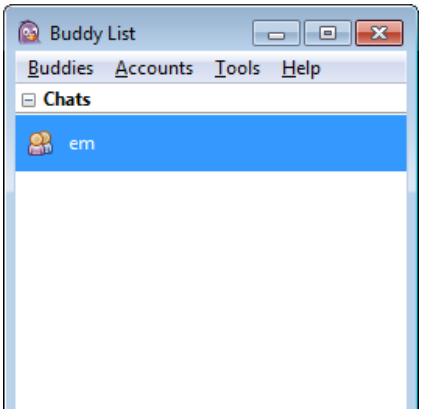
Click Browse and enter the domain that your OpenFire server is running on. You will be presented with a list of available services including a service entitled 'conference.<yourdomain>' similar to the list below:



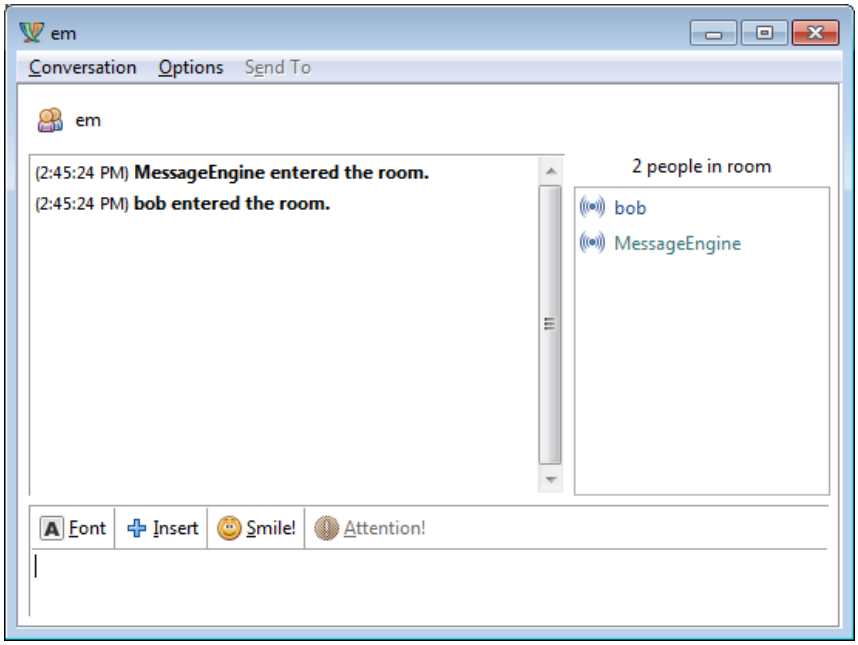
Expand the conference service and you should see the Enterprise Messaging chat room available for choice. Select it and click **Add**. The default fields in the next dialog box should be left as is.



Select **Add** again and close any remaining dialog boxes. You should now see the chat room added to your available chat services:



Now double click on the `'em'` chat room to join the room. You should see the following:



As you see above, both `'bob'` and `'MessageEngine'` have entered the room. The MessageEngine joins the room automatically using the Swiften XMPP library when it starts up. Repeat the above process for the account `'louie'`. This only needs to be done one time for each account. Now `'bob'` and `'louie'` can interact with one another while watching Device data being published to the room from the Message Engine. Follow the next steps for starting a device and be ready to observe the chat room after you start up one or more devices.

Start One or More Devices

The Devices simulate the equivalent of industrial type devices which emit diagnostic measurements including temperature and pressure. The devices send this diagnostic data to the Message Engine via OpenDDS to be reported into the chat room. Each device needs a unique identifier so those in the chat room can tell which device is emitting the data. Also each device must know how often to report the

diagnostic information. Since a device also uses OpenDDS to communicate with the Message Engine it needs to know how to discover the Message Engine via the DCPSInfoRepo, so we will supply a configuration file to accomplish that. Based on this information follow these steps to start a device:

1. In a new terminal session navigate to the project root bin directory: the project root bin directory: `cd c:\dev\messageengine\bin`
2. Set the proper environment variables: `setup.cmd`
3. Start a device:
   ```
   c:\dev\messageengine\bin>  Device.exe dev1 30 -DCPSConfigFile me.ini
   ```
4. Optionally a second device can be started in a different terminal:
   ```
   c:\dev\messageengine\bin>  Device.exe dev2 40 -DCPSConfigFile me.ini
   ```

The parameter `'dev1'` is the unique identifier and the value `'30'` is the number of seconds between diagnostic reports. Finally the devices use the same configuration file that the Message Engine does or OpenDDS discovery and communication, so the `'me.ini'` file found in the bin directory is included.

At this point you should be seeing a report every 30 seconds and 40 seconds from device dev1 and dev2 for temperature and pressure. Also the Message Engine reports the number of devices that are in the room when a device starts and leaves.

Send Commands to a Device

Another feature that has been added to the example scenario is the ability to send a command to a device from the chat session. The device source code adds a simulation feature that reports temperature and pressure measurements. A human user in the chat room can send a particular device a command to reset it's pressure to a specific value like so:

```
cmd: dev1->set_pressure 100
```

The `'cmd:'` prefix tells the Message Engine that the following command is addressed to device `'dev1'` and that the command is to set the pressure to 100. All of the participants in the chat session receive the command, of course. However, the devices have used a feature of OpenDDS to filter any messages coming to them to only allow those that are prefixed with `'cmd:'` and to only allow those messages that are addressed to their specific identifier (e.g. dev1, dev2, etc.) to be allowed through.

Example Session

Here is a sample of what you might see when Bob and Louie monitor two devices started in two separate terminal sessions:

```
(5:25:57 PM) louie entered the room.
(5:26:15 PM) bob entered the room.
(5:26:37 PM) bob: hey louie!
(5:26:55 PM) louie: hi bob! Looks like the Message Engine is up....let's see what the
devices are doing...
(5:27:58 PM) MessageEngine: 1 device is in the room
(5:28:06 PM) MessageEngine: 2 devices are in the room...
```

```
(5:28:26 PM) MessageEngine: [dev1] [Sensor Publication]Temperature = 200.000000,
Pressure = 50.000000
(5:28:31 PM) MessageEngine: [dev2] [Sensor Publication]Temperature = 200.000000,
Pressure = 50.000000
(5:28:47 PM) bob: ok... looks like dev1 and dev2 are online...
```

At this point in the chat log, bob and louie see that the MessageEngine was in the room when they logged in and that **dev1** and **dev2** are reporting their diagnostics.

```
(5:28:56 PM) MessageEngine: [dev1] [Sensor Publication]Temperature = 200.000000,
Pressure = 50.000000
(5:29:03 PM) louie: yeah... the boss wants more output from dev1 though...can you set
the pressure up to about 90 and get it going?
(5:29:11 PM) MessageEngine: [dev2] [Sensor Publication]Temperature = 200.000000,
Pressure = 50.000000
(5:29:22 PM) bob: Sure...
(5:29:25 PM) bob: cmd: dev1->set_pressure 90
(5:29:25 PM) MessageEngine: [dev1] [Acknowledged]set_pressure 90.000000
```

Louie notes that the boss wants more production from dev1 so, Bob sends a command to **dev1** to set its pressure to 90. **dev1** replies that it has done so.

```
(5:29:26 PM) MessageEngine: [dev1] [Sensor Publication]Temperature = 201.507507,
Pressure = 90.000000
(5:29:51 PM) MessageEngine: [dev2] [Sensor Publication]Temperature = 200.000000,
Pressure = 50.000000
(5:29:56 PM) MessageEngine: [dev1] [Sensor Publication]Temperature = 203.015015,
Pressure = 90.000000
(5:30:26 PM) MessageEngine: [dev1] [Sensor Publication]Temperature = 204.522522,
Pressure = 90.000000
(5:30:31 PM) MessageEngine: [dev2] [Sensor Publication]Temperature = 200.000000,
Pressure = 50.000000
(5:30:32 PM) louie: ok... it's up, but the temp is rising a bit too fast...you better
drop the pressure down about 10 or 12
(5:30:50 PM) bob: cmd: dev1->set_pressure 80
(5:30:50 PM) MessageEngine: [dev1] [Acknowledged]set_pressure 80.000000
```

After a bit, Louie notes that the temperature seems to be rising too fast and asks Bob to drop the pressure some more and Bob sends another **set_pressure** command with **dev1** acknowledging again.

```
(5:30:56 PM) MessageEngine: [dev1] [Sensor Publication]Temperature = 202.052521,
Pressure = 80.000000
(5:31:11 PM) MessageEngine: [dev2] [Sensor Publication]Temperature = 200.000000,
```

```
Pressure = 50.000000
(5:31:22 PM) bob: ok...I dropped it 10...let's see if the pressure comes down....
(5:31:26 PM) MessageEngine: [dev1] [Sensor Publication]Temperature = 199.582520,
Pressure = 80.000000
(5:31:51 PM) MessageEngine: [dev2] [Sensor Publication]Temperature = 200.000000,
Pressure = 50.000000
(5:31:56 PM) MessageEngine: [dev1] [Sensor Publication]Temperature = 199.582520,
Pressure = 80.000000
(5:32:02 PM) louie: Whew!....its settling down now...the pressure looks about right
too...thanks for the help!
(5:32:15 PM) bob: Sure...let's go to lunch!
(5:32:26 PM) MessageEngine: [dev1] [Sensor Publication]Temperature = 199.582520,
Pressure = 80.000000
(5:32:31 PM) MessageEngine: [dev2] [Sensor Publication]Temperature = 200.000000,
Pressure = 50.000000
(5:32:36 PM) MessageEngine: 1 device is in the room
(5:32:43 PM) MessageEngine: No devices are in the room...
```

The drop in pressure by Bob gets the temperature down. At the end of the session the devices are shutdown as well and the MessageEngine reports how many devices are in the room along the way.