



**OBJECT
COMPUTING**
REIMAGINE TOGETHER

WEBINAR

Introducing DDS XTypes

Presented by OCI's OpenDDS Team
January 28, 2021



© 2021, Object Computing, Inc. (OCI). All rights reserved.

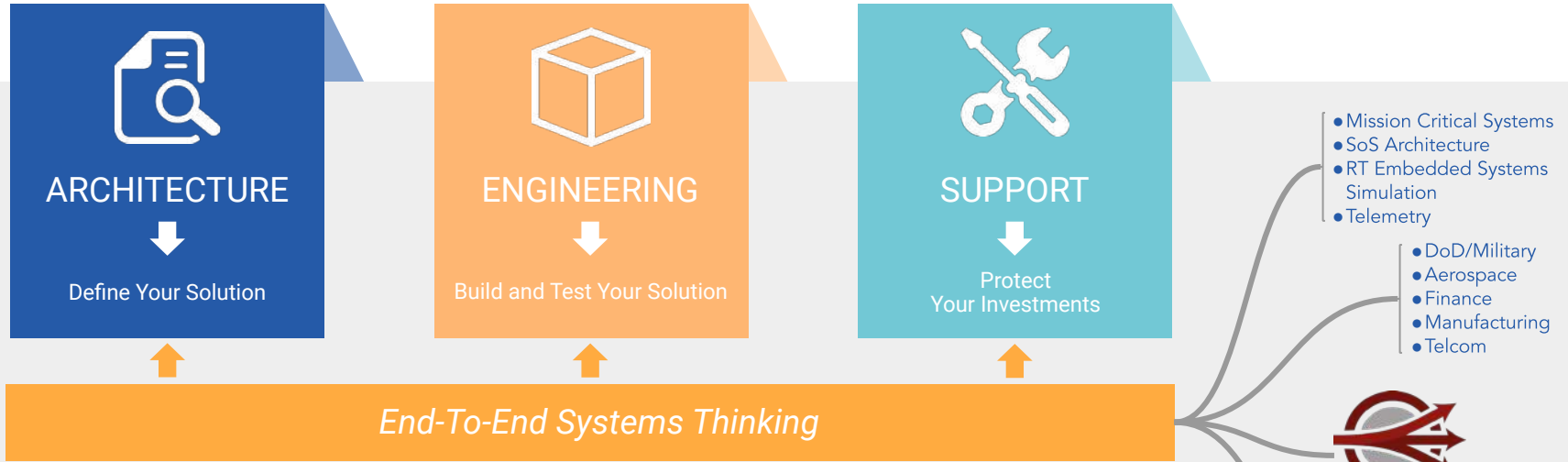
No part of these slides may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior, written permission of OCI.



OpenDDS Background

Adam Mitz

Solutions Engineering Services



Competencies

- Systems / Software Engineering
- Software Architecture
- Requirements Management
- Modeling (MBSE)
- Architecture Frameworks
- Compliance

Applied Experience

- Detailed Software Design
- Software Development
- Systems Integration
- Agile/Lean Methods
- Technology Agnostic
- Automated Testing
- Continuous Integration/Delivery
- Quality Measurement

Added Value

- Strategic Planning
- Training
- Industry Standards (OMG Member)
- Flexibility – your location or ours; your team management or ours
- Hybrid models



OpenDDS is the secure and scalable connectivity framework
connecting and powering next-generation systems



SECURE



SCALABLE



INTEROPERABLE

Standards-based Publish/Subscribe Solution



Specifies



Data Distribution Service™

Implemented By



OBJECT
COMPUTING

Resulting In



OpenDDS®

OpenDDS is an open source and widely adopted standards-based real-time publish/subscribe solution for distributed systems.

www.omg.org/spec/category/data-distribution-service-dds-foundation.org

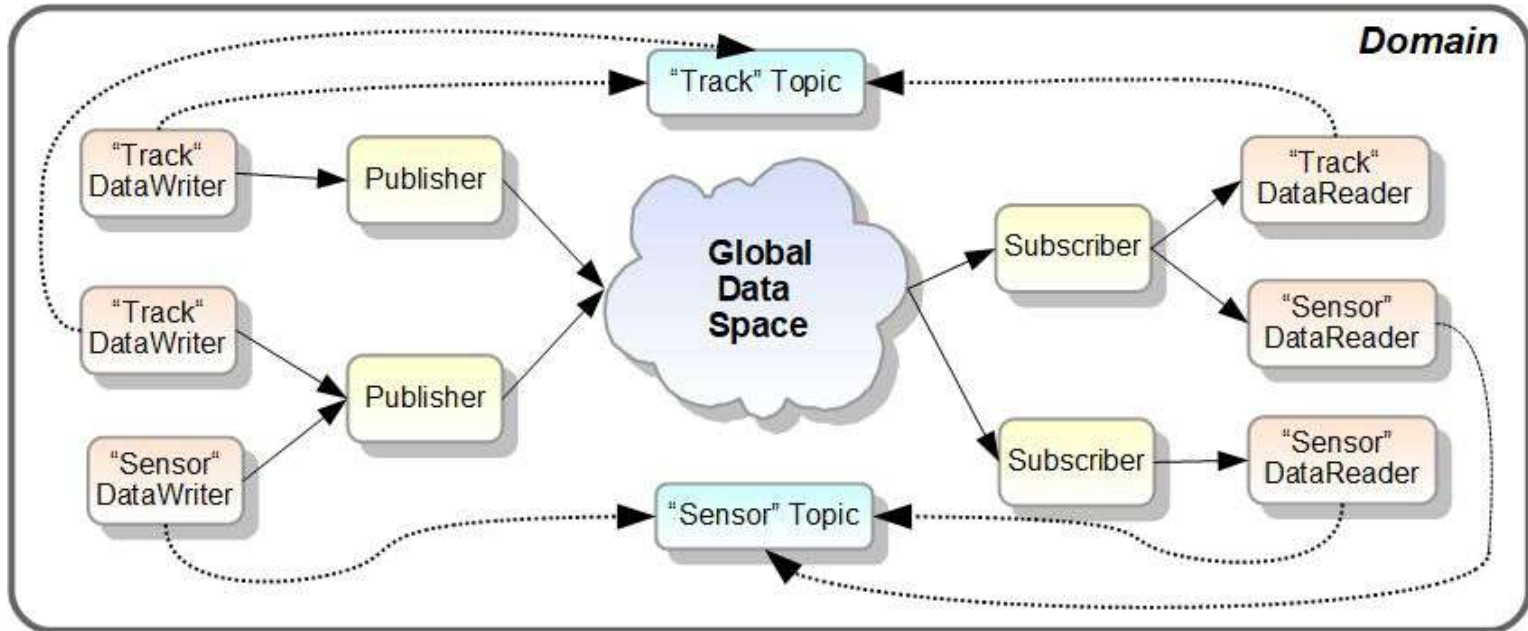
- opendds.org
- <https://github.com/objectcomputing/OpenDDS>



DDS Terms and Concepts

- Sender of data: DataWriter
- Receiver of data: DataReader
- Coordination point: Topic

DataWriters and DataReaders each act as caches. Their contents are synced over the network according to QoS policies.





OpenDDS Key Features

- Complete DDS-DCPS API: QoS policies, Content Filter, MultiTopic
 - C++ and Java
- Code generation from IDL
 - C++ (classic and C++11) and Java
- Cross-platform: desktop, mobile, embedded, cloud
- Pluggable Transport and Discovery
- Interoperable Wire Protocol (DDSI-RTPS)
- DDS Security
- RTPS over the Internet (RtpsRelay, IETF STUN and ICE)
- XTypes



Introducing XTypes

Mike Kuznetsov



Why XTypes, Extensible Topic Types in OpenDDS?

1: Distributed System Evolution

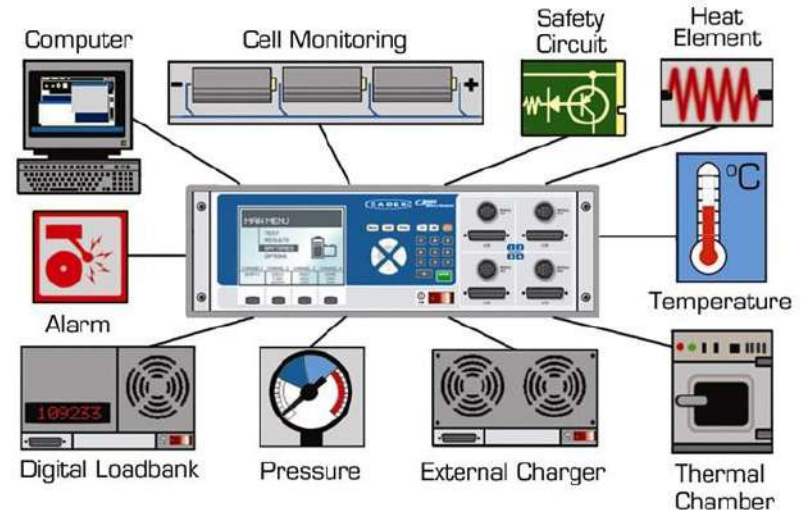
- Before XTypes: An assumption built into DDS model is that all applications agree on data type definitions for each Topic that they use. All participants (publishers and subscribers) in the distributed DDS system have **fixed data format knowledge** at compile time.
- This assumption is not practical as systems must be able to **evolve while remaining compatible** and interoperable. As new participants are added to the system, their data format may deviate from the initial design. Before XTypes this would necessitate adding new topics and / or upgrading existing participants with 'knowledge' of new data type format.
- With XTypes: Existing participants are aware of possible variances in the data format and handle addition of new-format participants **without software upgrades**.



Why XTypes, Extensible Topic Types in OpenDDS?

2: Data Variability Under One Topic

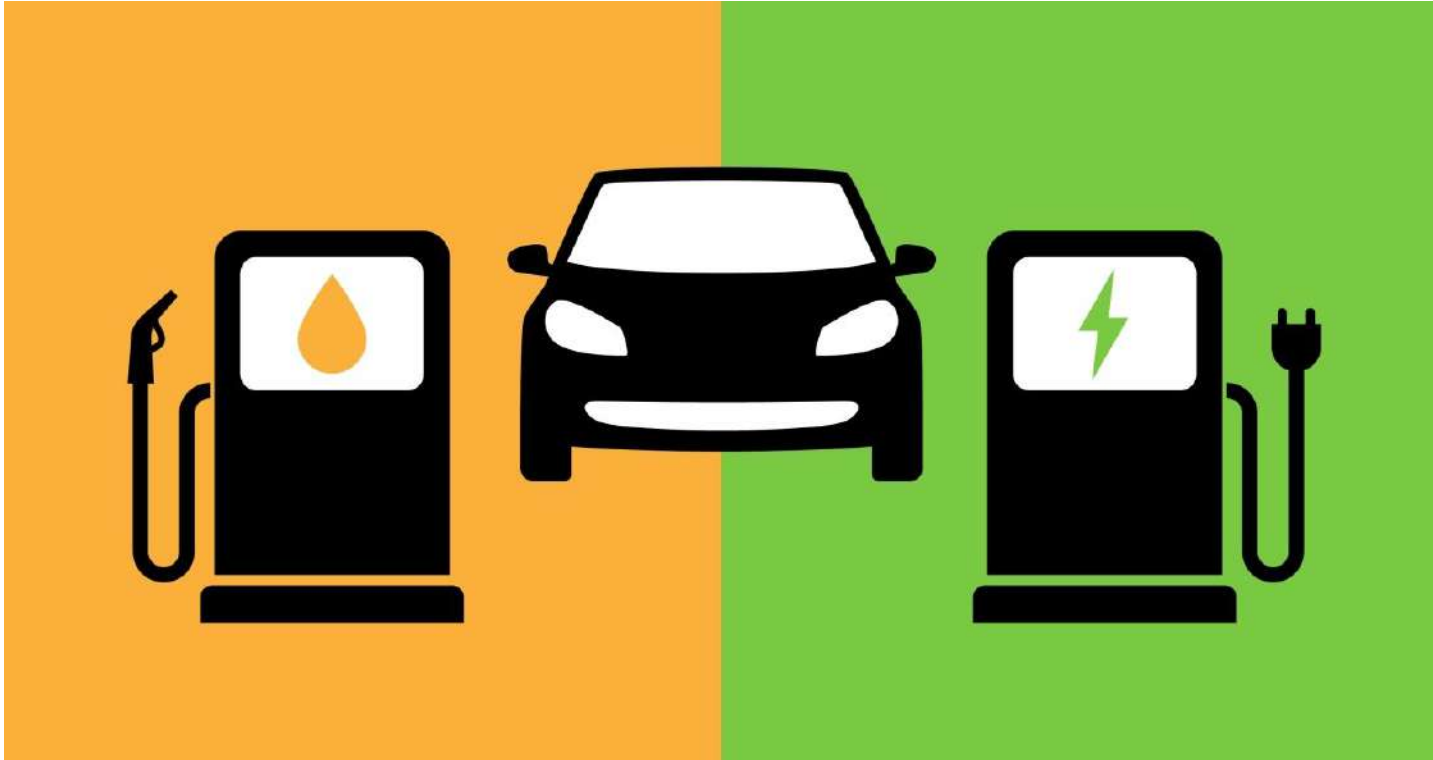
- Another use case is when system design requirements dictate DDS participants data format differentiation, but also communicating on the same topic.
- Typical battery lab test unit collects data from a variety of sources
- Driven by specific battery and test type, each test unit may have different collection of instruments and sensors, but publishes its data to a central location subscriber for test control and data acquisition
- Use of XTypes allows DDS participants to communicate with different data format on the same topic



Data Type Evolution Example



Car rental company adding EVs to its fleet





Data Type Evolution Example (cont.)

Car rental company adding EVs to its fleet

- Company uses Vehicle Remote Health Monitoring System
- Data from all vehicles is sent to the Service Center
- Data may include variables such as oil level, tire pressure, etc.
- Existing vehicles use internal combustion engines (ICE) which can be powered by gasoline or diesel.
- With addition of the Hybrid Electric Vehicles (HEV) and electric vehicles (EV), new data variables need to be added: Powertrain battery state of charge (SOC, %) and Cell Voltages (V).

Original data type

```
@topic
struct VehicleHealth {
  @key string vin;
  short oil_level;
  sequence<double, 4> tire_pressure;
};
```

Possible new data type (works for ICE, HEV or EV)

```
@topic
struct VehicleHealth {
  @key string vin;
  short oil_level;
  sequence<double, 4> tire_pressure;
  double battery_soc;
  sequence<double> cell_voltages;
};
```



Mechanism of Annotations

OpenDDS uses IDL annotations to specify some properties for data types that it transmits and processes.

- The `@topic` annotation marks a data type that can be used as a topic's type. This must be a structure or a union.
- The `@key` annotation identifies a field that is used as a key for this topic type. A topic type may have zero or more key fields. These keys are used to identify different DDS Instances within a topic.

```
@topic
struct VehicleHealth {
    @key string vin;
    short oil_level;
    sequence<double, 4> tire_pressure;
    double battery_soc;
    sequence<double> cell_voltages;
};
```

- Some annotations new to XTypes are `@final`, `@appendable`, `@mutable`, and `@try_construct`



Extensibility

Fred Hornsey

Appendable



In our VehicleHealth struct, using `appendable` is easy because it's already `appendable` by default. However `@appendable` can be applied to a struct for making that intention clear if desired.

IDL in older ICE vehicles

```
@topic
@appendable
struct VehicleHealth {
    @key string vin;
    short oil_level;
    sequence<double, 4> tire_pressure;
};
```

IDL in EVs and newer ICE vehicles

```
@topic
@appendable
struct VehicleHealth {
    @key string vin;
    short oil_level;
    sequence<double, 4> tire_pressure;
    double battery_soc;
    sequence<double> cell_voltages;
};
```

Members of `appendable` structs can't be removed, renamed, or reordered without breaking capability. New members must always go at the end.

Mutable



Mutable structs and unions are much more flexible than appendable ones. They work based on an integer that identifies each member. These IDs allows them to appear in any order and for certain members to be excluded by the writer or skipped by the reader.

We will cover more about member IDs later. For now we will just use `@autoid(HASH)`, which generates IDs based on the name of the member.

If we know we want to make `VehicleHealth` mutable beforehand, we can write this as the first version:

```
@topic
@mutable
@autoid(HASH)
struct VehicleHealth {
    @key string vin;
    short oil_level;
    sequence<double, 4> tire_pressure;
};
```

Mutable



When EVs are added to the fleet, but new ICE vehicles are also being added, the relevant fields can be added to the IDL in the same way as appendable:

```
@topic
@mutable
@autoid(HASH)
struct VehicleHealth {
    @key string vin;
    short oil_level;
    sequence<double, 4> tire_pressure;
    double battery_soc;
    sequence<double> cell_voltages;
};
```

The main difference between this and appendable is that if the values are the same as their default values, the fields may be omitted from the message on the wire and so it can be smaller.

Mutable



If at some point all ICE vehicle were removed from the fleet, we could remove the `oil_level` field altogether.

```
@topic
@mutable
@autoid(HASH)
struct VehicleHealth {
  @key string vin;
  sequence<double, 4> tire_pressure;
  double battery_soc;
  sequence<double> cell_voltages;
};
```

Existing EVs with `oil_level` in their IDL would continue to be compatible with the new IDL.

Key fields can't be modified in this way. Removing or modifying a key field will result in the types being incompatible.

Unions



Unions can also be annotated with extensibility annotations.

For example say we had a sequence of union for miscellaneous details about the status of a vehicle:

```
typedef float Location[3];
enum DetailType {OdometerDetail, LocationDetail};

@appendable
union Detail switch (DetailType) {
  case OdometerDetail:
    unsigned long miles;
  case LocationDetail:
    Location location;
};
```

Unions



If we wanted to know if an electric vehicle was charging and what the power level of the charger was, we could add that to IDL like so:

```
typedef float Location[3];
enum DetailType {OdometerDetail, LocationDetail, ChargingDetail};

@appendable
union Detail switch (DetailType) {
  case OdometerDetail:
    unsigned long miles;
  case LocationDetail:
    Location location;
  case ChargingDetail:
    float watts;
};
```

If we detected irregularities in the power level it could help diagnose issues with the vehicle.

Extensibility



In XTypes, structs and unions can differ from one participant to another while still being compatible with one another.

- How the structs and unions can or can't be modified depends on the extensibility IDL annotation applied.
 - **@appendable**
 - The default extensibility
 - New members can be added to structs and unions, but existing members can't be removed or reordered.
 - **@mutable**
 - Members can be added, removed, reordered, and possibly renamed while retaining compatibility with older IDL.
 - **@final**
 - Structs and unions should not be changed or else readers and writers will not be able to associate.



Try-Construct

Clayton Calabrese



Try-Construct Overview

- `@try_construct` is an annotation that defines how a reader recovers from an object construction failure, and then constructs a sample from a published sample of a different but assignable type
- 3 values for this annotation
 - `DISCARD`
 - `USE_DEFAULT`
 - `TRIM`
- Annotation goes on members of a type, unlike extensibility which goes on the type itself
- If a member exists in both the reader and writer and there is no `Try-Construct` annotation, `DISCARD` behavior is implied.
- If a member is added to the reader that does not exist in the writer, it gets the default value for that type.
- If a member exists on the writer that does not exist on the reader, the member is ignored for `Try-Construct`.
- Only the reader's `Try-Construct` annotation matters



Discard

Your rental company adds a gas pump, which communicates with cars to get their fuel types and then selects automatically for each customer.

This is set up using an enumerated type `FuelType`.

Over time, car technology improves and adds additional types of vehicles, such as electric.

New vehicles have an additional value in their enum: `electric`.

```
@mutable enum FuelType {Diesel, Premium,
Standard, Electric};

// New Vehicle Writer
@topic
@mutable
struct Vehicle {
    FuelType ft;
};

@mutable enum FuelType {Diesel, Premium,
Standard};

// Old Vehicle Writer + Old Gas Pump Reader
@topic
@mutable
struct Vehicle {
    @try_construct(DISCARD) FuelType ft;
};
```



Discard

Vehicle sends Gas Pump a sample containing the `FuelType: Electric`

Gas Pump receives `Electric`, which does not correspond to any values in `FuelType`, and the sample is rejected.

When do you want to utilize `DISCARD`?

- When a member is absolutely necessary for a sample to be understood
- When a member cannot be abbreviated or replaced without loss of crucial information

```
@mutable enum FuelType {Diesel, Premium, Standard, Electric};
```

```
// New Vehicle Writer  
@topic  
@mutable  
struct Vehicle {  
    FuelType ft;  
};
```

```
@mutable enum FuelType {Diesel, Premium, Standard};
```

```
// Old Vehicle Writer + Old Gas Pump Reader  
@topic  
@mutable  
struct Vehicle {  
    @try_construct(DISCARD) FuelType ft;  
};
```



Use Default

You have the same Old Gas Pump reader setup for the same scenario as before.

You don't like the idea of just discarding samples because even if customers don't buy your gasoline, you still want to advertise your company on the display.

Like before, car technology improves and electric vehicles emerge to become a large part of the market.

You prepare for this by having the first value in your enum be `Default`. The first value in an enumerated value list is the default; it will be used if an unknown value is received.

```
@mutable enum FuelType {Default, Diesel,  
Premium, Standard, Electric};
```

```
// New Vehicle Writer  
@topic  
@mutable  
struct Vehicle {  
    FuelType ft;  
};
```

```
@mutable enum FuelType {Default, Diesel,  
Premium, Standard};
```

```
// Old Vehicle Writer + Old Gas Pump Reader  
@topic  
@mutable  
struct Vehicle {  
    @try_construct(USE_DEFAULT) FuelType ft;  
};
```



Use Default

A new vehicle writer sends the gas pump a sample containing the `FuelType: Electric`

Gas pump receives `Electric`, which is not in its list of enum values. The sample then has the value received replaced with the default `Default` and is accepted.

Your code recognizes an unknown vehicle type has been detected and flashes your company logo on the gas pump display.

What are defaults for different types listed in XTypes specification, but generally corresponds to the 0 or empty value for a type.

When do you want to utilize `USE_DEFAULT`?

- When not all writers have all members the reader expects. Example would be supporting legacy devices that might not produce all the data a new version does.

```
@mutable enum FuelType {Default, Diesel,  
Premium, Standard, Electric};
```

```
// New Vehicle Writer
```

```
@topic  
@mutable  
struct Vehicle {  
    FuelType ft;  
};
```

```
@mutable enum FuelType {Default, Diesel,  
Premium, Standard};
```

```
// Old Vehicle Writer + Old Gas Pump Reader
```

```
@topic  
@mutable  
struct Vehicle {  
    @try_construct(USE_DEFAULT) FuelType ft;  
};
```

Trim

You have a system setup that periodically receives samples from each vehicle in your fleet and logs the sample to your database.

As your business has grown, the amount of data and samples you've collected over time has increased. You have thus become concerned about the amount of data you are saving and want a way to reduce stored sample size.

You realize that companies have unique model names that you can shorten to save space. You settle on 5 characters as the most you will need to uniquely identify a type of vehicle.



```
typedef string<5> model5;

// Vehicle Writer
@topic
@mutable
struct CarInfo {
    @key string vin;
    string make;
    string model_name;
};

// Database Reader
@topic
@mutable
struct CarInfo {
    @key string vin;
    string make;
    @try_construct(TRIM) model5 model_name;
};
```



Trim

Vehicle sends database reader a sample containing the make: "Toyota" and `model_name`: "Corolla"

Database reader receives this sample, but "Corolla" is too long for `model5`. It is shortened to the maximum size, 5: "Corol," and the sample is accepted.

Your code then adds this smaller sample into the database.

Trim applies only to Bounded Strings and Bounded Sequences.

When do you want to utilize **TRIM**?

- When you are trying to reduce the size of received samples
- When loss of data precision is unimportant

```
typedef string<5> model5;

// Vehicle Writer
@topic
@mutable
struct CarInfo {
    @key string vin;
    string make;
    string model_name;
};

// Database Reader
@topic
@mutable
struct CarInfo {
    @key string vin;
    string make;
    @try_construct(TRIM) model5 model_name;
};
```



Try-Construct Advanced Overview

- **Try-Construct** only occurs on object construction failures, so it will only be activated on:
 - bounded sequences
 - bounded strings
 - enumerated types
 - types with the above nested in them, such as structs, unions, arrays, and sequences.
- **Discard** takes precedence over **trim** and **use_default**. As long as one discard is triggered at the same level, the entire sample will be discarded.
- In nested types, a **discard** is considered an object construction failure at the level above, but **use_default** and **trim** are not.



Advanced XTypes

Son Dinh



Renaming Fields

XTypes supports renaming fields while keeping type compatibility between writer and reader

- Explicitly assign member IDs using **@id** or **@hashid** annotations
- Member names can be ignored during type compatibility check (to be added after 3.16)

Example

Older data type

```
@topic
@appendable
struct VehicleHealth {
  @id(1) @key string vin;
  @id(2) short oil_level;
  @id(3) sequence<double, 4> tire_pressure;
};
```

Newer data type

```
@topic
@appendable
struct VehicleHealth {
  @id(1) @key string identity;
  @id(2) short oil_level;
  @id(3) sequence<double, 4> tire_pressure;
  @id(4) double battery_soc;
  @id(5) sequence<double> cell_voltages;
};
```



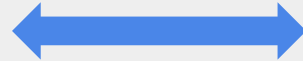


Compatibility With Non-XTypes

- Can be done using Final or Appendable extensibility kind together with a specific data representation (XCDR1)
 - Data is encoded similar to that of non-XTypes systems

Non-XTypes system

```
@topic
struct VehicleHealth {
  @key string vin;
  short oil_level;
  sequence<double, 4> tire_pressure;
};
```



XTypes system

```
@topic
@final
struct VehicleHealth {
  @key string vin;
  short oil_level;
  sequence<double, 4> tire_pressure;
};
```

- More details in OpenDDS Developer's Guide document



Under The Hood of XTypes

- TypeIdentifier & TypeObject
 - Objects describing the data types of the communicating peers
 - Used to perform type compatibility check between peers
- Type lookup service
 - Internal service to exchange TypeObjects between peers
 - Incorporated into the discovery process to get remote types for type compatibility check
 - If secure discovery is used, type lookup is also secured
- Data representations
 - Encoding format is determined by the extensibility kind of the data type and the encoding version being used (XCDR1 or XCDR2)



Wrap Up

Adam Mitz



What Else Is in XTypes?

- XTypes spec some features not yet implemented:
 - Dynamic Types API
Participant application doesn't need code generated from IDL
 - Additional annotations
Custom default values, optional fields
 - IDL version 4 extensions
Struct inheritance
- We welcome your involvement
 - GitHub issues
 - Code contributions
 - Sponsorship




Next Steps

- Download OpenDDS 3.16 (or newer)
 - github.com/objectcomputing/OpenDDS or opendds.org
 - Developer's Guide PDF (opendds.org/documents)
- "Introduction to OpenDDS Programming" online training March 3-4, 2021
 - objectcomputing.com/services/training/catalog/middleware/opendds-programming-cpp-and-java
- How can OCI help you with OpenDDS?
 - Custom training
 - Architecture and design consulting
 - Porting, extending, integrating
 - Performance and scalability analysis
 - Research and development
 - Ad-hoc support
- Contact us (next page) or visit
 - objectcomputing.com/products/opendds/opendds-consulting-and-support



THANK YOU!

LET'S CONNECT

 +1 (314) 579.0066

 info@objectcomputing.com

 objectcomputing.com