OpenDDS Developer's Guide

Table of Contents

Preface

What Is OpenDDS?

OpenDDS is an open source implementation of a group of related Object Management Group (OMG) specifications.

1) **Data Distribution Service (DDS) for Real-Time Systems v1.4** (OMG document formal/2015-04-10). This specification details the core functionality implemented by OpenDDS for real-time publish and subscribe applications and is described throughout this document.

2) **The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDSI-RTPS) v2.3** (OMG document formal/2019-04-03). Although the document number is v2.3, it specifies protocol version 2.4. This specification describes the requirements for interoperability between DDS implementations.

3) **DDS Security v1.1** (OMG document formal/2018-04-01) extends DDS with capabilities for authentication and encryption. OpenDDS's support for the DDS Security specification is described below in Chapter 14.

4) **Extensible and Dynamic Topic Types for DDS (XTypes) v1.3** (OMG document formal/2020-02-04) defines details of the type system used for the data exchanged on

DDS Topics, including how schema and data are encoded for network transmission. Details of DDS-XTypes are below in Chapter 16.

OpenDDS is sponsored by Object Computing, Inc. (OCI) and is available via https://www.opendds.org.

Licensing Terms

OpenDDS 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 OpenDDS software. For details of the licensing terms, see the file named `LICENSE` that is included in the OpenDDS source code distribution or visit http://www.opendds.org/license.html.

OpenDDS also utilizes other open source software products, including MPC (Make Project Creator), ACE (the ADAPTIVE Communication Environment), and TAO (The ACE ORB). More information about these products is available from OCI's web site at http://www.objectcomputing.com/products.

OpenDDS is open source and the development team welcomes contributions of code, tests, and ideas. Active participation by users ensures a robust implementation. Contact OCI if you are interested in contributing to the development of OpenDDS. Please note that any code that is contributed to and becomes part of the OpenDDS open source code base is subject to the same licensing terms as the rest of the OpenDDS code base.

About This Guide

This Developer's Guide corresponds to OpenDDS version 3.18. This guide is primarily focused on the specifics of using and configuring OpenDDS to build distributed publish-subscribe applications. While it does give a general overview of the OMG Data Distribution Service, this guide is not intended to provide comprehensive coverage of the specification. The intent of this guide is to help you become proficient with OpenDDS as quickly as possible.

Highlights of the 3.18 Release

NOTE: Numbers in parenthesis are GitHub pull requests

Additions:

- IDL4 explicitly-named integer types (#2814, #2857, #2862)

  - The existing IDL integer types, short, long, and long long, now have new aliases, which are int16, int32, and int64 respectively, along with unsigned versions of them like uint32.

  - There are also two new distinct integer types: int8 and uint8.

- RTPS:

  - Added the LeaseExtension RTPS discovery config option that extends the lease of discovered participants (#2869)

  - Added the SedpPassiveConnectDuration RTPS discovery config option (#2847)

  - Improved reassembly of fragments by adding an expiration to them (#2848)

    - Added the SedpFragmentReassemblyTimeout RTPS discovery config option to control this.

  - Use preallocated buffers in RtpsUdpDataLink (#2901)

    - Added the anticipated_fragments RTPS transport config option to scale how much memory is preallocated for packet customization and fragments buffers.

  - To try to reduce possible network congestion, temporarily increase the heartbeat send period while they go unanswered (#2911, #2913, #2915)

  - Added the `SendBufferSize` and `RecvBufferSize` RTPS discovery config options to set the buffer sizes for the sockets used for discovery (#2935)

- The QoS XML Handler can now take input from a string (#2439)

Fixes:

- RTPS:

  - Fixed RtpsUdpDataLink leaking remote writers on failed associations (#2812)

  - Fixed TypeObject encoding compatibly and consistency issues (#2749, #2804)

  - Fixed local/local RTPS associations not properly notifying liveliness (#2783)

  - Fixed durable data not being delivered in the right order (#2728)

  - RtpsUdpDataLink will no longer purge durable data (#2690)

- Fixed writers of fragmented data samples having issues reconnecting with readers (#2930)

- Fixed non-durable readers receiving historical samples (#2914)

- Fixed occasional deadlocks in DataReaderImpl and SubscriberImpl associated with presentation QoS (#2886, #2903)

- Fixed issues with security permissions and governance XML file parsing (#2896, #2912, #2916)

  - <id_range>s in domain id sets can now omit <max> in permissions and governance files.

    - This means <id_range><min>0</min></id_range> will now mean all domains.

  - Fix date/time in <not_before> and <not_after> in permissions grants always being interpreted as local time and fix issues with parsing time zone offsets.

  - Fixed settings for multiple <domain_rule>s in governance files ending up just being copies of the first domain rule.

  - Allow using 1/0 in addition to TRUE/FALSE and true/false for boolean values.

- RtpsRelay fixes and improvements (#2899, #2895, #2887, #2884, #2864, #2881, #2776, #2761, #2751, #2717, #2710, #2927, #2929, #2928, #2921)

- Fixed Topics returned from find_topic not being distinct entities (#2731)

- Try to workaround faulty DNS when getting the FQDN (#2677)

- Fixed configure --wireshark-cmake not working on Linux (#2679)

- Fixed an invalid memory write in SporadicTask (#2685)

- make install:

  - Use $(INSTALL_LIB) instead of hardcoded lib for Java-related files (#2740)

  - Use relative links in $(INSTALL_PREFIX)/share (#2780)

  - Fix headers unnecessarily getting installed to $(INSTALL_PREFIX)/include/DCPS (#2864)

Notes:

- Fixed a spelling error for the DCPSChunkAssociationMultiplier config option (#2723)

  - The old spelling, DCPSChunkAssociationMutltiplier (with an extra "t" in "Multiplier"), will continue to also be accepted for the time being.

ACE/TAO Version Compatibility

OpenDDS 3.18 is compatible with the current patch level of OCI TAO 2.2a (which includes ACE), as well as the current DOC Group micro release in the ACE 6.x / TAO 2.x series. See the `$DDS_ROOT/README.md` file for details.

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 OpenDDS source code distribution in the `$DDS_ROOT/DevGuideExamples/` directory. MPC files are provided with the examples for generating build-tool specific files, such as GNU Makefiles or Visual C++ project and solution files. A Perl script named `run_test.pl` is provided with each example so you can easily run it.

Related Documents

Throughout this guide, we refer to various specifications published by the Object Management Group (OMG) and from other sources.

OMG 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, the OMG DDS version 1.4 specification is referenced as `formal/2015-04-10`.

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/07-01-01` becomes http://www.omg.org/cgi-bin/doc?formal/07-01-01. Providing this destination to a web browser should take you to a site from which you can download the referenced specification document.

Additional documentation on OpenDDS is produced and maintained by Object Computing, Inc. and is available from the *OpenDDS Website* at https://www.opendds.org.

Here are some documents of interest and their locations:

Document	Location
Data Distribution Service (DDS) for Real-Time Systems v1.4 (OMG Document formal/2015-04-10)	http://www.omg.org/spec/DDS/1.4/PDF
The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDSI-RTPS) v2.3 (OMG Document formal/2019-04-03)	http://www.omg.org/spec/DDSI-RTPS/2.3/PDF
OMG Data Distribution Portal	http://portals.omg.org/dds/
OpenDDS Build Instructions, Architecture, and Doxygen Documentation	http://www.opendds.org/documentation.html
OpenDDS Frequently Asked Questions	http://www.opendds.org/faq.html

Supported Platforms

OCI regularly builds and tests OpenDDS on a wide variety of platforms, operating systems, and compilers. We continually update OpenDDS to support additional platforms. See the `$DDS_ROOT/README.md` file in the distribution 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 DDS.

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

Support offerings for OpenDDS include:

Consulting services to aid in the design of extensible, scalable, and robust publish-subscribe solutions, including the validation of domain-specific approaches, service selection, product customization and extension, and migrating your applications to OpenDDS from other publish-subscribe technologies and products.

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 OpenDDS-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 email `<sales@objectcomputing.com>` for more information.

OCI 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 (including DDS), 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.objectcomputing.com/training *for updates to our Educational Programs.*

On-Site or Remote Classes

We can provide the following courses at your company's facility or remotely, 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.objectcomputing.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 CORBA services 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++ or with Java

In this four-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 is intended for experienced C++ or Java developers.

OpenDDS Modeling Software Development Kit (SDK)

In this two-day course, developers and architects gain hands-on experience using the OpenDDS Modeling SDK to design and build publish/subscribe applications that use OpenDDS. The Eclipse-based, open source Modeling SDK enables developers to define an application's middleware components and data structures as a UML model, then generate the code to implement the model using OpenDDS. The generated code can then be compiled and linked with the application to provide seamless middleware support to the application.

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.

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

CHAPTER 1

Introduction

OpenDDS is an open source implementation of the OMG Data Distribution Service (DDS) for Real-Time Systems Specification v1.4 (OMG Document `formal/2015-04-10`) and the Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDSI-RTPS) v2.3 (OMG Document `formal/2019-04-03`). OpenDDS also implements the DDS Security Specification v1.1 (OMG Document `formal/2018-04-01`) and DDS XTypes v1.3 (OMG Document `formal/2020-02-04`). OpenDDS is sponsored by Object Computing, Inc. (OCI) and is available at <u>https://www.opendds.org/</u>. This developer's guide is based on the version 3.18 release of OpenDDS.

DDS defines a service for efficiently distributing application data between participants in a distributed application. This service is not tied to CORBA. The specification provides a Platform Independent Model (PIM) as well as a Platform Specific Model (PSM) that maps the PIM onto an OMG IDL implementation.

For additional details about DDS, developers should refer to the DDS specification (OMG Document `formal/2015-04-10`) as it contains in-depth coverage of all the service's features.

OpenDDS is the open-source C++ implementation of OMG's DDS specification developed and commercially supported by Object Computing, Inc. (OCI). It is available for download from <u>https://www.opendds.org/downloads.html</u> and is compatible with the latest patch level of OCI TAO version 2.2a (includes ACE), and the latest DOC Group release of ACE+TAO in the ACE 6.x / TAO 2.x series.

Note *OpenDDS currently implements the OMG DDS version 1.4 specification. See the compliance information in or at <u>https://www.opendds.org/</u> for more information.*

1.1.   DCPS Overview

In this section we introduce the main concepts and entities of the DCPS layer and discuss how they interact and work together.

1.1.1   Basic Concepts

Figure 1-1 shows an overview of the DDS DCPS layer. The following subsections define the concepts shown in this diagram.

Figure 1-1   DCPS Conceptual Overview
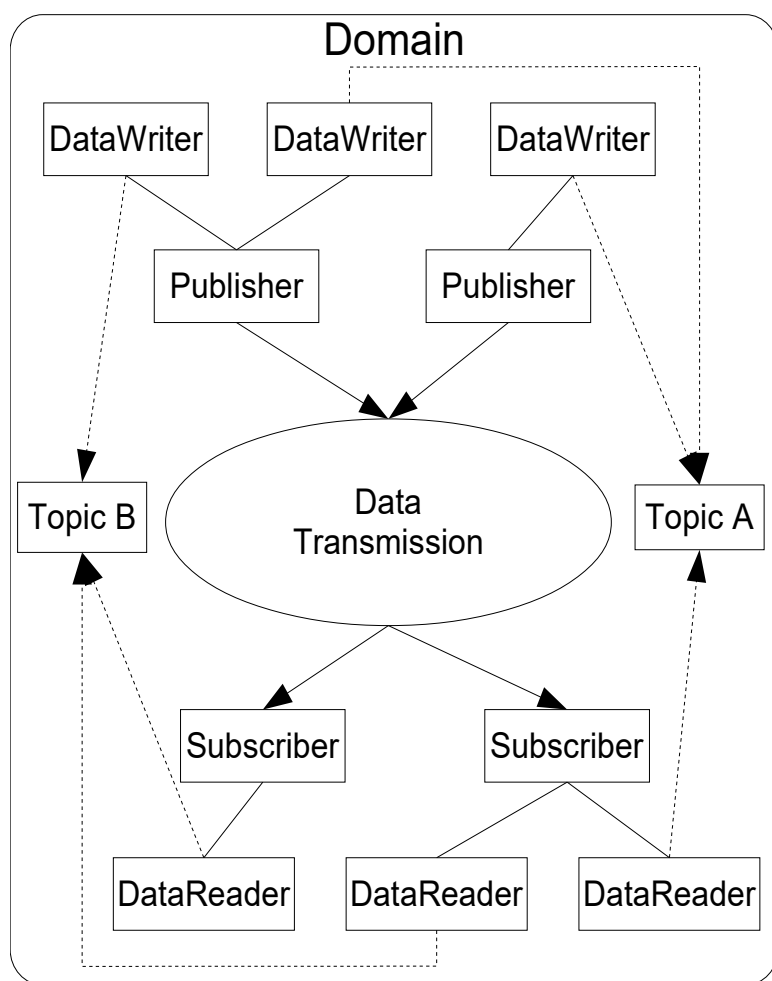
1.1.1.1   Domain

The *domain* is the fundamental partitioning unit within DCPS. Each of the other entities belongs to a domain and can only interact with other entities in that same domain.

OpenDDS Developer's Guide

Application code is free to interact with multiple domains but must do so via separate entities that belong to the different domains.

1.1.1.2 DomainParticipant

A *domain participant* is the entry-point for an application to interact within a particular domain. The domain participant is a factory for many of the objects involved in writing or reading data.

1.1.1.3 Topic

The *topic* is the fundamental means of interaction between publishing and subscribing applications. Each topic has a unique name within the domain and a specific data type that it publishes. Each topic data type can specify zero or more fields that make up its *key*. When publishing data, the publishing process always specifies the topic. Subscribers request data via the topic. In DCPS terminology you publish individual data *samples* for different *instances* on a topic. Each instance is associated with a unique value for the key. A publishing process publishes multiple data samples on the same instance by using the same key value for each sample.

1.1.1.4 DataWriter

The *data writer* is used by the publishing application code to pass values to the DDS. Each data writer is bound to a particular topic. The application uses the data writer's type-specific interface to publish samples on that topic. The data writer is responsible for marshaling the data and passing it to the publisher for transmission.

1.1.1.5 Publisher

The *publisher* is responsible for taking the published data and disseminating it to all relevant subscribers in the domain. The exact mechanism employed is left to the service implementation.

1.1.1.6 Subscriber

The *subscriber* receives the data from the publisher and passes it to any relevant data readers that are connected to it.

1.1.1.7 DataReader

The *data reader* takes data from the subscriber, demarshals it into the appropriate type for that topic, and delivers the sample to the application. Each data reader is bound to a particular topic. The application uses the data reader's type-specific interfaces to receive the samples.

1.1.2 Built-In Topics

The DDS specification defines a number of topics that are built-in to the DDS implementation. Subscribing to these *built-in topics* gives application developers access to the state of the domain being used including which topics are registered, which data readers and data writers are connected and disconnected, and the QoS settings of the various entities. While subscribed, the application receives samples indicating changes in the entities within the domain.

The following table shows the built-in topics defined within the DDS specification:

Table 1-1 Built-in Topics

Topic Name	Description
`DCPSParticipant`	Each instance represents a domain participant.
`DCPSTopic`	Each instance represents a normal (not built-in) topic.
`DCPSPublication`	Each instance represents a data writer.
`DCPSSubscription`	Each instance represents a data reader.

1.1.3 Quality Of Service Policies

The DDS specification defines a number of Quality of Service (QoS) policies that are used by applications to specify their QoS requirements to the service. Participants specify what behavior they require from the service and the service decides how to achieve these behaviors. These policies can be applied to the various DCPS entities (topic, data writer, data reader, publisher, subscriber, domain participant) although not all policies are valid for all types of entities.

Subscribers and publishers are matched using a request-versus-offered (RxO) model. Subscribers *request* a set of policies that are minimally required. Publishers *offer* a set of QoS policies to potential subscribers. The DDS implementation then attempts to match the requested policies with the offered policies; if these policies are compatible then the association is formed.

The QoS policies currently implemented by OpenDDS are discussed in detail in Chapter 3.

1.1.4   **Listeners**

The DCPS layer defines a callback interface for each entity that allows an application processes to "listen" for certain state changes or events pertaining to that entity. For example, a Data Reader Listener is notified when there are data values available for reading.

1.1.5   **Conditions**

Conditions and *Wait Sets* allow an alternative to listeners in detecting events of interest in DDS. The general pattern is

The application creates a specific kind of `Condition` object, such as a `StatusCondition`, and attaches it to a `WaitSet`.

- The application waits on the `WaitSet` until one or more conditions become true.

- The application calls operations on the corresponding entity objects to extract the necessary information.

- The `DataReader` interface also has operations that take a `ReadCondition` argument.

- `QueryCondition` objects are provided as part of the implementation of the Content-Subscription Profile. The `QueryCondition` interface extends the `ReadCondition` interface.

1.2   **OpenDDS Implementation**

1.2.1   **Compliance**

OpenDDS complies with the OMG DDS and the OMG DDSI-RTPS specifications. Details of that compliance follows here. OpenDDS also implements the OMG DDS Security specification. Details of compliance to that specification are in section 14.8. Details of XTypes compliance are in sections 16.7 and 16.8.

1.2.1.1   **DDS Compliance**

Section 2 of the DDS specification defines five compliance points for a DDS implementation:

1) Minimum Profile

2) Content-Subscription Profile

3) Persistence Profile

4) Ownership Profile

5) Object Model Profile

OpenDDS complies with the entire DDS specification (including all optional profiles). This includes the implementation of all Quality of Service policies with the following notes:

- RELIABILITY.kind = RELIABLE is supported by the RTPS_UDP transport, the TCP transport, or the IP Multicast transport (when configured as reliable).

- TRANSPORT_PRIORITY is not implemented as changeable.

Although version 1.5 of the DDS specification is not yet published, OpenDDS incorporates some changes planned for that version that are required for a robust implementation:

- DDS15-257: The IDL type BuiltinTopicKey_t is a struct containing an array of 16 octets

1.2.1.2 DDSI-RTPS Compliance

The OpenDDS implementation complies with the requirements of the OMG DDSI-RTPS specification.

OpenDDS RTPS Implementation Notes

The OMG DDSI-RTPS specification (formal/2019-04-03) supplies statements for implementation, but not required for compliance. The following items should be taken into consideration when utilizing the OpenDDS RTPS functionality for transport and/or discovery. Section numbers of the DDSI-RTPS specification are supplied with each item for further reference.

Items not implemented in OpenDDS:

1) Writer-side content filtering (8.7.3)
   OpenDDS may still drop samples that aren't needed (due to content filtering) by any associated readers — this is done above the transport layer

2) Coherent sets for `PRESENTATION` QoS (8.7.5)

3) Directed writes (8.7.6)

   – OpenDDS will use the Directed Write parameter if it's present on incoming messages (for example, messages generated by a different DDS implementation)

4) Property lists (8.7.7)

5) Original writer info for `DURABLE` data (8.7.8) -- this would only be used for transient and persistent durability, which are not supported by the RTPS specification (8.7.2.2.1)

6) Key Hashes (8.7.9) are not generated, but they are optional

7) `nackSuppressionDuration` (Table 8.47) and `heartbeatSuppressionDuration` (Table 8.62).

Note *Items 3 and 4 above are described in the DDSI-RTPS specification. However, they do not have a corresponding concept in the DDS specification.*

1.2.1.3 IDL Compliance

OMG IDL is used in a few different ways in the OpenDDS code base and downstream applications that use it:

- Files that come with OpenDDS such as `dds/DdsDcpsTopic.idl` define parts of the API between the middleware libraries and the application. This is known as the OMG IDL Platform Specific Model (PSM).

- Users of OpenDDS author IDL files in addition to source code files in C++ or Java.

This section only describes the latter use.

The IDL specification (version 4.2) uses the term "building block" to define subsets of the overall IDL grammar that may be supported by certain tools. OpenDDS supports the following building blocks, with notes/caveats listed below each:

- Core Data Types

  ◦ Support for the "fixed" data type (fixed point decimal) is incomplete.

- Anonymous Types

  ◦ There is limited support for anonymous types when they appear as sequence/array instantiations directly as struct field types. Using an explicitly-named type is recommended.

- Annotations

  ◦ See sections 2.1.1 and 16.6 for details on which built-in annotations are supported.

  ◦ User-defined annotation types are also supported.

- Extended Data Types

- ○ The integer types int8, int16, int32, and int64 along with unsigned versions of them like uint32, are supported.

- ○ The rest of the building block is not supported.

1.2.2 Extensions To The DDS Specification

Data types, interfaces, and constants in the **DDS** IDL module (C++ namespace, Java package) correspond directly to the DDS specification with very few exceptions:

- DDS::SampleInfo contains an extra field starting with "opendds_reserved"

- Type-specific DataReaders (including those for Built-in Topics) have additional operations read_instance_w_condition() and take_instance_w_condition().

Additional extended behavior is provided by various classes and interfaces in the OpenDDS module/namespace/package. Those include features like Recorder and Replayer (see chapter 12) and also:

- OpenDDS::DCPS::TypeSupport adds the unregister_type() operation not found in the DDS spec.

- OpenDDS::DCPS::ALL_STATUS_MASK, NO_STATUS_MASK, and DEFAULT_STATUS_MASK are useful constants for the DDS::StatusMask type used by DDS::Entity, DDS::StatusCondition, and the various create_*() operations.

1.2.3 OpenDDS Architecture

This section gives a brief overview of the OpenDDS implementation, its features, and some of its components. The `$DDS_ROOT` environment variable should point to the base directory of the OpenDDS distribution. Source code for OpenDDS can be found under the `$DDS_ROOT/dds/` directory. DDS tests can be found under `$DDS_ROOT/tests/`.

1.2.3.1 Design Philosophy

The OpenDDS implementation and API is based on a fairly strict interpretation of the OMG IDL PSM. In almost all cases the OMG's IDL-to-C++ Language Mapping is used to define how the IDL in the DDS specification is mapped into the C++ APIs that OpenDDS exposes to the client.

The main deviation from the OMG IDL PSM is that local interfaces are used for the entities and various other interfaces. These are defined as unconstrained (non-local) interfaces in the DDS specification. Defining them as local interfaces improves performance, reduces

memory usage, simplifies the client's interaction with these interfaces, and makes it easier for clients to build their own implementations.

1.2.3.2 Extensible Transport Framework (ETF)

OpenDDS uses the IDL interfaces defined by the DDS specification to initialize and control service usage. Data transmission is accomplished via an OpenDDS-specific transport framework that allows the service to be used with a variety of transport protocols. This is referred to as *pluggable transports* and makes the extensibility of OpenDDS an important part of its architecture. OpenDDS currently supports TCP/IP, UDP/IP, IP multicast, shared-memory, and RTPS_UDP transport protocols as shown in Figure 1-2. Transports are typically specified via configuration files and are attached to various entities in the publisher and subscriber processes. Refer to Section 7.4.4 for details on configuring ETF components.
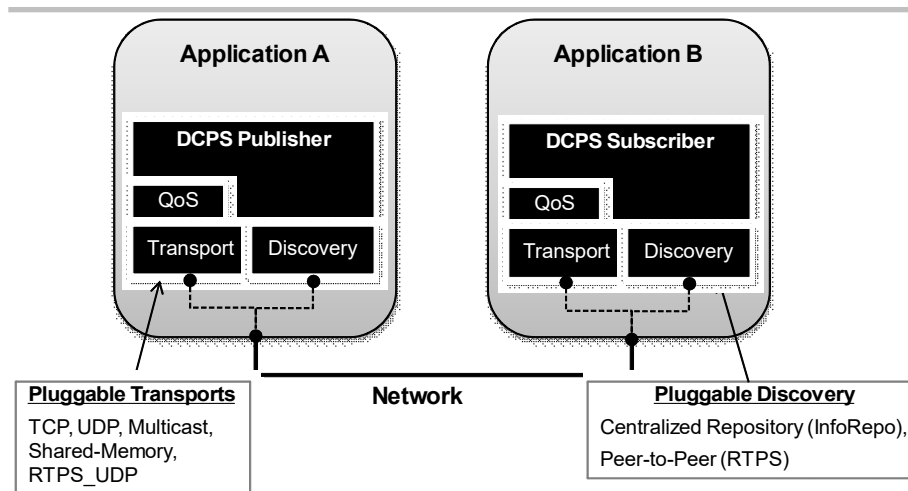
Figure 1-2 OpenDDS Extensible Transport Framework

The ETF enables application developers to implement their own customized transports. Implementing a custom transport involves specializing a number of classes defined in the transport framework. The udp transport provides a good foundation developers may use when creating their own implementation. See the $DDS_ROOT/dds/DCPS/transport/udp/ directory for details.

1.2.3.3 DDS Discovery

DDS applications must discover one another via some central agent or through some distributed scheme. An important feature of OpenDDS is that DDS applications can be configured to perform discovery using the DCPSInfoRepo or RTPS discovery, but utilize a different transport type for data transfer between data writers and data readers. The OMG

DDS specification (`formal/2015-04-10`) leaves the details of discovery to the implementation. In the case of interoperability between DDS implementations, the OMG DDSI-RTPS (`formal/2014-09-01`) specification provides requirements for a peer-to-peer style of discovery.

OpenDDS provides two options for discovery.

1) Information Repository: a centralized repository style that runs as a separate process allowing publishers and subscribers to discover one another centrally or

2) RTPS Discovery: a peer-to-peer style of discovery that utilizes the RTPS protocol to advertise availability and location information.

Interoperability with other DDS implementations must utilize the peer-to-peer method, but can be useful in OpenDDS-only deployments.

Centralized Discovery with DCPSInfoRepo

OpenDDS implements a standalone service called the DCPS Information Repository (DCPSInfoRepo) to achieve the centralized discovery method. It is implemented as a CORBA server. When a client requests a subscription for a topic, the DCPS Information Repository locates the topic and notifies any existing publishers of the location of the new subscriber. The DCPSInfoRepo process needs to be running whenever OpenDDS is being used in a non-RTPS configuration. An RTPS configuration does not use the DCPSInfoRepo. The DCPSInfoRepo is not involved in data propagation, its role is limited in scope to OpenDDS applications discovering one another.
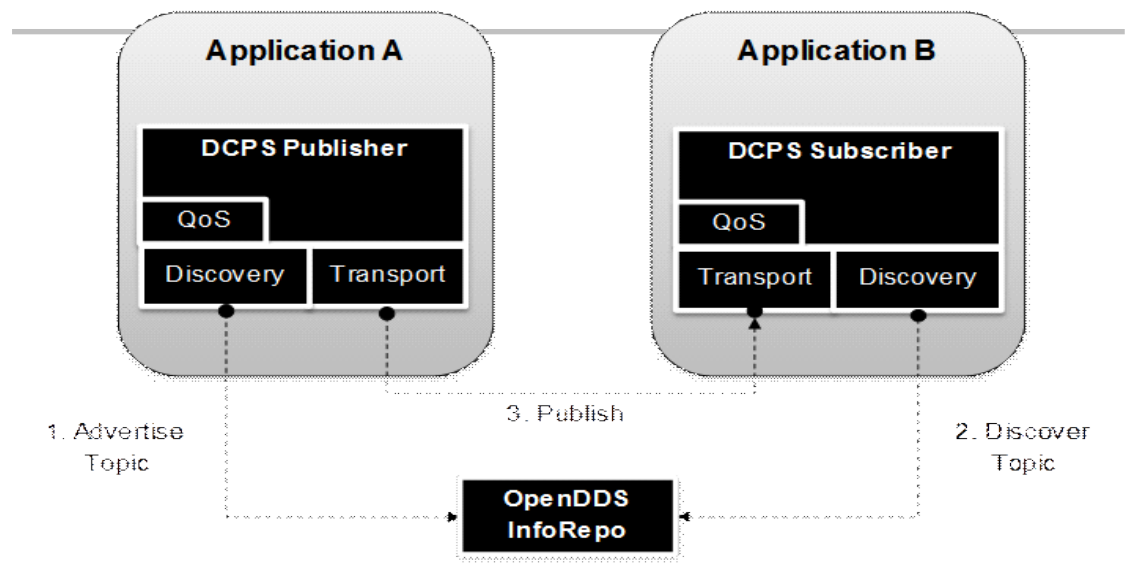
Figure 1-3: Centralized Discovery with OpenDDS InfoRepo

Application developers are free to run multiple information repositories with each managing their own non-overlapping sets of DCPS domains.

It is also possible to operate domains with more than a single repository, thus forming a distributed virtual repository. This is known as *Repository Federation*. In order for individual repositories to participate in a federation, each one must specify its own federation identifier value (a 32-bit numeric value) upon start-up. See 9.2 for further information about repository federations.

Peer-to-Peer Discovery with RTPS

DDS applications requiring a Peer-to-Peer discovery pattern can be accommodated by OpenDDS capabilities. This style of discovery is accomplished only through the use of the RTPS protocol as of the current release. This simple form of discovery is accomplished through simple configuration of DDS application data readers and data writers running in application processes as shown in Figure 1-4. As each participating process activates the DDSI-RTPS discovery mechanisms in OpenDDS for their data readers and writers, network endpoints are created with either default or configured network ports such that DDS participants can begin advertising the availability of their data readers and data writers. After a period of time, those seeking one another based on criteria will find each other and establish a connection based on the configured pluggable transport as discussed in

Extensible Transport Framework (ETF). A more detailed description of this flexible configuration approach is discussed in Section 7.4.1.1 and Section 7.4.5.5.
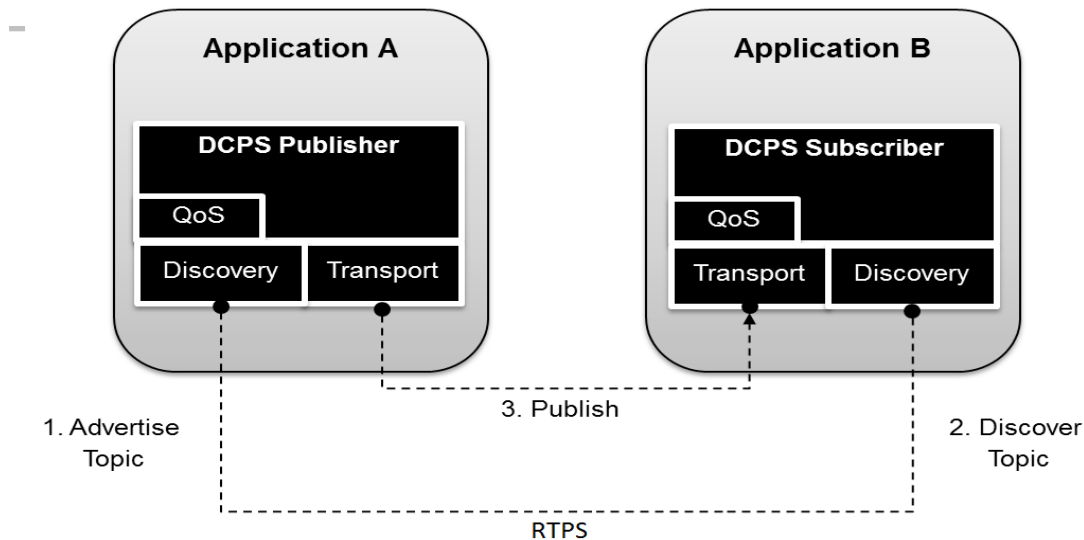
Figure 1-4: Peer-to-peer Discovery with RTPS

The following are additional implementation limits that developers need to take into consideration when developing and deploying applications that use RTPS discovery:

1) Domain IDs should be between 0 and 231 (inclusive) due to the way UDP ports are assigned to domain IDs. In each OpenDDS process, up to 120 domain participants are supported in each domain.

2) Topic names and type identifiers are limited to 256 characters.

3) OpenDDS's native multicast transport does not work with RTPS Discovery due to the way GUIDs are assigned (a warning will be issued if this is attempted).

For more details in how RTPS discovery occurs, a very good reference to read can be found in Section 8.5 of the Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDSI-RTPS) v2.2 (OMG Document `formal/2014-09-01`).

1.2.3.4 Threading

OpenDDS creates its own ORB (when one is required) as well as a separate thread upon which to run that ORB. It also uses its own threads to process incoming and outgoing transport I/O. A separate thread is created to cleanup resources upon unexpected

connection closure. Your application may get called back from these threads via the Listener mechanism of DCPS.

When publishing a sample via DDS, OpenDDS normally attempts to send the sample to any connected subscribers using the calling thread. If the send call blocks, then the sample may be queued for sending on a separate service thread. This behavior depends on the QoS policies described in Chapter 3.

All incoming data in the subscriber is read by a service thread and queued for reading by the application. DataReader listeners are called from the service thread.

1.2.3.5 Configuration

OpenDDS includes a file-based configuration framework for configuring both global items such as debug level, memory allocation, and discovery, as well as transport implementation details for publishers and subscribers. Configuration can also be achieved directly in code, however, it is recommended that configuration be externalized for ease of maintenance and reduction in runtime errors. The complete set of configuration options are described in Chapter 7.

1.3 Installation

The steps on how to build OpenDDS can be found in `$DDS_ROOT/INSTALL.md`.

To build OpenDDS with DDS Security, see section 14.1 below.

To avoid compiling OpenDDS code that you will not be using, there are certain features than can be excluded from being built. The features are discussed below.

Users requiring a small-footprint configuration or compatibility with safety-oriented platforms should consider using the OpenDDS Safety Profile, which is described in chapter 13 of this guide.

1.3.1 Building With A Feature Enabled Or Disabled

Most features are supported by the *configure* script. The *configure* script creates config files with the correct content and then runs MPC. If you are using the *configure* script, run it with the "--help" command line option and look for the feature you wish to enable/disable. If you are not using the *configure* script, continue reading below for instructions on running MPC directly.

For the features described below, MPC is used for enabling (the default) a feature or disabling the feature. For a feature named *feature*, the following steps are used to disable the feature from the build:

1) Use the command line "features" argument to MPC:

   `mwc.pl -type <type> -features feature=0 DDS.mwc`

   Or alternatively, add the line `feature=0` to the file `$ACE_ROOT/bin/MakeProjectCreator/config/default.features` and regenerate the project files using MPC.

2) If you are using the `gnuace` MPC project type (which is the case if you will be using GNU make as your build system), add line "`feature=0`" to the file `$ACE_ROOT/include/makeinclude/platform_macros.GNU`.

To explicitly enable the feature, use `feature=1` above.

Note	*You can also use the* `$DDS_ROOT/configure` *script to enable or disable features. To disable the feature, pass* `--no-feature` *to the script, to enable pass* `--feature`*. In this case '-' is used instead of '_' in the feature name. For example, to disable feature* `content_subscription` *discussed below, pass* `--no-content-subscription` *to the configure script.*

1.3.2 Disabling The Building Of Built-In Topic Support

Feature Name: `built_in_topics`

You can reduce the footprint of the core DDS library by up to 30% by disabling Built-in Topic Support. See Chapter 6 for a description of Built-In Topics.

1.3.3 Disabling The Building Of Compliance Profile Features

The DDS specification defines *compliance profiles* to provide a common terminology for indicating certain feature sets that a DDS implementation may or may not support. These profiles are given below, along with the name of the MPC feature to use to disable support for that profile or components of that profile.

Many of the profile options involve QoS settings. If you attempt to use a QoS value that is incompatible with a disabled profile, a runtime error will occur. If a profile involves a class, a compile time error will occur if you try to use the class and the profile is disabled.

1.3.3.1 Content-Subscription Profile

Feature Name: `content_subscription`

This profile adds the classes `ContentFilteredTopic`, `QueryCondition`, and `MultiTopic` discussed in Chapter 5.

In addition, individual classes can be excluded by using the features given in the table below.

Table 1-2: Content-Subscription Class Features

Class	Feature
ContentFilteredTopic	`content_filtered_topic`
QueryCondition	`query_condition`
MultiTopic	`multi_topic`

1.3.3.2 Persistence Profile

Feature Name: `persistence_profile`

This profile adds the QoS policy `DURABILITY_SERVICE` and the settings 'TRANSIENT' and 'PERSISTENT' of the `DURABILITY` QoS policy `kind`.

1.3.3.3 Ownership Profile

Feature Name: `ownership_profile`

This profile adds:

• the setting 'EXCLUSIVE' of the `OWNERSHIP` kind

• support for the `OWNERSHIP_STRENGTH` policy

• setting a `depth > 1` for the `HISTORY` QoS policy.

Note *Some users may wish to exclude support for the Exclusive OWNERSHIP policy and its associated OWNERSHIP_STRENGTH without impacting use of HISTORY. In order to support this configuration, OpenDDS also has the MPC feature ownership_kind_exclusive (configure script option --no-ownership-kind-exclusive).*

1.3.3.4 Object Model Profile

Feature Name: `object_model_profile`

This profile includes support for the `PRESENTATION` access_scope setting of 'GROUP'.

Note *Currently, the* `PRESENTATION` *access_scope of* 'TOPIC' *is also excluded when* `object_model_profile` *is disabled.*

1.4 Building Applications that use OpenDDS

This section applies to any C++ code that directly or indirectly includes OpenDDS headers. For Java applications, see Chapter 10 below.

C++ source code that includes OpenDDS headers can be built using either build system: MPC or CMake.

1.4.1 MPC: The Makefile, Project, And Workspace Creator

OpenDDS is itself built with MPC, so development systems that are set up to use OpenDDS already have MPC available. The OpenDDS configure script creates a "setenv" script with environment settings (`setenv.cmd` on Windows; `setenv.sh` on Linux/macOS). This environment contains the `PATH` and `MPC_ROOT` settings necessary to use MPC.

MPC's source tree (in `MPC_ROOT`) contains a "docs" directory with both HTML and plain text documentation (`USAGE` and `README` files).

The example walk-through in section 2.1 uses MPC as its build system. The OpenDDS source tree contains many tests and examples that are built with MPC. These can be used as starting points for application MPC files.

1.4.2 CMake

Applications can also be built with CMake (https://cmake.org). See the included documentation in the OpenDDS source tree: `docs/cmake.md`

The OpenDDS source tree also includes examples of using CMake. They are listed in the `cmake.md` file.

1.4.3   **Custom Build Systems**

Users of OpenDDS are strongly encouraged to select one of the two options listed above (MPC or CMake) to generate consistent build files on any supported platform. If this is not possible, users of OpenDDS must make sure that all code generator, compiler, and linker settings in the custom build setup result in API- and ABI-compatible code. To do this, start with an MPC or CMake-generated project file (makefile or Visual Studio project file) and make sure all relevant settings are represented in the custom build system. This is often done through a combination of inspecting the project file and running the build with verbose output to see how the toolchain (code generators, compiler, linker) is invoked. Contact Object Computing, Inc. (OCI) via https://objectcomputing.com/products/opendds/consulting-support to have our expert software engineers work on this for you.

CHAPTER 2

Getting Started

2.1 Using DCPS

This chapter focuses on an example application using DCPS to distribute data from a single publisher process to a single subscriber process. It is based on a simple messenger application where a single publisher publishes messages and a single subscriber subscribes to them. We use the default QoS properties and the default TCP/IP transport. Full source code for this example may be found under the `$DDS_ROOT/DevGuideExamples/DCPS/Messenger/` directory. Additional DDS and DCPS features are discussed in later chapters.

2.1.1 Defining Data Types With IDL

2.1.1.1 Identifying Topic Types

Each data type used by DDS is defined using OMG Interface Definition Language (IDL). OpenDDS uses IDL annotations[1] to identify the data types that it transmits and processes. These data types are processed by the TAO IDL compiler and the OpenDDS IDL compiler to generate the necessary code to transmit data of these types with OpenDDS. Here is the IDL file that defines our Message data type:

```
module Messenger {
```

---

[1] For backwards compatibility, OpenDDS also parses `#pragma` directives which were used before release 3.14. This guide will describe IDL annotations only.

```
  @topic
  struct Message {
    string from;
    string subject;
    @key long subject_id;
    string text;
    long count;
  };
};
```

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 structure or union may contain basic types (short, long, float, etc.), enumerations, strings, sequences, arrays, structures, and unions. See section 1.2.1.3 for more details on the use of IDL for OpenDDS topic types. The IDL above defines the structure Message in the Messenger module for use in this example.

2.1.1.2 Keys

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. Keys can be of scalar type, structures or unions containing key fields, or arrays of any of these constructs.

Multiple keys are specified with separate `@key` annotations. In the above example, we identify the `subject_id` member of `Messenger::Message` as a key. Each sample published with a unique `subject_id` value will be defined as belonging to a different DDS Instance within the same topic. Since we are using the default QoS policies, subsequent samples with the same subject_id value are treated as replacement values for that DDS Instance.

`@key` can be applied to a structure field of the following types:

- Any primitive, such as booleans, integers, characters, and strings.

- Other structures that have a defined key or set of keys. For example:

```
struct StructA {
  @key long key;
};

struct StructB {
  @key StructA main_info;
  long other_info;
};

@topic
struct StructC {
  @key StructA keya; // keya.key is one key
  @key StructB keyb; // keyb.main_info.key is another
  DDS::OctetSeq data;
};
```

In this example, every type from the key marked on the topic type down to what

primitive data types to use as the key is annotated with @key. That isn't strictly necessary though, as the next section shows.

- Other structures that don't have any defined keys. In the following example, it's implied that all the fields in InnerStruct are keys.

```
struct InnerStruct {
  long a;
  short b;
  char c;
};

@topic
struct OuterStruct {
  @key InnerStruct value;
                          // value.a, value.b, and value.c are all keys
};
```

If none of the fields in a struct are marked with @key or @key(TRUE), then when the struct is used in another struct and marked as a key, all the fields in the struct are assumed to keys. Fields marked with @key(FALSE) are always excluded from being a key, such as in this example:

```
struct InnerStruct {
  long a;
  short b;
  @key(FALSE) char c;
};

@topic
struct OuterStruct {
  @key InnerStruct value;
                          // Now just value.a and value.b are the keys
};
```

- Unions can also be used as keys if their discriminator is marked as a key. There is an example of a keyed union topic type in the next section, but keep in mind a union being used as a key doesn't have to be a topic type.

- Arrays of any of the previous data types. @key can't be applied to sequences, even if the base type would be valid in an array. Also @key, when applied to arrays, it makes every element in the array part of the key. They can't be applied to individual array elements.

2.1.1.3 Union Topic Types

Unions can be used as topic types. Here is an example:

```
enum TypeKind {
  STRING_TYPE,
  LONG_TYPE,
  FLOAT_TYPE
};

@topic
```

```
union MyUnionType switch (@key TypeKind) {
case STRING_TYPE:
  string string_value;
case LONG_TYPE:
  long long_value;
case FLOAT_TYPE:
  float float_value;
};
```

Unions can be keyed like structures, but only the union discriminator can be a key, so the set of possible DDS Instances of topics using keyed unions are values of the discriminator. Designating a key for a union topic type is done by putting @key before the discriminator type like in the example above. Like structures, it is also possible to have no key fields, in which case @key would be omitted and there would be only one DDS Instance.

2.1.1.4   Topic Types vs. Nested Types

In addition to @topic, the set of IDL types OpenDDS can use can also be controlled using @nested and @default_nested. Types that are "nested" are the opposite of topic types; they can't be used for the top-level type of a topic, but they can be nested inside the top-level type (at any level of nesting). All types are nested by default in OpenDDS to reduce the code generated for type support, but there a number of ways to change this:

- The type can be annotated with @topic (see section 2.1.1.1), or with @nested(FALSE), which is equivalent to @topic.

- The enclosing module can be annotated with @default_nested(FALSE).

- The global default for opendds_idl can be changed by adding --no-default-nested, in which case it would be as if all valid types were marked with @topic. If desired for IDL compatibility with other DDS implementations or based on preference, this can be done through the build system:

  - When using MPC, add dcps_ts_flags += --no-default-nested to the project.

  - When using CMake, this can be done by setting either the OPENDDS_DEFAULT_NESTED global variable to FALSE or adding --no-default-nested to the OPENDDS_IDL_OPTIONS parameter for OPENDDS_TARGET_SOURCES.
    See docs/cmake.md in the source for more information about using OpenDDS with CMake.

In cases where the module default is not nested, you can reverse this by using @nested or @nested(TRUE) for structures/unions and @default_nested or @default_nested(TRUE) for

modules. NOTE: the @`topic` annotation doesn't take a boolean argument, so *@topic(FALSE)* would cause an error in the OpenDDS IDL Compiler.

2.1.2 Processing The IDL

This section uses the OMG IDL-to-C++ mapping ("C++ classic") as part of the walk-through. OpenDDS also supports the OMG IDL-to-C++11 mapping, see section 8.2 for details.

The OpenDDS IDL is first processed by the TAO IDL compiler.

tao_idl *Messenger.idl*

In addition, we need to process the IDL file with the OpenDDS IDL compiler to generate the serialization and key support code that OpenDDS requires to marshal and demarshal the Message, as well as the type support code for the data readers and writers. This IDL compiler is located in `$DDS_ROOT/bin` and generates three files for each IDL file processed. The three files all begin with the original IDL file name and would appear as follows:

- *<filename>TypeSupport.idl*

- *<filename>TypeSupportImpl.h*

- *<filename>TypeSupportImpl.cpp*

For example, running `opendds_idl` as follows

opendds_idl *Messenger.idl*

generates *MessengerTypeSupport.idl*, *MessengerTypeSupportImpl.h*, and *MessengerTypeSupportImpl.cpp*. The IDL file contains the `MessageTypeSupport`, `MessageDataWriter`, and `MessageDataReader` interface definitions. These are type-specific DDS interfaces that we use later to register our data type with the domain, publish samples of that data type, and receive published samples. The implementation files contain implementations for these interfaces. The generated IDL file should itself be compiled with the TAO IDL compiler to generate stubs and skeletons. These and the implementation file should be linked with your OpenDDS applications that use the Message type. The OpenDDS IDL compiler has a number of options that specialize the generated code. These options are described in Chapter 8.

Typically, you do not directly invoke the TAO or OpenDDS IDL compilers as above, but let your build system do it for you. Two different build systems are supported for projects that use OpenDDS:

- MPC, the "Make Project Creator" which is used to build OpenDDS itself and the majority of its included tests and examples

- CMake, a build system that's commonly used across the industry (cmake.org)

Even if you will eventually use some custom build system that's not one of the two listed above, start by building an example OpenDDS application using one of the supported build systems and then migrate the code generator command lines, compiler options, etc., to the custom build system.

The remainder of this section will assume MPC. For more details on using CMake, see the included documentation in the OpenDDS repository: `docs/cmake.md`

The code generation process is simplified when using MPC, by inheriting from the dcps base project. Here is the MPC file section common to both the publisher and subscriber

```
project(*idl): dcps {
  // This project ensures the common components get built first.

  TypeSupport_Files {
    Messenger.idl
  }
  custom_only = 1
}
```

The dcps parent project adds the Type Support custom build rules. The TypeSupport_Files section above tells MPC to generate the Message type support files from `Messenger.idl` using the OpenDDS IDL complier. Here is the publisher section:

```
project(*Publisher): dcpsexe_with_tcp {
  exename   = publisher
  after    += *idl

  TypeSupport_Files {
    Messenger.idl
  }

  Source_Files {
    Publisher.cpp
  }
}
```

The `dcpsexe_with_tcp` project links in the DCPS library.

For completeness, here is the subscriber section of the MPC file:

```
project(*Subscriber): dcpsexe_with_tcp {

  exename   = subscriber
  after    += *idl

  TypeSupport_Files {
    Messenger.idl
  }

  Source_Files {
    Subscriber.cpp
```

```
    DataReaderListenerImpl.cpp
  }
}
```

2.1.3 A Simple Message Publisher

In this section we describe the steps involved in setting up a simple OpenDDS publication process. The code is broken into logical sections and explained as we present each section. We omit some uninteresting sections of the code (such as #include directives, error handling, and cross-process synchronization). The full source code for this sample publisher is found in the *Publisher.cpp* and *Writer.cpp* files in $DDS_ROOT/DevGuideExamples/DCPS/Messenger/.

2.1.3.1 Initializing the Participant

The first section of main() initializes the current process as an OpenDDS participant.

```
int main (int argc, char *argv[]) {
  try {
    DDS::DomainParticipantFactory_var dpf =
      TheParticipantFactoryWithArgs(argc, argv);
    DDS::DomainParticipant_var participant =
      dpf->create_participant(42, // domain ID
                              PARTICIPANT_QOS_DEFAULT,
                              0,  // No listener required
                              OpenDDS::DCPS::DEFAULT_STATUS_MASK);
    if (!participant) {
      std::cerr << "create_participant failed." << std::endl;
      return 1;
    }
```

The TheParticipantFactoryWithArgs macro is defined in Service_Participant.h and initializes the Domain Participant Factory with the command line arguments. These command line arguments are used to initialize the ORB that the OpenDDS service uses as well as the service itself. This allows us to pass ORB_init() options on the command line as well as OpenDDS configuration options of the form -DCPS*. Available OpenDDS options are fully described in Chapter 7.

The create_participant() operation uses the domain participant factory to register this process as a participant in the domain specified by the ID of 42. The participant uses the default QoS policies and no listeners. Use of the OpenDDS default status mask ensures all relevant communication status changes (e.g., data available, liveliness lost) in the middleware are communicated to the application (e.g., via callbacks on listeners).

Users may define any number of domains using IDs in the range (0x0 ~ 0x7FFFFFFF). All other values are reserved for internal use by the implementation.

OpenDDS Developer's Guide

The Domain Participant object reference returned is then used to register our Message data type.

2.1.3.2   Registering the Data Type and Creating a Topic

First, we create a `MessageTypeSupportImpl` object, then register the type with a type name using the `register_type()` operation. In this example, we register the type with a nil string type name, which causes the `MessageTypeSupport` interface repository identifier to be used as the type name. A specific type name such as "*Message*" can be used as well.

```
Messenger::MessageTypeSupport_var mts =
  new Messenger::MessageTypeSupportImpl();
if (DDS::RETCODE_OK != mts->register_type(participant, "")) {
  std::cerr << "register_type failed." << std::endl;
  return 1;
}
```

Next, we obtain the registered type name from the type support object and create the topic by passing the type name to the participant in the `create_topic()` operation.

```
 CORBA::String_var type_name = mts->get_type_name ();

  DDS::Topic_var topic =
    participant->create_topic ("Movie Discussion List",
                               type_name,
                               TOPIC_QOS_DEFAULT,
                               0,   // No listener required
                               OpenDDS::DCPS::DEFAULT_STATUS_MASK);
  if (!topic) {
    std::cerr << "create_topic failed." << std::endl;
    return 1;
  }
```

We have created a topic named "*Movie Discussion List*" with the registered type and the default QoS policies.

2.1.3.3   Creating a Publisher

Now, we are ready to create the publisher with the default publisher QoS.

```
  DDS::Publisher_var pub =
    participant->create_publisher(PUBLISHER_QOS_DEFAULT,
                                  0,    // No listener required
                                  OpenDDS::DCPS::DEFAULT_STATUS_MASK);
  if (!pub) {
    std::cerr << "create_publisher failed." << std::endl;
    return 1;
  }
```

2.1.3.4   Creating a DataWriter and Waiting for the Subscriber

With the publisher in place, we create the data writer.

```
// Create the datawriter
DDS::DataWriter_var writer =
  pub->create_datawriter(topic,
                         DATAWRITER_QOS_DEFAULT,
                         0,    // No listener required
                         OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (!writer) {
  std::cerr << "create_datawriter failed." << std::endl;
  return 1;
}
```

When we create the data writer we pass the topic object reference, the default QoS policies, and a null listener reference. We now narrow the data writer reference to a `MessageDataWriter` object reference so we can use the type-specific publication operations.

```
Messenger::MessageDataWriter_var message_writer =
    Messenger::MessageDataWriter::_narrow(writer);
```

The example code uses *conditions* and *wait* sets so the publisher waits for the subscriber to become connected and fully initialized. In a simple example like this, failure to wait for the subscriber may cause the publisher to publish its samples before the subscriber is connected.

The basic steps involved in waiting for the subscriber are:

1) Get the status condition from the data writer we created

2) Enable the Publication Matched status in the condition

3) Create a wait set

4) Attach the status condition to the wait set

5) Get the publication matched status

6) If the current count of matches is one or more, detach the condition from the wait set and proceed to publication

7) Wait on the wait set (can be bounded by a specified period of time)

8) Loop back around to step 5

Here is the corresponding code:

```
// Block until Subscriber is available
DDS::StatusCondition_var condition =
    writer->get_statuscondition();
    condition->set_enabled_statuses(
     DDS::PUBLICATION_MATCHED_STATUS);
```

```
DDS::WaitSet_var ws = new DDS::WaitSet;
ws->attach_condition(condition);

while (true) {
  DDS::PublicationMatchedStatus matches;
  if (writer->get_publication_matched_status(matches)
      != DDS::RETCODE_OK) {
    std::cerr << "get_publication_matched_status failed!"
              << std::endl;
    return 1;
  }

  if (matches.current_count >= 1) {
    break;
  }

  DDS::ConditionSeq conditions;
  DDS::Duration_t timeout = { 60, 0 };
  if (ws->wait(conditions, timeout) != DDS::RETCODE_OK) {
    std::cerr << "wait failed!" << std::endl;
    return 1;
  }

}

ws->detach_condition(condition);
```

For more details about status, conditions, and wait sets, see Chapter 4.

2.1.3.5   Sample Publication

The message publication is quite straightforward:

```
// Write samples
Messenger::Message message;
message.subject_id = 99;
message.from       = "Comic Book Guy";
message.subject    = "Review";
message.text       = "Worst. Movie. Ever.";
message.count      = 0;
for (int i = 0; i < 10; ++i) {
  DDS::ReturnCode_t error = message_writer->write(message,
                                                  DDS::HANDLE_NIL);
  ++message.count;
  ++message.subject_id;
  if (error != DDS::RETCODE_OK) {
    // Log or otherwise handle the error condition
    return 1;
  }
}
```

For each loop iteration, calling write() causes a message to be distributed to all connected subscribers that are registered for our topic. Since the subject_id is the key for Message, each time subject_id is incremented and write() is called, a new instance is created (see 1.1.1.3). The second argument to write() specifies the instance on which we are publishing the sample. It should be passed either a handle returned by register_instance() or DDS::HANDLE_NIL. Passing a DDS::HANDLE_NIL value indicates that the data writer should

determine the instance by inspecting the key of the sample. See Section 2.2.1 for details on using instance handles during publication.

2.1.4   Setting Up The Subscriber

Much of the subscriber's code is identical or analogous to the publisher that we just finished exploring. We will progress quickly through the similar parts and refer you to the discussion above for details. The full source code for this sample subscriber is found in the *Subscriber.cpp* and *DataReaderListener.cpp* files in $DDS_ROOT/DevGuideExamples/DCPS/Messenger/.

2.1.4.1   Initializing the Participant

The beginning of the subscriber is identical to the publisher as we initialize the service and join our domain:

```
int main (int argc, char *argv[])
{
  try {
    DDS::DomainParticipantFactory_var dpf =
      TheParticipantFactoryWithArgs(argc, argv);
    DDS::DomainParticipant_var participant =
      dpf->create_participant(42, // Domain ID
                              PARTICIPANT_QOS_DEFAULT,
                              0,  // No listener required
                              OpenDDS::DCPS::DEFAULT_STATUS_MASK);
    if (!participant) {
      std::cerr << "create_participant failed." << std::endl;
      return 1;
    }
```

2.1.4.2   Registering the Data Type and Creating a Topic

Next, we initialize the message type and topic. Note that if the topic has already been initialized in this domain with the same data type and compatible QoS, the create_topic() invocation returns a reference corresponding to the existing topic. If the type or QoS specified in our create_topic() invocation do not match that of the existing topic then the invocation fails. There is also a find_topic() operation our subscriber could use to simply retrieve an existing topic.

```
    Messenger::MessageTypeSupport_var mts =
      new Messenger::MessageTypeSupportImpl();
    if (DDS::RETCODE_OK != mts->register_type(participant, "")) {
      std::cerr << "Failed to register the MessageTypeSupport." << std::endl;
      return 1;
    }

    CORBA::String_var type_name = mts->get_type_name ();

    DDS::Topic_var topic =
      participant->create_topic("Movie Discussion List",
                                type_name,
```

OpenDDS Developer's Guide

```
                                   TOPIC_QOS_DEFAULT,
                                   0,    // No listener required
                                   OpenDDS::DCPS::DEFAULT_STATUS_MASK);
      if (!topic) {
        std::cerr << "Failed to create_topic." << std::endl;
        return 1;
      }
```

2.1.4.3 Creating the subscriber

Next, we create the subscriber with the default QoS.

```
// Create the subscriber
DDS::Subscriber_var sub =
  participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT,
                                 0,    // No listener required
                                 OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (!sub) {
  std::cerr << "Failed to create_subscriber." << std::endl;
  return 1;
}
```

2.1.4.4 Creating a DataReader and Listener

We need to associate a listener object with the data reader we create, so we can use it to detect when data is available. The code below constructs the listener object. The `DataReaderListenerImpl` class is shown in the next subsection.

```
DDS::DataReaderListener_var listener(new DataReaderListenerImpl);
```

The listener is allocated on the heap and assigned to a `DataReaderListener_var` object. This type provides reference counting behavior so the listener is automatically cleaned up when the last reference to it is removed. This usage is typical for heap allocations in OpenDDS application code and frees the application developer from having to actively manage the lifespan of the allocated objects.

Now we can create the data reader and associate it with our topic, the default QoS properties, and the listener object we just created.

```
// Create the Datareader
DDS::DataReader_var dr =
  sub->create_datareader(topic,
                         DATAREADER_QOS_DEFAULT,
                         listener,
                         OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (!dr) {
  std::cerr << "create_datareader failed." << std::endl;
  return 1;
}
```

This thread is now free to perform other application work. Our listener object will be called on an OpenDDS thread when a sample is available.

2.1.5   **The Data Reader Listener Implementation**

Our listener class implements the `DDS::DataReaderListener` interface defined by the DDS specification. The `DataReaderListener` is wrapped within a `DCPS::LocalObject` which resolves ambiguously-inherited members such as `_narrow` and `_ptr_type`. The interface defines a number of operations we must implement, each of which is invoked to inform us of different events. The `OpenDDS::DCPS::DataReaderListener` defines operations for OpenDDS's special needs such as disconnecting and reconnected event updates. Here is the interface definition:

```
module DDS {
  local interface DataReaderListener : Listener {
    void on_requested_deadline_missed(in DataReader reader,
                                      in RequestedDeadlineMissedStatus status);
    void on_requested_incompatible_qos(in DataReader reader,
                                       in RequestedIncompatibleQosStatus status);
    void on_sample_rejected(in DataReader reader,
                            in SampleRejectedStatus status);
    void on_liveliness_changed(in DataReader reader,
                               in LivelinessChangedStatus status);
    void on_data_available(in DataReader reader);
    void on_subscription_matched(in DataReader reader,
                                 in SubscriptionMatchedStatus status);
    void on_sample_lost(in DataReader reader, in SampleLostStatus status);
  };
};
```

Our example listener class stubs out most of these listener operations with simple print statements. The only operation that is really needed for this example is `on_data_available()` and it is the only member function of this class we need to explore.

```
void DataReaderListenerImpl::on_data_available(DDS::DataReader_ptr reader)
{
  ++num_reads_;

  try {
    Messenger::MessageDataReader_var reader_i =
        Messenger::MessageDataReader::_narrow(reader);
    if (!reader_i) {
      std::cerr << "read: _narrow failed." << std::endl;
      return;
    }
```

The code above narrows the generic data reader passed into the listener to the type-specific `MessageDataReader` interface. The following code takes the next sample from the message reader. If the take is successful and returns valid data, we print out each of the message's fields.

```
    Messenger::Message message;
    DDS::SampleInfo si;
    DDS::ReturnCode_t status = reader_i->take_next_sample(message, si);

    if (status == DDS::RETCODE_OK) {

      if (si.valid_data == 1) {

        std::cout << "Message: subject    = " << message.subject.in() << std::endl
```

```
              << "        subject_id  = " << message.subject_id   << std::endl
              << "        from        = " << message.from.in()    << std::endl
              << "        count       = " << message.count         << std::endl
              << "        text        = " << message.text.in()    << std::endl;
      }
      else if (si.instance_state == DDS::NOT_ALIVE_DISPOSED_INSTANCE_STATE)
      {
        std::cout << "instance is disposed" << std::endl;
      }
      else if (si.instance_state == DDS::NOT_ALIVE_NO_WRITERS_INSTANCE_STATE)
      {
        std::cout << "instance is unregistered" << std::endl;
      }
      else
      {
        std::cerr << "ERROR: received unknown instance state "
                << si.instance_state << std::endl;
      }
    } else if (status == DDS::RETCODE_NO_DATA) {
        cerr << "ERROR: reader received DDS::RETCODE_NO_DATA!" << std::endl;
    } else {
        cerr << "ERROR: read Message: Error: " <<  status << std::endl;
    }
```

Note the sample read may contain invalid data. The valid_data flag indicates if the sample
has valid data. There are two samples with invalid data delivered to the listener callback for
notification purposes. One is the *dispose* notification, which is received when the
`DataWriter` calls `dispose()` explicitly. The other is the *unregistered* notification, which is
received when the `DataWriter` calls `unregister()` explicitly. The dispose notification is
delivered with the instance state set to `NOT_ALIVE_DISPOSED_INSTANCE_STATE` and the
unregister notification is delivered with the instance state set to
`NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`.

If additional samples are available, the service calls this function again. However, reading
values a single sample at a time is not the most efficient way to process incoming data. The
Data Reader interface provides a number of different options for processing data in a more
efficient manner. We discuss some of these operations in Section 2.2.

2.1.6 Cleaning Up In OpenDDS Clients

After we are finished in the publisher and subscriber, we can use the following code to clean
up the OpenDDS-related objects:

```
participant->delete_contained_entities();
dpf->delete_participant(participant);
TheServiceParticipant->shutdown ();
```

The domain participant's `delete_contained_entities()` operation deletes all the topics,
subscribers, and publishers created with that participant. Once this is done, we can use the
domain participant factory to delete our domain participant.

Since the publication and subscription of data within DDS is decoupled, data is not
guaranteed to be delivered if a publication is disassociated (shutdown) prior to all data that

has been sent having been received by the subscriptions. If the application requires that all published data be received, the `wait_for_acknowledgements()` operation is available to allow the publication to wait until all written data has been received. Data readers must have a `RELIABLE` setting for the `RELIABILITY` QoS (which is the default) in order for `wait_for_acknowledgements()` to work. This operation is called on individual `DataWriters` and includes a timeout value to bound the time to wait. The following code illustrates the use of `wait_for_acknowledgements()` to block for up to 15 seconds to wait for subscriptions to acknowledge receipt of all written data:

```
DDS::Duration_t shutdown_delay = {15, 0};
DDS::ReturnCode_t result;
result = writer->wait_for_acknowledgments(shutdown_delay);
if( result != DDS::RETCODE_OK) {
  std::cerr << "Failed while waiting for acknowledgment of "
            << "data being received by subscriptions, some data "
            << "may not have been delivered." << std::endl;
}
```

2.1.7   **Running The Example**

We are now ready to run our simple example. Running each of these commands in its own window should enable you to most easily understand the output.

First we will start a `DCPSInfoRepo` service so our publishers and subscribers can find one another.

Note   *This step is not necessary if you are using peer-to-peer discovery by configuring your environment to use RTPS discovery.*

The `DCPSInfoRepo` executable is found in $DDS_ROOT/bin/DCPSInfoRepo. When we start the `DCPSInfoRepo` we need to ensure that publisher and subscriber application processes can also find the started `DCPSInfoRepo`. This information can be provided in one of three ways: a.) parameters on the command line , b.) generated and placed in a shared file for applications to use, or c.) parameters placed in a configuration file for other processes to use. For our simple example here we will use option 'b' by generating the location properties of the `DCPSInfoRepo` into a file so that our simple publisher and subscriber can read it in and connect to it.

From your current directory type:

Windows:

`%DDS_ROOT%\bin\DCPSInfoRepo -o simple.ior`

Unix:

`$DDS_ROOT/bin/DCPSInfoRepo -o simple.ior`

The `-o` parameter instructs the `DCPSInfoRepo` to generate its connection information to the file *simple.ior* for use by the publisher and subscriber. In a separate window navigate to the same directory that contains the *simple.ior* file and start the subscriber application in our example by typing:

Windows:

```
subscriber -DCPSInfoRepo file://simple.ior
```

Unix:

```
./subscriber -DCPSInfoRepo file://simple.ior
```

The command line parameters direct the application to use the specified file to locate the `DCPSInfoRepo`. Our subscriber is now waiting for messages to be sent, so we will now start the publisher in a separate window with the same parameters:

Windows:

```
publisher -DCPSInfoRepo file://simple.ior
```

Unix

```
./publisher -DCPSInfoRepo file://simple.ior
```

The publisher connects to the `DCPSInfoRepo` to find the location of any subscribers and begins to publish messages as well as write them to the console. In the subscriber window, you should also now be seeing console output from the subscriber that is reading messages from the topic demonstrating a simple publish and subscribe application.

You can read more about configuring your application for RTPS and other more advanced configuration options in Section 7.3.3 and Section 7.4.5.5 . To read more about configuring and using the `DCPSInfoRepo` go to Section 7.3 and Chapter 9. To find more about setting and using QoS features that modify the behavior of your application read Chapter 3.

2.1.8   **Running Our Example With RTPS**

The prior OpenDDS example has demonstrated how to build and execute an OpenDDS application using basic OpenDDS configurations and centralized discovery using the `DCPSInfoRepo` service. The following details what is needed to run the same example using RTPS for discovery and with an interoperable transport. This is important in scenarios when your OpenDDS application needs to interoperate with a non-OpenDDS implementation of the DDS specification or if you do not want to use centralized discovery in your deployment of OpenDDS.

The coding and building of the Messenger example above is not changed for using RTPS, so you will not need to modify or rebuild your publisher and subscriber services. This is a

strength of the OpenDDS architecture in that to enable the RTPS capabilities, it is an exercise of configuration. Chapter 7 will cover more details concerning the configuration of all the available transports including RTPS, however, for this exercise we will enable RTPS for the Messenger example using a configuration file that the publisher and subscriber will share.

Navigate to the directory where your publisher and subscriber have been built. Create a new text file named *rtps.ini* and populate it with the following content:

```
[common]
DCPSGlobalTransportConfig=$file
DCPSDefaultDiscovery=DEFAULT_RTPS

[transport/the_rtps_transport]
transport_type=rtps_udp
```

More details of configuration files are specified in upcoming chapters, but the two lines of interest are called out for setting the discovery method and the data transport protocol to RTPS.

Now lets re-run our example with RTPS enabled by starting the subscriber process first and then the publisher to begin sending data. It is best to start them in separate windows to see the two working separately.

Start the subscriber with the -DCPSConfigFile command line parameter to point to the newly created configuration file...

Windows:

```
subscriber -DCPSConfigFile rtps.ini
```

Unix:

```
./subscriber -DCPSConfigFile rtps.ini
```

Now start the publisher with the same parameter...

Windows:

```
publisher -DCPSConfigFile rtps.ini
```

Unix:

```
./publisher -DCPSConfigFile rtps.ini
```

Since there is no centralized discovery in the RTPS specification, there are provisions to allow for wait times to allow discovery to occur. The specification sets the default to 30 seconds. When the two above processes are started there may be up to a 30 second delay depending on how far apart they are started from each other. This time can be adjusted in OpenDDS configuration files discussed later Section 7.3.3.

Because the architecture of OpenDDS allows for pluggable discovery and pluggable transports the two configuration entries called out in the *rtps.ini* file above can be changed independently with one using RTPS and the other not using RTPS (e.g. centralized discovery using DCPSInfoRepo). Setting them both to RTPS in our example makes this application fully interoperable with other non-OpenDDS implementations.

2.2 Data Handling Optimizations

2.2.1 Registering And Using Instances In The Publisher

The previous example implicitly specifies the instance it is publishing via the sample's data fields. When write() is called, the data writer queries the sample's key fields to determine the instance. The publisher also has the option to explicitly register the instance by calling register_instance() on the data writer:

```
Messenger::Message message;
message.subject_id = 99;
DDS::InstanceHandle_t handle = message_writer->register_instance(message);
```

After we populate the Message structure we called the register_instance() function to register the instance. The instance is identified by the subject_id value of 99 (because we earlier specified that field as the key).

We can later use the returned instance handle when we publish a sample:

```
DDS::ReturnCode_t ret = data_writer->write(message, handle);
```

Publishing samples using the instance handle may be slightly more efficient than forcing the writer to query for the instance and is much more efficient when publishing the first sample on an instance. Without explicit registration, the first write causes resource allocation by OpenDDS for that instance.

Because resource limitations can cause instance registration to fail, many applications consider registration as part of setting up the publisher and always do it when initializing the data writer.

2.2.2 Reading Multiple Samples

The DDS specification provides a number of operations for reading and writing data samples. In the examples above we used the take_next_sample() operation, to read the next sample and "take" ownership of it from the reader. The Message Data Reader also has the following take operations.

- take()—Take a sequence of up to max_samples values from the reader

- take_instance()—Take a sequence of values for a specified instance

- take_next_instance()—Take a sequence of samples belonging to the same instance, without specifying the instance.

There are also "read" operations corresponding to each of these "take" operations that obtain the same values, but leave the samples in the reader and simply mark them as read in the SampleInfo.

Since these other operations read a sequence of values, they are more efficient when samples are arriving quickly. Here is a sample call to take() that reads up to 5 samples at a time.

```
MessageSeq messages(5);
DDS::SampleInfoSeq sampleInfos(5);
DDS::ReturnCode_t status =
    message_dr->take(messages,
                     sampleInfos,
                     5,
                     DDS::ANY_SAMPLE_STATE,
                     DDS::ANY_VIEW_STATE,
                     DDS::ANY_INSTANCE_STATE);
```

The three state parameters potentially specialize which samples are returned from the reader. See the DDS specification for details on their usage.

2.2.3   Zero-Copy Read

The read and take operations that return a sequence of samples provide the user with the option of obtaining a copy of the samples (single-copy read) or a reference to the samples (zero-copy read). The zero-copy read can have significant performance improvements over the single-copy read for large sample types. Testing has shown that samples of 8KB or less do not gain much by using zero-copy reads but there is little performance penalty for using zero-copy on small samples.

The application developer can specify the use of the zero-copy read optimization by calling take() or read() with a sample sequence constructed with a max_len of zero. The message sequence and sample info sequence constructors both take max_len as their first parameter and specify a default value of zero. The following example code is taken from DevGuideExamples/DCPS/Messenger_ZeroCopy/DataReaderListenerImpl.cpp:

```
Messenger::MessageSeq messages;
DDS::SampleInfoSeq info;

// get references to the samples  (zero-copy read of the samples)
DDS::ReturnCode_t status = dr->take(messages,
                                    info,
```

```
                                    DDS::LENGTH_UNLIMITED,
                                    DDS::ANY_SAMPLE_STATE,
                                    DDS::ANY_VIEW_STATE,
                                    DDS::ANY_INSTANCE_STATE);
```

After both zero-copy takes/reads and single-copy takes/reads, the sample and info sequences' length are set to the number of samples read. For the zero-copy reads, the `max_len` is set to a `value >= length`.

Since the application code has asked for a zero-copy loan of the data, it must return that loan when it is finished with the data:

```
    dr->return_loan(messages, info);
```

Calling `return_loan()` results in the sequences' `max_len` being set to 0 and its owns member set to false, allowing the same sequences to be used for another zero-copy read.

If the first parameter of the data sample sequence constructor and info sequence constructor were changed to a value greater than zero, then the sample values returned would be copies. When values are copied, the application developer has the option of calling `return_loan()`, but is not required to do so.

If the `max_len` (the first) parameter of the sequence constructor is not specified, it defaults to 0; hence using zero-copy reads. Because of this default, a sequence will automatically call `return_loan()` on itself when it is destroyed. To conform with the DDS specification and be portable to other implementations of DDS, applications should not rely on this automatic `return_loan()` feature.

The second parameter to the sample and info sequences is the maximum slots available in the sequence. If the `read()` or `take()` operation's `max_samples` parameter is larger than this value, then the maximum samples returned by `read()` or `take()` will be limited by this parameter of the sequence constructor.

Although the application can change the length of a zero-copy sequence, by calling the `length(len)` operation, you are advised against doing so because this call results in copying the data and creating a single-copy sequence of samples.

CHAPTER 3

Quality of Service

3.1 Introduction

The previous examples use default QoS policies for the various entities. This chapter discusses the QoS policies which are implemented in OpenDDS and the details of their usage. See the DDS specification for further information about the policies discussed in this chapter.

3.2 QoS Policies

Each policy defines a structure to specify its data. Each entity supports a subset of the policies and defines a QoS structure that is composed of the supported policy structures. The set of allowable policies for a given entity is constrained by the policy structures nested in its QoS structure. For example, the Publisher's QoS structure is defined in the specification's IDL as follows:

```
module DDS {
  struct PublisherQos {
    PresentationQosPolicy presentation;
    PartitionQosPolicy partition;
    GroupDataQosPolicy group_data;
    EntityFactoryQosPolicy entity_factory;
  };
};
```

Setting policies is as simple as obtaining a structure with the default values already set,

modifying the individual policy structures as necessary, and then applying the QoS structure to an entity (usually when it is created). We show examples of how to obtain the default QoS policies for various entity types in Section 3.2.1.

Applications can change the QoS of any entity by calling the set_qos() operation on the entity. If the QoS is changeable, existing associations are removed if they are no longer compatible and new associations are added if they become compatible. The `DCPSInfoRepo` re-evaluates the QoS compatibility and associations according to the QoS specification. If the compatibility checking fails, the call to set_qos() will return an error. The association re-evaluation may result in removal of existing associations or addition of new associations.

If the user attempts to change a QoS policy that is immutable (not changeable), then `set_qos()` returns `DDS::RETCODE_IMMUTABLE_POLICY`.

A subset of the QoS policies are changeable. Some changeable QoS policies, such as `USER_DATA`, `TOPIC_DATA`, `GROUP_DATA`, `LIFESPAN`, `OWNERSHIP_STRENGTH`, `TIME_BASED_FILTER`, `ENTITY_FACTORY`, `WRITER_DATA_LIFECYCLE`, and `READER_DATA_LIFECYCLE`, do not require compatibility and association re-evaluation. The `DEADLINE` and `LATENCY_BUDGET` QoS policies require compatibility re-evaluation, but not for association. The `PARTITION` QoS policy does not require compatibility re-evaluation, but does require association re-evaluation. The DDS specification lists `TRANSPORT_PRIORITY` as changeable, but the OpenDDS implementation does not support dynamically modifying this policy.

3.2.1   Default QoS Policy Values

Applications obtain the default QoS policies for an entity by instantiating a QoS structure of the appropriate type for the entity and passing it by reference to the appropriate `get_default_entity_qos()` operation on the appropriate factory entity. (For example, you would use a domain participant to obtain the default QoS for a publisher or subscriber.) The following examples illustrate how to obtain the default policies for publisher, subscriber, topic, domain participant, data writer, and data reader.

```
// Get default Publisher QoS from a DomainParticipant:
DDS::PublisherQos pub_qos;
DDS::ReturnCode_t ret;
ret = domain_participant->get_default_publisher_qos(pub_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default publisher QoS" << std::endl;
}

// Get default Subscriber QoS from a DomainParticipant:
DDS::SubscriberQos sub_qos;
ret = domain_participant->get_default_subscriber_qos(sub_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default subscriber QoS" << std::endl;
}
```

```
// Get default Topic QoS from a DomainParticipant:
DDS::TopicQos topic_qos;
ret = domain_participant->get_default_topic_qos(topic_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default topic QoS" << std::endl;
}

// Get default DomainParticipant QoS from a DomainParticipantFactory:
DDS::DomainParticipantQos dp_qos;
ret = domain_participant_factory->get_default_participant_qos(dp_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default participant QoS" << std::endl;
}

// Get default DataWriter QoS from a Publisher:
DDS::DataWriterQos dw_qos;
ret = pub->get_default_datawriter_qos(dw_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default data writer QoS" << std::endl;
}

// Get default DataReader QoS from a Subscriber:
DDS::DataReaderQos dr_qos;
ret = sub->get_default_datareader_qos(dr_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default data reader QoS" << std::endl;
}
```

The following tables summarize the default QoS policies for each entity type in OpenDDS to which policies can be applied.

Table 3-1 Default DomainParticipant QoS Policies

Policy	Member	Default Value
USER_DATA	value	(empty sequence)
ENTITY_FACTORY	autoenable_created_entities	true

Table 3-2 Default Topic QoS Policies

Policy	Member	Default Value
TOPIC_DATA	value	(empty sequence)
DURABILITY	kind service_cleanup_delay.sec service_cleanup_delay.nanosec	VOLATILE_DURABILITY_QOS DURATION_ZERO_SEC DURATION_ZERO_NSEC
DURABILITY_SERVICE	service_cleanup_delay.sec service_cleanup_delay.nanosec history_kind history_depth max_samples max_instances max_samples_per_instance	DURATION_ZERO_SEC DURATION_ZERO_NSEC KEEP_LAST_HISTORY_QOS 1 LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
DEADLINE	period.sec period.nanosec	DURATION_INFINITE_SEC DURATION_INFINITE_NSEC
LATENCY_BUDGET	duration.sec duration.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC

Policy	Member	Default Value
LIVELINESS	kind lease_duration.sec lease_duration.nanosec	AUTOMATIC_LIVELINESS_QOS DURATION_INFINITE_SEC DURATION_INFINITE_NSEC
RELIABILITY	kind max_blocking_time.sec max_blocking_time.nanosec	BEST_EFFORT_RELIABILITY_QOS DURATION_INFINITE_SEC DURATION_INFINITE_NSEC
DESTINATION_ORDER	kind	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
HISTORY	kind depth	KEEP_LAST_HISTORY_QOS 1
RESOURCE_LIMITS	max_samples max_instances max_samples_per_instance	LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
TRANSPORT_PRIORITY	value	0
LIFESPAN	duration.sec duration.nanosec	DURATION_INFINITE_SEC DURATION_INFINITE_NSEC
OWNERSHIP	kind	SHARED_OWNERSHIP_QOS

Table 3-3 Default Publisher QoS Policies

Policy	Member	Default Value
PRESENTATION	access_scope coherent_access ordered_access	INSTANCE_PRESENTATION_QOS 0 0
PARTITION	name	(empty sequence)
GROUP_DATA	value	(empty sequence)
ENTITY_FACTORY	autoenable_created_entities	true

Table 3-4 Default Subscriber QoS Policies

Policy	Member	Default Value
PRESENTATION	access_scope coherent_access ordered_access	INSTANCE_PRESENTATION_QOS 0 0
PARTITION	name	(empty sequence)
GROUP_DATA	value	(empty sequence)
ENTITY_FACTORY	autoenable_created_entities	true

Table 3-5 Default DataWriter QoS Policies

Policy	Member	Default Value
DURABILITY	kind service_cleanup_delay.sec service_cleanup_delay.nanosec	VOLATILE_DURABILITY_QOS DURATION_ZERO_SEC DURATION_ZERO_NSEC
DURABILITY_SERVICE	service_cleanup_delay.sec service_cleanup_delay.nanosec history_kind history_depth max_samples max_instances max_samples_per_instance	DURATION_ZERO_SEC DURATION_ZERO_NSEC KEEP_LAST_HISTORY_QOS 1 LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED

Policy	Member	Default Value
DEADLINE	period.sec period.nanosec	DURATION_INFINITE_SEC DURATION_INFINITE_NSEC
LATENCY_BUDGET	duration.sec duration.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC
LIVELINESS	kind lease_duration.sec lease_duration.nanosec	AUTOMATIC_LIVELINESS_QOS DURATION_INFINITE_SEC DURATION_INFINITE_NSEC
RELIABILITY	kind max_blocking_time.sec max_blocking_time.nanosec	RELIABLE_RELIABILITY_QOS[2] 0 100000000 (100 ms)
DESTINATION_ORDER	kind	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
HISTORY	kind depth	KEEP_LAST_HISTORY_QOS 1
RESOURCE_LIMITS	max_samples max_instances max_samples_per_instance	LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
TRANSPORT_PRIORITY	value	0
LIFESPAN	duration.sec duration.nanosec	DURATION_INFINITE_SEC DURATION_INFINITE_NSEC
USER_DATA	value	(empty sequence)
OWNERSHIP	kind	SHARED_OWNERSHIP_QOS
OWNERSHIP_STRENGTH	value	0
WRITER_DATA_LIFECYCLE	autodispose_unregistered_instances	1

Table 3-6 Default DataReader QoS Policies

Policy	Member	Default Value
DURABILITY	kind service_cleanup_delay.sec service_cleanup_delay.nanosec	VOLATILE_DURABILITY_QOS DURATION_ZERO_SEC DURATION_ZERO_NSEC
DEADLINE	period.sec period.nanosec	DURATION_INFINITE_SEC DURATION_INFINITE_NSEC
LATENCY_BUDGET	duration.sec duration.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC
LIVELINESS	kind lease_duration.sec lease_duration.nanosec	AUTOMATIC_LIVELINESS_QOS DURATION_INFINITE_SEC DURATION_INFINITE_NSEC
RELIABILITY	kind max_blocking_time.sec max_blocking_time.nanosec	BEST_EFFORT_RELIABILITY_QOS DURATION_INFINITE_SEC DURATION_INFINITE_NSEC
DESTINATION_ORDER	kind	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
HISTORY	kind depth	KEEP_LAST_HISTORY_QOS 1
RESOURCE_LIMITS	max_samples max_instances max_samples_per_instance	LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
USER_DATA	value	(empty sequence)
OWNERSHIP	kind	SHARED_OWNERSHIP_QOS
TIME_BASED_FILTER	minimum_separation.sec minimum_separation.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC

2   For OpenDDS versions, up to 2.0, the default reliability kind for data writers is best effort. For versions 2.0.1 and later, this is changed to reliable (to conform to the DDS specification).

Policy	Member	Default Value
READER_DATA_LIFECYCLE	autopurge_nowriter_samples_delay.sec autopurge_nowriter_samples_delay.nanosec autopurge_disposed_samples_delay.sec autopurge_disposed_samples_delay.nanosec	DURATION_INFINITE_SEC DURATION_INFINITE_NSEC DURATION_INFINITE_SEC DURATION_INFINITE_NSEC

3.2.2   LIVELINESS

The LIVELINESS policy applies to the topic, data reader, and data writer entities via the liveliness member of their respective QoS structures. Setting this policy on a topic means it is in effect for all data readers and data writers on that topic. Below is the IDL related to the liveliness QoS policy:

```
enum LivelinessQosPolicyKind {
  AUTOMATIC_LIVELINESS_QOS,
  MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
  MANUAL_BY_TOPIC_LIVELINESS_QOS
};

struct LivelinessQosPolicy {
  LivelinessQosPolicyKind kind;
  Duration_t lease_duration;
};
```

The LIVELINESS policy controls when and how the service determines whether participants are alive, meaning they are still reachable and active. The kind member setting indicates whether liveliness is asserted automatically by the service or manually by the specified entity. A setting of AUTOMATIC_LIVELINESS_QOS means that the service will send a liveliness indication if the participant has not sent any network traffic for the lease_duration. The MANUAL_BY_PARTICIPANT_LIVELINESS_QOS or MANUAL_BY_TOPIC_LIVELINESS_QOS setting means the specified entity (data writer for the "by topic" setting or domain participant for the "by participant" setting) must either write a sample or manually assert its liveliness within a specified heartbeat interval. The desired heartbeat interval is specified by the lease_duration member. The default lease duration is a pre-defined infinite value, which disables any liveliness testing.

To manually assert liveliness without publishing a sample, the application must call the assert_liveliness() operation on the data writer (for the "by topic" setting) or on the domain participant (for the "by participant" setting) within the specified heartbeat interval.

Data writers specify (*offer*) their own liveliness criteria and data readers specify (*request*) the desired liveliness of their writers. Writers that are not heard from within the lease duration (either by writing a sample or by asserting liveliness) cause a change in the LIVELINESS_CHANGED_STATUS communication status and notification to the application (e.g., by calling the data reader listener's on_liveliness_changed() callback operation or by signaling any related wait sets).

    O p e n D D S   D e v e l o p e r ' s   G u i d e

This policy is considered during the establishment of associations between data writers and data readers. The value of both sides of the association must be compatible in order for an association to be established. Compatibility is determined by comparing the data reader's requested liveliness with the data writer's offered liveliness. Both the kind of liveliness (automatic, manual by topic, manual by participant) and the value of the lease duration are considered in determining compatibility. The writer's offered kind of liveliness must be greater than or equal to the reader's requested kind of liveliness. The liveliness kind values are ordered as follows:

```
MANUAL_BY_TOPIC_LIVELINESS_QOS >
MANUAL_BY_PARTICIPANT_LIVELINESS_QOS >
AUTOMATIC_LIVELINESS_QOS
```

In addition, the writer's offered lease duration must be less than or equal to the reader's requested lease duration. Both of these conditions must be met for the offered and requested liveliness policy settings to be considered compatible and the association established.

3.2.3   RELIABILITY

The RELIABILITY policy applies to the topic, data reader, and data writer entities via the reliability member of their respective QoS structures. Below is the IDL related to the reliability QoS policy:

```
enum ReliabilityQosPolicyKind {
  BEST_EFFORT_RELIABILITY_QOS,
  RELIABLE_RELIABILITY_QOS
};

struct ReliabilityQosPolicy {
  ReliabilityQosPolicyKind kind;
  Duration_t max_blocking_time;
};
```

This policy controls how data readers and writers treat the data samples they process. The "best effort" value (BEST_EFFORT_RELIABILITY_QOS) makes no promises as to the reliability of the samples and could be expected to drop samples under some circumstances. The "reliable" value (RELIABLE_RELIABILITY_QOS) indicates that the service should eventually deliver all values to eligible data readers.

The max_blocking_time member of this policy is used when the history QoS policy is set to "keep all" and the writer is unable to proceed because of resource limits. When this situation occurs and the writer blocks for more than the specified time, then the write fails with a timeout return code. The default for this policy for data readers and topics is "best effort," while the default value for data writers is "reliable."

This policy is considered during the creation of associations between data writers and data readers. The value of both sides of the association must be compatible in order for an association to be created. The reliability kind of data writer must be greater than or equal to the value of data reader.

3.2.4   HISTORY

The `HISTORY` policy determines how samples are held in the data writer and data reader for a particular instance. For data writers these values are held until the publisher retrieves them and successfully sends them to all connected subscribers. For data readers these values are held until "taken" by the application. This policy applies to the topic, data reader, and data writer entities via the history member of their respective QoS structures. Below is the IDL related to the history QoS policy:

```
enum HistoryQosPolicyKind {
  KEEP_LAST_HISTORY_QOS,
  KEEP_ALL_HISTORY_QOS
};

struct HistoryQosPolicy {
  HistoryQosPolicyKind kind;
  long depth;
};
```

The "keep all" value (`KEEP_ALL_HISTORY_QOS`) specifies that all possible samples for that instance should be kept. When "keep all" is specified and the number of unread samples is equal to the "resource limits" field of `max_samples_per_instance` then any incoming samples are rejected.

The "keep last" value (`KEEP_LAST_HISTORY_QOS`) specifies that only the last `depth` values should be kept. When a data writer contains depth samples of a given instance, a write of new samples for that instance are queued for delivery and the oldest unsent samples are discarded. When a data reader contains depth samples of a given instance, any incoming samples for that instance are kept and the oldest samples are discarded.

This policy defaults to a "keep last" with a `depth` of one.

3.2.5   DURABILITY

The `DURABILITY` policy controls whether data writers should maintain samples after they have been sent to known subscribers. This policy applies to the topic, data reader, and data writer entities via the durability member of their respective QoS structures. Below is the IDL related to the durability QoS policy:

```
enum DurabilityQosPolicyKind {
  VOLATILE_DURABILITY_QOS,          // Least Durability
```

```
   TRANSIENT_LOCAL_DURABILITY_QOS,
   TRANSIENT_DURABILITY_QOS,
   PERSISTENT_DURABILITY_QOS         // Greatest Durability
};

struct DurabilityQosPolicy {
  DurabilityQosPolicyKind kind;
};
```

By default the kind is VOLATILE_DURABILITY_QOS.

A durability kind of VOLATILE_DURABILITY_QOS means samples are discarded after being sent to all known subscribers. As a side effect, subscribers cannot recover samples sent before they connect.

A durability kind of TRANSIENT_LOCAL_DURABILITY_QOS means that data readers that are associated/connected with a data writer will be sent all of the samples in the data writer's history.

A durability kind of TRANSIENT_DURABILITY_QOS means that samples outlive a data writer and last as long as the process is alive. The samples are kept in memory, but are not persisted to permanent storage. A data reader subscribed to the same topic and partition within the same domain will be sent all of the cached samples that belong to the same topic/partition.

A durability kind of PERSISTENT_DURABILITY_QOS provides basically the same functionality as transient durability except the cached samples are persisted and will survive process destruction.

When transient or persistent durability is specified, the DURABILITY_SERVICE QoS policy specifies additional tuning parameters for the durability cache.

The durability policy is considered during the creation of associations between data writers and data readers. The value of both sides of the association must be compatible in order for an association to be created. The durability kind value of the data writer must be greater than or equal to the corresponding value of the data reader. The durability kind values are ordered as follows:

```
PERSISTENT_DURABILITY_QOS >
TRANSIENT_DURABILITY_QOS >
TRANSIENT_LOCAL_DURABILITY_QOS >
VOLATILE_DURABILITY_QOS
```

3.2.6   DURABILITY_SERVICE

The DURABILITY_SERVICE policy controls deletion of samples in TRANSIENT or PERSISTENT durability cache. This policy applies to the topic and data writer entities via the durability_service member of their respective QoS structures and provides a way to specify

HISTORY and RESOURCE_LIMITS for the sample cache. Below is the IDL related to the durability service QoS policy:

```
struct DurabilityServiceQosPolicy {
  Duration_t             service_cleanup_delay;
  HistoryQosPolicyKind   history_kind;
  long                   history_depth;
  long                   max_samples;
  long                   max_instances;
  long                   max_samples_per_instance;
};
```

The history and resource limits members are analogous to, although independent of, those found in the HISTORY and RESOURCE_LIMITS policies. The service_cleanup_delay can be set to a desired value. By default, it is set to zero, which means never clean up cached samples.

3.2.7 RESOURCE_LIMITS

The RESOURCE_LIMITS policy determines the amount of resources the service can consume in order to meet the requested QoS. This policy applies to the topic, data reader, and data writer entities via the resource_limits member of their respective QoS structures. Below is the IDL related to the resource limits QoS policy.

```
struct ResourceLimitsQosPolicy {
  long max_samples;
  long max_instances;
  long max_samples_per_instance;
};
```

The max_samples member specifies the maximum number of samples a single data writer or data reader can manage across all of its instances. The max_instances member specifies the maximum number of instances that a data writer or data reader can manage. The max_samples_per_instance member specifies the maximum number of samples that can be managed for an individual instance in a single data writer or data reader. The values of all these members default to unlimited (DDS::LENGTH_UNLIMITED).

Resources are used by the data writer to queue samples written to the data writer but not yet sent to all data readers because of backpressure from the transport. Resources are used by the data reader to queue samples that have been received, but not yet read/taken from the data reader.

3.2.8 PARTITION

The PARTITION QoS policy allows the creation of logical partitions within a domain. It only allows data readers and data writers to be associated if they have matched partition strings.

OpenDDS Developer's Guide

This policy applies to the publisher and subscriber entities via the partition member of their respective QoS structures. Below is the IDL related to the partition QoS policy.

```
struct PartitionQosPolicy {
  StringSeq name;
};
```

The name member defaults to an empty sequence of strings. The default partition name is an empty string and causes the entity to participate in the default partition. The partition names may contain wildcard characters as defined by the POSIX `fnmatch` function (POSIX 1003.2-1992 section B.6).

The establishment of data reader and data writer associations depends on matching partition strings on the publication and subscription ends. Failure to match partitions is not considered a failure and does not trigger any callbacks or set any status values.

The value of this policy may be changed at any time. Changes to this policy may cause associations to be removed or added.

3.2.9 DEADLINE

The `DEADLINE` QoS policy allows the application to detect when data is not written or read within a specified amount of time. This policy applies to the topic, data writer, and data reader entities via the deadline member of their respective QoS structures. Below is the IDL related to the deadline QoS policy.

```
struct DeadlineQosPolicy {
  Duration_t period;
};
```

The default value of the `period` member is infinite, which requires no behavior. When this policy is set to a finite value, then the data writer monitors the changes to data made by the application and indicates failure to honor the policy by setting the corresponding status condition and triggering the `on_offered_deadline_missed()` listener callback. A data reader that detects that the data has not changed before the period has expired sets the corresponding status condition and triggers the `on_requested_deadline_missed()` listener callback.

This policy is considered during the creation of associations between data writers and data readers. The value of both sides of the association must be compatible in order for an association to be created. The deadline period of the data reader must be greater than or equal to the corresponding value of data writer.

The value of this policy may change after the associated entity is enabled. In the case where the policy of a data reader or data writer is made, the change is successfully applied only if

the change remains consistent with the remote end of all associations in which the reader or writer is participating. If the policy of a topic is changed, it will affect only data readers and writers that are created after the change has been made. Any existing readers or writers, and any existing associations between them, will not be affected by the topic policy value change.

3.2.10 LIFESPAN

The `LIFESPAN` QoS policy allows the application to specify when a sample expires. Expired samples will not be delivered to subscribers. This policy applies to the topic and data writer entities via the lifespan member of their respective QoS structures. Below is the IDL related to the lifespan QoS policy.

```
struct LifespanQosPolicy {
  Duration_t duration;
}
```

The default value of the `duration` member is infinite, which means samples never expire. OpenDDS currently supports expired sample detection on the publisher side when using a `DURABILITY kind` other than `VOLATILE`. The current OpenDDS implementation may not remove samples from the data writer and data reader caches when they expire after being placed in the cache.

The value of this policy may be changed at any time. Changes to this policy affect only data written after the change.

3.2.11 USER_DATA

The `USER_DATA` policy applies to the domain participant, data reader, and data writer entities via the user_data member of their respective QoS structures. Below is the IDL related to the user data QoS policy:

```
struct UserDataQosPolicy {
  sequence<octet> value;
};
```

By default, the `value` member is not set. It can be set to any sequence of octets which can be used to attach information to the created entity. The value of the `USER_DATA` policy is available in respective built-in topic data. The remote application can obtain the information via the built-in topic and use it for its own purposes. For example, the application could attach security credentials via the `USER_DATA` policy that can be used by the remote application to authenticate the source.

3.2.12 TOPIC_DATA

The `TOPIC_DATA` policy applies to topic entities via the topic_data member of TopicQoS structures. Below is the IDL related to the topic data QoS policy:

```
struct TopicDataQosPolicy {
  sequence<octet> value;
};
```

By default, the `value` is not set. It can be set to attach additional information to the created topic. The value of the `TOPIC_DATA` policy is available in data writer, data reader, and topic built-in topic data. The remote application can obtain the information via the built-in topic and use it in an application-defined way.

3.2.13 GROUP_DATA

The `GROUP_DATA` policy applies to the publisher and subscriber entities via the group_data member of their respective QoS structures. Below is the IDL related to the group data QoS policy:

```
struct GroupDataQosPolicy {
  sequence<octet> value;
};
```

By default, the `value` member is not set. It can be set to attach additional information to the created entities. The value of the `GROUP_DATA` policy is propagated via built-in topics. The data writer built-in topic data contains the `GROUP_DATA` from the publisher and the data reader built-in topic data contains the `GROUP_DATA` from the subscriber. The `GROUP_DATA` policy could be used to implement matching mechanisms similar to those of the `PARTITION` policy described in 1.1.6 except the decision could be made based on an application-defined policy.

3.2.14 TRANSPORT_PRIORITY

The `TRANSPORT_PRIORITY` policy applies to topic and data writer entities via the transport_priority member of their respective QoS policy structures. Below is the IDL related to the TransportPriority QoS policy:

```
struct TransportPriorityQosPolicy {
  long value;
};
```

The default value member of `transport_priority` is zero. This policy is considered a hint to the transport layer to indicate at what priority to send messages. Higher values indicate higher priority. OpenDDS maps the priority value directly onto thread and DiffServ

codepoint values. A default priority of zero will not modify either threads or codepoints in messages.

OpenDDS will attempt to set the thread priority of the sending transport as well as any associated receiving transport. Transport priority values are mapped from zero (default) through the maximum thread priority linearly without scaling. If the lowest thread priority is different from zero, then it is mapped to the transport priority value of zero. Where priority values on a system are inverted (higher numeric values are lower priority), OpenDDS maps these to an increasing priority value starting at zero. Priority values lower than the minimum (lowest) thread priority on a system are mapped to that lowest priority. Priority values greater than the maximum (highest) thread priority on a system are mapped to that highest priority. On most systems, thread priorities can only be set when the process scheduler has been set to allow these operations. Setting the process scheduler is generally a privileged operation and will require system privileges to perform. On POSIX based systems, the system calls of `sched_get_priority_min()` and `sched_get_priority_max()` are used to determine the system range of thread priorities.

OpenDDS will attempt to set the DiffServ codepoint on the socket used to send data for the data writer if it is supported by the transport implementation. If the network hardware honors the codepoint values, higher codepoint values will result in better (faster) transport for higher priority samples. The default value of zero will be mapped to the (default) codepoint of zero. Priority values from 1 through 63 are then mapped to the corresponding codepoint values, and higher priority values are mapped to the highest codepoint value (63).

OpenDDS does not currently support modifications of the transport_priority policy values after creation of the data writer. This can be worked around by creating new data writers as different priority values are required.

3.2.15   LATENCY_BUDGET

The `LATENCY_BUDGET` policy applies to topic, data reader, and data writer entities via the latency_budget member of their respective QoS policy structures. Below is the IDL related to the LatencyBudget QoS policy:

```
struct LatencyBudgetQosPolicy {
  Duration_t duration;
};
```

The default value of `duration` is zero indicating that the delay should be minimized. This policy is considered a hint to the transport layer to indicate the urgency of samples being sent. OpenDDS uses the value to bound a delay interval for reporting unacceptable delay in transporting samples from publication to subscription. This policy is used for monitoring

purposes only at this time. Use the `TRANSPORT_PRIORITY` policy to modify the sending of samples. The data writer policy value is used only for compatibility comparisons and if left at the default value of zero will result in all requested duration values from data readers being matched.

An additional listener extension has been added to allow reporting delays in excess of the policy duration setting. The `OpenDDS::DCPS::DataReaderListener` interface has an additional operation for notification that samples were received with a measured transport delay greater than the latency_budget policy duration. The IDL for this method is:

```
struct BudgetExceededStatus {
  long total_count;
  long total_count_change;
  DDS::InstanceHandle_t last_instance_handle;
};

void on_budget_exceeded(
        in DDS::DataReader reader,
        in BudgetExceededStatus status);
```

To use the extended listener callback you will need to derive the listener implementation from the extended interface, as shown in the following code fragment:

```
class DataReaderListenerImpl
      : public virtual
        OpenDDS::DCPS::LocalObject<OpenDDS::DCPS::DataReaderListener>
```

Then you must provide a non-null implementation for the on_budget_exceeded() operation. Note that you will need to provide empty implementations for the following extended operations as well:

```
on_subscription_disconnected()
on_subscription_reconnected()
on_subscription_lost()
on_connection_deleted()
```

OpenDDS also makes the summary latency statistics available via an extended interface of the data reader. This extended interface is located in the `OpenDDS::DCPS` module and the IDL is defined as:

```
struct LatencyStatistics {
  GUID_t        publication;
  unsigned long n;
  double        maximum;
  double        minimum;
  double        mean;
  double        variance;
};

typedef sequence<LatencyStatistics> LatencyStatisticsSeq;

local interface DataReaderEx : DDS::DataReader {
```

```
    /// Obtain a sequence of statistics summaries.
    void get_latency_stats( inout LatencyStatisticsSeq stats);

    /// Clear any intermediate statistical values.
    void reset_latency_stats();

    /// Statistics gathering enable state.
    attribute boolean statistics_enabled;
  };
```

To gather this statistical summary data you will need to use the extended interface. You can do so simply by dynamically casting the OpenDDS data reader pointer and calling the operations directly. In the following example, we assume that reader is initialized correctly by calling DDS::Subscriber::create_datareader():

```
  DDS::DataReader_var reader;
  // ...

  // To start collecting new data.
  dynamic_cast<OpenDDS::DCPS::DataReaderImpl*>(reader.in())->
    reset_latency_stats();
  dynamic_cast<OpenDDS::DCPS::DataReaderImpl*>(reader.in())->
    statistics_enabled(true);

  // ...

  // To collect data.
  OpenDDS::DCPS::LatencyStatisticsSeq stats;
  dynamic_cast<OpenDDS::DCPS::DataReaderImpl*>(reader.in())->
    get_latency_stats(stats);
  for (unsigned long i = 0; i < stats.length(); ++i)
  {
    std::cout << "stats[" << i << "]:" << std::endl;
    std::cout << "         n = " << stats[i].n << std::endl;
    std::cout << "       max = " << stats[i].maximum << std::endl;
    std::cout << "       min = " << stats[i].minimum << std::endl;
    std::cout << "      mean = " << stats[i].mean << std::endl;
    std::cout << "  variance = " << stats[i].variance << std::endl;
  }
```

3.2.16   ENTITY_FACTORY

The ENTITY_FACTORY policy controls whether entities are automatically enabled when they are created. Below is the IDL related to the Entity Factory QoS policy:

```
struct EntityFactoryQosPolicy {
  boolean autoenable_created_entities;
};
```

This policy can be applied to entities that serve as factories for other entities and controls whether or not entities created by those factories are automatically enabled upon creation. This policy can be applied to the domain participant factory (as a factory for domain participants), domain participant (as a factory for publishers, subscribers, and topics), publisher (as a factory for data writers), or subscriber (as a factory for data readers). The

default value for the `autoenable_created_entities` member is `true`, indicating that entities are automatically enabled when they are created. Applications that wish to explicitly enable entities some time after they are created should set the value of the `autoenable_created_entities` member of this policy to `false` and apply the policy to the appropriate factory entities. The application must then manually enable the entity by calling the entity's `enable()` operation.

The value of this policy may be changed at any time. Changes to this policy affect only entities created after the change.

3.2.17 PRESENTATION

The `PRESENTATION` QoS policy controls how changes to instances by publishers are presented to data readers. It affects the relative ordering of these changes and the scope of this ordering. Additionally, this policy introduces the concept of coherent change sets. Here is the IDL for the Presentation QoS:

```
enum PresentationQosPolicyAccessScopeKind {
  INSTANCE_PRESENTATION_QOS,
  TOPIC_PRESENTATION_QOS,
  GROUP_PRESENTATION_QOS
};

struct PresentationQosPolicy {
  PresentationQosPolicyAccessScopeKind access_scope;
  boolean coherent_access;
  boolean ordered_access;
};
```

The scope of these changes (`access_scope`) specifies the level in which an application may be made aware:

–   `INSTANCE_PRESENTATION_QOS` (the default) indicates that changes occur to instances independently. Instance access essentially acts as a no-op with respect to coherent_access and ordered_access. Setting either of these values to true has no observable affect within the subscribing application.

–   `TOPIC_PRESENTATION_QOS` indicates that accepted changes are limited to all instances within the same data reader or data writer.

–   `GROUP_PRESENTATION_QOS` indicates that accepted changes are limited to all instances within the same publisher or subscriber.

Coherent changes (`coherent_access`) allow one or more changes to an instance be made available to an associated data reader as a single change. If a data reader does not receive the entire set of coherent changes made by a publisher, then none of the changes are made

available. The semantics of coherent changes are similar in nature to those found in transactions provided by many relational databases. By default, `coherent_access` is `false`.

Changes may also be made available to associated data readers in the order sent by the publisher (`ordered_access`). This is similar in nature to the `DESTINATION_ORDER QoS` policy, however `ordered_access` permits data to be ordered independently of instance ordering. By default, `ordered_access` is `false`.

| Note | *This policy controls the ordering and scope of samples made available to the subscriber, but the subscriber application must use the proper logic in reading samples to guarantee the requested behavior. For more details, see Section 2.2.2.5.1.9 of the Version 1.4 DDS Specification.* | **Note** |

3.2.18   DESTINATION_ORDER

The `DESTINATION_ORDER` QoS policy controls the order in which samples within a given instance are made available to a data reader. If a history depth of one (the default) is specified, the instance will reflect the most recent value written by all data writers to that instance. Here is the IDL for the Destination Order Qos:

```
enum DestinationOrderQosPolicyKind {
  BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS,
  BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
};

struct DestinationOrderQosPolicy {
  DestinationOrderQosPolicyKind kind;
};
```

The `BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS` value (the default) indicates that samples within an instance are ordered in the order in which they were received by the data reader. Note that samples are not necessarily received in the order sent by the same data writer. To enforce this type of ordering, the `BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS` value should be used.

The `BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS` value indicates that samples within an instance are ordered based on a timestamp provided by the data writer. It should be noted that if multiple data writers write to the same instance, care should be taken to ensure that clocks are synchronized to prevent incorrect ordering on the data reader.

3.2.19   WRITER_DATA_LIFECYCLE

The `WRITER_DATA_LIFECYCLE` QoS policy controls the lifecycle of data instances managed by a data writer. Here is the IDL for the Writer Data Lifecycle QoS policy:

```
struct WriterDataLifecycleQosPolicy {
  boolean autodispose_unregistered_instances;
};
```

When `autodispose_unregistered_instances` is set to `true` (the default), a data writer disposes an instance when it is unregistered. In some cases, it may be desirable to prevent an instance from being disposed when an instance is unregistered. This policy could, for example, allow an `EXCLUSIVE` data writer to gracefully defer to the next data writer without affecting the instance state. Deleting a data writer implicitly unregisters all of its instances prior to deletion.

3.2.20 READER_DATA_LIFECYCLE

The `READER_DATA_LIFECYCLE` QoS policy controls the lifecycle of data instances managed by a data reader. Here is the IDL for the Reader Data Lifecycle QoS policy:

```
struct ReaderDataLifecycleQosPolicy {
  Duration_t autopurge_nowriter_samples_delay;
  Duration_t autopurge_disposed_samples_delay;
};
```

Normally, a data reader maintains data for all instances until there are no more associated data writers for the instance, the instance has been disposed, or the data has been taken by the user.

In some cases, it may be desirable to constrain the reclamation of these resources. This policy could, for example, permit a late-joining data writer to prolong the lifetime of an instance in fail-over situations.

The `autopurge_nowriter_samples_delay` controls how long the data reader waits before reclaiming resources once an instance transitions to the `NOT_ALIVE_NO_WRITERS` state. By default, `autopurge_nowriter_samples_delay` is infinite.

The `autopurge_disposed_samples_delay` controls how long the data reader waits before reclaiming resources once an instance transitions to the `NOT_ALIVE_DISPOSED` state. By default, `autopurge_disposed_samples_delay` is infinite.

3.2.21 TIME_BASED_FILTER

The `TIME_BASED_FILTER` QoS policy controls how often a data reader may be interested in changes in values to a data instance. Here is the IDL for the Time Based Filter QoS:

```
struct TimeBasedFilterQosPolicy {
  Duration_t minimum_separation;
};
```

An interval (`minimum_separation`) may be specified on the data reader. This interval defines a minimum delay between instance value changes; this permits the data reader to throttle changes without affecting the state of the associated data writer. By default, minimum_separation is zero, which indicates that no data is filtered. This QoS policy does not conserve bandwidth as instance value changes are still sent to the subscriber process. It only affects which samples are made available via the data reader.

3.2.22   OWNERSHIP

The `OWNERSHIP` policy controls whether more than one Data Writer is able to write samples for the same data-object instance. Ownership can be `EXCLUSIVE` or `SHARED`. Below is the IDL related to the Ownership QoS policy:

```
enum OwnershipQosPolicyKind {
  SHARED_OWNERSHIP_QOS,
  EXCLUSIVE_OWNERSHIP_QOS
};

struct OwnershipQosPolicy {
  OwnershipQosPolicyKind kind;
};
```

If the kind member is set to `SHARED_OWNERSHIP_QOS`, more than one Data Writer is allowed to update the same data-object instance. If the kind member is set to `EXCLUSIVE_OWNERSHIP_QOS`, only one Data Writer is allowed to update a given data-object instance (i.e., the Data Writer is considered to be the *owner* of the instance) and associated Data Readers will only see samples written by that Data Writer. The owner of the instance is determined by value of the `OWNERSHIP_STRENGTH` policy; the data writer with the highest value of strength is considered the owner of the data-object instance. Other factors may also influence ownership, such as whether the data writer with the highest strength is "alive" (as defined by the `LIVELINESS` policy) and has not violated its offered publication deadline constraints (as defined by the `DEADLINE` policy).

3.2.23   OWNERSHIP_STRENGTH

The `OWNERSHIP_STRENGTH` policy is used in conjunction with the `OWNERSHIP` policy, when the `OWNERSHIP` kind is set to `EXCLUSIVE`. Below is the IDL related to the Ownership Strength QoS policy:

```
struct OwnershipStrengthQosPolicy {
  long value;
};
```

The value member is used to determine which Data Writer is the *owner* of the data-object instance. The default value is zero.

3.3 Policy Example

The following sample code illustrates some policies being set and applied for a publisher.

```
DDS::DataWriterQos dw_qos;
pub->get_default_datawriter_qos (dw_qos);

dw_qos.history.kind = DDS::KEEP_ALL_HISTORY_QOS;

dw_qos.reliability.kind = DDS::RELIABLE_RELIABILITY_QOS;
dw_qos.reliability.max_blocking_time.sec = 10;
dw_qos.reliability.max_blocking_time.nanosec = 0;

dw_qos.resource_limits.max_samples_per_instance = 100;

DDS::DataWriter_var dw =
  pub->create_datawriter(topic,
                         dw_qos,
                         0,    // No listener
                         OpenDDS::DCPS::DEFAULT_STATUS_MASK);
```

This code creates a publisher with the following qualities:

- HISTORY set to Keep All

- RELIABILITY set to Reliable with a maximum blocking time of 10 seconds

- The maximum samples per instance resource limit set to 100

This means that when 100 samples are waiting to be delivered, the writer can block up to 10 seconds before returning an error code. These same QoS settings on the Data Reader side would mean that up to 100 unread samples are queued by the framework before any are rejected. Rejected samples are dropped and the SampleRejectedStatus is updated.

CHAPTER 4

Conditions and Listeners

4.1   Introduction

The DDS specification defines two separate mechanisms for notifying applications of DCPS communication status changes. Most of the status types define a structure that contains information related to the change of status and can be detected by the application using conditions or listeners. The different status types are described in 4.2.

Each entity type (domain participant, topic, publisher, subscriber, data reader, and data writer) defines its own corresponding listener interface. Applications can implement this interface and then attach their listener implementation to the entity. Each listener interface contains an operation for each status that can be reported for that entity. The listener is asynchronously called back with the appropriate operation whenever a qualifying status change occurs. Details of the different listener types are discussed in 4.3.

Conditions are used in conjunction with Wait Sets to let applications synchronously wait on events. The basic usage pattern for conditions involves creating the condition objects, attaching them to a wait set, and then waiting on the wait set until one of the conditions is triggered. The result of wait tells the application which conditions were triggered, allowing the application to take the appropriate actions to get the corresponding status information. Conditions are described in greater detail in 4.4.

4.2 Communication Status Types

Each status type is associated with a particular entity type. This section is organized by the entity types, with the corresponding statuses described in subsections under the associated entity type.

Most of the statuses below are plain communication statuses. The exceptions are `DATA_ON_READERS` and `DATA_AVAILABLE` which are read statuses. Plain communication statuses define an IDL data structure. Their corresponding section below describes this structure and its fields. The read statuses are simple notifications to the application which then reads or takes the samples as desired.

Incremental values in the status data structure report a change since the last time the status was accessed. A status is considered accessed when a listener is called for that status or the status is read from its entity.

Fields in the status data structure with a type of `InstanceHandle_t` identify an entity (topic, data reader, data writer, etc.) by the instance handle used for that entity in the Built-In-Topics.

4.2.1 Topic Status Types

4.2.1.1 Inconsistent Topic Status

The `INCONSISTENT_TOPIC` status indicates that a topic was attempted to be registered that already exists with different characteristics. Typically, the existing topic may have a different type associated with it. The IDL associated with the Inconsistent Topic Status is listed below:

```
struct InconsistentTopicStatus {
  long total_count;
  long total_count_change;
};
```

The `total_count` value is the cumulative count of topics that have been reported as inconsistent. The `total_count_change` value is the incremental count of inconsistent topics since the last time this status was accessed.

4.2.2 Subscriber Status Types

4.2.2.1 Data On Readers Status

The `DATA_ON_READERS` status indicates that new data is available on some of the data readers associated with the subscriber. This status is considered a read status and does not define an IDL structure. Applications receiving this status can call `get_datareaders()` on the subscriber to get the set of data readers with data available.

4.2.3 Data Reader Status Types

4.2.3.1 Sample Rejected Status

The `SAMPLE_REJECTED` status indicates that a sample received by the data reader has been rejected. The IDL associated with the Sample Rejected Status is listed below:

```
enum SampleRejectedStatusKind {
  NOT_REJECTED,
  REJECTED_BY_INSTANCES_LIMIT,
  REJECTED_BY_SAMPLES_LIMIT,
  REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT
};

struct SampleRejectedStatus {
  long total_count;
  long total_count_change;
  SampleRejectedStatusKind last_reason;
  InstanceHandle_t last_instance_handle;
};
```

The `total_count` value is the cumulative count of samples that have been reported as rejected. The `total_count_change` value is the incremental count of rejected samples since the last time this status was accessed. The `last_reason` value is the reason the most recently rejected sample was rejected. The `last_instance_handle` value indicates the instance of the last rejected sample.

4.2.3.2 Liveliness Changed Status

The `LIVELINESS_CHANGED` status indicates that there have been liveliness changes for one or more data writers that are publishing instances for this data reader. The IDL associated with the Liveliness Changed Status is listed below:

```
struct LivelinessChangedStatus {
  long alive_count;
  long not_alive_count;
  long alive_count_change;
  long not_alive_count_change;
```

```
  InstanceHandle_t last_publication_handle;
};
```

The `alive_count` value is the total number of data writers currently active on the topic this data reader is reading. The `not_alive_count` value is the total number of data writers writing to the data reader's topic that are no longer asserting their liveliness. The `alive_count_change` value is the change in the alive count since the last time the status was accessed. The `not_alive_count_change` value is the change in the not alive count since the last time the status was accessed. The `last_publication_handle` is the handle of the last data writer whose liveliness has changed.

4.2.3.3 Requested Deadline Missed Status

The `REQUESTED_DEADLINE_MISSED` status indicates that the deadline requested via the Deadline QoS policy was not respected for a specific instance. The IDL associated with the Requested Deadline Missed Status is listed below:

```
struct RequestedDeadlineMissedStatus {
  long total_count;
  long total_count_change;
  InstanceHandle_t last_instance_handle;
};
```

The `total_count` value is the cumulative count of missed requested deadlines that have been reported. The `total_count_change` value is the incremental count of missed requested deadlines since the last time this status was accessed. The `last_instance_handle` value indicates the instance of the last missed deadline.

4.2.3.4 Requested Incompatible QoS Status

The `REQUESTED_INCOMPATIBLE_QOS` status indicates that one or more QoS policy values that were requested were incompatible with what was offered. The IDL associated with the Requested Incompatible QoS Status is listed below:

```
struct QosPolicyCount {
  QosPolicyId_t policy_id;
  long count;
};

typedef sequence<QosPolicyCount> QosPolicyCountSeq;

struct RequestedIncompatibleQosStatus {
  long total_count;
  long total_count_change;
  QosPolicyId_t last_policy_id;
  QosPolicyCountSeq policies;
};
```

The total_count value is the cumulative count of times data writers with incompatible QoS have been reported. The total_count_change value is the incremental count of incompatible data writers since the last time this status was accessed. The last_policy_id value identifies one of the QoS policies that was incompatible in the last incompatibility detected. The policies value is a sequence of values that indicates the total number of incompatibilities that have been detected for each QoS policy.

4.2.3.5 Data Available Status

The DATA_AVAILABLE status indicates that samples are available on the data writer. This status is considered a read status and does not define an IDL structure. Applications receiving this status can use the various take and read operations on the data reader to retrieve the data.

4.2.3.6 Sample Lost Status

The SAMPLE_LOST status indicates that a sample has been lost and never received by the data reader. The IDL associated with the Sample Lost Status is listed below:

```
struct SampleLostStatus {
  long total_count;
  long total_count_change;
};
```

The total_count value is the cumulative count of samples reported as lost. The total_count_change value is the incremental count of lost samples since the last time this status was accessed.

4.2.3.7 Subscription Matched Status

The SUBSCRIPTION_MATCHED status indicates that either a compatible data writer has been matched or a previously matched data writer has ceased to be matched. The IDL associated with the Subscription Matched Status is listed below:

```
struct SubscriptionMatchedStatus {
  long total_count;
  long total_count_change;
  long current_count;
  long current_count_change;
  InstanceHandle_t last_publication_handle;
};
```

The total_count value is the cumulative count of data writers that have compatibly matched this data reader. The total_count_change value is the incremental change in the total count since the last time this status was accessed. The current_count value is the

current number of data writers matched to this data reader. The `current_count_change` value is the change in the current count since the last time this status was accessed. The `last_publication_handle` value is a handle for the last data writer matched.

4.2.4 Data Writer Status Types

4.2.4.1 Liveliness Lost Status

The `LIVELINESS_LOST` status indicates that the liveliness that the data writer committed through its Liveliness QoS has not been respected. This means that any connected data readers will consider this data writer no longer active.The IDL associated with the Liveliness Lost Status is listed below:

```
struct LivelinessLostStatus {
  long total_count;
  long total_count_change;
};
```

The `total_count` value is the cumulative count of times that an alive data writer has become not alive. The `total_count_change` value is the incremental change in the total count since the last time this status was accessed.

4.2.4.2 Offered Deadline Missed Status

The `OFFERED_DEADLINE_MISSED` status indicates that the deadline offered by the data writer has been missed for one or more instances. The IDL associated with the Offered Deadline Missed Status is listed below:

```
struct OfferedDeadlineMissedStatus {
  long total_count;
  long total_count_change;
  InstanceHandle_t last_instance_handle;
};
```

The `total_count` value is the cumulative count of times that deadlines have been missed for an instance. The `total_count_change` value is the incremental change in the total count since the last time this status was accessed. The `last_instance_handle` value indicates the last instance that has missed a deadline.

4.2.4.3 Offered Incompatible QoS Status

The `OFFERED_INCOMPATIBLE_QOS` status indicates that an offered QoS was incompatible with the requested QoS of a data reader. The IDL associated with the Offered Incompatible QoS Status is listed below:

```
struct QosPolicyCount {
  QosPolicyId_t policy_id;
  long count;
};
typedef sequence<QosPolicyCount> QosPolicyCountSeq;

struct OfferedIncompatibleQosStatus {
  long total_count;
  long total_count_change;
  QosPolicyId_t last_policy_id;
  QosPolicyCountSeq policies;
};
```

The `total_count` value is the cumulative count of times that data readers with incompatible QoS have been found. The `total_count_change` value is the incremental change in the total count since the last time this status was accessed. The `last_policy_id` value identifies one of the QoS policies that was incompatible in the last incompatibility detected. The `policies` value is a sequence of values that indicates the total number of incompatibilities that have been detected for each QoS policy.

4.2.4.4 Publication Matched Status

The `PUBLICATION_MATCHED` status indicates that either a compatible data reader has been matched or a previously matched data reader has ceased to be matched. The IDL associated with the Publication Matched Status is listed below:

```
struct PublicationMatchedStatus {
  long total_count;
  long total_count_change;
  long current_count;
  long current_count_change;
  InstanceHandle_t last_subscription_handle;
};
```

The `total_count` value is the cumulative count of data readers that have compatibly matched this data writer. The `total_count_change` value is the incremental change in the total count since the last time this status was accessed. The `current_count` value is the current number of data readers matched to this data writer. The `current_count_change` value is the change in the current count since the last time this status was accessed. The `last_subscription_handle` value is a handle for the last data reader matched.

4.3 Listeners

Each entity defines its own listener interface based on the statuses it can report. Any entity's listener interface also inherits from the listeners of its owned entities, allowing it to handle statuses for owned entities as well. For example, a subscriber listener directly

defines an operation to handle Data On Readers statuses and inherits from the data reader listener as well.

Each status operation takes the general form of on_<status_name>(<entity>, <status_struct>), where <status_name> is the name of the status being reported, <entity> is a reference to the entity the status is reported for, and <status_struct> is the structure with details of the status. Read statuses omit the second parameter. For example, here is the operation for the Sample Lost status:

```
void on_sample_lost(in DataReader the_reader, in SampleLostStatus status);
```

Listeners can either be passed to the factory function used to create their entity or explicitly set by calling set_listener() on the entity after it is created. Both of these functions also take a status mask as a parameter. The mask indicates which statuses are enabled in that listener. Mask bit values for each status are defined in DdsDcpsInfrastructure.idl:

```
module DDS {
  typedef unsigned long StatusKind;
  typedef unsigned long StatusMask; // bit-mask StatusKind

  const StatusKind INCONSISTENT_TOPIC_STATUS       = 0x0001 << 0;
  const StatusKind OFFERED_DEADLINE_MISSED_STATUS   = 0x0001 << 1;
  const StatusKind REQUESTED_DEADLINE_MISSED_STATUS = 0x0001 << 2;
  const StatusKind OFFERED_INCOMPATIBLE_QOS_STATUS  = 0x0001 << 5;
  const StatusKind REQUESTED_INCOMPATIBLE_QOS_STATUS= 0x0001 << 6;
  const StatusKind SAMPLE_LOST_STATUS               = 0x0001 << 7;
  const StatusKind SAMPLE_REJECTED_STATUS           = 0x0001 << 8;
  const StatusKind DATA_ON_READERS_STATUS           = 0x0001 << 9;
  const StatusKind DATA_AVAILABLE_STATUS            = 0x0001 << 10;
  const StatusKind LIVELINESS_LOST_STATUS           = 0x0001 << 11;
  const StatusKind LIVELINESS_CHANGED_STATUS        = 0x0001 << 12;
  const StatusKind PUBLICATION_MATCHED_STATUS       = 0x0001 << 13;
  const StatusKind SUBSCRIPTION_MATCHED_STATUS      = 0x0001 << 14;
};
```

Simply do a bit-wise "or" of the desired status bits to construct a mask for your listener.

Here is an example of attaching a listener to a data reader (for just Data Available statuses):

```
DDS::DataReaderListener_var listener (new DataReaderListenerImpl);
// Create the Datareader
DDS::DataReader_var dr = sub->create_datareader(
  topic,
  DATAREADER_QOS_DEFAULT,
  listener,
  DDS::DATA_AVAILABLE_STATUS);
```

Here is an example showing how to change the listener using set_listener():

```
dr->set_listener(listener,
                 DDS::DATA_AVAILABLE_STATUS |
                 DDS::LIVELINESS_CHANGED_STATUS);
```

When a plain communication status changes, OpenDDS invokes the most specific relevant listener operation. This means, for example, that a data reader's listener would take precedence over the subscriber's listener for statuses related to the data reader.

The following sections define the different listener interfaces. For more details on the individual statuses, see 4.2.

4.3.1 Topic Listener

```
interface TopicListener : Listener {
  void on_inconsistent_topic(in Topic the_topic,
                             in InconsistentTopicStatus status);
};
```

4.3.2 Data Writer Listener

```
interface DataWriterListener : Listener {
  void on_offered_deadline_missed(in DataWriter writer,
                                  in OfferedDeadlineMissedStatus status);
  void on_offered_incompatible_qos(in DataWriter writer,
                                   in OfferedIncompatibleQosStatus status);
  void on_liveliness_lost(in DataWriter writer,
                          in LivelinessLostStatus status);
  void on_publication_matched(in DataWriter writer,
                              in PublicationMatchedStatus status);
};
```

4.3.3 Publisher Listener

```
interface PublisherListener : DataWriterListener {
};
```

4.3.4 Data Reader Listener

```
interface DataReaderListener : Listener {
  void on_requested_deadline_missed(in DataReader the_reader,
                                    in RequestedDeadlineMissedStatus status);
  void on_requested_incompatible_qos(in DataReader the_reader,
                                     in RequestedIncompatibleQosStatus status);
  void on_sample_rejected(in DataReader the_reader,
                          in SampleRejectedStatus status);
  void on_liveliness_changed(in DataReader the_reader,
                             in LivelinessChangedStatus status);
  void on_data_available(in DataReader the_reader);
  void on_subscription_matched(in DataReader the_reader,
                               in SubscriptionMatchedStatus status);
  void on_sample_lost(in DataReader the_reader,
                      in SampleLostStatus status);
};
```

4.3.5 Subscriber Listener

```
interface SubscriberListener : DataReaderListener {
  void on_data_on_readers(in Subscriber the_subscriber);
};
```

4.3.6 Domain Participant Listener

```
interface DomainParticipantListener : TopicListener,
                                      PublisherListener,
                                      SubscriberListener {
};
```

4.4 Conditions

The DDS specification defines four types of condition:

- Status Condition
- Read Condition
- Query Condition
- Guard Condition

4.4.1 Status Condition

Each entity has a status condition object associated with it and a `get_statuscondition()` operation that lets applications access the status condition. Each condition has a set of enabled statuses that can trigger that condition. Attaching one or more conditions to a wait set allows application developers to wait on the condition's status set. Once an enabled status is triggered, the wait call returns from the wait set and the developer can query the relevant status condition on the entity. Querying the status condition resets the status.

4.4.1.1 Status Condition Example

This example enables the Offered Incompatible QoS status on a data writer, waits for it, and then queries it when it triggers. The first step is to get the status condition from the data writer, enable the desired status, and attach it to a wait set:

```
DDS::StatusCondition_var cond = data_writer->get_statuscondition();
cond->set_enabled_statuses(DDS::OFFERED_INCOMPATIBLE_QOS_STATUS);

DDS::WaitSet_var ws = new DDS::WaitSet;
ws->attach_condition(cond);
```

Now we can wait ten seconds for the condition:

OpenDDS Developer's Guide

```
DDS::ConditionSeq active;
DDS::Duration_t ten_seconds = {10, 0};
int result = ws->wait(active, ten_seconds);
```

The result of this operation is either a timeout or a set of triggered conditions in the active sequence:

```
if (result == DDS::RETCODE_TIMEOUT) {
  cout << "Wait timed out" << std::endl;
} else if (result == DDS::RETCODE_OK) {
  DDS::OfferedIncompatibleQosStatus incompatibleStatus;
  data_writer->get_offered_incompatible_qos(incompatibleStatus);
  // Access status fields as desired...
}
```

Developers have the option of attaching multiple conditions to a single wait set as well as enabling multiple statuses per condition.

4.4.2   Additional Condition Types

The DDS specification also defines three other types of conditions: read conditions, query conditions, and guard conditions. These conditions do not directly involve the processing of statuses but allow the integration of other activities into the condition and wait set mechanisms. These are other conditions are briefly described here. For more information see the DDS specification or the OpenDDS tests in $DDS_ROOT/tests/.

4.4.2.1   Read Conditions

Read conditions are created using the data reader and the same masks that are passed to the read and take operations. When waiting on this condition, it is triggered whenever samples match the specified masks. Those samples can then be retrieved using the `read_w_condition()` and `take_w_condition()` operations which take the read condition as a parameter.

4.4.2.2   Query Conditions

Query conditions are a specialized form of read conditions that are created with a limited form of an SQL-like query. This allows applications to filter the data samples that trigger the condition and then are read use the normal read condition mechanisms. See Section 5.3 for more information about query conditions.

4.4.2.3   Guard Conditions

The guard condition is a simple interface that allows the application to create its own condition object and trigger it when application events (external to OpenDDS) occur.

CHAPTER 5

Content-Subscription Profile

5.1 Introduction

The Content-Subscription Profile of DDS consists of three features which enable a data reader's behavior to be influenced by the content of the data samples it receives. These three features are:

- Content-Filtered Topic
- Query Condition
- Multi Topic

The content-filtered topic and multi topic interfaces inherit from the `TopicDescription` interface (and not from the `Topic` interface, as the names may suggest).

Content-filtered topic and query condition allow filtering (selection) of data samples using a SQL-like parameterized query string. Additionally, query condition allows sorting the result set returned from a data reader's `read()` or `take()` operation. Multi topic also has this selection capability as well as the ability to aggregate data from different data writers into a single data type and data reader.

If you are not planning on using the Content-Subscription Profile features in your application, you can configure OpenDDS to remove support for it at build time. See page 15 for information on disabling this support.

5.2 Content-Filtered Topic

The domain participant interface contains operations for creating and deleting a content-filtered topic. Creating a content-filtered topic requires the following parameters:

- Name

  Assigns a name to this content-filtered topic which could later be used with the `lookup_topicdescription()` operation.

- Related topic

  Specifies the topic that this content-filtered topic is based on. This is the same topic that matched data writers will use to publish data samples.

- Filter expression

  An SQL-like expression (see section 5.2.1) which defines the subset of samples published on the related topic that should be received by the content-filtered topic's data readers.

- Expression parameters

  The filter expression can contain parameter placeholders. This argument provides initial values for those parameters. The expression parameters can be changed after the content-filtered topic is created (the filter expression cannot be changed).

Once the content-filtered topic has been created, it is used by the subscriber's `create_datareader()` operation to obtain a content-filtering data reader. This data reader is functionally equivalent to a normal data reader except that incoming data samples which do not meet the filter expression's criteria are dropped.

Filter expressions are first evaluated at the publisher so that data samples which would be ignored by the subscriber can be dropped before even getting to the transport. This feature can be turned off with `-DCPSPublisherContentFilter 0` or the equivalent setting in the `[common]` section of the configuration file. The behavior of non-default `DEADLINE` or `LIVELINESS` QoS policies may be affected by this policy. Special consideration must be given to how the "missing" samples impact the QoS behavior, see the document in `docs/design/CONTENT_SUBSCRIPTION`.

Note *RTPS_UDP transport does not always do Writer-side filtering. It does not currently implement transport level filtering, but may be able to filter above the transport layer.*

5.2.1   Filter Expressions

The formal grammar for filter expressions is defined in Annex A of the DDS specification. This section provides an informal summary of that grammar. Query expressions (5.3.1) and topic expressions (5.4.1) are also defined in Annex A.

Filter expressions are combinations of one or more predicates. Each predicate is a logical expression taking one of two forms:

- `<arg1> <RelOp> <arg2>`

  - `arg1` and `arg2` are arguments which may be either a literal value (integer, character, floating-point, string, or enumeration), a parameter placeholder of the form `%n` (where n is a zero-based index into the parameter sequence), or a field reference.

  - At least one of the arguments must be a field reference, which is the name of an IDL struct field, optionally followed by any number of '.' and another field name to represent nested structures.

  - `RelOp` is a relational operator from the list: `=`, `>`, `>=`, `<`, `<=`, `<>`, and 'like'. 'like' is a wildcard match using `%` to match any number of characters and `_` to match a single character.

  - Examples of this form of predicate include: `a = 'z'`, `b <> 'str'`, `c < d`, `e = 'enumerator'`, `f >= 3.14e3`, `27 > g`, `h <> i`, `j.k.l like %0`

- `<arg1> [NOT] BETWEEN <arg2> AND <arg3>`

  - In this form, argument 1 must be a field reference and arguments 2 and 3 must each be a literal value or parameter placeholder.

Any number of predicates can be combined through the use of parenthesis and the Boolean operators `AND`, `OR`, and `NOT` to form a filter expression.

5.2.2   Expression Parameters

Expression parameters allow more flexibility since the filter can effectively change at runtime. To use expression parameters, add parameter placeholders in the filter expression wherever a literal would be used. For example, an expression to select all samples that have a string field with a fixed value (`m = 'A'`) could instead use a placeholder which would be written as `m = %0`. Placeholders consist of a percent sign followed by a decimal integer between 0 and 99 inclusive.

Using a filter that contains placeholders requires values for each placeholder which is used in the expression to be provided by the application in the corresponding index of the expression parameters sequence (placeholder %0 is sequence[0]). The application can set the parameter sequence when the content-filtered topic is created (`create_contentfilteredtopic`) or after it already exists by using `set_expression_parameters`. A valid value for each used placeholder must be in the parameters sequence whenever the filter is evaluated, for example when a data reader using the content-filtered topic is enabled.

The type used for the parameters sequence in the DDS-DCPS API is a sequence of strings. The application must format this string based on how the parameter is used:

- For a number (integer or floating point), provide the decimal representation in the same way it would appear as a C++ or Java literal.

- For a character or string, provide the character(s) directly without quoting

- For an enumerated type, provide one of the enumerators as if it was a string

5.2.3 Filtering And Dispose/Unregister Samples

DataReaders without filtering can see samples with the `valid_data` field of SampleInfo set to false. This happens when the matching DataWriter disposes or unregisters the instance. Content filtering (whether achieved through Content-Filtered Topics, Query Conditions, or Multi Topics) will filter such samples when the filter expression explicitly uses key fields. Filter expressions that don't meet that criteria will result in no such samples passing the filter.

5.2.4 Content-Filtered Topic Example

The code snippet below creates a content-filtered topic for the Message type. First, here is the IDL for Message:

```
module Messenger {
  @topic
  struct Message {
    long id;
  };
};
```

Next we have the code that creates the data reader:

```
CORBA::String_var type_name = message_type_support->get_type_name();
DDS::Topic_var topic = dp->create_topic("MyTopic",
                                        type_name,
                                        TOPIC_QOS_DEFAULT, 0, 0);
DDS::ContentFilteredTopic_var cft =
  participant->create_contentfilteredtopic("MyTopic-Filtered",
```

```
                                        topic,
                                        "id > 1",
                                        StringSeq());
    DDS::DataReader_var dr =
      subscriber->create_datareader(cft,
                                    DATAREADER_QOS_DEFAULT, 0, 0);
```

The data reader 'dr' will only receive samples that have values of 'id' greater than 1.

5.3   Query Condition

The query condition interface inherits from the read condition interface, therefore query conditions have all of the capabilities of read conditions along with the additional capabilities described in this section. One of those inherited capabilities is that the query condition can be used like any other condition with a wait set (see Section 4.4).

The `DataReader` interface contains operations for creating (`create_querycondition`) and deleting (`delete_readcondition`) a query condition. Creating a query condition requires the following parameters:

- Sample, view, and instance state masks
  These are the same state masks that would be passed to `create_readcondition()`, `read()`, or `take()`.

- Query expression
  An SQL-like expression (see 5.3.1) describing a subset of samples which cause the condition to be triggered. This same expression is used to filter the data set returned from a `read_w_condition()` or `take_w_condition()` operation. It may also impose a sort order (`ORDER BY`) on that data set.

- Query parameters
  The query expression can contain parameter placeholders. This argument provides initial values for those parameters. The query parameters can be changed after the query condition is created (the query expression cannot be changed).

A particular query condition can be used with a wait set (`attach_condition`), with a data reader (`read_w_condition`, `take_w_condition`, `read_next_instance_w_condition`, `take_next_instance_w_condition`), or both. When used with a wait set, the `ORDER BY` clause has no effect on triggering the wait set. When used with a data reader's `read*()` or `take*()` operation, the resulting data set will only contain samples which match the query expression and they will be ordered by the `ORDER BY` fields, if an `ORDER BY` clause is present.

5.3.1   Query Expressions

Query expressions are a super set of filter expressions (see section 5.2.1). Following the filter expression, the query expression can optionally have an `ORDER BY` keyword followed by a comma-separated list of field references. If the `ORDER BY` clause is present, the filter expression may be empty. The following strings are examples of query expressions:

- m > 100 ORDER BY n
- ORDER BY p.q, r, s.t.u
- NOT v LIKE 'z%'

Query expressions can use parameter placeholders in the same way that filter expressions (for content-filtered topics) use them. See section 5.2.2 for details.

5.3.2   Query Condition Example

The following code snippet creates and uses a query condition for a type that uses struct 'Message' with field 'key' (an integral type).

```
DDS::QueryCondition_var dr_qc =
  dr->create_querycondition(DDS::ANY_SAMPLE_STATE,
                            DDS::ANY_VIEW_STATE,
                            DDS::ALIVE_INSTANCE_STATE,
                            "key > 1",
                            DDS::StringSeq());
DDS::WaitSet_var ws = new DDS::WaitSet;
ws->attach_condition(dr_qc);
DDS::ConditionSeq active;
DDS::Duration_t three_sec = {3, 0};
DDS::ReturnCode_t ret = ws->wait(active, three_sec);
  // error handling not shown
ws->detach_condition(dr_qc);
MessageDataReader_var mdr = MessageDataReader::_narrow(dr);
MessageSeq data;
DDS::SampleInfoSeq infoseq;
ret = mdr->take_w_condition(data, infoseq, DDS::LENGTH_UNLIMITED, dr_qc);
  // error handling not shown
dr->delete_readcondition(dr_qc);
```

Any sample received with `key <= 1` would neither trigger the condition (to satisfy the wait) nor be returned in the 'data' sequence from `take_w_condition()`.

5.4   Multi Topic

Multi topic is a more complex feature than the other two Content-Subscription features, therefore describing it requires some new terminology.

The `MultiTopic` interface inherits from the `TopicDescription` interface, just like `ContentFilteredTopic` does. A data reader created for the multi topic is known as a "multi

topic data reader." A multi topic data reader receives samples belonging to any number of regular topics. These topics are known as its "constituent topics." The multi topic has a DCPS data type known as the "resulting type." The multi topic data reader implements the type-specific data reader interface for the resulting type. For example, if the resulting type is Message, then the multi topic data reader can be narrowed to the `MessageDataReader` interface.

The multi topic's topic expression (see section 5.4.1) describes how the distinct fields of the incoming data (on the constituent topics) are mapped to the fields of the resulting type.

The domain participant interface contains operations for creating and deleting a multi topic. Creating a multi topic requires the following parameters:

- Name

  Assigns a name to this multi topic which could later be used with the `lookup_topicdescription()` operation.

- Type name

  Specifies the resulting type of the multi topic. This type must have its type support registered before creating the multi topic.

- Topic expression (also known as subscription expression)

  An SQL-like expression (see section 5.4.1) which defines the mapping of constituent topic fields to resulting type fields. It can also specify a filter (`WHERE` clause).

- Expression parameters

  The topic expression can contain parameter placeholders. This argument provides initial values for those parameters. The expression parameters can be changed after the multi topic is created (the topic expression cannot be changed).

Once the multi topic has been created, it is used by the subscriber's `create_datareader()` operation to obtain a multi topic data reader. This data reader is used by the application to receive the constructed samples of the resulting type. The manner in which these samples are constructed is described below in section 5.4.2.2.

5.4.1 Topic Expressions

Topic expressions use a syntax that is very similar to a complete SQL query:

`SELECT <aggregation> FROM <selection> [WHERE <condition>]`

- The aggregation can be either a "*" or a comma separated list of field specifiers. Each field specifier has the following syntax:

  - `<constituent_field> [[AS] <resulting_field>]]`

- constituent_field is a field reference (see section 5.2.1) to a field in one of the constituent topics (which topic is not specified).

- The optional resulting_field is a field reference to a field in the resulting type. If present, the resulting_field is the destination for the constituent_field in the constructed sample. If absent, the constituent_field data is assigned to a field with the same name in the resulting type. The optional "AS" has no effect.

- If a "*" is used as the aggregation, each field in the resulting type is assigned the value from a same-named field in one of the constituent topic types.

- The selection lists one or more constituent topic names. Topic names are separated by a "join" keyword (all 3 join keywords are equivalent):

   - <topic> [{NATURAL INNER | NATURAL | INNER NATURAL} JOIN <topic>]...

   - Topic names must contain only letters, digits, and dashes (but may not start with a digit).

   - The natural join operation is commutative and associative, thus the order of topics has no impact.

   - The semantics of the natural join are that any fields with the same name are treated as "join keys" for the purpose of combining data from the topics in which those keys appear. The join operation is described in more detail in the subsequent sections of this chapter.

- The condition has the exact same syntax and semantics as the filter expression (see section 5.2.1). Field references in the condition must match field names in the resulting types, not field names in the constituent topic types. The condition in the topic expression can use parameter placeholders in the same way that filter expressions (for content-filtered topics) use them. See section 5.2.2 for details.

5.4.2 Usage Notes

5.4.2.1 Join Keys and DCPS Data Keys

The concept of DCPS data keys (@key) has already been discussed in Section 2.1.1. Join keys for the multi topic are a distinct but related concept.

A join key is any field name that occurs in the struct for more than one constituent topic. The existence of the join key enforces a constraint on how data samples of those topics are combined into a constructed sample (see section 5.4.2.2). Specifically, the value of that key must be equal for those data samples from the constituent topics to be combined into a

sample of the resulting type. If multiple join keys are common to the same two or more topics, the values of all keys must be equal in order for the data to be combined.

The DDS specification requires that join key fields have the same type. Additionally, OpenDDS imposes two requirements on how the IDL must define DCPS data keys to work with multi topics:

1) Each join key field must also be a DCPS data key for the types of its constituent topics.

2) The resulting type must contain each of the join keys, and those fields must be DCPS data keys for the resulting type.

The example in section 5.4.3.1 meets both of these requirements. Note that it is not necessary to list the join keys in the aggregation (SELECT clause).

5.4.2.2 How Resulting Samples are Constructed

Although many concepts in multi topic are borrowed from the domain of relational databases, a real-time middleware such as DDS is not a database. Instead of processing a batch of data at a time, each sample arriving at the data reader from one of the constituent topics triggers multi-topic-specific processing that results in the construction of zero, one, or many samples of the resulting type and insertion of those constructed samples into the multi topic data reader.

Specifically, the arrival of a sample on constituent topic "A" with type "TA" results in the following steps in the multi topic data reader (this is a simplification of the actual algorithm):

1) A sample of the resulting type is constructed, and fields from TA which exist in the resulting type and are in the aggregation (or are join keys) are copied from the incoming sample to the constructed sample.

2) Each topic "B" which has at least one join key in common with A is considered for a join operation. The join reads READ_SAMPLE_STATE samples on topic B with key values matching those in the constructed sample. The result of the join may be zero, one, or many samples. Fields from TB are copied to the resulting sample as described in step 1.

3) Join keys of topic "B" (connecting it to other topics) are then processed as described in step 2, and this continues to all other topics that are connected by join keys.

4) Any constituent topics that were not visited in steps 2 or 3 are processed as "cross joins" (also known as cross-product joins). These are joins with no key constraints.

5) If any constructed samples result, they are inserted into the multi topic data reader's internal data structures as if they had arrived via the normal mechanisms. Application listeners and conditions are notified.

5.4.2.3 Use with Subscriber Listeners

If the application has registered a subscriber listener for read condition status changes (`DATA_ON_READERS_STATUS`) with the same subscriber that also contains a multi topic, then the application must invoke `notify_datareaders()` in its implementation of the subscriber listener's `on_data_on_readers()` callback method. This requirement is necessary because the multi topic internally uses data reader listeners, which are preempted when a subscriber listener is registered.

5.4.3 Multi Topic Example

This example is based on the example topic expression used in Annex A section A.3 of the DDS specification. It illustrates how the properties of the multi topic join operation can be used to correlate data from separate topics (and possibly distinct publishers).

5.4.3.1 IDL and Topic Expression

Often times we will use the same string as both the topic name and topic type. In this example we will use distinct strings for the type names and topic names, in order to illustrate when each is used.

Here is the IDL for the constituent topic data types:

```
@topic
struct LocationInfo {
  @key unsigned long flight_id;
  long x;
  long y;
  long z;
};

@topic
struct PlanInfo {
  @key unsigned long flight_id;
  string flight_name;
  string tailno;
};
```

Note that the names and types of the key fields match, so they are designed to be used as join keys. The resulting type (below) also has that key field.

Next we have the IDL for the resulting data type:

```
@topic
struct Resulting {
```

```
  @key unsigned long flight_id;
  string flight_name;
  long x;
  long y;
  long height;
};
```

Based on this IDL, the following topic expression can be used to combine data from a topic *Location* which uses type `LocationInfo` and a topic *FlightPlan* which uses type `PlanInfo`:

```
SELECT flight_name, x, y, z AS height FROM Location NATURAL JOIN FlightPlan WHERE height
< 1000 AND x <23
```

Taken together, the IDL and the topic expression describe how this multi topic will work. The multi topic data reader will construct samples which belong to instances keyed by `flight_id`. The instance of the resulting type will only come into existence once the corresponding instances are available from both the *Location* and *FlightPlan* topics. Some other domain participant or participants within the domain will publish data on those topics, and they don't even need to be aware of one another. Since they each use the same `flight_id` to refer to flights, the multi topic can correlate the incoming data from disparate sources.

5.4.3.2 Creating the Multi Topic Data Reader

Creating a data reader for the multi topic consists of a few steps. First the type support for the resulting type is registered, then the multi topic itself is created, followed by the data reader:

```
  ResultingTypeSupport_var ts_res = new ResultingTypeSupportImpl;
  ts_res->register_type(dp, "");
  CORBA::String_var type_name = ts_res->get_type_name();
  DDS::MultiTopic_var mt =
    dp->create_multitopic("MyMultiTopic",
                          type_name,
                          "SELECT flight_name, x, y, z AS height "
                            "FROM Location NATURAL JOIN FlightPlan "
                            "WHERE height < 1000 AND x<23",
                          DDS::StringSeq());
  DDS::DataReader_var dr =
    sub->create_datareader(mt,
                           DATAREADER_QOS_DEFAULT,
                           NULL,
                           OpenDDS::DCPS::DEFAULT_STATUS_MASK);
```

5.4.3.3 Reading Data with the Multi Topic Data Reader

From an API perspective, the multi topic data reader is identical to any other typed data reader for the resulting type. This example uses a wait set and a read condition in order to block until data is available.

```
  DDS::WaitSet_var ws = new DDS::WaitSet;
  DDS::ReadCondition_var rc =
    dr->create_readcondition(DDS::ANY_SAMPLE_STATE,
```

```
                              DDS::ANY_VIEW_STATE,
                              DDS::ANY_INSTANCE_STATE);
ws->attach_condition(rc);
DDS::Duration_t infinite = {DDS::DURATION_INFINITE_SEC,
                            DDS::DURATION_INFINITE_NSEC};
DDS::ConditionSeq active;
ws->wait(active, infinite); // error handling not shown
ws->detach_condition(rc);
ResultingDataReader_var res_dr = ResultingDataReader::_narrow(dr);
ResultingSeq data;
DDS::SampleInfoSeq info;
res_dr->take_w_condition(data, info, DDS::LENGTH_UNLIMITED, rc);
```

CHAPTER 6

Built-In Topics

6.1   Introduction

In OpenDDS, Built-In-Topics are created and published by default to exchange information about DDS participants operating in the deployment. When OpenDDS is used in a centralized discovery approach using the `DCPSInfoRepo` service, the Built-In-Topics are published by this service. For DDSI-RTPS discovery, the internal OpenDDS implementation instantiated in a process populates the caches of the Built-In Topic DataReaders. See Section 7.3.3 for a description of RTPS discovery configuration.

The IDL struct `BuiltinTopicKey_t` is used by the Built-In Topics. This structure contains an array of 16 octets (bytes) which corresponds to an InfoRepo identifier or a DDSI-RTPS GUID.

6.2   Built-In Topics for DCPSInfoRepo Configuration

When starting the `DCPSInfoRepo` a command line option of `-NOBITS` may be used to suppress publication of built-in topics.

Four separate topics are defined for each domain. Each is dedicated to a particular entity (domain participant, topic, data writer, data reader) and publishes instances describing the state for each entity in the domain.

Subscriptions to built-in topics are automatically created for each domain participant. A participant's support for Built-In-Topics can be toggled via the `DCPSBit` configuration option (see the table in Section 7.2) (Note: this option cannot be used for RTPS discovery). To view the built-in topic data, simply obtain the built-in Subscriber and then use it to access the Data Reader for the built-in topic of interest. The Data Reader can then be used like any other Data Reader.

Sections 6.3 through 6.6 provide details on the data published for each of the four built-in topics. An example showing how to read from a built-in topic follows those sections.

If you are not planning on using Built-in-Topics in your application, you can configure OpenDDS to remove Built-In-Topic support at build time. Doing so can reduce the footprint of the core DDS library by up to 30%. See Section 1.3.2 for information on disabling Built-In-Topic support.

6.3   DCPSParticipant Topic

The `DCPSParticipant` topic publishes information about the Domain Participants of the Domain. Here is the IDL that defines the structure published for this topic:

```
struct ParticipantBuiltinTopicData {
  BuiltinTopicKey_t key;
  UserDataQosPolicy user_data;
};
```

Each Domain Participant is defined by a unique key and is its own instance within this topic.

6.4   DCPSTopic Topic

Note   *OpenDDS does not support this Built-In-Topic when configured for RTPS discovery.*

The `DCPSTopic` topic publishes information about the topics in the domain. Here is the IDL that defines the structure published for this topic:

```
struct TopicBuiltinTopicData {
  BuiltinTopicKey_t key;
  string name;
  string type_name;
  DurabilityQosPolicy durability;
  QosPolicy deadline;
  LatencyBudgetQosPolicy latency_budget;
  LivelinessQosPolicy liveliness;
  ReliabilityQosPolicy reliability;
  TransportPriorityQosPolicy transport_priority;
  LifespanQosPolicy lifespan;
  DestinationOrderQosPolicy destination_order;
  HistoryQosPolicy history;
  ResourceLimitsQosPolicy resource_limits;
```

```
    OwnershipQosPolicy ownership;
    TopicDataQosPolicy topic_data;
};
```

Each topic is identified by a unique key and is its own instance within this built-in topic. The members above identify the name of the topic, the name of the topic type, and the set of QoS policies for that topic.

6.5   DCPSPublication Topic

The `DCPSPublication` topic publishes information about the Data Writers in the Domain. Here is the IDL that defines the structure published for this topic:

```
struct PublicationBuiltinTopicData {
  BuiltinTopicKey_t key;
  BuiltinTopicKey_t participant_key;
  string topic_name;
  string type_name;
  DurabilityQosPolicy durability;
  DeadlineQosPolicy deadline;
  LatencyBudgetQosPolicy latency_budget;
  LivelinessQosPolicy liveliness;
  ReliabilityQosPolicy reliability;
  LifespanQosPolicy lifespan;
  UserDataQosPolicy user_data;
  OwnershipStrengthQosPolicy ownership_strength;
  PresentationQosPolicy presentation;
  PartitionQosPolicy partition;
  TopicDataQosPolicy topic_data;
  GroupDataQosPolicy group_data;
};
```

Each Data Writer is assigned a unique key when it is created and defines its own instance within this topic. The fields above identify the Domain Participant (via its key) that the Data Writer belongs to, the topic name and type, and the various QoS policies applied to the Data Writer.

6.6   DCPSSubscription Topic

The `DCPSSubscription` topic publishes information about the Data Readers in the Domain. Here is the IDL that defines the structure published for this topic:

```
struct SubscriptionBuiltinTopicData {
  BuiltinTopicKey_t key;
  BuiltinTopicKey_t participant_key;
  string topic_name;
  string type_name;
  DurabilityQosPolicy durability;
  DeadlineQosPolicy deadline;
  LatencyBudgetQosPolicy latency_budget;
  LivelinessQosPolicy liveliness;
  ReliabilityQosPolicy reliability;
```

```
        DestinationOrderQosPolicy destination_order;
        UserDataQosPolicy user_data;
        TimeBasedFilterQosPolicy time_based_filter;
        PresentationQosPolicy presentation;
        PartitionQosPolicy partition;
        TopicDataQosPolicy topic_data;
        GroupDataQosPolicy group_data;
      };
```

Each Data Reader is assigned a unique key when it is created and defines its own instance within this topic. The fields above identify the Domain Participant (via its key) that the Data Reader belongs to, the topic name and type, and the various QoS policies applied to the Data Reader.

6.7   Built-In Topic Subscription Example

The following code uses a domain participant to get the built-in subscriber. It then uses the subscriber to get the Data Reader for the `DCPSParticipant` topic and subsequently reads samples for that reader.

```
    Subscriber_var bit_subscriber = participant->get_builtin_subscriber();
    DDS::DataReader_var dr =
      bit_subscriber->lookup_datareader(BUILT_IN_PARTICIPANT_TOPIC);
    DDS::ParticipantBuiltinTopicDataDataReader_var part_dr =
      DDS::ParticipantBuiltinTopicDataDataReader::_narrow(dr);

    DDS::ParticipantBuiltinTopicDataSeq part_data;
    DDS::SampleInfoSeq infos;
    DDS::ReturnCode_t ret = part_dr->read(part_data, infos, 20,
                                          DDS::ANY_SAMPLE_STATE,
                                          DDS::ANY_VIEW_STATE,
                                          DDS::ANY_INSTANCE_STATE);

    // Check return status and read the participant data
```

The code for the other built-in topics is similar.

6.8   OpenDDS-specific Built-In Topics

6.8.1   OpenDDSParticipantLocation Topic

The Built-In Topic "OpenDDSParticipantLocation" is published by the DDSI-RTPS discovery implementation to give applications visibility into the details of how each remote participant is connected over the network.

The IDL for OpenDDSParticipantLocation is in `dds/DdsDcpsCore.idl` in the `OpenDDS::DCPS` module. If the RtpsRelay (15.2) and/or IETF ICE (15.3) are enabled, their usage is reflected in the OpenDDSParticipantLocation topic data.

6.8.2   **OpenDDSConnectionRecord Topic**

The Built-In Topic "OpenDDSConnectionRecord" is published by the DDSI-RTPS discovery implementation and RTPS_UDP transport implementation when support for IETF ICE is enabled. See section 15.3 for details on OpenDDS's support for IETF ICE. The IDL for OpenDDSConnectionRecord is in `dds/DdsDcpsCore.idl` in the `OpenDDS::DCPS` module.

6.8.3   **OpenDDSInternalThread Topic**

The Built-In Topic "OpenDDSInternalThread" is published when OpenDDS is configured with DCPSThreadStatusInterval (see section 7.2). When enabled, the DataReader for this Built-In Topic will report the health (responsiveness) of threads created and managed by OpenDDS within the current process. The IDL for OpenDDSInternalThread is in `dds/DdsDcpsCore.idl` in the `OpenDDS::DCPS` module.

CHAPTER 7

Run-time Configuration

7.1 Configuration Approach

OpenDDS includes a file-based configuration framework for configuring global options and options related to specific publishers and subscribers such as discovery and transport configuration. OpenDDS also allows configuration via the command line for a limited number of options and via a configuration API. This chapter summarizes the configuration options supported by OpenDDS.

OpenDDS configuration is concerned with three main areas:

1) **Common Configuration Options** – configure the behavior of DCPS entities at a global level. This allows separately deployed processes in a computing environment to share common settings for the specified behavior (e.g. all readers and writers should use RTPS discovery).

2) **Discovery Configuration Options** – configure the behavior of the discovery mechanism(s). OpenDDS supports multiple approaches for discovering and associating writers and readers as detailed in Section 7.3.

3) **Transport Configuration Options** – configure the Extensible Transport Framework (ETF) which abstracts the transport layer from the DCPS layer of OpenDDS. Each pluggable transport can be configured separately.

The configuration file for OpenDDS is a human-readable ini-style text file. Table 7-1 shows a list of the available configuration section types as they relate to the area of OpenDDS that they configure.

Table 7-1 Configuration File Sections

Focus Area	File Section Title
Global Settings	`[common]`
Discovery	`[domain]`
	`[repository]`
	`[rtps_discovery]`
Static Discovery	`[endpoint]`
	`[topic]`
	`[datawriterqos]`
	`[datareaderqos]`
	`[publisherqos]`
	`[subscriberqos]`
Transport	`[config]`
	`[transport]`

For each of the section types with the exception of `[common]`, the syntax of a section header takes the form of `[section type/instance]`. For example, a `[repository]` section type would always be used in a configuration file like so:

`[repository/repo_1]` where `repository` is the section type and `repo_1` is an instance name of a repository configuration. How to use instances to configure discovery and transports is explained further in Sections 7.3 and 7.4.

The `-DCPSConfigFile` command-line argument can be used to pass the location of a configuration file to OpenDDS. For example:

Windows:

```
publisher -DCPSConfigFile pub.ini
```

Unix:

```
./publisher -DCPSConfigFile pub.ini
```

Command-line arguments are passed to the service participant singleton when initializing the domain participant factory. This is accomplished by using the `TheParticipantFactoryWithArgs` macro:

```
#include <dds/DCPS/Service_Participant.h>

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

  DDS::DomainParticipantFactory_var dpf =
    TheParticipantFactoryWithArgs(argc, argv);
```

To set a default configuration file to load, use
`TheServiceParticipant->default_configuration_file(ACE_TCHAR* path)`, like in the
following example:

```
#include <dds/DCPS/Service_Participant.h>

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

  TheServiceParticipant->default_configuration_file(ACE_TEXT("pub.ini"));

  DDS::DomainParticipantFactory_var dpf =
    TheParticipantFactoryWithArgs(argc, argv);
```

`pub.ini` would be used unless `-DCPSConfigFile` is passed to override the default
configuration file.

The `Service_Participant` class also provides methods that allow an application to
configure the DDS service. See the header file
$DDS_ROOT/dds/DCPS/Service_Participant.h for details.

The following subsections detail each of the configuration file sections and the available
options related to those sections.

7.2 Common Configuration Options

The [common] section of an OpenDDS configuration file contains options such as the
debugging output level, the location of the `DCPSInfoRepo` process, and memory
preallocation settings. A sample [common] section follows:

```
[common]
DCPSDebugLevel=0
DCPSInfoRepo=localhost:12345
DCPSLivelinessFactor=80
DCPSChunks=20
DCPSChunksAssociationMultiplier=10
DCPSBitLookupDurationMsec=2000
DCPSPendingTimeout=30
```

It is not necessary to specify every option.

Option values in the [common] section with names that begin with "DCPS" can be overridden by a command-line argument. The command-line argument has the same name as the configuration option with a "-" prepended to it. For example:

```
subscriber -DCPSInfoRepo localhost:12345
```

The following table summarizes the [common] configuration options:

Table 7-2 Common Configuration Options

Option	Description	Default
DCPSBit=[1|0]	Toggle Built-In-Topic support.	1
DCPSBitLookupDurationMsec=msec	The maximum duration in milliseconds that the framework will wait for latent Built-In Topic information when retrieving BIT data given an instance handle. The participant code may get an instance handle for a remote entity before the framework receives and processes the related BIT information. The framework waits for up to the given amount of time before it fails the operation.	2000
DCPSBitTransportIPAddress=addr	IP address identifying the local interface to be used by tcp transport for the Built-In Topics. **NOTE**: This property is only applicable to a DCPSInfoRepo configuration.	INADDR_ANY
DCPSBitTransportPort=port	Port used by the tcp transport for Built-In Topics.If the default of '0' is used, the operating system will choose a port to use. **NOTE**: This property is only applicable to a DCPSInfoRepo configuration.	0
DCPSChunks=n	Configurable number of chunks that a data writer's and reader's cached allocators will preallocate when the RESOURCE_LIMITS QoS value is infinite. When all of the preallocated chunks are in use, OpenDDS allocates from the heap.	20
DCPSChunkAssociationMultiplier=n	Multiplier for the DCPSChunks or resource_limits.max_samples value to determine the total number of shallow copy chunks that are preallocated. Set this to a value greater than the number of connections so the preallocated chunk handles do not run out. A sample written to multiple data readers will not be copied multiple times but there is a shallow copy handle to that sample used to manage the delivery to each data reader. The size of the handle is small so there is not great need to set this value close to the number of connections.	10

Option	Description	Default
DCPSDebugLevel=n	Integer value that controls the amount of debug information the DCPS layer prints. Valid values are 0 through 10.	0
ORBLogFile=filename	Change log message destination to the file specified, which is opened in appending mode. See the note below this table regarding the ORB prefix.	None: use standard error
ORBVerboseLogging=[0|1|2]	Add a prefix to each log message, using a format defined by the ACE library: 0 – no prefix 1 – verbose "lite": adds timestamp and priority 2 – verbose: in addition to "lite" has host name, PID, program name See the note below this table regarding the ORB prefix.	0
DCPSDefaultAddress=addr	Default value for the host portion of local_address for transport instances containing a local_address. Only applied when DCPSDefaultAddress is set to a non-empty value and no local_address is specified in the transport. Other subsystems (such as DDSI-RTPS Discovery) use DCPSDefaultAddress as a default value as well.	
DCPSDefaultDiscovery=[DEFAULT_REPO| DEFAULT_RTPS| DEFAULT_STATIC| user-defined configuration instance name]	Specifies a discovery configuration to use for any domain not explicitly configured. DEFAULT_REPO translates to using the DCPSInfoRepo. DEFAULT_RTPS specifies the use of RTPS for discovery. DEFAULT_STATIC specifies the use of static discovery. See Section 7.3 for details about configuring discovery.	DEFAULT_REPO
DCPSGlobalTransportConfig=name	Specifies the name of the transport configuration that should be used as the global configuration. This configuration is used by all entities that do not otherwise specify a transport configuration. A special value of $file uses a transport configuration that includes all transport instances defined in the configuration file.	The default configuration is used as described in 7.4.1
DCPSInfoRepo=objref	Object reference for locating the DCPS Information Repository. This can either be a full CORBA IOR or a simple host:port string.	file://repo.ior
DCPSLivelinessFactor=n	Percent of the liveliness lease duration after which a liveliness message is sent. A value of 80 implies a 20% cushion of latency from the last detected heartbeat message.	80
DCPSMonitor=[0|1]	Use the OpenDDS_monitor library to publish data on monitoring topics (see dds/monitor/README).	0

Option	Description	Default
DCPSPendingTimeout=sec	The maximum duration in seconds a data writer will block to allow unsent samples to drain on deletion. By default, this option blocks indefinitely.	0
DCPSPersistentDataDir=path	The path on the file system where durable data will be stored. If the directory does not exist it will be created automatically.	OpenDDS-durable-data-dir
DCPSPublisherContentFilter=[1|0]	Controls the filter expression evaluation policy for content filtered topics. When enabled (1), the publisher may drop any samples, before handing them off to the transport when these samples would have been ignored by all subscribers.	1
DCPSSecurity=[0|1]	This setting is only available when OpenDDS is compiled with DDS Security enabled. If set to 1, enable DDS Security framework and built-in plugins. Each Domain Participant using security must be created with certain QoS policy values. See chapter 14: DDS Security for more information.	0
DCPSSecurityDebug=CAT[,CAT...]	This setting is only available when OpenDDS is compiled with DDS Security enabled. This controls the security debug logging granularity by category. See Section 7.6.3 "Security Logging" for details.	0
DCPSSecurityDebugLevel=n	This setting is only available when OpenDDS is compiled with DDS Security enabled. This controls the security debug logging granularity by debug level. See Section 7.6.3 "Security Logging" for details.	N/A
DCPSSecurityFakeEncryption=[0|1]	This setting is only available when OpenDDS is compiled with DDS Security enabled. This option, when set to 1, disables all encryption by making encryption and decryption no-ops. OpenDDS still generates keys and performs other security bookkeeping, so this option is useful for debugging the security infrastructure by making it possible to manually inspect all messages.	0
pool_size=n_bytes	Size of safety profile memory pool, in bytes.	41943040 (40 MiB)
pool_granularity=n_bytes	Granularity of safety profile memory pool in bytes. Must be multiple of 8.	8

Option	Description	Default

Scheduler=[
 	
  SCHED_RR|
 	
  SCHED_FIFO|
 	
  SCHED_OTHER]
	Selects the thread scheduler to use. Setting the scheduler to a value other than the default requires privileges on most systems. A value of SCHED_RR, SCHED_FIFO, or SCHED_OTHER can be set. SCHED_OTHER is the default scheduler on most systems; SCHED_RR is a round robin scheduling algorithm; and SCHED_FIFO allows each thread to run until it either blocks or completes before switching to a different thread.	SCHED_OTHER

scheduler_slice=usec	Some operating systems, such as SunOS, require a time slice value to be set when selecting schedulers other than the default. For those systems, this option can be used to set a value in microseconds.	none
DCPSBidirGIOP=[0|1]	Use TAO's BiDirectional GIOP feature for interaction with the DCPSInfoRepo. With BiDir enabled, fewer sockets are needed since the same socket can be used for both client and server roles.	1
DCPSThreadStatusInterval=sec	Enable internal thread status reporting (see section 6.8.3) using the specified reporting interval, in seconds.	0 (disabled)

DCPSTypeObjectEncoding=[
  Normal |
  WriteOldFormat |
  ReadOldFormat]
	Before version 3.18, OpenDDS had a bug in the encoding used for TypeObject (from XTypes) and related data types.
	
	If this application needs to be compatible with an application built with an older OpenDDS (that has XTypes), select one of WriteOldFormat or ReadOldFormat.
	
	Using WriteOldFormat means that the TypeInformation written by this application will be understood by legacy applications.
	
	Using WriteOldFormat or ReadOldFormat means that TypeInformation written in the legacy format will be understood by this application.
	
	These options are designed to enable a phased migration from the incorrect implementation (pre-3.18) to a compliant one. In the first phase, legacy applications can coexist with WriteOldFormat. In the second phase (once all legacy applications have been upgraded), WriteOldFormat can communicate with ReadOldFormat. In the final phase (once all WriteOldFormat applications have been upgraded), ReadOldFormat applications can be transitioned to Normal.
		Normal

The DCPSInfoRepo option's value is passed to CORBA::ORB::string_to_object() and can be any Object URL type understandable by TAO (file, IOR, corbaloc, corbaname). A simplified endpoint description of the form <host>:<port> is also accepted. It is equivalent to corbaloc::<host>:<port>/DCPSInfoRepo.

Certain options that begin with "ORB" instead of "DCPS" are listed in the table above. They are named differently since they are inherited from TAO. The options starting with "ORB" listed in this table are implemented directly by OpenDDS (not passed to TAO) and are supported either on the command line (using a "-" prefix) or in the configuration file. Other command-line options that begin with "-ORB" are passed to TAO's `ORB_init` if DCPSInfoRepo discovery is used.

The `DCPSChunks` option allows application developers to tune the amount of memory preallocated when the `RESOURCE_LIMITS` are set to infinite. Once the allocated memory is exhausted, additional chunks are allocated/deallocated from the heap. This feature of allocating from the heap when the preallocated memory is exhausted provides flexibility but performance will decrease when the preallocated memory is exhausted.

7.3   Discovery Configuration

In DDS implementations, participants are instantiated in application processes and must discover one another in order to communicate. A DDS implementation uses the feature of domains to give context to the data being exchanged between DDS participants in the same domain. When DDS applications are written, participants are assigned to a domain and need to ensure their configuration allows each participant to discover the other participants in the same domain.

OpenDDS offers a centralized discovery mechanism, a peer-to-peer discovery mechanism, and a static discovery mechanism. The centralized mechanism uses a separate service running a `DCPSInfoRepo` process. The RTPS peer-to-peer mechanism uses the DDSI-RTPS discovery protocol standard to achieve non-centralized discovery. The static discovery mechanism uses the configuration file to determine which writers and readers should be associated and uses the underlying transport to determine which writers and readers exist. A number of configuration options exist to meet the deployment needs of DDS applications. Except for static discovery, each mechanism uses default values if no configuration is supplied either via the command line or configuration file.

The following sections show how to configure the advanced discovery capabilities. For example, some deployments may need to use multiple `DCPSInfoRepo` services or DDSI-RTPS discovery to satisfy interoperability requirements.

7.3.1   Domain Configuration

An OpenDDS configuration file uses the `[domain]` section type to configure one or more discovery domains with each domain pointing to a discovery configuration in the same file or a default discovery configuration. OpenDDS applications can use a centralized discovery

approach using the `DCPSInfoRepo` service or a peer-to-peer discovery approach using the RTPS discovery protocol standard or a combination of the two in the same deployment. The section type for the `DCPSInfoRepo` method is `[repository]` and the section type for an RTPS discovery configuration is `[rtps_discovery]`. The static discovery mechanism does not have a dedicated section. Instead, users are expected to refer to the `DEFAULT_STATIC` instance. A single domain can refer to only one type of discovery section.

See Sections 7.3.2 for configuring `[repository]` sections, 7.3.3 for configuring `[rtps_discovery]`, and 7.3.4 for configuring static discovery.

Ultimately a domain is assigned an integer value and a configuration file can support this in two ways. The first is to simply make the instance value the integer value assigned to the domain as shown here:

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1
    (more properties...)
```

Our example configures a single domain identified by the domain keyword and followed by an instance value of /1. The instance value after the slash in this case is the integer value assigned to the domain. An alternative syntax for this same content is to use a more recognizable (friendly) name instead of a number for the domain name and then add the `DomainId` property to the section to give the integer value. Here is an example:

```
[domain/books]
DomainId=1
DiscoveryConfig=DiscoveryConfig1
```

The domain is given a friendly name of books. The `DomainId` property assigns the integer value of 1 needed by a DDS application reading the configuration. Multiple domain instances can be identified in a single configuration file in this format.

Once one or more domain instances are established, the discovery properties must be identified for that domain. The `DiscoveryConfig` property must either point to another section that holds the discovery configuration or specify one of the internal default values for discovery (e.g. `DEFAULT_REPO`, `DEFAULT_RTPS`, or `DEFAULT_STATIC`). The instance name in our example is `DiscoveryConfig1`. This instance name must be associated with a section type of either `[repository]` or `[rtps_discovery]`.

Here is an extension of our example:

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=host1.mydomain.com:12345
```

In this case our domain points to a `[repository]` section which is used for an OpenDDS DCPSInfoRepo service. See Section 7.3.2 for more details.

There are going to be occasions when specific domains are not identified in the configuration file. For example, if an OpenDDS application assigns a domain ID of 3 to its participants and the above example does not supply a configuration for domain id of 3 then the following can be used:

```
[common]
DCPSInfoRepo=host3.mydomain.com:12345
DCPSDefaultDiscovery=DEFAULT_REPO

[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=host1.mydomain.com:12345
```

The `DCPSDefaultDiscovery` property tells the application to assign any participant that doesn't have a domain id found in the configuration file to use a discovery type of `DEFAULT_REPO` which means "use a `DCPSInfoRepo` service" and that `DCPSInfoRepo` service can be found at *host3.mydomain.com:12345*.

As shown in Table 7-2 the `DCPSDefaultDiscovery` property has three other values that can be used. The `DEFAULT_RTPS` constant value informs participants that don't have a domain configuration to use RTPS discovery to find other participants. Similarly, the `DEFAULT_STATIC` constant value informs the participants that don't have a domain configuration to use static discovery to find other participants.

The final option for the `DCPSDefaultDiscovery` property is to tell an application to use one of the defined discovery configurations to be the default configuration for any participant domain that isn't called out in the file. Here is an example:

```
[common]

DCPSDefaultDiscovery=DiscoveryConfig2

[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=host1.mydomain.com:12345

[domain/2]
DiscoveryConfig=DiscoveryConfig2

[repository/DiscoveryConfig2]
RepositoryIor=host2.mydomain.com:12345
```

By adding the `DCPSDefaultDiscovery` property to the `[common]` section, any participant that hasn't been assigned to a domain id of 1 or 2 will use the configuration of

O p e n D D S   D e v e l o p e r ' s   G u i d e

`DiscoveryConfig2`. For more explanation of a similar configuration for RTPS discovery see Section 7.3.3.

Here are the available properties for the [domain] section.

Table 7-3 Domain Section Configuration Properties

Option	Description
`DomainId=n`	An integer value representing a Domain being associated with a repository.
`DomainRepoKey=k`	Key value of the mapped repository (Deprecated. Provided for backward compatibility).
`DiscoveryConfig=config instance name`	A user-defined string that refers to the instance name of a [`repository`] or [`rtps_discovery`] section in the same configuration file or one of the internal default values (`DEFAULT_REPO`, `DEFAULT_RTPS`, or `DEFAULT_STATIC`). (Also see the `DCPSDefaultDiscovery` property in Table 7-2)
`DefaultTransportConfig=config`	A user-defined string that refers to the instance name of a [`config`] section. See Section 7.4.

7.3.2   Configuring Applications For DCPSInfoRepo

An OpenDDS `DCPSInfoRepo` is a service on a local or remote node used for participant discovery. Configuring how participants should find `DCPSInfoRepo` is the purpose of this section. Assume for example that the `DCPSInfoRepo` service is started on a host and port of *myhost.mydomain.com:12345*. Applications can make their OpenDDS participants aware of how to find this service through command line options or by reading a configuration file.

In our Getting Started example from 2.1.7, "Running the Example" the executables were given a command line parameter to find the `DCPSInfoRepo` service like so:

`publisher -DCPSInfoRepo file://repo.ior`

This assumes that the `DCPSInfoRepo` has been started with the following syntax:

Windows:

`%DDS_ROOT%\bin\DCPSInfoRepo -o repo.ior`

Unix:

`$DDS_ROOT/bin/DCPSInfoRepo -o repo.ior`

The `DCPSInfoRepo` service generates its location object information in this file and participants need to read this file to ultimately connect. The use of file based IORs to find a discovery service, however, is not practical in most production environments, so

O p e n D D S   D e v e l o p e r ' s   G u i d e

applications instead can use a command line option like the following to simply point to the host and port where the `DCPSInfoRepo` is running.

```
publisher -DCPSInfoRepo myhost.mydomain.com:12345
```

The above assumes that the `DCPSInfoRepo` has been started on a host (*myhost.mydomain.com*) as follows:

Windows:

```
%DDS_ROOT%\bin\DCPSInfoRepo -ORBListenEndpoints iiop://:12345
```

Unix:

```
$DDS_ROOT/bin/DCPSInfoRepo -ORBListenEndpoints iiop://:12345
```

If an application needs to use a configuration file for other settings, it would become more convenient to place discovery content in the file and reduce command line complexity and clutter. The use of a configuration file also introduces the opportunity for multiple application processes to share common OpenDDS configuration. The above example can easily be moved to the `[common]` section of a configuration file (assume a file of *pub.ini*):

```
[common]
DCPSInfoRepo=myhost.mydomain.com:12345
```

The command line to start our executable would now change to the following:

```
publisher -DCSPConfigFile pub.ini
```

Reinforcing our example from the discussion of domains in section , a configuration file can specify domains with discovery configuration assigned to those domains. In this case the `RepositoryIor` property is used to take the same information that would be supplied on a command line to point to a running `DCPSInfoRepo` service. Two domains are configured here:

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=myhost.mydomain.com:12345

[domain/2]
DiscoveryConfig=DiscoveryConfig2

[repository/DiscoveryConfig2]
RepositoryIor=host2.mydomain.com:12345
```

The `DiscoveryConfig` property under `[domain/1]` instructs all participants in domain 1 to use the configuration defined in an instance called `DiscoveryConfig1`. In the above, this is

mapped to a `[repository]` section that gives the `RepositoryIor` value of *myhost.mydomain.com:12345*.

Finally, when configuring a `DCPSInfoRepo` the `DiscoveryConfig` property under a domain instance entry can also contain the value of `DEFAULT_REPO` which instructs a participant using this instance to use the definition of the property `DCPSInfoRepo` wherever it has been supplied. Consider the following configuration file as an example:

```
[common]
DCPSInfoRepo=localhost:12345

[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=myhost.mydomain.com:12345

[domain/2]
DiscoveryConfig=DEFAULT_REPO
```

In this case any participant in domain 2 would be instructed to refer to the discovery property of `DCPSInfoRepo`, which is defined in the `[common]` section of our example. If the `DCPSInfoRepo` value is not supplied in the `[common]` section, it could alternatively be supplied as a parameter to the command line like so:

```
publisher -DCPSInfoRepo localhost:12345 -DCPSConfigFile pub.ini
```

This sets the value of `DCPSInfoRepo` such that if participants reading the configuration file pub.ini encounters `DEFAULT_REPO`, there is a value for it. If `DCPSInfoRepo` is not defined in a configuration file or on the command line, then the OpenDDS default value for `DCPSInfoRepo` is *file://repo.ior*. As mentioned prior, this is not likely to be the most useful in production environments and should lead to setting the value of `DCPSInfoRepo` by one of the means described in this section.

7.3.2.1 Configuring for Multiple DCPSInfoRepo Instances

The DDS entities in a single OpenDDS process can be associated with multiple DCPS information repositories (`DCPSInfoRepo`).

The repository information and domain associations can be configured using a configuration file, or via application API. Internal defaults, command line arguments, and configuration file options will work as-is for existing applications that do not want to use multiple `DCPSInfoRepo` associations.

Refer to Figure 7-1 as an example of a process that uses multiple `DCPSInfoRepo` repositories. Processes A and B are typical application processes that have been configured to communicate with one another and discover one another in `InfoRepo_1`. This is a simple

O p e n D D S   D e v e l o p e r ' s   G u i d e

use of basic discovery. However, an additional layer of context has been applied with the use of a specified domain (Domain 1). DDS entities (data readers/data writers) are restricted to communicate to other entities within that same domain. This provides a useful method of separating traffic when needed by an application. Processes C and D are configured the same way, but operate in Domain 2 and use InfoRepo_2. The challenge comes when you have an application process that needs to use multiple domains and have separate discovery services. This is Process E in our example. It contains two subscribers, one subscribing to publications from InfoRepo_1 and the other subscribing to publications in InfoRepo_2. What allows this configuration to work can be found in the *configE.ini* file.
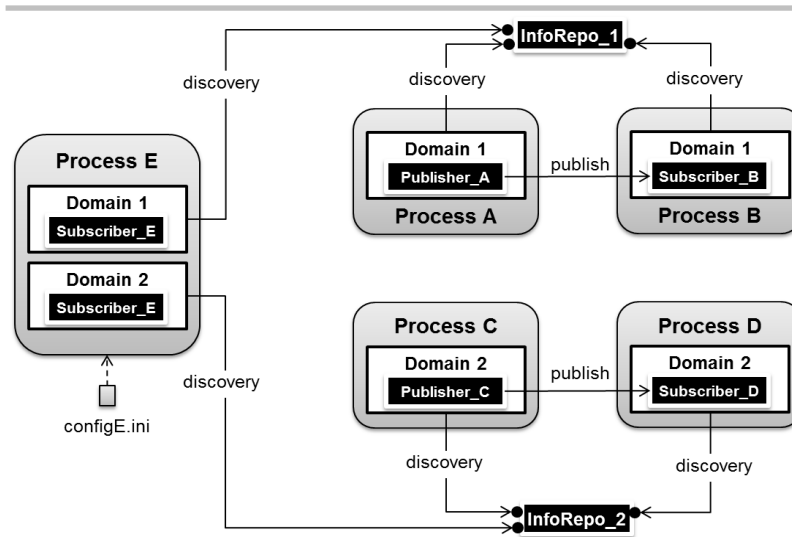
Figure 7-1 Multiple DCPSInfoRepo Configuration

We will now look at the configuration file (referred to as *configE.ini*) to demonstrate how Process E can communicate to both domains and separate DCPSInfoRepo services. For this example we will only show the discovery aspects of the configuration and not show transport content.

configE.ini

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=host1.mydomain.com:12345

[domain/2]
DiscoveryConfig=DiscoveryConfig2

[repository/DiscoveryConfig2]
RepositoryIor=host2.mydomain.com:12345
```

OpenDDS Developer's Guide

When Process E in Figure 7-1 reads in the above configuration it finds the occurrence of multiple domain sections. As described in Section each domain has an instance integer and a property of `DiscoveryConfig` defined.

For the first domain (`[domain/1]`), the `DiscoveryConfig` property is supplied with the user-defined name of `DiscoveryConfig1` value. This property causes the OpenDDS implementation to find a section title of either `repository` or `rtps_discovery` and an instance name of `DiscoveryConfig1`. In our example, a `[repository/DiscoveryConfig1]` section title is found and this becomes the discovery configuration for domain instance `[domain/1]` (integer value 1). The section found now tells us that the address of the `DCPSInfoRepo` that this domain should use can be found by using the `RepositoryIor` property value. In particular it is *host1.mydomain.com* and port *12345*. The values of the `RepositoryIor` can be a full CORBA IOR or a simple *host:port* string.

A second domain section title `[domain/2]` is found in this configuration file along with it's corresponding repository section `[repository/DiscoveryConfig2]` that represents the configuration for the second domain of interest and the `InfoRepo_2` repository. There may be any number of repository or domain sections within a single configuration file.

Note *Domains not explicitly configured are automatically associated with the default discovery configuration.*

Note *Individual DCPSInfoRepos can be associated with multiple domains, however domains cannot be shared between multiple DCPSInfoRepos.*

Here are the valid properties for a `[repository]` section.

Table 7-4 Multiple repository configuration sections

Option	Description
`RepositoryIor=ior`	Repository IOR or host:port.
`RepositoryKey=key`	Unique key value for the repository. (Deprecated. Provided for backward compatibility)

7.3.3 Configuring For DDSI-RTPS Discovery

The OMG DDSI-RTPS specification gives the following simple description that forms the basis for the discovery approach used by OpenDDS and the two different protocols used to accomplish the discovery operations. The excerpt from the OMG DDSI-RTPS specification Section 8.5.1 is as follows:

"The RTPS specification splits up the discovery protocol into two independent protocols:

1. Participant Discovery Protocol

2. Endpoint Discovery Protocol

A Participant Discovery Protocol (PDP) specifies how Participants discover each other in the network. Once two Participants have discovered each other, they exchange information on the Endpoints they contain using an Endpoint Discovery Protocol (EDP). Apart from this causality relationship, both protocols can be considered independent."

The configuration options discussed in this section allow a user to specify property values to change the behavior of the Simple Participant Discovery Protocol (SPDP) and/or the Simple Endpoint Discovery Protocol (SEDP) default settings.

DDSI-RTPS can be configured for a single domain or for multiple domains as was done in Section 7.3.2.1.

A simple configuration is achieved by specifying a property in the `[common]` section of our example configuration file.

configE.ini (for RTPS)

```
[common]
DCPSDefaultDiscovery=DEFAULT_RTPS
```

All default values for DDSI-RTPS discovery are adopted in this form. A variant of this same basic configuration is to specify a section to hold more specific parameters of RTPS discovery. The following example uses the `[common]` section to point to an instance of an `[rtps_discovery]` section followed by an instance name of `TheRTPSConfig` which is supplied by the user.

```
[common]
DCPSDefaultDiscovery=TheRTPSConfig

[rtps_discovery/TheRTPSConfig]
ResendPeriod=5
```

The instance `[rtps_discovery/TheRTPSConfig]` is now the location where properties that vary the default DDSI-RTPS settings get specified. In our example the `ResendPeriod=5` entry sets the number of seconds between periodic announcements of available data readers / data writers and to detect the presence of other data readers / data writers on the network. This would override the default of 30 seconds.

If your OpenDDS deployment uses multiple domains, the following configuration approach combines the use of the `[domain]` section title with `[rtps_discovery]` to allow a user to specify particular settings by domain. It might look like this:

configE.ini

```
[common]
DCPSDebugLevel=0
```

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[rtps_discovery/DiscoveryConfig1]
ResendPeriod=5

[domain/2]
DiscoveryConfig=DiscoveryConfig2

[rtps_discovery/DiscoveryConfig2]
ResendPeriod=5
SedpMulticast=0
```

Some important implementation notes regarding DDSI-RTPS discovery in OpenDDS are as follows:

1) Domain IDs should be between 0 and 231 (inclusive) due to the way UDP ports are assigned to domain IDs. In each OpenDDS process, up to 120 domain participants are supported in each domain.

2) OpenDDS's multicast transport (7.4.5.4) does not work with RTPS Discovery due to the way GUIDs are assigned (a warning will be issued if this is attempted).

The OMG DDSI-RTPS specification details several properties that can be adjusted from their defaults that influence the behavior of DDSI-RTPS discovery. Those properties, along with options specific to OpenDDS's RTPS Discovery implementation, are listed in Table 7-5.

Table 7-5 RTPS Discovery Configuration Options

Option	Description	Default
ResendPeriod=sec	The number of seconds that a process waits between the announcement of participants (see section 8.5.3 in the OMG DDSI-RTPS specification for details).	30
MinResendDelay=msec	The minimum time in milliseconds between participant announcements.	100
QuickResendRatio=frac	Tuning parameter that configures local SPDP resends as a fraction of the resend period.	0.1
LeaseDuration=sec	Sent as part of the participant announcement. It tells the peer participants that if they don't hear from this participant for the specified duration, then this participant can be considered "not alive."	300
LeaseExtension=sec	Extends the lease of discovered participants by the set amount of seconds. Useful on spotty connections to reduce load on the RtpsRelay.	0

Option	Description	Default
PB=port	Port Base number. This number sets the starting point for deriving port numbers used for Simple Endpoint Discovery Protocol (SEDP). This property is used in conjunction with DG, PG, D0 (or DX), and D1 to construct the necessary Endpoints for RTPS discovery communication. (see section 9.6.1.1 in the OMG DDSI-RTPS specification in how these Endpoints are constructed)	7400
DG=n	An integer value representing the Domain Gain. This is a multiplier that assists in formulating Multicast or Unicast ports for RTPS.	250
PG=n	An integer that assists in configuring SPDP Unicast ports and serves as an offset multiplier as participants are assigned addresses using the formula: PB + DG * domainId + d1 + PG * participantId (see section 9.6.1.1 in the OMG DDSI-RTPS specification in how these Endpoints are constructed)	2
D0=n	An integer value that assists in providing an offset for calculating an assignable port in SPDP Multicast configurations. The formula used is: PB + DG * domainId + d0 (see section 9.6.1.1 in the OMG DDSI-RTPS specification in how these Endpoints are constructed)	0
D1=n	An integer value that assists in providing an offset for calculating an assignable port in SPDP Unicast configurations. The formula used is: PB + DG * domainId + d1 + PG * participantId (see section 9.6.1.1 in the OMG DDSI-RTPS specification in how these Endpoints are constructed)	10

Option	Description	Default
`SedpMaxMessageSize=n`	Set the maximum SEDP message size. The default is the maximum UDP message size. See max_message_size in table 7-17.	65466
`SedpMulticast=[0|1]`	A boolean value (0 or 1) that determines whether Multicast is used for the SEDP traffic. When set to 1, Multicast is used. When set to zero (0) Unicast for SEDP is used.	1
`SedpLocalAddress=addr:[port]`	Configure the transport instance created and used by SEDP to bind to the specified local address and port. In order to leave the port unspecified, it can be omitted from the setting but the trailing : must be present.	`System default address`
`SpdpLocalAddress=addr[:port]`	Address of a local interface, which will be used by SPDP to bind to that specific interface.	`DCPSDefaultAddress, or IPADDR_ANY`
`SedpAdvertisedLocalAddress= addr:[port]`	Sets the address advertised by SEDP. Typically used when the participant is behind a firewall or NAT. In order to leave the port unspecified, it can be omitted from the setting but the trailing : must be present.	
`SedpSendDelay=msec`	Time in milliseconds for a built-in (SEDP) Writer to wait before sending data.	10
`SedpHeartbeatPeriod=msec`	Time in milliseconds for a built-in (SEDP) Writer to announce the availability of data.	1000
`SedpNakResponseDelay=msec`	Time in milliseconds for a built-in (SEDP) Writer to delay the response to a negative acknowledgment.	200
`DX=n`	An integer value that assists in providing an offset for calculating a port in SEDP Multicast configurations. The formula used is: `PB + DG * domainId + dx` This is only valid when `SedpMulticast=1`. This is an OpenDDS extension and not part of the OMG DDSI-RTPS specification.	2
`SpdpSendAddrs= [host:port],[host:port]...`	A list (comma or whitespace separated) of host:port pairs used as destinations for SPDP content. This can be a combination of Unicast and Multicast addresses.	
`MaxSpdpSequenceMsgResetChecks=n`	Remove a discovered participant after this number of SPDP messages with earlier sequence numbers.	3

Option	Description	Default
PeriodicDirectedSpdp=[0|1]	A boolean value that determines whether directed SPDP messages are sent to all participants once every resend period. This setting should be enabled for participants that cannot use multicast to send SPDP announcements, e.g., an RtpsRelay.	0
UndirectedSpdp=[0|1]	A boolean value that determines whether undirected SPDP messages are sent. This setting should be disabled for participants that cannot use multicast to send SPDP announcements, e.g., an RtpsRelay.	1
InteropMulticastOverride= group_address	A network address specifying the multicast group to be used for SPDP discovery. This overrides the interoperability group of the specification. It can be used, for example, to specify use of a routed group address to provide a larger discovery scope.	239.255.0.1
TTL=n	The value of the Time-To-Live (TTL) field of multicast datagrams sent as part of discovery. This value specifies the number of hops the datagram will traverse before being discarded by the network. The default value of 1 means that all data is restricted to the local network subnet.	1
MulticastInterface=iface	Specifies the network interface to be used by this discovery instance. This uses a platform-specific format that identifies the network interface. On Linux systems this would be something like eth0. If this value is not configured, the Common Configuration value DCPSDefaultAddress is used to set the multicast interface.	The system default interface is used
GuidInterface=iface	Specifies the network interface to use when determining which local MAC address should appear in a GUID generated by this node.	The system / ACE library default is used
SpdpRtpsRelayAddress=host:port	Specifies the address of the RtpsRelay for SPDP messages. See section 15.2.	
SpdpRtpsRelaySendPeriod=period	Specifies the interval between SPDP announcements sent to the RtpsRelay. See section 15.2.	30 seconds

O p e n D D S   D e v e l o p e r ' s   G u i d e

Option	Description	Default
`SedpRtpsRelayAddress=host:port`	Specifies the address of the RtpsRelay for SEDP messages. See section 15.2.	
`RtpsRelayOnly=[0|1]`	Only send RTPS message to the RtpsRelay (for debugging). See section 15.2.	0
`UseRtpsRelay=[0|1]`	Send messages to the RtpsRelay. Messages will only be sent if SpdpRtpsRelayAddress and/or SedpRtpsRelayAddress is set. See section 15.2.	0
`SpdpStunServerAddress=host:port`	Specifies the address of the STUN server to use for SPDP when using ICE. See section 15.3	
`SedpStunServerAddress=host:port`	Specifies the address of the STUN server to use for SEDP when using ICE. See section 15.3.	
`UseIce=[0|1]`	Enable or disable ICE for both SPDP and SEDP. See section 15.3.	0
`IceTa=msec`	Minimum interval between ICE sends. See section 15.3.	50
`IceConnectivityCheckTTL=sec`	Maximum duration of connectivity check. See section 15.3.	300
`IceChecklistPeriod=sec`	Attempt to cycle through all of the connectivity checks for a candidate in this amount of time. See section 15.3.	10
`IceIndicationPeriod=sec`	Send STUN indications to peers to maintain NAT bindings at this period. See section 15.3.	15
`IceNominatedTTL=sec`	Forget a valid candidate if an indication is not received in this amount of time. See section 15.3.	300
`IceServerReflexiveAddressPeriod=sec`	Send a messages to the STUN server at this period. See section 15.3.	30
`IceServerReflexiveIndicationCount=integer`	Send this many indications before sending a new binding request to the STUN server. See section 15.3.	10
`IceDeferredTriggeredCheckTTL=sec`	Purge deferred checks after this amount of time. See section 15.3.	300
`IceChangePasswordPeriod=sec`	Change the ICE password after this amount of time. See section 15.3.	300
`MaxAuthTime=sec`	Set the maximum time for authentication with DDS Security.	300
`AuthResendPeriod=sec`	Resend authentication messages after this amount of time.	1

Option	Description	Default
`SecureParticipantUserData=[0|1]`	If DDS Security is enabled, the Participant's USER_DATA QoS is omitted from unsecured discovery messages.	0
`UseXTypes=[0|1]`	Enables discovery extensions from the XTypes specification. Participants exchange top-level type information in endpoint announcements and extended type information using the Type Lookup Service.	1
`TypeLookupServiceReplyTimeout=msec`	If a request is sent to a peer's Type Lookup Service (see UseXTypes above), wait up to this duration (in milliseconds) for a reply.	5000 (5 seconds)
SedpResponsiveMode=[0|1]	Causes the built-in SEDP endpoints to send additional messages which may reduce latency.	0
SedpPassiveConnectDuration=msec	Sets the duration that a passive endpoint will wait for a connection.	60000 (1 minute)
SendBufferSize=bytes	Socket send buffer size for both SPDP and SEDP. A value of zero indicates that the system default value is used.	0
RecvBufferSize=bytes	Socket receive buffer size for both SPDP and SEDP. A value of zero indicates that the system default value is used.	0

Note *If the environment variable* `OPENDDS_RTPS_DEFAULT_D0` *is set, its value is used as the* `D0` *default value.*

7.3.4   Configuring For Static Discovery

Static discovery may be used when a DDS domain has a fixed number of processes and data readers/writers that are all known *a priori*. Data readers and writers are collectively known as *endpoints*. Using only the configuration file, the static discovery mechanism must be able to determine a network address and the QoS settings for each endpoint. The static discovery mechanism uses this information to determine all potential associations between readers and writers. A domain participant learns about the existence of an endpoint through hints supplied by the underlying transport.

Currently, static discovery can only be used for endpoints using the RTPS UDP transport.

Static discovery introduces the following configuration file sections: `[topic/*]`, `[datawriterqos/*]`, `[datareaderqos/*]`, `[publisherqos/*]`, `[subscriberqos/*]`, and `[endpoint/*]`. The `[topic/*]` (Table 7-6) section is used to introduce a topic. The `[datawriterqos/*]` (Table 7-7), `[datareaderqos/*]` (Table 7-8), `[publisherqos/*]` (Table

7-9), and `[subscriberqos/*]` (Table 7-10) sections are used to describe a QoS of the associated type. The `[endpoint/*]` (Table 7-11) section describes a data reader or writer.

Data reader and writer objects must be identified by the user so that the static discovery mechanism can associate them with the correct `[endpoint/*]` section in the configuration file. This is done by setting the `user_data` of the `DomainParticipantQos` to an octet sequence of length 6. The representation of this octet sequence occurs in the `participant` value of an `[endpoint/*]` section as a string with two hexadecimal digits per octet. Similarly, the `user_data` of the `DataReaderQos` or `DataWriterQos` must be set to an octet sequence of length 3 corresponding to the `entity` value in the `[endpoint/*]` section. For example, suppose the configuration file contains the following:

```
[topic/MyTopic]
type_name=TestMsg::TestMsg

[endpoint/MyReader]
type=reader
topic=MyTopic
config=MyConfig
domain=34
participant=0123456789ab
entity=cdef01

[config/MyConfig]
transports=MyTransport

[transport/MyTransport]
transport_type=rtps_udp
use_multicast=0
local_address=1.2.3.4:30000
```

The corresponding code to configure the `DomainParticipantQos` is:

```
DDS::DomainParticipantQos dp_qos;
domainParticipantFactory->get_default_participant_qos(dp_qos);
dp_qos.user_data.value.length(6);
dp_qos.user_data.value[0] = 0x01;
dp_qos.user_data.value[1] = 0x23;
dp_qos.user_data.value[2] = 0x45;
dp_qos.user_data.value[3] = 0x67;
dp_qos.user_data.value[4] = 0x89;
dp_qos.user_data.value[5] = 0xab;
```

The code to configure the DataReaderQos is similar:

```
DDS::DataReaderQos qos;
subscriber->get_default_datareader_qos(qos);
qos.user_data.value.length(3);
qos.user_data.value[0] = 0xcd;
qos.user_data.value[1] = 0xef;
qos.user_data.value[2] = 0x01;
```

The domain id, which is 34 in the example, should be passed to the call to `create_participant`.

In the example, the endpoint configuration for `MyReader` references `MyConfig` which in turn references `MyTransport`. Transport configuration is described in Section 7.4. The important detail for static discovery is that at least one of the transports contains a known network address (`1.2.3.4:30000`). An error will be issued if an address cannot be determined for an endpoint. The static discovery implementation also checks that the QoS of a data reader or data writer object matches the QoS specified in the configuration file.

Table 7-6 [topic/*] Configuration Options

Option	Description	Default
`name=string`	The name of the topic.	Instance name of section
`type_name=string`	Identifier which uniquely defines the sample type. This is typically a CORBA interface repository type name.	Required

Table 7-7 [datawriterqos/*] Configuration Options

Option	Description	Default
`durability.kind=[` `   VOLATILE|TRANSIENT_LOCAL]`	See Section 3.2.5.	See Table 3-5.
`deadline.period.sec=[` `   numeric|DURATION_INFINITE_SEC]`	See Section 3.2.9.	See Table 3-5.
`deadline.period.nanosec=[` `   numeric|DURATION_INFINITE_NANOSEC]`	See Section 3.2.9.	See Table 3-5.
`latency_budget.duration.sec=[` `   numeric|DURATION_INFINITE_SEC]`	See Section 3.2.15.	See Table 3-5.
`latency_budget.duration.nanosec=[` `   numeric|DURATION_INFINITE_NANOSEC]`	See Section 3.2.15.	See Table 3-5.

`liveliness.kind=[`
`   AUTOMATIC|`
`   MANUAL_BY_TOPIC|`
`   MANUAL_BY_PARTICIPANT]`
	See Section 3.2.2.	See Table 3-5.

`liveliness.lease_duration.sec=[` `   numeric|DURATION_INFINITE_SEC]`	See Section 3.2.2.	See Table 3-5.
`liveliness.lease_duration.nanosec=[` `   numeric|DURATION_INFINITE_NANOSEC]`	See Section 3.2.2.	See Table 3-5.
`reliability.kind=[BEST_EFFORT|RELIABILE]`	See Section 3.2.3.	See Table 3-5.
`reliability.max_blocking_time.sec=[` `   numeric|DURATION_INFINITE_SEC]`	See Section 3.2.3.	See Table 3-5.
`reliability.max_blocking_time.nanosec=[` `   numeric|DURATION_INFINITE_NANOSEC]`	See Section 3.2.3.	See Table 3-5.

Option	Description	Default

destination_order.kind=[
  BY_SOURCE_TIMESTAMP|
  BY_RECEPTION_TIMESTAMP]
	See Section 3.2.18.	See Table 3-5.

history.kind=[KEEP_LAST|KEEP_ALL]	See Section 3.2.4.	See Table 3-5.
history.depth=numeric	See Section 3.2.4.	See Table 3-5.
resource_limits.max_samples=numeric	See Section 3.2.7.	See Table 3-5.
resource_limits.max_instances=numeric	See Section 3.2.7.	See Table 3-5.
resource_limits.max_samples_per_instance= numeric	See Section 3.2.7.	See Table 3-5.
transport_priority.value=numeric	See Section 3.2.14.	See Table 3-5.
lifespan.duration.sec=[numeric|DURATION_INFINITE_SEC]	See Section 3.2.10.	See Table 3-5.
lifespan.duration.nanosec=[numeric|DURATION_INFINITE_NANOSEC]	See Section 3.2.10.	See Table 3-5.
ownership.kind=[SHARED|EXCLUSIVE]	See Section 3.2.22.	See Table 3-5.
ownership_strength.value=numeric	See Section 3.2.23.	See Table 3-5.

Table 7-8 [datareaderqos/*] Configuration Options

Option	Description	Default
durability.kind=[VOLATILE|TRANSIENT_LOCAL]	See Section 3.2.5.	See Table 3-6.
deadline.period.sec=[numeric|DURATION_INFINITE_SEC]	See Section 3.2.9.	See Table 3-6.
deadline.period.nanosec=[numeric|DURATION_INFINITE_NANOSEC]	See Section 3.2.9.	See Table 3-6.
latency_budget.duration.sec=[numeric|DURATION_INFINITE_SEC]	See Section 3.2.15.	See Table 3-6.
latency_budget.duration.nanosec=[numeric|DURATION_INFINITE_NANOSEC]	See Section 3.2.15.	See Table 3-6.

liveliness.kind=[
  AUTOMATIC|
  MANUAL_BY_TOPIC|
  MANUAL_BY_PARTICIPANT]
	See Section 3.2.2.	See Table 3-6.

liveliness.lease_duration.sec=[numeric|DURATION_INFINITE_SEC]	See Section 3.2.2.	See Table 3-6.
liveliness.lease_duration.nanosec=[numeric|DURATION_INFINITE_NANOSEC]	See Section 3.2.2.	See Table 3-6.
reliability.kind=[BEST_EFFORT|RELIABILE]	See Section 3.2.3.	See Table 3-6.
reliability.max_blocking_time.sec=[numeric|DURATION_INFINITE_SEC]	See Section 3.2.3.	See Table 3-6.

Option	Description	Default
`reliability.max_blocking_time.nanosec=[` `    numeric|DURATION_INFINITE_NANOSEC]`	See Section 3.2.3.	See Table 3-6.

`destination_order.kind=[`
`    BY_SOURCE_TIMESTAMP|`
`    BY_RECEPTION_TIMESTAMP]`
	See Section 3.2.18.	See Table 3-6.

`history.kind=[KEEP_LAST|KEEP_ALL]`	See Section 3.2.4.	See Table 3-6.
`history.depth=numeric`	See Section 3.2.4.	See Table 3-6.
`resource_limits.max_samples=numeric`	See Section 3.2.7.	See Table 3-6.
`resource_limits.max_instances=numeric`	See Section 3.2.7.	See Table 3-6.
`resource_limits.max_samples_per_instance=` `    numeric`	See Section 3.2.7.	See Table 3-6.
`time_based_filter.minimum_separation.sec=[` `    numeric|DURATION_INFINITE_SEC]`	See Section 3.2.21.	See Table 3-6.
`time_based_filter.minimum_separation.nanosec=[` `    numeric|DURATION_INFINITE_NANOSEC]`	See Section 3.2.21.	See Table 3-6.

`reader_data_lifecycle.`
`autopurge_nowriter_samples_delay.sec=[`
`    numeric|DURATION_INFINITE_SEC]`
	See Section 3.2.20.	See Table 3-6.

`reader_data_lifecycle.`
`autopurge_nowriter_samples_delay.nanosec=[`
`    numeric|DURATION_INFINITE_NANOSEC]`
	See Section 3.2.20.	See Table 3-6.

`reader_data_lifecycle.`
`autopurge_dispose_samples_delay.sec=[`
`    numeric|DURATION_INFINITE_SEC]`
	See Section 3.2.20.	See Table 3-6.

`reader_data_lifecycle.`
`autopurge_dispose_samples_delay.nanosec=[`
`    numeric|DURATION_INFINITE_NANOSEC]`
	See Section 3.2.20.	See Table 3-6.

Table 7-9 [publisherqos/*] Configuration Options

Option	Description	Default
`presentation.access_scope=[INSTANCE|TOPIC|GROUP]`	See Section 3.2.17.	See Table 3-3.
`presentation.coherent_access=[true|false]`	See Section 3.2.17.	See Table 3-3.
`presentation.ordered_access=[true|false]`	See Section 3.2.17.	See Table 3-3.
`partition.name=name0,name1,...`	See Section 3.2.8.	See Table 3-3.

Table 7-10 [subscriberqos/*] Configuration Options

Option	Description	Default
`presentation.access_scope=[INSTANCE|TOPIC|GROUP]`	See Section 3.2.17.	See Table 3-4.
`presentation.coherent_access=[true|false]`	See Section 3.2.17.	See Table 3-4.
`presentation.ordered_access=[true|false]`	See Section 3.2.17.	See Table 3-4.

Option	Description	Default
`partition.name=name0,name1,...`	See Section 3.2.8.	See Table 3-4.

Table 7-11 [endpoint/*] Configuration Options

Option	Description	Default
`domain=numeric`	Domain id for endpoint in range 0-231. Used to form GUID of endpoint.	Required
`participant=hexstring`	String of 12 hexadecimal digits. Used to form GUID of endpoint. All endpoints with the same domain/participant combination should be in the same process.	Required
`entity=hexstring`	String of 6 hexadecimal digits. Used to form GUID of endpoint. The combination of domain/participant/entity should be unique.	Required
`type=[reader|writer]`	Determines if the entity is a data reader or data writer.	Required
`topic=name`	Refers to a [`topic/*`] section.	Required
`datawriterqos=name`	Refers to a [`datawriterqos/*`] section.	See Table 3-5.
`datareaderqos=name`	Refers to a [`datareaderqos/*`] section.	See Table 3-6.
`publisherqos=name`	Refers to a [`publisherqos/*`] section.	See Table 3-3.
`subscriberqos=name`	Refers to a [`subscriberqos/*`] section.	See Table 3-4.
`config`	Refers to a transport configuration in a [`config/*`] section. This is used to determine a network address for the endpoint.	

7.4 Transport Configuration

Beginning with OpenDDS 3.0, a new transport configuration design has been implemented. The basic goals of this design were to:

- Allow simple deployments to ignore transport configuration and deploy using intelligent defaults (with no transport code required in the publisher or subscriber).

- Enable flexible deployment of applications using only configuration files and command line options.

- Allow deployments that mix transports within individual data writers and writers. Publishers and subscribers negotiate the appropriate transport implementation to use based on the details of the transport configuration, QoS settings, and network reachability.

- Support a broader range of application deployments in complex networks.

- Support optimized transport development (such as collocated and shared memory transports - note that these are not currently implemented).

- Integrate support for the `RELIABILITY` QoS policy with the underlying transport.

- Whenever possible, avoid dependence on the ACE Service Configurator and its configuration files.

Unfortunately, implementing these new capabilities involved breaking of backward compatibility with OpenDDS transport configuration code and files from previous releases. See `$DDS_ROOT/docs/OpenDDS_3.0_Transition.txt` for information on how to convert your existing application to use the new transport configuration design.

7.4.1 Overview

7.4.1.1 Transport Concepts

This section provides an overview of the concepts involved in transport configuration and how they interact.

Each data reader and writer uses a *Transport Configuration* consisting of an ordered set of *Transport Instances*. Each Transport Instance specifies a Transport Implementation (i.e. tcp, udp, multicast, shmem, or rtps_udp) and can customize the configuration parameters defined by that transport. Transport Configurations and Transport Instances are managed by the *Transport Registry* and can be created via configuration files or through programming APIs.

Transport Configurations can be specified for Domain Participants, Publishers, Subscribers, Data Writers, and Data Readers. When a Data Reader or Writer is enabled, it uses the most specific configuration it can locate, either directly bound to it or accessible through its parent entity. For example, if a Data Writer specifies a Transport Configuration, it always uses it. If the Data Writer does not specify a configuration, it tries to use that of its Publisher or Domain Participant in that order. If none of these entities have a transport configuration specified, the *Global Transport Configuration* is obtained from the Transport Registry. The Global Transport Configuration can be specified by the user via either configuration file, command line option, or a member function call on the Transport Registry. If not defined by the user, a default transport configuration is used which contains all available transport implementations with their default configuration parameters. If you don't specifically load or link in any other transport implementations, OpenDDS uses the tcp transport for all communication.

7.4.1.2 How OpenDDS Selects a Transport

Currently, the behavior for OpenDDS is that Data Writers actively connect to Data Readers, which are passively awaiting those connections. Data Readers "listen" for connections on each of the Transport Instances that are defined in their Transport Configuration. Data Writers use their Transport Instances to "connect" to those of the Data Readers. Because the logical connections discussed here don't correspond to the physical connections of the transport, OpenDDS often refers to them as *Data Links*.

When a Data Writer tries to connect to a Data Reader, it first attempts to see if there is an existing data link that it can use to communicate with that Data Reader. The Data Writer iterates (in definition order) through each of its Transport Instances and looks for an existing data link to the Transport Instances that the reader defined. If an existing data link is found it is used for all subsequent communication between the Data Writer and Reader.

If no existing data link is found, the Data Writer attempts to connect using the different Transport Instances in the order they are defined in its Transport Configuration. Any Transport Instances not "matched" by the other side are skipped. For example, if the writer specifies udp and tcp transport instances and the reader only specifies tcp, the udp transport instance is ignored. Matching algorithms may also be affected by QoS parameters, configuration of the instances, and other specifics of the transport implementation. The first pair of Transport Instances that successfully "connect" results in a data link that is used for all subsequent data sample publication.

7.4.2 Configuration File Examples

The following examples explain the basic features of transport configuration via files and describe some common use cases. These are followed by full reference documentation for these features.

7.4.2.1 Single Transport Configuration

The simplest way to provide a transport configuration for your application is to use the OpenDDS configuration file. Here is a sample configuration file that might be used by an application running on a computer with two network interfaces that only wants to communicate using one of them:

```
[common]
DCPSGlobalTransportConfig=myconfig

[config/myconfig]
transports=mytcp

[transport/mytcp]
```

```
transport_type=tcp
local_address=myhost
```

This file does the following (starting from the bottom up):

1) Defines a transport instance named `mytcp` with a transport type of tcp and the local address specified as `myhost`, which is the host name corresponding to the network interface we want to use.

2) Defines a transport configuration named `myconfig` that uses the transport instance `mytcp` as its only transport.

3) Makes the transport configuration named `myconfig` the global transport configuration for all entities in this process.

A process using this configuration file utilizes our customized transport configuration for all Data Readers and Writers created by it (unless we specifically bind another configuration in the code as described in 7.4.2.3).

7.4.2.2   Using Mixed Transports

This example configures an application to primarily use multicast and to "fall back" to tcp when it is unable to use multicast. Here is the configuration file:

```
[common]
DCPSGlobalTransportConfig=myconfig

[config/myconfig]
transports=mymulticast,mytcp

[transport/mymulticast]
transport_type=multicast

[transport/mytcp]
transport_type=tcp
```

The transport configuration named `myconfig` now includes two transport instances, `mymulticast` and `mytcp`. Neither of these transport instances specify any parameters besides transport_type, so they use the default configuration of these transport implementations. Users are free to use any of the transport-specific configuration parameters that are listed in the following reference sections.

Assuming that all participating processes use this configuration file, the application attempts to use multicast to initiate communication between data writers and readers. If the initial multicast communication fails for any reason (possibly because an intervening router is not passing multicast traffic) tcp is used to initiate the connection.

7.4.2.3 **Using Multiple Configurations**

For many applications, one configuration is not equally applicable to all communication within a given process. These applications must create multiple Transport Configurations and then assign them to the different entities of the process.

For this example consider an application hosted on a computer with two network interfaces that requires communication of some data over one interface and the remainder over the other interface. Here is our configuration file:

```
[common]
DCPSGlobalTransportConfig=config_a

[config/config_a]
transports=tcp_a

[config/config_b]
transports=tcp_b

[transport/tcp_a]
transport_type=tcp
local_address=hosta

[transport/tcp_b]
transport_type=tcp
local_address=hostb
```

Assuming `hosta` and `hostb` are the host names assigned to the two network interfaces, we now have separate configurations that can use tcp on the respective networks. The above file sets the "A" side configuration as the default, meaning we must manually bind any entities we want to use the other side to the "B" side configuration.

OpenDDS provides two mechanisms to assign configurations to entities:

- Via source code by attaching a configuration to an entity (reader, writer, publisher, subscriber, or domain participant)

- Via configuration file by associating a configuration with a domain

Here is the source code mechanism (using a domain participant):

```
DDS::DomainParticipant_var dp =
        dpf->create_participant(MY_DOMAIN,
                                PARTICIPANT_QOS_DEFAULT,
                                DDS::DomainParticipantListener::_nil(),
                                OpenDDS::DCPS::DEFAULT_STATUS_MASK);

OpenDDS::DCPS::TransportRegistry::instance()->bind_config("config_b", dp);
```

Any Data Writers or Readers owned by this Domain Participant should now use the "B" side configuration.

Note	*When directly binding a configuration to a data writer or reader, the* `bind_config` *call must occur before the reader or writer is enabled. This is not an issue when binding configurations to Domain Participants, Publishers, or Subscribers. See Section 3.2.16 for details on how to create entities that are not enabled.*

7.4.3   # Transport Registry Example

OpenDDS allows developers to also define transport configurations and instances via C++ APIs. The `OpenDDS::DCPS::TransportRegistry` class is used to construct `OpenDDS::DCPS::TransportConfig` and `OpenDDS::DCPS::TransportInst` objects. The `TransportConfig` and `TransportInst` classes contain public data member corresponding to the options defined below. This section contains the code equivalent of the simple transport configuration file described in . First, we need to include the correct header files:

```
#include <dds/DCPS/transport/framework/TransportRegistry.h>
#include <dds/DCPS/transport/framework/TransportConfig.h>
#include <dds/DCPS/transport/framework/TransportInst.h>
#include <dds/DCPS/transport/tcp/TcpInst.h>

using namespace OpenDDS::DCPS;
```

Next we create the transport configuration, create the transport instance, configure the transport instance, and then add the instance to the configuration's collection of instances:

```
TransportConfig_rch cfg = TheTransportRegistry->create_config("myconfig");
TransportInst_rch inst = TheTransportRegistry->create_inst("mytcp", // name
                                                   "tcp");  // type

// Must cast to TcpInst to get access to transport-specific options
TcpInst_rch tcp_inst = dynamic_rchandle_cast<TcpInst>(inst);
tcp_inst->local_address_str_ = "myhost";

// Add the inst to the config
cfg->instances_.push_back(inst);
```

Lastly, we can make our newly defined transport configuration the global transport configuration:

```
TheTransportRegistry->global_config(cfg);
```

This code should be executed before any Data Readers or Writers are enabled.

See the header files included above for the full list of public data members and member functions that can be used. See the option descriptions in the following sections for a full understanding of the semantics of these settings.

Stepping back and comparing this code to the original configuration file from , the configuration file is much simpler than the corresponding C++ code and has the added advantage of being modifiable at run-time. It is easy to see why we recommend that almost all applications should use the configuration file mechanism for transport configuration.

7.4.4   Transport Configuration Options

Transport Configurations are specified in the OpenDDS configuration file via sections with the format of [config/<name>], where <name> is a unique name for that configuration within that process. The following table summarizes the options when specifying a transport configuration:

Table 7-12 Transport Configuration Options

Option	Description	Default
transports=*inst1[,inst2][,...]*	The ordered list of transport instance names that this configuration will utilize. This field is required for every transport configuration.	none
swap_bytes=*[0|1]*	A value of 0 causes DDS to serialize data in the source machine's native endianness; a value of 1 causes DDS to serialize data in the opposite endianness. The receiving side will adjust the data for its endianness so there is no need to match this option between machines. The purpose of this option is to allow the developer to decide which side will make the endian adjustment, if necessary.	0
passive_connect_duration=*msec*	Timeout (milliseconds) for initial passive connection establishment. A value of zero would wait indefinitely (not recommended).	10000 (10 sec)

The passive_connect_duration option is typically set to a non-zero, positive integer. Without a suitable connection timeout, the subscriber endpoint can potentially enter a state of deadlock while waiting for the remote side to initiate a connection. Because there can be multiple transport instances on both the publisher and subscriber side, this option needs to be set to a high enough value to allow the publisher to iterate through the combinations until it succeeds.

In addition to the user-defined configurations, OpenDDS can implicitly define two transport configurations. The first is the default configuration and includes all transport implementations that are linked into the process. If none are found, then only tcp is used. Each of these transport instances uses the default configuration for that transport

implementation. This is the global transport configuration used when the user does not define one.

The second implicit transport configuration is defined whenever an OpenDDS configuration file is used. It is given the same name as the file being read and includes all the transport instances defined in that file, in the alphabetical order of their names. The user can most easily utilize this configuration by specifying the `DCPSGlobalTransportConfiguration=$file` option in the same file. The $file value always binds to the implicit file configuration of the current file.

7.4.5 Transport Instance Options

Transport Instances are specified in the OpenDDS configuration file via sections with the format of [transport/<name>], where <name> is a unique name for that instance within that process. Each Transport Instance must specify the `transport_type` option with a valid transport implementation type. The following sections list the other options that can be specified, starting with those options common to all transport types and following with those specific to each transport type.

When using dynamic libraries, the OpenDDS transport libraries are dynamically loaded whenever an instance of that type is defined in a configuration file. When using custom transport implementations or static linking, the application developer is responsible for ensuring that the transport implementation code is linked with their executables.

7.4.5.1 Configuration Options Common to All Transports

The following table summarizes the transport configuration options that are common to all transports:

Table 7-13 Common Transport Configuration Options

Option	Description	Default
transport_type=*transport*	Type of the transport; the list of available transports can be extended programmatically via the transport framework. tcp, udp, multicast, shmem, and rtps_udp are included with OpenDDS.	none
queue_messages_per_pool=*n*	When backpressure is detected, messages to be sent are queued. When the message queue must grow, it grows by this number.	10

Option	Description	Default
queue_initial_pools=*n*	The initial number of pools for the backpressure queue. The default settings of the two backpressure queue values preallocate space for 50 messages (5 pools of 10 messages).	5
max_packet_size=*n*	The maximum size of a transport packet, including its transport header, sample header, and sample data.	2147481599
max_samples_per_packet=*n*	Maximum number of samples in a transport packet.	10
optimum_packet_size=*n*	Transport packets greater than this size will be sent over the wire even if there are still queued samples to be sent. This value may impact performance depending on your network configuration and application nature.	4096 (4 KiB)
thread_per_connection= *[0|1]*	Enable or disable the thread per connection send strategy. By default, this option is disabled.	0
datalink_release_delay=*msec*	The datalink_release_delay is the delay (in milliseconds) for datalink release after no associations. Increasing this value may reduce the overhead of re-establishment when reader/writer associations are added and removed frequently.	10000 (10 sec)

Enabling the thread_per_connection option will increase performance when writing to multiple data readers on different process as long as the overhead of thread context switching does not outweigh the benefits of parallel writes. This balance of network performance to context switching overhead is best determined by experimenting. If a machine has multiple network cards, it may improve performance by creating a transport for each network card.

7.4.5.2 **TCP/IP Transport Configuration Options**

There are a number of configurable options for the tcp transport. A properly configured transport provides added resilience to underlying stack disturbances. Almost all of the options available to customize the connection and reconnection strategies have reasonable defaults, but ultimately these values should to be chosen based upon a careful study of the quality of the network and the desired QoS in the specific DDS application and target environment.

The local_address option is used by the peer to establish a connection. By default, the TCP transport selects an ephemeral port number on the NIC with the FQDN (fully qualified domain name) resolved. Therefore, you may wish to explicitly set the address if you have

multiple NICs or if you wish to specify the port number. When you configure inter-host communication, the local_address can not be localhost and should be configured with an externally visible address (i.e. 192.168.0.2), or you can leave it unspecified in which case the FQDN and an ephemeral port will be used.

FQDN resolution is dependent upon system configuration. In the absence of a FQDN (e.g. *example.objectcomputing.com*), OpenDDS will use any discovered short names (e.g. example). If that fails, it will use the name resolved from the loopback address (e.g. localhost).

Note *OpenDDS IPv6 support requires that the underlying ACE/TAO components be built with IPv6 support enabled. The local_address needs to be an IPv6 decimal address or a FQDN with port number. The FQDN must be resolvable to an IPv6 address.*

The `tcp` transport exists as an independent library and needs to be linked in order to use it. When using a dynamically-linked build, OpenDDS automatically loads the transport library whenever it is referenced in a configuration file or as the default transport when no other transports are specified.

When the `tcp` library is built statically, your application must link directly against the library. To do this, your application must first include the proper header for service initialization: *<dds/DCPS/transport/tcp/Tcp.h>*.

You can also configure the publisher and subscriber transport implementations programatically, as described in 7.4.3. Configuring subscribers and publishers should be identical, but different addresses/ports should be assigned to each Transport Instance.

The following table summarizes the transport configuration options that are unique to the `tcp` transport:

Table 7-14 TCP/IP Configuration Options

Option	Description	Default
`active_conn_timeout_period=msec`	The time period (milliseconds) for the active connection side to wait for the connection to be established. If not connected within this period then the on_publication_lost() callbacks will be called.	5000 (5 sec)
`conn_retry_attempts=n`	Number of reconnect attempts before giving up and calling the on_publication_lost() and on_subscription_lost() callbacks.	3
`conn_retry_initial_delay=msec`	Initial delay (milliseconds) for reconnect attempt. As soon as a lost connection is detected, a reconnect is attempted. If this reconnect fails, a second attempt is made after this specified delay.	500

OpenDDS Developer's Guide

Option	Description	Default
conn_retry_backoff_multiplier=*n*	The backoff multiplier for reconnection tries. After the initial delay described above, subsequent delays are determined by the product of this multiplier and the previous delay. For example, with a conn_retry_initial_delay of 500 and a conn_retry_backoff_multiplier of 1.5, the second reconnect attempt will be 0.5 seconds after the first retry connect fails; the third attempt will be 0.75 seconds after the second retry connect fails; the fourth attempt will be 1.125 seconds after the third retry connect fails.	2.0
enable_nagle_algorithm=*[0|1]*	Enable or disable the Nagle's algorithm. By default, it is disabled. Enabling the Nagle's algorithm may increase throughput at the expense of increased latency.	0
local_address=*host:port*	Hostname and port of the connection acceptor. The default value is the FQDN and port 0, which means the OS will choose the port. If only the host is specified and the port number is omitted, the ':' is still required on the host specifier.	fqdn:0
max_output_pause_period=*msec*	Maximum period (milliseconds) of not being able to send queued messages. If there are samples queued and no output for longer than this period then the connection will be closed and on_*_lost() callbacks will be called. The default value of zero means that this check is not made.	0
passive_reconnect_duration=*msec*	The time period (milliseconds) for the passive connection side to wait for the connection to be reconnected. If not reconnected within this period then the on_*_lost() callbacks will be called.	2000 (2 sec)
pub_address=*host:port*	Override the address sent to peers with the configured string. This can be used for firewall traversal and other advanced network configurations.	

TCP/IP Reconnection Options

When a TCP/IP connection gets closed OpenDDS attempts to reconnect. The reconnection process is (a successful reconnect ends this sequence):

• Upon detecting a lost connection immediately attempt reconnect.

O p e n D D S   D e v e l o p e r ' s   G u i d e

- If that fails, then wait `conn_retry_initial_delay` milliseconds and attempt reconnect.

- While we have not tried more than `conn_retry_attempts`, wait (previous wait time * `conn_retry_backoff_multiplier`) milliseconds and attempt to reconnect.

7.4.5.3 UDP/IP Transport Configuration Options

The `udp` transport is a bare bones transport that supports best-effort delivery only. Like `tcp`, `local_address`, it supports both IPv4 and IPv6 addresses.

`udp` exists as an independent library and therefore needs to be linked and configured like other transport libraries. When using a dynamic library build, OpenDDS automatically loads the library when it is referenced in a configuration file. When the `udp` library is built statically, your application must link directly against the library. Additionally, your application must also include the proper header for service initialization: *`<dds/DCPS/transport/udp/Udp.h>`*.

The following table summarizes the transport configuration options that are unique to the `udp` transport:

Table 7-15 UDP/IP Configuration Options

Option	Description	Default
`local_address=host:port`	Hostname and port of the listening socket. Defaults to a value picked by the underlying OS. The port can be omitted, in which case the value should end in ":".	`fqdn:0`
`send_buffer_size=n`	Total send buffer size in bytes for UDP payload.	Platform value of `ACE_DEFAULT_MAX_SOCKET_BUFSIZ`
`rcv_buffer_size=n`	Total receive buffer size in bytes for UDP payload.	Platform value of `ACE_DEFAULT_MAX_SOCKET_BUFSIZ`

7.4.5.4 IP Multicast Transport Configuration Options

The `multicast` transport provides unified support for best-effort and reliable delivery based on a transport configuration parameter.

Best-effort delivery imposes the least amount of overhead as data is exchanged between peers, however it does not provide any guarantee of delivery. Data may be lost due to unresponsive or unreachable peers or received in duplicate.

Reliable delivery provides for guaranteed delivery of data to associated peers with no duplication at the cost of additional processing and bandwidth. Reliable delivery is achieved

through two primary mechanisms: 2-way peer handshaking and negative acknowledgment of missing data. Each of these mechanisms are bounded to ensure deterministic behavior and is configurable to ensure the broadest applicability possible for user environments.

`multicast` supports a number of configuration options:

The `default_to_ipv6` and `port_offset` options affect how default multicast group addresses are selected. If `default_to_ipv6` is set to "1" (enabled), then the default IPv6 address will be used (`[FF01::80]`). The `port_offset` option determines the default port used when the group address is not set and defaults to `49152`.

The `group_address` option may be used to manually define a multicast group to join to exchange data. Both IPv4 and IPv6 addresses are supported. As with `tcp`, OpenDDS IPv6 support requires that the underlying ACE/TAO components be built with IPv6 support enabled.

On hosts with multiple network interfaces, it may be necessary to specify that the multicast group should be joined on a specific interface. The option `local_address` can be set to the IP address of the local interface that will receive multicast traffic.

If reliable delivery is desired, the `reliable` option may be specified (the default). The remainder of configuration options affect the reliability mechanisms used by the `multicast` transport:

The `syn_backoff`, `syn_interval`, and `syn_timeout` configuration options affect the handshaking mechanism. `syn_backoff` is the exponential base used when calculating the backoff delay between retries. The `syn_interval` option defines the minimum number of milliseconds to wait before retrying a handshake. The `syn_timeout` defines the maximum number of milliseconds to wait before giving up on the handshake.

Given the values of `syn_backoff` and `syn_interval`, it is possible to calculate the delays between handshake attempts (bounded by `syn_timeout`):

```
delay = syn_interval * syn_backoff ^ number_of_retries
```

For example, if the default configuration options are assumed, the delays between handshake attempts would be: 0, 250, 1000, 2000, 4000, and 8000 milliseconds respectively.

The `nak_depth`, `nak_interval`, and `nak_timeout` configuration options affect the Negative Acknowledgment mechanism. `nak_depth` determines the maximum number of datagrams retained by the transport to service incoming repair requests. The `nak_interval` configuration option defines the minimum number of milliseconds to wait between repair requests. This interval is randomized to prevent potential collisions between similarly

O p e n D D S   D e v e l o p e r ' s   G u i d e

associated peers. The *maximum* delay between repair requests is bounded to double the minimum value.

The `nak_timeout` configuration option defines the maximum amount of time to wait on a repair request before giving up.

The `nak_delay_intervals` configuration option defines the number of intervals between naks after the initial nak.

The `nak_max` configuration option limits the maximum number of times a missing sample will be nak'ed. Use this option so that naks will be not be sent repeatedly for unrecoverable packets before `nak_timeout`.

Currently, there are a couple of requirements above and beyond those already mandated by the ETF when using this transport:

• *At most*, one DDS domain may be used per multicast group;

• A given participant may only have a single `multicast` transport attached per multicast group; if you wish to send and receive samples on the same multicast group in the same process, independent participants must be used.

`multicast` exists as an independent library and therefore needs to be linked and configured like other transport libraries. When using a dynamic library build, OpenDDS automatically loads the library when it is referenced in a configuration file. When the `multicast` library is built statically, your application must link directly against the library. Additionally, your application must also include the proper header for service initialization: *<dds/DCPS/transport/multicast/Multicast.h>*.

The following table summarizes the transport configuration options that are unique to the `multicast` transport:

Table 7-16 IP Multicast Configuration Options

Option	Description	Default
`default_to_ipv6=`*[0|1]*	Enables IPv6 default group address selection. By default, this option is disabled.	0
`group_address=`*host:port*	The multicast group to join to send/receive data.	224.0.0.128:<port>, [FF01::80]:<port>
`local_address=`*address*	If non-empty, address of a local network interface which is used to join the multicast group.	
`nak_delay_intervals=`*n*	The number of intervals between naks after the initial nak.	4

O p e n D D S   D e v e l o p e r ' s   G u i d e

Option	Description	Default
nak_depth=*n*	The number of datagrams to retain in order to service repair requests (reliable only).	32
nak_interval=*msec*	The minimum number of milliseconds to wait between repair requests (reliable only).	500
nak_max=*n*	The maximum number of times a missing sample will be nak'ed.	3
nak_timeout=*msec*	The maximum number of milliseconds to wait before giving up on a repair response (reliable only).	30000 (30 sec)
port_offset=*n*	Used to set the port number when not specifying a group address. When a group address is specified, the port number within it is used. If no group address is specified, the port offset is used as a port number. This value should not be set less than 49152.	49152
rcv_buffer_size=*n*	The size of the socket receive buffer in bytes. A value of zero indicates that the system default value is used.	0
reliable=*[0|1]*	Enables reliable communication.	1
syn_backoff=*n*	The exponential base used during handshake retries; smaller values yield shorter delays between attempts.	2.0
syn_interval=*msec*	The minimum number of milliseconds to wait between handshake attempts during association.	250
syn_timeout=*msec*	The maximum number of milliseconds to wait before giving up on a handshake response during association. The default is 30 seconds.	30000 (30 sec)
ttl=*n*	The value of the time-to-live (ttl) field of any datagrams sent. The default value of one means that all data is restricted to the local network.	1
async_send=*[0|1]*	Send datagrams using Async I/O (on platforms that support it efficiently).	

7.4.5.5 RTPS_UDP Transport Configuration Options

The OpenDDS implementation of the OMG DDSI-RTPS (`formal/2014-09-01`) specification includes the transport protocols necessary to fulfill the specification requirements and those needed to be interoperable with other DDS implementations. The `rtps_udp` transport is one of the pluggable transports available to a developer and is necessary for interoperable

communication between implementations. This section will discuss the options available to the developer for configuring OpenDDS to use this transport.

To provide an RTPS variant of the single configuration example from Section , the configuration file below simply modifies the `transport_type` property to the value `rtps_udp`. All other items remain the same.

```
[common]
DCPSGlobalTransportConfig=myconfig

[config/myconfig]
transports=myrtps

[transport/myrtps]
transport_type=rtps_udp
local_address=myhost
```

To extend our examples to a mixed transport configuration as shown in Section , below shows the use of an `rtps_udp` transport mixed with a `tcp` transport. The interesting pattern that this allows for is a deployed OpenDDS application that can be, for example, communicating using `tcp` with other OpenDDS participants while communicating in an interoperability configuration with a non-OpenDDS participant using `rtps_udp`.

```
[common]
DCPSGlobalTransportConfig=myconfig

[config/myconfig]
transports=mytcp,myrtps

[transport/myrtps]
transport_type=rtps_udp

[transport/mytcp]
transport_type=tcp
```

Some implementation notes related to using the `rtps_udp` transport protocol are as follows:

1) `WRITER_DATA_LIFECYCLE` (8.7.2.2.7) notes that the same Data sub-message should dispose and unregister an instance. OpenDDS may use two Data sub-messages.

2) RTPS transport instances can not be shared by different Domain Participants.

3) Transport auto-selection (negotiation) is partially supported with RTPS such that the `rtps_udp` transport goes through a handshaking phase only in reliable mode.

Table 7-17 RTPS_UDP Configuration Options

Option	Description	Default
use_multicast=*[0|1]*	The `rtps_udp` transport can use Unicast or Multicast. When set to 0 (false) the transport uses Unicast, otherwise a value of 1 (true) will use Multicast.	1
multicast_group_address =*network_address*	When the transport is set to multicast, this is the multicast network address that should be used. If no port is specified for the network address, port 7401 will be used.	239.255.0.2:7401
multicast_interface=*iface*	Specifies the network interface to be used by this transport instance. This uses a platform-specific format that identifies the network interface. On Linux systems this would be something like `eth0`. If this value is not configured, the Common Configuration value `DCPSDefaultAddress` is used to set the multicast interface.	The system default interface is used
local_address=*addr:[port]*	Bind the socket to the given address and port. Port can be omitted but the trailing ":" is required.	System default
ipv6_local_address= *addr:[port]*	Bind the socket to the given address and port. Port can be omitted but the trailing ":" is required.	System default
advertised_address= *addr:[port]*	Sets the address advertised by the transport. Typically used when the participant is behind a firewall or NAT. Port can be omitted but the trailing ":" is required.	
ipv6_advertised_address= *addr:[port]*	Sets the address advertised by the transport. Typically used when the participant is behind a firewall or NAT. Port can be omitted but the trailing ":" is required.	
send_delay=*msec*	Time in milliseconds for an RTPS Writer to wait before sending data.	10
nak_depth=*n*	The number of data samples to retain in order to service repair requests (reliable only).	32
nak_response_delay=*msec*	Protocol tuning parameter that allows the RTPS Writer to delay the response (expressed in milliseconds) to a request for data from a negative acknowledgment. (see table 8.47 in the OMG DDSI-RTPS specification)	200

Option	Description	Default
heartbeat_period=*msec*	Protocol tuning parameter that specifies in milliseconds how often an RTPS Writer announces the availability of data. (see table 8.47 in the OMG DDSI-RTPS specification)	1000 (1 sec)
heartbeat_response_delay=*msec*	Protocol tuning parameter in milliseconds that allows the RTPS Reader to delay the sending of a positive or negative acknowledgment. This parameter is used to reduce the occurrences of network storms. (see Section 8.4.12.2 of the OMG DDSI-RTPS specification (formal/2014-09-01))	500
ResponsiveMode=*[0|1]*	Causes reliable writers and readers to send additional messages which may reduce latency.	0
max_message_size=*n*	The maximum message size. The default is the maximum UDP message size.	65466
quick_reply_ratio=*r*	Tuning parameter that governs how fast new announcements occur relative to the heartbeat_period and heartbeat_response_delay.	0.1
ttl=*n*	The value of the time-to-live (ttl) field of any multicast datagrams sent. This value specifies the number of hops the datagram will traverse before being discarded by the network. The default value of 1 means that all data is restricted to the local network subnet.	1
DataRtpsRelayAddress=*host:port*	Specifies the address of the RtpsRelay for RTPS messages. See section 15.2.	
RtpsRelayOnly=*[0|1]*	Only send RTPS message to the RtpsRelay (for debugging). See section 15.2.	0
UseRtpsRelay=*[0|1]*	Send messages to the RtpsRelay. Messages will only be sent if DataRtpsRelayAddress is set. See section 15.2.	0
DataStunServerAddress=*host:port*	Specifies the address of the STUN server to use for RTPS when using ICE. See section 15.3.	
UseIce=*[0|1]*	Enable or disable ICE for this transport instance. See section 15.3.	0

7.4.5.6   Shared-Memory Transport Configuration Options

The following table summarizes the transport configuration options that are unique to the shmem transport. This transport type is supported Unix-like platforms with POSIX/XSI shared memory and on Windows platforms. The shared memory transport type can only provide

communication between transport instances on the same host. As part of transport negotiation (see 7.4.2.2), if there are multiple transport instances available for communication between hosts, the shared memory transport instances will be skipped so that other types can be used.

Table 7-18 Shared-Memory Transport Configuration Options

Option	Description	Default
`pool_size=bytes`	The size of the single shared-memory pool allocated.	16777216 (16 MiB)
`datalink_control_size=bytes`	The size of the control area allocated for each data link. This allocation comes out of the shared-memory pool defined by pool_size.	4096 (4 KiB)

7.5 Discovery and Transport Configuration Templates

OpenDDS supports dynamic configuration of RTPS discovery and transports by means of configuration templates in OpenDDS configuration files. This feature adds 3 optional file sections, `[DomainRange]`, `[transport_template]`, and `[Customization]`, as well as a new transport property, `instantiation_rule`, which specifies when transport instances are created. Configuration templates are processed at application startup; however, creation of domain, discovery, and transport objects is deferred until a participant is created in a corresponding domain.

A traditional OpenDDS application with 5 participants in different domains will have a config.ini file with 5 separate but nearly identical `[domain]` sections. The same functionality can be accomplished with a single `[DomainRange/1-5]` section using templates.

`[Customization]` sections can be used in `[rtps_discovery]` template sections to add the domain ID to the multicast override address. This creates a unique address for each domain. `[Customization]` sections can also be used with `[transport_template]` sections to modify the transport multicast group addresses and address ports by domain ID. The `[transport_template]` rule, `instantiation_rule=per_participant`, configures OpenDDS to create a separate transport instance for each domain participant. This allows applications to have multiple participants per domain when using RTPS.

7.5.1 Configuring Discovery For A Set Of Similar Domains

Domain range sections are similar to domain sections and use the same configuration properties with 3 notable differences.

- Domain ranges must have a beginning and end domain, such as `[DomainRange/1-5]`.

- Domain ranges use the `DiscoveryTemplate` property rather than the `DiscoveryConfig` property to denote the corresponding `[rtps_discovery]` section.

- Domain ranges can have an optional `Customization` property that maps to a named `[Customization]` section

See section 7.5.4 for a `[DomainRange]` example.

7.5.2 Configuring A Set Of Similar Transports

Transport template sections are specified as `[transport_template/name]`. They are similar to `[transport]` sections and use the same configuration properties as well as an optional `Customization` property that maps to a named `[Customization]` section. To associate a transport template with a domain range in a configuration file, set the `DCPSGlobalTransportConfig` property in the `[common]` section to the name of the `[config]` whose transports property is the name of the transport template. For example, for a global config setting

```
[common]
DCPSGlobalTransportConfig=primary_config
```

a corresponding config could be

```
[config/primary_config]
transports=auto_config_rtps
```

and the partial transport template would be

```
[transport_template/auto_config_rtps]
transport_type=rtps_udp
```

Domain participants that belong to a domain that is configured by a template can bind to non-global transport configurations using the `bind_config` function. See section 7.4.2.3 for a discussion of `bind_config`.

If the [transport_template] sets the property instantiation_rule=per_participant, a separate transport instance will be created for each participant in the domain.
See section 7.5.4 for a [transport_template] example.

7.5.3   Adding Customizations

[Customization] sections can modify the InteropMulticastOverride property in [rtps_discovery] sections and the multicast_group_address property in [transport_template] sections.

- InteropMulticastOverride=AddDomainId adds the domain id to the last octet of the InteropMulticastOverride address
- multicast_group_address=add_domain_id_to_ip_addr adds the domain ID to the last octet of the multicast group address
- multicast_group_address=add_domain_id_to_port uses the domain ID in the port calculation for the multicast group address

7.5.4   Example Config.ini

The following is an example configuration file for domains 2 through 10. It includes customizations to add the domain ID to the discovery InteropMulticastOverride address and customizations to add the domain ID to the transport's multicast group IP address and port.

```
[common]
DCPSGlobalTransportConfig=the_config


[DomainRange/2-10]
DiscoveryTemplate=DiscoveryConfigTemplate


[Customization/discovery_customization]
InteropMulticastOverride=AddDomainId


[Customization/transport_customization]
multicast_group_address=add_domain_id_to_ip_addr,add_domain_id_to_port


[rtps_discovery/DiscoveryConfigTemplate]
InteropMulticastOverride=239.255.4.0
Customization=discovery_customization
SedpMulticast=1
```

```
[config/the_config]
transports=auto_config_rtps


[transport_template/auto_config_rtps]
transport_type=rtps_udp
instantiation_rule=per_participant
Customization=transport_customization
multicast_group_address=239.255.2.0
```

7.6 Logging

By default, the OpenDDS framework will only log when there is a serious error that is not indicated by a return code. An OpenDDS user may increase the amount of logging via controls at the DCPS, Transport, or Security layers.

The default destination of these log messages is the process's standard error stream. See Table 7-2 Common Configuration Options for options controlling the destination and formatting of log messages.

7.6.1 DCPS Layer Logging

Logging in the DCPS layer of OpenDDS is controlled by the `DCPSDebugLevel` configuration option and command-line option. It can also be set in application code using:

```
OpenDDS::DCPS::set_DCPS_debug_level(level)
```

The *level* defaults to a value of 0 and has values of 0 to 10 as defined below:

- 0 - logs that indicate serious errors that are not indicated by return codes (almost none).
- 1 - logs that should happen once per process or are warnings
- 2 - logs that should happen once per DDS entity
- 4 - logs that are related to administrative interfaces
- 6 - logs that should happen every Nth sample write/read
- 8 - logs that should happen once per sample write/read
- 10 - logs that may happen more than once per sample write/read

7.6.2 Transport Layer Logging

OpenDDS transport layer logging is controlled via the `DCPSTransportDebugLevel` configuration option. For example, to add transport layer logging to any OpenDDS application, add the following option to the command line:

```
-DCPSTransportDebugLevel level
```

The transport layer logging level can also be configured by setting the variable:

```
OpenDDS::DCPS::Transport_debug_level = level;
```

Valid transport logging levels range from 0 to 5 with increasing verbosity of output.

Note *Transport logging level 6 is available to generate system trace logs. Using this level is not recommended as the amount of data generated can be overwhelming and is mostly of interest only to OpenDDS developers. Setting the logging level to 6 requires defining the* `DDS_BLD_DEBUG_LEVEL` *macro to* 6 *and rebuilding OpenDDS.*

7.6.3 Security Logging

When OpenDDS is compiled with security enabled, logging for security can be enabled using `DCPSecurityDebug` (See Table 7-2 Common Configuration Options). Unlike the other logging controls, security logging is divided into categories, although `DCPSSecurityDebugLevel` is also provided, which controls the categories in a similar manner and using the same scale as `DCPSDebugLevel`.

Table 7-19 Security Debug Logging Categories

Option	Debug Level	Description
N/A	0	The default. Security related messages are not logged.
access_warn	1	Logs warnings from permission and governance file parsing.
auth_warn	3	Log warnings from the authentication and handshake that happen when two secure participants discover each other.
encdec_error	3	Logs errors from the encryption and decryption of RTPS messages.
bookkeeping	4	Logs generation of crypto handles and keys for local DDS entities and tracking crypto handles and keys for remote DDS entities.
auth_debug	4	Logs debug information from the authentication and handshake that happen when two secure participants discover each other.
encdec_warn	4	Logs warnings from the encryption and decryption of RTPS messages.
encdec_debug	8	Logs debug information from the encryption and decryption of RTPS messages.
showkeys	9	Logs the whole key when generating it, receiving it, and using it.
chlookup	10	Very verbosely prints the steps being taken when looking up a crypto handle for decrypting. This is most useful to see what keys a participant has.
all	10	Enable all the security related logging.

Categories are passed to DCPSecurityDebug using a comma limited list:

-DCPSSecurityDebug=access_warn,showkeys

Unknown categories will cause error messages, but will not cause the DomainParticipantFactory to fail to initialize.

Like the other debug levels, security logging can also be programmatically configured. All the following are equivalent:

```
OpenDDS::DCPS::security_debug.access_warn = true;
OpenDDS::DCPS::security_debug.set_debug_level(1);
OpenDDS::DCPS::security_debug.parse_flags(ACE_TEXT("access_warn"));
```

CHAPTER 8

opendds_idl

opendds_idl is one of the code generators used in the process of building OpenDDS and OpenDDS applications. It can be used in a number of different ways to customize how source code is generated from IDL files. See section 2.1.2 for an overview of the default usage pattern.

The OpenDDS IDL compiler is invoked using the opendds_idl executable, located in $DDS_ROOT/bin/ (on the PATH). It parses a single IDL file and generates the serialization and key support code that OpenDDS requires to marshal and demarshal the types described in the IDL file, as well as the type support code for the data readers and writers. For each IDL file processed, such as *xyz.idl*, it generates three files: *xyzTypeSupport.idl*, *xyzTypeSupportImpl.h*, and *xyzTypeSupportImpl.cpp*. In the typical usage, opendds_idl is passed a number of options and the IDL file name as a parameter. For example,

opendds_idl [options...] Foo.idl

The remaining sections of this chapter describe all of the command-line options and the ways that opendds_idl can be used to generate alternate mappings.

8.1   **opendds_idl Command Line Options**

The following table summarizes the options supported by opendds_idl.

Table 8-1 opendds_idl Command Line Options

Option	Description	Default
-v	Enables verbose execution	Quiet execution
-h	Prints a help (usage) message and exits	N/A
-V	Prints version numbers of both TAO and OpenDDS	N/A
--idl-version VERSION	Set the version of IDL to use.	4
--list-idl-versions	List the versions of IDL at least partially supported.	N/A
--syntax-only	Just check syntax of input files, exiting after parsing.	Goes on to generate code
-Wb,export_macro=macro	Export macro used for generating C++ implementation code. --export is equivalent to -Wb,export_macro	No export macro used
-Wb,export_include=file	Additional header to #include in generated code — this header #defines the export macro	No additional include
-Wb,pch_include=file	Pre-compiled header file to include in generated C++ files	No pre-compiled header included
-Dname[=value]	Define a preprocessor macro	N/A
-Idir	Add dir to the preprocessor include path	N/A
-o outputdir	Output directory where opendds_idl should place the generated files.	The current directory
-Wb,java	Enable OpenDDS Java Bindings for generated TypeSupport implementation classes	No Java support
-Gitl	Generates "Intermediate Type Language" descriptions of datatypes. These files are used by the Wireshark dissector or other external applications.	Not generated
-GfaceTS	Generates FACE (Future Airborne Capability Environment) Transport Services API	Not generated
-Gv8	Generate type support for converting data samples to/from V8 JavaScript objects -Wb,v8 is an alternative form of this option	Not generated
-Grapidjson	Generate type support for converting data samples to/from RapidJSON objects	Not generated
-Lface	Generates IDL-to-C++ mapping for FACE	Not generated
-Lspcpp	Generates IDL-to-C++ mapping for Safety Profile	Not generated
-Lc++11	Generates IDL-to-C++11 mapping	Not generated

Option	Description	Default
`-Wb,tao_include_prefix=s`	Prefix the string *s* to #include directives meant to include headers generated by `tao_idl`	N/A
`-St`	Suppress generation of IDL TypeCodes when one of the `-L` options are present.	IDL TypeCodes generated
`--unknown-annotations VAL`	For IDL version 4, control the reaction to unknown annotations. The options are: • `warn-once`, the default, warn once per annotation with the same name. • `warn-all`, warn for every use of an unknown annotation. • `error`, similar to `warn-all`, but causes the compiler to exit with an error status when finished. • `ignore`, ignore all unknown annotations.	`warn-once`
`--no-dcps-data-type-warnings`	Don't warn about #pragma DCPS_DATA_TYPE	`Warnings are issued, use annotations to silence them`
`--[no-]default-nested`	Un-annotated types/modules are treated as nested. See section 2.1.1.4 for details.	`Types are nested by default.`
`--default-extensibility VAL`	Set the default XTypes Extensibility – see section 16.6.3	`appendable`
`--default-autoid VAL`	Set the default XTypes auto member-id assignment strategy: sequential or hash – see section 16.6.5.2	`sequential`
`--default-try-construct VAL`	Set the default XTypes try-construct strategy: discard, use-default, or trim – see section 16.6.4	`Discard`

The code generation options allow the application developer to use the generated code in a wide variety of environments. Since IDL may contain preprocessing directives (`#include`, `#define`, etc.), the C++ preprocessor is invoked by `opendds_idl`. The `-I` and `-D` options allow customization of the preprocessing step. The `-Wb,export_macro` option lets you add an export macro to your class definitions. This is required if the generated code is going to reside in a shared library and the compiler (such as Visual C++ or GCC) uses the export macro (`dllexport` on Visual C++ / overriding hidden visibility on GCC). The `-Wb,pch_include` option is required if the generated implementation code is to be used in a project that uses precompiled headers.

8.2   Using the IDL-to-C++11 Mapping

The IDL-to-C++11 Mapping is a separate specification from the OMG. Like the "classic" IDL-to-C++ Mapping, IDL-to-C++11 describes how IDL constructs (structs, sequences, unions, etc.) should appear in C++. Since the IDL-to-C++11 Mapping assumes a C++11 (or higher) compiler and standard library, the code generated is easier to use and looks more natural to C++ developers who are not familiar with the classic mapping. For example, IDL strings, arrays, and sequences map to their equivalents in the `std` namespace: `string`, `array`, and `vector`. All of the details of the mapping are spelled out in the specification document (available at https://www.omg.org/spec/CPP11), however the easiest way to get started with the mapping is to generate code from IDL and examine the generated header file.

In `opendds_idl`'s default mode (as described in section 2.1.2), responsibility for generating the language mapping is delegated to `tao_idl` (using the IDL-to-C++ classic mapping). In this case, `opendds_idl` is only responsible for generating the OpenDDS-specific additions such as `TypeSupport.idl` and the marshal/demarshal functions.

Contrast this with using `opendds_idl` for IDL-to-C++11. In this case, `opendds_idl` takes over responsibility for generating the language mapping. This is indicated by the `-Lc++11` command-line option.

Starting with a user-written file `Foo.idl`, running "`opendds_idl -Lc++11 <other options> Foo.idl`" generates these output files:

- `FooTypeSupport.idl`
  - IDL local interfaces for *TypeSupport, *DataWriter, *DataReader
- `FooC.h`
  - IDL-to-C++11 language mapping
- `FooTypeSupportImpl.h` and `.cpp`
  - Additional source code needed for OpenDDS

FooTypeSupport.idl is the same as it was when using the classic mapping. After it's generated by `opendds_idl`, it needs to be processed by `tao_idl` to generate `FooTypeSupportC.h`, `FooTypeSupportC.inl`, and `FooTypeSupportC.cpp`.

Unlike when using the classic mapping, `Foo.idl` is not processed by `tao_idl`.

`Foo.idl` can contain the following IDL features:

- modules, typedefs, and constants

- basic types

- constructed types: enums, structs and unions

  - Note that setting a union value through a modifier method automatically sets the discriminator. In cases where there are multiple possible values for the discriminator, a 2-argument modifier method is provided. Using this is preferred to using _d().

  - If you chose to use the _d() method of the generated union types, note the following requirement from the specification: "The _d discriminator modifier can only be used to set the discriminant to a value within the same union member." OpenDDS treats this as a precondition (it is not checked within the implementation).

- strings (narrow and wide), sequences, and arrays

  - Bounded strings and sequences are supported, but bounds checks are not currently enforced. Due to this limitation, distinct types are not used for bounded instantiations.

- annotations – see section 2.1.1

- #includes of IDL files that are also used with the IDL-to-C++11 mapping

When using MPC to generate projects, the `opendds_cxx11` base project should be used to inherit the correct settings for code generation. If the generated code will be part of a shared library, use the `-Wb,export_include` option (in addition to `-Wb,export_macro`) so that the generated headers have an `#include` for the export header.

When using CMake to generate projects, see the CMake module documentation included in the OpenDDS repository (`docs/cmake.md`).

CHAPTER 9

The DCPS Information

Repository

9.1 DCPS Information Repository Options

The table below shows the command line options for the `DCPSInfoRepo` server:

Table 9-1 DCPS Information Repository Options

Option	Description	Default
`-o file`	Write the IOR of the `DCPSInfo` object to the specified file	`repo.ior`
`-NOBITS`	Disable the publication of built-in topics	Built-in topics are published
`-a address`	Listening address for built-in topics (when built-in topics are published).	Random port
`-z`	Turn on verbose transport logging	Minimal transport logging.
`-r`	Resurrect from persistent file	1(true)
`-FederationId <id>`	Unique identifier for this repository within any federation. This is supplied as a 32 bit decimal numeric value.	N/A

Option	Description	Default
-FederateWith <ref>	Repository federation reference at which to join a federation. This is supplied as a valid CORBA object reference in string form: stringified IOR, file: or corbaloc: reference string.	N/A
-?	Display the command line usage and exit	N/A

OpenDDS clients often use the IOR file that `DCPSInfoRepo` outputs to locate the service. The `-o` option allows you to place the IOR file into an application-specific directory or file name. This file can subsequently be used by clients with the *file://* IOR prefix.

Applications that do not use built-in topics may want to disable them with `-NOBITS` to reduce the load on the server. If you are publishing the built-in topics, then the `-a` option lets you pick the listen address of the tcp transport that is used for these topics.

Using the `-z` option causes the invocation of many transport-level debug messages. This option is only effective when the DCPS library is built with the `DCPS_TRANS_VERBOSE_DEBUG` environment variable defined.

The `-FederationId` and `-FederateWith` options are used to control the federation of multiple `DCPSInfoRepo` servers into a single logical repository. See 9.2 for descriptions of the federation capabilities and how to use these options.

File persistence is implemented as an ACE Service object and is controlled via service config directives. Currently available configuration options are:

Table 9-2 InfoRepo persistence directives

Options	Description	Defaults
-file	Name of the persistent file	*InforepoPersist*
-reset	Wipe out old persistent data.	0 (false)

The following directive:

```
static PersistenceUpdater_Static_Service "-file info.pr -reset 1"
```

will persist `DCPSInfoRepo` updates to local file *info.pr*. If a file by that name already exists, its contents will be erased. Used with the command-line option `-r`, the `DCPSInfoRepo` can be reincarnated to a prior state. When using persistence, start the `DCPSInfoRepo` process using a TCP fixed port number with the following command line option. This allows existing clients to reconnect to a restarted InfoRepo.

```
-ORBListenEndpoints iiop://:<port>
```

O p e n D D S   D e v e l o p e r ' s   G u i d e

9.2 Repository Federation

Note *Repository federation should be considered an experimental feature.*

Repository Federation allows multiple DCPS Information Repository servers to collaborate with one another into a single federated service. This allows applications obtaining service metadata and events from one repository to obtain them from another if the original repository is no longer available.

While the motivation to create this feature was the ability to provide a measure of fault tolerance to the DDS service metadata, other use cases can benefit from this feature as well. This includes the ability of initially separate systems to become federated and gain the ability to pass data between applications that were not originally reachable. An example of this would include two platforms which have independently established internal DDS services passing data between applications; at some point during operation the systems become reachable to each other and federating repositories allows data to pass between applications on the different platforms.

The current federation capabilities in OpenDDS provide only the ability to statically specify a federation of repositories at startup of applications and repositories. A mechanism to dynamically discover and join a federation is planned for a future OpenDDS release.

OpenDDS automatically detects the loss of a repository by using the `LIVELINESS` Quality of Service policy on a Built-in Topic. When a federation is used, the `LIVELINESS` QoS policy is modified to a non-infinite value. When `LIVELINESS` is lost for a Built-in Topic an application will initiate a failover sequence causing it to associate with a different repository server. Because the federation implementation currently uses a Built-in Topic `ParticipantDataDataReaderListener` entity, applications should not install their own listeners for this topic. Doing so would affect the federation implementation's capability to detect repository failures.

The federation implementation distributes repository data within the federation using a reserved DDS domain. The default domain used for federation is defined by the constant `Federator::DEFAULT_FEDERATIONDOMAIN`.

Currently only static specification of federation topology is available. This means that each DCPS Information Repository, as well as each application using a federated DDS service, needs to include federation configuration as part of its configuration data. This is done by specifying each available repository within the federation to each participating process and assigning each repository to a different key value in the configuration files as described in Section 7.3.2.1.

Each application and repository must include the same set of repositories in its configuration information. Failover sequencing will attempt to reach the next repository in numeric sequence (wrapping from the last to the first) of the repository key values. This sequence is unique to each application configured, and should be different to avoid overloading any individual repository.

Once the topology information has been specified, then repositories will need to be started with two additional command line arguments. These are shown in Table 9-1. One, `-FederationId <value>`, specifies the unique identifier for a repository within the federation. This is a 32 bit numeric value and needs to be unique for all possible federation topologies.

The second command line argument required is `-FederateWith <ref>`. This causes the repository to join a federation at the <ref> object reference after initialization and before accepting connections from applications.

Only repositories which are started with a federation identification number may participate in a federation. The first repository started should not be given a `-FederateWith` command line directive. All others are required to have this directive in order to establish the initial federation. There is a command line tool (`federation`) supplied that can be used to establish federation associations if this is not done at startup. See Section 9.2.1 for a description. It is possible, with the current static-only implementation, that the failure of a repository before a federation topology is entirely established could result in a partially unusable service. Due to this current limitation, it is highly recommended to always establish the federation topology of repositories prior to starting the applications.

9.2.1 Federation Management

A new command line tool has been provided to allow some minimal run-time management of repository federation. This tool allows repositories started without the `-FederateWith` option to be commanded to participate in a federation. Since the operation of the federated repositories and failover sequencing depends on the presence of connected topology, it is recommended that this tool be used before starting applications that will be using the federated set of repositories.

The command is named `repoctl` and is located in the `$DDS_ROOT/bin/` directory. It has a command format syntax of:

```
repoctl <cmd> <arguments>
```

Where each individual command has its own format as shown in Table 9-3. Some options contain endpoint information. This information consists of an optional host specification,

separated from a required port specification by a colon. This endpoint information is used to create a CORBA object reference using the corbaloc: syntax in order to locate the 'Federator' object of the repository server.

Table 9-3 repoctl Repository Management Command

Command	Syntax	Description
join	repoctl join <target> <peer> [<federation domain>]	Calls the <peer> to join <target> to the federation. <federation domain> is passed if present, or the default Federation Domain value is passed.
leave	repoctl leave <target>	Causes the <target> to gracefully leave the federation, removing all managed associations between applications using <target> as a repository with applications that are not using <target> as a repository.
shutdown	repoctl shutdown <target>	Causes the <target> to shutdown without removing any managed associations. This is the same effect as a repository which has crashed during operation.
kill	repoctl kill <target>	Kills the <target> repository regardless of its federation status.
help	repoctl help	Prints a usage message and quits.

A join command specifies two repository servers (by endpoint) and asks the second to join the first in a federation:

```
repoctl join 2112 otherhost:1812
```

This generates a CORBA object reference of *corbaloc::otherhost:1812/Federator* that the federator connects to and invokes a join operation. The join operation invocation passes the default Federation Domain value (because we did not specify one) and the location of the joining repository which is obtained by resolving the object reference *corbaloc::localhost:2112/Federator*.

A full description of the command arguments are shown in Table 9-4.

Table 9-4 Federation Management Command Arguments

Option	Description
<target>	This is endpoint information that can be used to locate the Federator::Manager CORBA interface of a repository which is used to manage federation behavior. This is used to command leave and shutdown federation operations and to identify the joining repository for the join command.

O p e n D D S   D e v e l o p e r ' s   G u i d e

Option	Description
`<peer>`	This is endpoint information that can be used to locate the `Federator::Manager` CORBA interface of a repository which is used to manage federation behavior. This is used to command join federation operations.
`<federation domain>`	This is the domain specification used by federation participants to distribute service metadata amongst the federated repositories. This only needs to be specified if more than one federation exists among the same set of repositories, which is currently not supported. The default domain is sufficient for single federations.

9.2.2 Federation Example

In order to illustrate the setup and use of a federation, this section walks through a simple example that establishes a federation and a working service that uses it.

This example is based on a two repository federation, with the simple Message publisher and subscriber from 2.1 configured to use the federated repositories.

9.2.2.1 Configuring the Federation Example

There are two configuration files to create for this example one each for the message publisher and subscriber.

The Message Publisher configuration *pub.ini* for this example is as follows:

```
[common]
DCPSDebugLevel=0

[domain/information]
DomainId=42
DomainRepoKey=1

[repository/primary]
RepositoryKey=1
RepositoryIor=corbaloc::localhost:2112/InfoRepo

[repository/secondary]
RepositoryKey=2
RepositoryIor=file://repo.ior
```

Note that the `DCPSInfo` attribute/value pair has been omitted from the `[common]` section. This has been replaced by the `[domain/user]` section as described in 7.5. The user domain is 42, so that domain is configured to use the primary repository for service metadata and events.

The `[repository/primary]` and `[repository/secondary]` sections define the primary and secondary repositories to use within the federation (of two repositories) for this

application. The `RepositoryKey` attribute is an internal key value used to uniquely identify the repository (and allow the domain to be associated with it, as in the preceding `[domain/information]` section). The `RepositoryIor` attributes contain string values of resolvable object references to reach the specified repository. The primary repository is referenced at port 2112 of the *localhost* and is expected to be available via the TAO `IORTable` with an object name of `/InfoRepo`. The secondary repository is expected to provide an IOR value via a file named *repo.ior* in the local directory.

The subscriber process is configured with the *sub.ini* file as follows:

```
[common]
DCPSDebugLevel=0

[domain/information]
DomainId=42
DomainRepoKey=1

[repository/primary]
RepositoryKey=1
RepositoryIor=file://repo.ior

[repository/secondary]
RepositoryKey=2
RepositoryIor=corbaloc::localhost:2112/InfoRepo
```

Note that this is the same as the *pub.ini* file except the subscriber has specified that the repository located at port 2112 of the *localhost* is the secondary and the repository located by the *repo.ior* file is the primary. This is opposite of the assignment for the publisher. It means that the publisher is started using the repository at port 2112 for metadata and events while the subscriber is started using the repository located by the IOR contained in the file. In each case, if a repository is detected as unavailable the application will attempt to use the other repository if it can be reached.

The repositories do not need any special configuration specifications in order to participate in federation, and so no files are required for them in this example.

9.2.2.2   **Running the Federation Example**

The example is executed by first starting the repositories and federating them, then starting the application publisher and subscriber processes the same way as was done in the example of Section 2.1.7.

Start the first repository as:

```
$DDS/bin/DCPSInfoRepo -ORBSvcConf tcp.conf -o repo.ior -FederationId 1024
```

The `-o repo.ior` option ensures that the repository IOR will be placed into the file as expected by the configuration files. The `-FederationId 1024` option assigns the value 1024

OpenDDS Developer's Guide

to this repository as its unique id within the federation. The `-ORBSvcConf tcp.conf` option is the same as in the previous example.

Start the second repository as:

```
$DDS/bin/DCPSInfoRepo -ORBSvcConf tcp.conf \
  -ORBListenEndpoints iiop://localhost:2112 \
  -FederationId 2048 -FederateWith file://repo.ior
```

Note that this is all intended to be on a single command line. The `-ORBSvcConf` *tcp.conf* option is the same as in the previous example. The `-ORBListenEndpoints` `iiop://`*localhost*`:2112` option ensures that the repository will be listening on the port that the previous configuration files are expecting. The `-FederationId 2048` option assigns the value 2048 as the repositories unique id within the federation. The `-FederateWith` `file://repo.ior` option initiates federation with the repository located at the IOR contained within the named file - which was written by the previously started repository.

Once the repositories have been started and federation has been established (this will be done automatically after the second repository has initialized), the application publisher and subscriber processes can be started and should execute as they did for the previous example in Section 2.1.7.

CHAPTER 10

Java Bindings

10.1   Introduction

OpenDDS provides JNI bindings. Java applications can make use of the complete OpenDDS middleware just like C++ applications.

See the *$DDS_ROOT/java/INSTALL* file for information on getting started, including the prerequisites and dependencies.

Java versions 9 and up use the [Java Platform Module System](#). To use OpenDDS with one of these Java versions, set the MPC feature java_pre_jpms to 0. OpenDDS's configure script will attempt to detect the Java version and set this automatically.

See the *$DDS_ROOT/java/FAQ* file for information on common issues encountered while developing applications with the Java bindings.

10.2   IDL and Code Generation

The OpenDDS Java binding is more than just a library that lives in one or two .jar files. The DDS specification defines the interaction between a DDS application and the DDS middleware. In particular, DDS applications send and receive messages that are strongly-typed and those types are defined by the application developer in IDL.

In order for the application to interact with the middleware in terms of these user-defined types, code must be generated at compile-time based on this IDL. C++, Java, and even some additional IDL code is generated. In most cases, application developers do not need to be concerned with the details of all the generated files. Scripts included with OpenDDS automate this process so that the end result is a native library (*.so* or *.dll*) and a Java library (*.jar* or just a *classes* directory) that together contain all of the generated code.

Below is a description of the generated files and which tools generate them. In this example, *Foo.idl* contains a single struct Bar contained in module Baz (IDL modules are similar to C++ namespaces and Java packages). To the right of each file name is the name of the tool that generates it, followed by some notes on its purpose.

Table 10-1 Generated files descriptions

File	Generation Tool
Foo.idl	Developer-written description of the DDS sample type
Foo{C,S}.{h,inl,cpp}	tao_idl: C++ representation of the IDL
FooTypeSupport.idl	opendds_idl: DDS type-specific interfaces
FooTypeSupport{C,S}.{h,inl,cpp}	tao_idl
Baz/BarSeq{Helper,Holder}.java	idl2jni
Baz/BarData{Reader,Writer}.java*	idl2jni
Baz/BarTypeSupport.java*	idl2jni (except TypeSupportImpl, see below)
FooTypeSupportJC.{h,cpp}	idl2jni: JNI native method implementations
FooTypeSupportImpl.{h,cpp}	opendds_idl: DDS type-specific C++ impl.
Baz/BarTypeSupportImpl.java	opendds_idl: DDS type-specific Java impl.
Baz/Bar.java*	idl2jni: Java representation of IDL struct
FooJC.{h,cpp}	idl2jni: JNI native method implementations

Foo.idl:

```
module Baz {
  @topic
  struct Bar {
    long x;
  };
};
```

10.3　Setting up an OpenDDS Java Project

These instructions assume you have completed the installation steps in the *$DDS_ROOT/java/INSTALL* document, including having the various environment variables defined.

1) Start with an empty directory that will be used for your IDL and the code generated from it. $DDS_ROOT/java/tests/messenger/messenger_idl/ is set up this way.

2) Create an IDL file describing the data structure you will be using with OpenDDS. See *Messenger.idl* for an example. This file will contain at least struct/union annotated with @topic. For the sake of these instructions, we will call the file *Foo.idl*.

3) The C++ generated classes will be packaged in a shared library to be loaded at run-time by the JVM. This requires the packaged classes to be exported for external visibility. ACE provides a utility script for generating the correct export macros. The script usage is shown here:

Unix:

```
$ACE_ROOT/bin/generate_export_file.pl Foo > Foo_Export.h
```

Windows:

```
%ACE_ROOT%\bin\generate_export_file.pl Foo > Foo_Export.h
```

4) Create an MPC file, Foo.mpc, from this template:

```
project: dcps_java {

idlflags      += -Wb,stub_export_include=Foo_Export.h \
                 -Wb,stub_export_macro=Foo_Export
dcps_ts_flags += -Wb,export_macro=Foo_Export
idl2jniflags  += -Wb,stub_export_include=Foo_Export.h \
                 -Wb,stub_export_macro=Foo_Export
dynamicflags  += FOO_BUILD_DLL

specific {
  jarname       = DDS_Foo_types
}

TypeSupport_Files {
  Foo.idl
}
}
```

You can leave out the specific {…} block if you do not need to create a jar file. In this case you can directly use the Java .class files which will be generated under the classes subdirectory of the current directory.

5) Run MPC to generate platform-specific build files.

Unix:

```
$ACE_ROOT/bin/mwc.pl -type gnuace
```

Windows:

```
%ACE_ROOT%\bin\mwc.pl -type [CompilerType]
```

CompilerType can be any supported MPC type (such as "vs2019")

Make sure this is running ActiveState Perl or Strawberry Perl.

6) Compile the generated C++ and Java code

Unix:

```
make (GNU make, so this may be "gmake" on Solaris systems)
```

Windows:

Build the generated *.sln* (Solution) file using your preferred method. This can be either the Visual Studio IDE or one of the command-line tools. If you use the IDE, start it from a command prompt using `devenv` so that it inherits the environment variables. Command-line tools for building include `msbuild` and invoking the IDE (`devenv`) with the appropriate arguments.

When this completes successfully you have a native library and a Java *.jar* file. The native library names are as follows:

Unix:

```
libFoo.so
```

Windows:

```
Foo.dll (Release) or Food.dll (Debug)
```

You can change the locations of these libraries (including the *.jar* file) by adding a line such as the following to the *Foo.mpc* file:

```
libout = $(PROJECT_ROOT)/lib
```

where `PROJECT_ROOT` can be any environment variable defined at build-time.

7) You now have all of the Java and C++ code needed to compile and run a Java OpenDDS application. The generated *.jar* file needs to be added to your `classpath`, along with the `.jar` files that come from OpenDDS (in the lib directory). The generated C++ library needs to be available for loading at run-time:

Unix:

Add the directory containing *libFoo.so* to the `LD_LIBRARY_PATH`.

Windows:

Add the directory containing *Foo.dll* (or *Food.dll*) to the `PATH`. If you are using the debug version (*Food.dll*) you will need to inform the OpenDDS middleware that it should not look for *Foo.dll*. To do this, add `-Dopendds.native.debug=1` to the Java VM arguments.

See the publisher and subscriber directories in $DDS_ROOT/java/tests/messenger/ for examples of publishing and subscribing applications using the OpenDDS Java bindings.

8) If you make subsequent changes to *Foo.idl*, start by re-running MPC (step #5 above). This is needed because certain changes to *Foo.idl* will affect which files are generated and need to be compiled.

10.4 **A Simple Message Publisher**

This section presents a simple OpenDDS Java publishing process. The complete code for this can be found at *$DDS_ROOT/java/tests/messenger/publisher/TestPublisher.java*. Uninteresting segments such as imports and error handling have been omitted here. The code has been broken down and explained in logical subsections.

10.4.1 **Initializing The Participant**

DDS applications are boot-strapped by obtaining an initial reference to the Participant Factory. A call to the static method `TheParticipantFactory.WithArgs()` returns a Factory reference. This also transparently initializes the C++ Participant Factory. We can then create Participants for specific domains.

```
public static void main(String[] args) {

    DomainParticipantFactory dpf =
        TheParticipantFactory.WithArgs(new StringSeqHolder(args));
    if (dpf == null) {
      System.err.println ("Domain Participant Factory not found");
      return;
    }
    final int DOMAIN_ID = 42;
    DomainParticipant dp = dpf.create_participant(DOMAIN_ID,
      PARTICIPANT_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
    if (dp == null) {
      System.err.println ("Domain Participant creation failed");
      return;
    }
```

Object creation failure is indicated by a null return. The third argument to `create_participant()` takes a Participant events listener. If one is not available, a null can be passed instead as done in our example.

10.4.2   Registering The Data Type And Creating A Topic

Next we register our data type with the `DomainParticipant` using the `register_type()` operation. We can specify a type name or pass an empty string. Passing an empty string indicates that the middleware should simply use the identifier generated by the IDL compiler for the type.

```
MessageTypeSupportImpl servant = new MessageTypeSupportImpl();
if (servant.register_type(dp, "") != RETCODE_OK.value) {
  System.err.println ("register_type failed");
  return;
}
```

Next we create a topic using the type support servant's registered name.

```
Topic top = dp.create_topic("Movie Discussion List",
                            servant.get_type_name(),
                            TOPIC_QOS_DEFAULT.get(), null,
                            DEFAULT_STATUS_MASK.value);
```

Now we have a topic named "*Movie Discussion List*" with the registered data type and default QoS policies.

10.4.3   Creating A Publisher

Next, we create a publisher:

```
Publisher pub = dp.create_publisher(
  PUBLISHER_QOS_DEFAULT.get(),
  null,
  DEFAULT_STATUS_MASK.value);
```

10.4.4   Creating A DataWriter And Registering An Instance

With the publisher, we can now create a DataWriter:

```
DataWriter dw = pub.create_datawriter(
  top, DATAWRITER_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
```

The `DataWriter` is for a specific topic. For our example, we use the default `DataWriter` QoS policies and a null `DataWriterListener`.

Next, we narrow the generic `DataWriter` to the type-specific `DataWriter` and register the instance we wish to publish. In our data definition IDL we had specified the subject_id field as the key, so it needs to be populated with the instance id (99 in our example):

OpenDDS Developer's Guide

```
MessageDataWriter mdw = MessageDataWriterHelper.narrow(dw);
Message msg = new Message();
msg.subject_id = 99;
int handle = mdw.register(msg);
```

Our example waits for any peers to be initialized and connected. It then publishes a few messages which are distributed to any subscribers of this topic in the same domain.

```
msg.from = "OpenDDS-Java";
msg.subject = "Review";
msg.text = "Worst. Movie. Ever.";
msg.count = 0;
int ret = mdw.write(msg, handle);
```

10.5 Setting up the Subscriber

Much of the initialization code for a subscriber is identical to the publisher. The subscriber needs to create a participant in the same domain, register an identical data type, and create the same named topic.

```
public static void main(String[] args) {

    DomainParticipantFactory dpf =
        TheParticipantFactory.WithArgs(new StringSeqHolder(args));
    if (dpf == null) {
      System.err.println ("Domain Participant Factory not found");
      return;
    }
    DomainParticipant dp = dpf.create_participant(42,
      PARTICIPANT_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
    if (dp == null) {
      System.err.println("Domain Participant creation failed");
      return;
    }

    MessageTypeSupportImpl servant = new MessageTypeSupportImpl();
    if (servant.register_type(dp, "") != RETCODE_OK.value) {
      System.err.println ("register_type failed");
      return;
    }
    Topic top = dp.create_topic("Movie Discussion List",
                                servant.get_type_name(),
                                TOPIC_QOS_DEFAULT.get(), null,
                                DEFAULT_STATUS_MASK.value);
```

10.5.1 Creating A Subscriber

As with the publisher, we create a subscriber:

```
Subscriber sub = dp.create_subscriber(
    SUBSCRIBER_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
```

10.5.2   Creating A DataReader And Listener

Providing a `DataReaderListener` to the middleware is the simplest way to be notified of the receipt of data and to access the data. We therefore create an instance of a `DataReaderListenerImpl` and pass it as a `DataReader` creation parameter:

```
DataReaderListenerImpl listener = new DataReaderListenerImpl();
 DataReader dr = sub.create_datareader(
    top, DATAREADER_QOS_DEFAULT.get(), listener,
    DEFAULT_STATUS_MASK.value);
```

Any incoming messages will be received by the Listener in the middleware's thread. The application thread is free to perform other tasks at this time.

10.6   The DataReader Listener Implementation

The application defined `DataReaderListenerImpl` needs to implement the specification's `DDS.DataReaderListener` interface. OpenDDS provides an abstract class `DDS._DataReaderListenerLocalBase`. The application's listener class extends this abstract class and implements the abstract methods to add application-specific functionality.

Our example `DataReaderListener` stubs out most of the Listener methods. The only method implemented is the message available callback from the middleware:

```
public class DataReaderListenerImpl extends DDS._DataReaderListenerLocalBase {

    private int num_reads_;

    public synchronized void on_data_available(DDS.DataReader reader) {
        ++num_reads_;
        MessageDataReader mdr = MessageDataReaderHelper.narrow(reader);
        if (mdr == null) {
          System.err.println ("read: narrow failed.");
          return;
        }
```

The Listener callback is passed a reference to a generic `DataReader`. The application narrows it to a type-specific `DataReader`:

```
        MessageHolder mh = new MessageHolder(new Message());
        SampleInfoHolder sih = new SampleInfoHolder(new SampleInfo(0, 0, 0,
            new DDS.Time_t(), 0, 0, 0, 0, 0, 0, 0, false));
        int status  = mdr.take_next_sample(mh, sih);
```

It then creates holder objects for the actual message and associated `SampleInfo` and takes the next sample from the `DataReader`. Once taken, that sample is removed from the `DataReader`'s available sample pool.

```
        if (status == RETCODE_OK.value) {

          System.out.println ("SampleInfo.sample_rank = "+ sih.value.sample_rank);
          System.out.println ("SampleInfo.instance_state = "+
                                 sih.value.instance_state);

          if (sih.value.valid_data) {

            System.out.println("Message: subject     = " + mh.value.subject);
            System.out.println("          subject_id = " + mh.value.subject_id);
            System.out.println("             from       = " + mh.value.from);
            System.out.println("             count      = " + mh.value.count);
            System.out.println("             text       = " + mh.value.text);
            System.out.println("SampleInfo.sample_rank = " +
                                 sih.value.sample_rank);
          }
          else if (sih.value.instance_state ==
                     NOT_ALIVE_DISPOSED_INSTANCE_STATE.value) {
            System.out.println ("instance is disposed");
          }
          else if (sih.value.instance_state ==
                     NOT_ALIVE_NO_WRITERS_INSTANCE_STATE.value) {
            System.out.println ("instance is unregistered");
          }
          else {
            System.out.println ("DataReaderListenerImpl::on_data_available: "+
                                 "received unknown instance state "+
                                 sih.value.instance_state);
          }

        } else if (status == RETCODE_NO_DATA.value) {
          System.err.println ("ERROR: reader received DDS::RETCODE_NO_DATA!");
        } else {
          System.err.println ("ERROR: read Message: Error: "+ status);
        }
      }

}
```

The `SampleInfo` contains meta-information regarding the message such as the message validity, instance state, etc.

10.7   Cleaning up OpenDDS Java Clients

An application should clean up its OpenDDS environment with the following steps:

```
        dp.delete_contained_entities();
```

Cleans up all topics, subscribers and publishers associated with that `Participant`.


```
        dpf.delete_participant(dp);
```

The `DomainParticipantFactory` reclaims any resources associated with the
`DomainParticipant`.

```
        TheServiceParticipant.shutdown();
```

Shuts down the `ServiceParticipant`. This cleans up all OpenDDS associated resources.
Cleaning up these resources is necessary to prevent the `DCPSInfoRepo` from forming
associations between endpoints which no longer exist.

10.8   Configuring the Example

OpenDDS offers a file-based configuration mechanism. The syntax of the configuration file is
similar to a Windows INI file. The properties are divided into named sections corresponding
to common and individual transports configuration.

The Messenger example has common properties for the `DCPSInfoRepo` objects location and
the global transport configuration:

```
[common]
DCPSInfoRepo=file://repo.ior
DCPSGlobalTransportConfig=$file
```

and a transport instance section with a transport type property:

```
[transport/1]
transport_type=tcp
```

The `[transport/1]` section contains configuration information for the transport instance
named "1". It is defined to be of type `tcp`. The global transport configuration setting above
causes this transport instance to be used by all readers and writers in the process.

See Chapter 7 for a complete description of all OpenDDS configuration parameters.

10.9   Running the Example

To run the Messenger Java OpenDDS application, use the following commands:

```
$DDS_ROOT/bin/DCPSInfoRepo -o repo.ior

$JAVA_HOME/bin/java -ea -cp
classes:$DDS_ROOT/lib/i2jrt.jar:$DDS_ROOT/lib/OpenDDS_DCPS.jar:classes TestPublisher -
DCPSConfigFile pub_tcp.ini
```

```
$JAVA_HOME/bin/java -ea -cp
classes:$DDS_ROOT/lib/i2jrt.jar:$DDS_ROOT/lib/OpenDDS_DCPS.jar:classes TestSubscriber -
DCPSConfigFile sub_tcp.ini
```

The -DCPSConfigFile command-line argument passes the location of the OpenDDS
configuration file.

10.10   Java Message Service (JMS) Support

OpenDDS provides partial support for JMS version 1.1
<*http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html*>. Enterprise Java
applications can make use of the complete OpenDDS middleware just like standard Java and
C++ applications.

See the INSTALL file in the $DDS_ROOT/java/jms/ directory for information on getting
started with the OpenDDS JMS support, including the prerequisites and dependencies.

CHAPTER 11

Modeling SDK

The OpenDDS Modeling SDK is a modeling tool that can be used by the application developer to define the required middleware components and data structures as a UML model and then generate the code to implement the model using OpenDDS. The generated code can then be compiled and linked with the application to provide seamless middleware support to the application.

11.1   Overview

11.1.1   Model Capture

UML models defining DCPS elements and policies along with data definitions are captured using the graphical model capture editors included in the Eclipse plug-ins. The elements of the UML models follow the structure of the DDS UML Platform Independent Model (PIM) defined in the DDS specification (OMG: `formal/2015-04-10`).

Opening a new OpenDDS model within the plug-ins begins with a top level main diagram. This diagram includes any package structures to be included in the model along with the local QoS policy definitions, data definitions, and DCPS elements of the model. Zero or more of the policy or data definition elements can be included. Zero or one DCPS elements definition can be included in any given model.

**Figure 11-1 Graphical modeling of
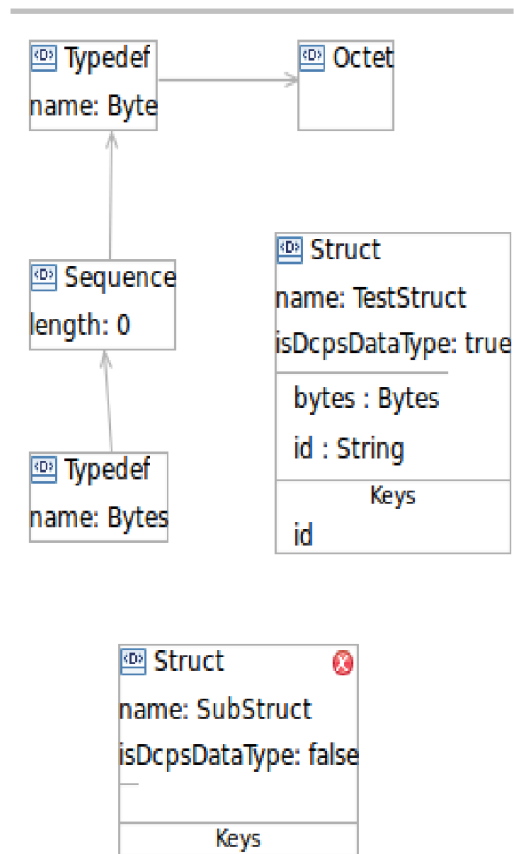the data definitions**

Creating separate models for QoS policies only, data definitions only, or DCPS elements
only is supported. References to other models allows externally defined models to be
included in a model. This allows sharing of data definitions and QoS policies among different
DCPS models as well as including externally defined data in a new set of data definitions.
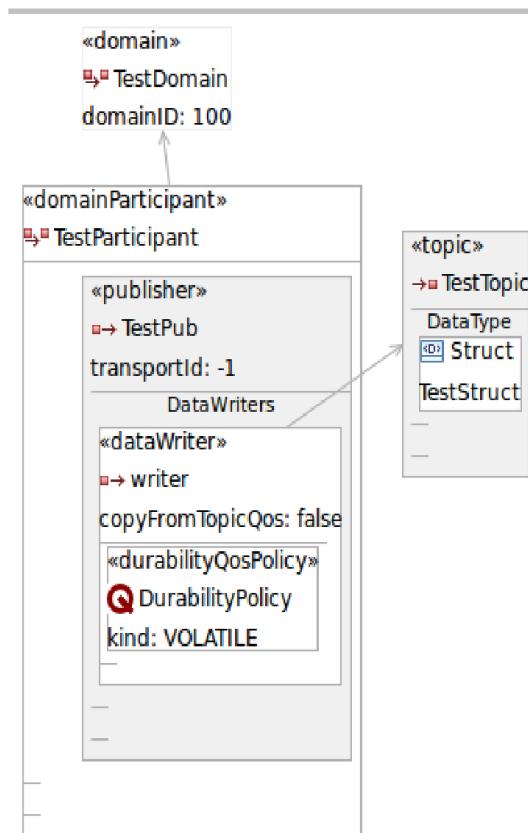
Figure 11-2 Graphical modeling of the DCPS entities

11.1.2 Code Generation

Once models have been captured, source code can be generated from them. This source code can then be compiled into link libraries providing the middleware elements defined in the model to applications that link the library. Code generation is done using a separate forms based editor.

Specifics of code generation are unique to the individual generation forms and are kept separate from the models for which generation is being performed. Code generation is performed on a single model at a time and includes the ability to tailor the generated code as well as specifying search paths to be used for locating resources at build time.

It is possible to generate model variations (distinct customizations of the same model) that can then be created within the same application or different applications. It is also possible to specify locations to search for header files and link libraries at build time.

See section 11.3.2, "Generated Code" for details.

11.1.3 Programming

In order to use the middleware defined by models, applications need to link in the generated code. This is done through header files and link libraries. Support for building applications using the MPC portable build tool is included in the generated files for a model.

See section 11.3, "Developing Applications" for details.

11.2 Installation and Getting Started

Unlike the OpenDDS Middleware which is compiled from source code by the developer, the compiled Modeling SDK is available for download via an Eclipse Update Site.

11.2.1 Prerequisites

• Java Runtime Environment (JRE)

• Eclipse IDE

11.2.2 Installation

1) From Eclipse, open the Help menu and select Install New Software.
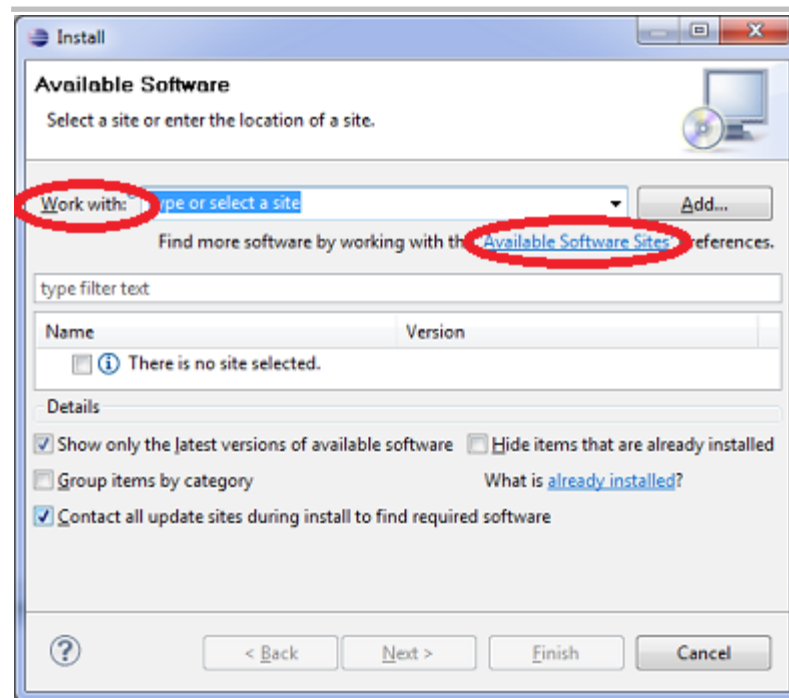
Figure 11-3 Eclipse Software Installation Dialog

9) Click the hyperlink for Available Software Sites.

10)   The standard `eclipse.org` sites (Eclipse Project Updates and Galileo) should be enabled. If they are disabled, enable them now.

11)   Add a new Site entry named OpenDDS with URL
       http://www.opendds.org/modeling/eclipse_44

12)   Click OK to close the Preferences dialog and return to the Install dialog.

13)   In the "Work with" combo box, select the new entry for OpenDDS.

14)   Select the "OpenDDS Modeling SDK" and click Next.

15)   Review the "Install Details" list and click Next. Review the license, select Accept (if you do accept it), and click Finish.

16)   Eclipse will download the OpenDDS plug-ins and various plug-ins from `eclipse.org` that they depend on. There will be a security warning because the OpenDDS plug-ins are not signed. There also may be a prompt to accept a certificate from `eclipse.org`.

17)   Eclipse will prompt the user to restart in order to use the newly installed software.

11.2.3 Getting Started

The OpenDDS Modeling SDK contains an Eclipse Perspective. Open it by going to the Window menu and selecting Open Perspective -> Other -> OpenDDS Modeling.

To get started using the OpenDDS Modeling SDK, see the help content installed in Eclipse. Start by going to the Help menu and selecting Help Contents. There is a top-level item for "*OpenDDS Modeling SDK Guide*" that contains all of the OpenDDS-specific content describing the modeling and code generation activities.

11.3 Developing Applications

In order to build an application using the OpenDDS Modeling SDK, one must understand a few key concepts. The concepts concern:

1) The support library
2) Generated model code
3) Application code

11.3.1 Modeling Support Library

The OpenDDS Modeling SDK includes a support library, found at *$DDS_ROOT/tools/modeling/codegen/model*. This support library, when combined with the code generated by the Modeling SDK, greatly reduces the amount of code needed to build an OpenDDS application.

The support library is a C++ library which is used by an OpenDDS Modeling SDK application. Two classes in the support library that most developers will need are the Application and Service classes.

11.3.1.1 The Application Class

The OpenDDS::Model::Application class takes care of initialization and finalization of the OpenDDS library. It is required for any application using OpenDDS to instantiate a single instance of the Application class, and further that the Application object not be destroyed while communicating using OpenDDS.

The Application class initializes the factory used to create OpenDDS participants. This factory requires the user-provided command line arguments. In order to provide them, the Application object must be provided the same command line arguments.

11.3.1.2 **The Service Class**

The `OpenDDS::Model::Service` class is responsible for the creation of OpenDDS entities described in an OpenDDS Modeling SDK model. Since the model can be generic, describing a much broader domain than an individual application uses, the Service class uses lazy instantiation to create OpenDDS entities.

In order to properly instantiate these entities, it must know:

- The relationships among the entities
- The transport configuration used by entities

11.3.2 **Generated Code**

The OpenDDS Modeling SDK generates model-specific code for use by an OpenDDS Modeling SDK application. Starting with a .codegen file (which refers to an *.opendds* model file), the files described in Table 11-1. The process of generating code is documented in the Eclipse help.

Table 11-1 Generated Files

File Name	Description
`<ModelName>.idl`	Data types from the model's `DataLib`
`<ModelName>_T.h`	C++ class from the model's `DcpsLib`
`<ModelName>_T.cpp`	C++ implementation of the model's `DcpsLib`
`<ModelName>.mpc`	MPC project file for the generated C++ library
`<ModelName>.mpb`	MPC base project for use by the application
`<ModelName>_paths.mpb`	MPC base project with paths, see section 11.3.3.7
`<ModelName>Traits.h`	Transport configuration from the .codegen file
`<ModelName>Traits.cpp`	Transport configuration from the .codegen file

11.3.2.1 **The DCPS Model Class**

The DCPS library models relationships between DDS entities, including `Topics`, `DomainParticipants`, `Publishers`, `Subscribers`, `DataWriters` and `DataReaders`, and their corresponding Domains.

For each DCPS library in your model, the OpenDDS Modeling SDK generates a class named after the DCPS library. This DCPS model class is named after the DCPS library, and is found in the *<ModelName>_T.h* file in the code generation target directory.

The model class contains an inner class, named Elements, defining enumerated identifiers for each DCPS entity modeled in the library and each type referenced by the library's Topics. This Elements class contains enumeration definitions for each of:

- DomainParticipants
- Types
- Topics
- Content Filtered Topics
- Multi Topics
- Publishers
- Subscribers
- Data Writers
- Data Readers

In addition, the DCPS model class captures the relationships between these entities. These relationships are used by the Service class when instantiating DCPS entities.

11.3.2.2   The Traits Class

Entities in a DCPS model reference their transport configuration by name. The Model Customization tab of the Codegen file editor is used to define the transport configuration for each name.

There can be more than one set of configurations defined for a specific code generation file. These sets of configurations are grouped into instances, each identified by a name. Multiple instances may be defined, representing different deployment scenarios for models using the application.

For each of these instances, a `Traits` class is generated. The traits class provides the transport configuration modeled in the Codegen editor for a specific transport configuration name.

11.3.2.3   The Service Typedef

The Service is a template which needs two parameters: (1) the entity model, in the DCPS model `Elements` class, (2) transport configuration, in a `Traits` class. The OpenDDS Modeling SDK generates one typedef for each combination of DCPS library and transport configuration model instance. The typedef is named `<InstanceName><DCPSLibraryName>Type`.

11.3.2.4   Data Library Generated Code

From the data library, IDL is generated, which is processed by the IDL compilers. The IDL compilers generate type support code, which is used to serialize and deserialize data types.

11.3.2.5   QoS Policy Library Generated Code

There are no specific compilation units generated from the QoS policy library. Instead, the DCPS library stores the QoS policies of the entities it models. This QoS policy is later queried by the Service class, which sets the QoS policy upon entity creation.

11.3.3   Application Code Requirements

11.3.3.1   Required headers

The application will need to include the `Traits` header, in addition to the *Tcp.h* header (for static linking). These will include everything required to build a publishing application. Here is the `#include` section of an example publishing application, *MinimalPublisher.cpp*.

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#endif

#include "model/MinimalTraits.h"
```

11.3.3.2   Exception Handling

It is recommended that Modeling SDK applications catch both `CORBA::Exception` objects and `std::exception` objects.

```
int ACE_TMAIN(int argc, ACE_TCHAR* argv[])
{
  try {
    // Create and use OpenDDS Modeling SDK (see sections below)
  } catch (const CORBA::Exception& e) {
    // Handle exception and return non-zero
  } catch (const OpenDDS::DCPS::Transport::Exception& te) {
    // Handle exception and return non-zero
  } catch (const std::exception& ex) {
    // Handle exception and return non-zero
  }
  return 0;
}
```

11.3.3.3   Instantiation

As stated above, an OpenDDS Modeling SDK application must create an `OpenDDS::Model::Application` object for the duration of its lifetime. This `Application` object, in turn, is passed to the constructor of the Service object specified by one of the typedef declarations in the traits headers.

The service is then used to create OpenDDS entities. The specific entity to create is specified using one of the enumerated identifiers specified in the `Elements` class. The Service provides this interface for entity creation:

```
DDS::DomainParticipant_var participant(Elements::Participants::Values part);
DDS::TopicDescription_var topic(Elements::Participants::Values part,
                                Elements::Topics::Values topic);
DDS::Publisher_var publisher(Elements::Publishers::Values publisher);
DDS::Subscriber_var subscriber(Elements::Subscribers::Values subscriber);
DDS::DataWriter_var writer(Elements::DataWriters::Values writer);
DDS::DataReader_var reader(Elements::DataReaders::Values reader);
```

It is important to note that the service also creates any required intermediate entities, such as `DomainParticipants`, `Publishers`, `Subscribers`, and `Topics`, when necessary.

11.3.3.4   Publisher Code

Using the `writer()` method shown above, *MinimalPublisher.cpp* continues:

```
int ACE_TMAIN(int argc, ACE_TCHAR* argv[])
{
  try {
    OpenDDS::Model::Application application(argc, argv);
    MinimalLib::DefaultMinimalType model(application, argc, argv);

    using OpenDDS::Model::MinimalLib::Elements;
    DDS::DataWriter_var writer = model.writer(Elements::DataWriters::writer);
```

What remains is to narrow the `DataWriter` to a type-specific data writer, and send samples.

```
    data1::MessageDataWriter_var msg_writer =
        data1::MessageDataWriter::_narrow(writer);
    data1::Message message;
    // Populate message and send
    message.text = "Worst. Movie. Ever.";
    DDS::ReturnCode_t error = msg_writer->write(message, DDS::HANDLE_NIL);
    if (error != DDS::RETCODE_OK) {
      // Handle error
    }
```

In total our publishing application, *MinimalPublisher.cpp*, looks like this:

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#endif

#include "model/MinimalTraits.h"

int ACE_TMAIN(int argc, ACE_TCHAR* argv[])
{
  try {
    OpenDDS::Model::Application application(argc, argv);
    MinimalLib::DefaultMinimalType model(application, argc, argv);
```

```
    using OpenDDS::Model::MinimalLib::Elements;
    DDS::DataWriter_var writer = model.writer(Elements::DataWriters::writer);

    data1::MessageDataWriter_var msg_writer =
        data1::MessageDataWriter::_narrow(writer);
    data1::Message message;
    // Populate message and send
    message.text = "Worst. Movie. Ever.";
    DDS::ReturnCode_t error = msg_writer->write(message, DDS::HANDLE_NIL);
    if (error != DDS::RETCODE_OK) {
      // Handle error
    }
  } catch (const CORBA::Exception& e) {
    // Handle exception and return non-zero
  } catch (const std::exception& ex) {
    // Handle exception and return non-zero
  }
  return 0;
}
```

Note this minimal example ignores logging and synchronization, which are issues that are not specific to the OpenDDS Modeling SDK.

11.3.3.5   Subscriber Code

The subscriber code is much like the publisher. For simplicity, OpenDDS Modeling SDK subscribers may want to take advantage of a base class for Reader Listeners, called `OpenDDS::Modeling::NullReaderListener`. The `NullReaderListener` implements the entire `DataReaderListener` interface and logs every callback.

Subscribers can create a listener by deriving a class from `NullReaderListener` and overriding the interfaces of interest, for example on_data_available.

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#endif

#include "model/MinimalTraits.h"
#include <model/NullReaderListener.h>

class ReaderListener : public OpenDDS::Model::NullReaderListener {
public:
  virtual void on_data_available(DDS::DataReader_ptr reader)
                       ACE_THROW_SPEC((CORBA::SystemException)) {
    data1::MessageDataReader_var reader_i =
      data1::MessageDataReader::_narrow(reader);

    if (!reader_i) {
      // Handle error
      ACE_OS::exit(-1);
    }

    data1::Message msg;
    DDS::SampleInfo info;

    // Read until no more messages
```

```
      while (true) {
        DDS::ReturnCode_t error = reader_i->take_next_sample(msg, info);
        if (error == DDS::RETCODE_OK) {
          if (info.valid_data) {
            std::cout << "Message: " << msg.text.in() << std::endl;
          }
        } else {
          if (error != DDS::RETCODE_NO_DATA) {
            // Handle error
          }
          break;
        }
      }
    }
};
```

In the main function, create a data reader from the service object:

```
    DDS::DataReader_var reader = model.reader(Elements::DataReaders::reader);
```

Naturally, the `DataReaderListener` must be associated with the data reader in order to get its callbacks.

```
    DDS::DataReaderListener_var listener(new ReaderListener);
    reader->set_listener(listener, OpenDDS::DCPS::DEFAULT_STATUS_MASK);
```

The remaining subscriber code has the same requirements of any OpenDDS Modeling SDK application, in that it must initialize the OpenDDS library through an `OpenDDS::Modeling::Application` object, and create a Service object with the proper DCPS model Elements class and traits class.

An example subscribing application, *MinimalSubscriber.cpp*, follows.

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#endif

#include "model/MinimalTraits.h"
#include <model/NullReaderListener.h>

class ReaderListener : public OpenDDS::Model::NullReaderListener {
public:
  virtual void on_data_available(DDS::DataReader_ptr reader)
                      ACE_THROW_SPEC((CORBA::SystemException)) {
    data1::MessageDataReader_var reader_i =
      data1::MessageDataReader::_narrow(reader);

    if (!reader_i) {
      // Handle error
      ACE_OS::exit(-1);
    }

    data1::Message msg;
    DDS::SampleInfo info;
```

```
      // Read until no more messages
      while (true) {
        DDS::ReturnCode_t error = reader_i->take_next_sample(msg, info);
        if (error == DDS::RETCODE_OK) {
          if (info.valid_data) {
            std::cout << "Message: " << msg.text.in() << std::endl;
          }
        } else {
          if (error != DDS::RETCODE_NO_DATA) {
            // Handle error
          }
          break;
        }
      }
    }
};

int ACE_TMAIN(int argc, ACE_TCHAR* argv[])
{
  try {
    OpenDDS::Model::Application application(argc, argv);
    MinimalLib::DefaultMinimalType model(application, argc, argv);

    using OpenDDS::Model::MinimalLib::Elements;

    DDS::DataReader_var reader = model.reader(Elements::DataReaders::reader);

    DDS::DataReaderListener_var listener(new ReaderListener);
    reader->set_listener(listener, OpenDDS::DCPS::DEFAULT_STATUS_MASK);

    // Call on_data_available in case there are samples which are waiting
    listener->on_data_available(reader);

    // At this point the application can wait for an exteral "stop" indication
    // such as blocking until the user terminates the program with Ctrl-C.

  } catch (const CORBA::Exception& e) {
    e._tao_print_exception("Exception caught in main():");
    return -1;
  } catch (const std::exception& ex) {
    // Handle error
    return -1;
  }
  return 0;
}
```

11.3.3.6 MPC Projects

In order to make use of the OpenDDS Modeling SDK support library, OpenDDS Modeling SDK MPC projects should inherit from the dds_model project base. This is in addition to the dcpsexe base from which non-Modeling SDK projects inherit.

```
project(*Publisher) : dcpsexe, dds_model {
  // project configuration
}
```

The generated model library will generate an MPC project file and base project file in the target directory, and take care of building the model shared library. OpenDDS modeling

applications must both (1) include the generated model library in their build and (2) ensure their projects are built after the generated model libraries.

```
project(*Publisher) : dcpsexe, dds_model {
  // project configuration
  libs  += Minimal
  after += Minimal
}
```

Both of these can be accomplished by inheriting from the model library's project base, named after the model library.

```
project(*Publisher) : dcpsexe, dds_model, Minimal {
  // project configuration
}
```

Note that the *Minimal.mpb* file must now be found by MPC during project file creation. This can be accomplished through the -include command line option.

Using either form, the MPC file must tell the build system where to look for the generated model library.

```
project(*Publisher) : dcpsexe, dds_model, Minimal {
  // project configuration
  libpaths += model
}
```

This setting based upon what was provided to the Target Folder setting in the Codegen file editor.

Finally, like any other MPC project, its source files must be included:

```
  Source_Files {
    MinimalPublisher.cpp
  }
```

The final MPC project looks like this for the publisher:

```
project(*Publisher) : dcpsexe, dds_model, Minimal {
  exename   = publisher
  libpaths += model

  Source_Files {
    MinimalPublisher.cpp
  }
}
```

And similar for the subscriber:

```
project(*Subscriber) : dcpsexe, dds_model, Minimal {
  exename   = subscriber
  libpaths += model
```

```
    Source_Files {
      MinimalSubscriber.cpp
    }
}
```

11.3.3.7 Dependencies Between Models

One final consideration — the generated model library could itself depend on other generated model libraries. For example, there could be an external data type library which is generated to a different directory.

This possibility could cause a great deal of maintenance of project files, as models change their dependencies over time. To help overcome this burden, the generated model library records the paths to all of its externally referenced model libraries in a separate MPB file named *<ModelName>_paths.mpb*. Inheriting from this paths base project will inherit the needed settings to include the dependent model as well.

Our full MPC file looks like this:

```
project(*Publisher) : dcpsexe, dds_model, Minimal, Minimal_paths {
  exename   = publisher
  libpaths += model

  Source_Files {
    MinimalPublisher.cpp
  }
}

project(*Subscriber) : dcpsexe, dds_model, Minimal, Minimal_paths {
  exename   = subscriber
  libpaths += model

  Source_Files {
    MinimalSubscriber.cpp
  }
}
```

CHAPTER 12

Alternate Interfaces to Data

The DDS-DCPS approach to data transfer using synchronization of strongly-typed caches (DataWriter and DataReader) is not appropriate for all applications. Therefore OpenDDS provides two different alternate interface approaches which are described in this chapter. These are not defined by OMG specifications and may change in future releases of OpenDDS, including minor updates. The two approaches are:

- Recorder and Replayer

  ◦ These interfaces allow the application to create untyped stand-ins for DataReaders and/or DataWriters

- Observer

  ◦ Observers play a role similar to the spec-defined Listeners (attached to DataReaders and/or DataWriters). Unlike the Listeners, Observers don't need to interact with the DataReader/Writer caches to access the data samples.

12.1 Recorder and Replayer

The Recorder feature of OpenDDS allows applications to record samples published on arbitrary topics without any prior knowledge of the data type used by that topic. Analogously, the Replayer feature allows these recorded samples to by re-published back into the same or other topics. What makes these features different from other Data Readers and Writers are their ability to work with any data type, even if unknown at application

build time. Effectively, the samples are treated as if each one contains an opaque byte sequence.

The purpose of this section is to describe the public API for OpenDDS to enable the recording/replaying use-case.

12.1.1   API Structure

Two new user-visible classes (that behave somewhat like their DDS Entity counterparts) are defined in the `OpenDDS::DCPS` namespace, along with the associated Listener interfaces. Listeners may be optionally implemented by the application. The `Recorder` class acts similarly to a `DataReader` and the `Replayer` class acts similarly to a `DataWriter`.

Both `Recorder` and `Replayer` make use of the underlying OpenDDS discovery and transport libraries as if they were `DataReader` and `DataWriter`, respectively. Regular OpenDDS applications in the domain will "see" the `Recorder` objects as if they were remote `DataReader`s and `Replayer`s as if they were `DataWriter`s.

12.1.2   Usage Model

The application creates any number of `Recorder`s and `Replayer`s as necessary. This could be based on using the Built-In Topics to dynamically discover which topics are active in the Domain. Creating a `Recorder` or `Replayer` requires the application to provide a topic name and type name (as in `DomainParticipant::create_topic()`) and also the relevant QoS data structures. The `Recorder` requires SubscriberQos and DataReaderQos whereas the `Replayer` requires PublisherQos and DataWriterQos. These values are used in discovery's reader/writer matching. See the section on QoS processing below for how the `Recorder` and `Replayer` use QoS. Here is the code needed to create a recorder:

```
OpenDDS::DCPS::Recorder_var recorder =
    service_participant->create_recorder(domain_participant,
                                         topic.in(),
                                         sub_qos,
                                         dr_qos,
                                         recorder_listener);
```

Data samples are made available to the application via the `RecorderListener` using a simple "one callback per sample" model. The sample is delivered as an `OpenDDS::DCPS::RawDataSample` object. This object includes the timestamp for that data sample as well as the marshaled sample value. Here is a class definition for a user-defined Recorder Listener.

```
class MessengerRecorderListener : public OpenDDS::DCPS::RecorderListener
{
```

```
public:
  MessengerRecorderListener();

  virtual void on_sample_data_received(OpenDDS::DCPS::Recorder*,
                                       const OpenDDS::DCPS::RawDataSample& sample);

  virtual void on_recorder_matched(OpenDDS::DCPS::Recorder*,
                                   const DDS::SubscriptionMatchedStatus& status );

};
```

The application can store the data wherever it sees fit (in memory, file system, database, etc.). At any later time, the application can provide that same sample to a `Replayer` object configured for the same topic. It's the application's responsibility to make sure the topic types match. Here is an example call that replays a sample to all readers connected on a replayer's topic:

```
replayer->write(sample);
```

Because the stored data is dependent on the definition of the data structure, it can't be used across different versions of OpenDDS or different versions of the IDL used by the OpenDDS participants.

12.1.3   QoS Processing

The lack of detailed knowledge about the data sample complicates the use of many normal DDS QoS properties on the `Replayer` side. The properties can be divided into a few categories:

- Supported
    - Liveliness
    - Time-Based Filter
    - Lifespan
    - Durability (transient local level, see details below)
    - Presentation (topic level only)
    - Transport Priority (pass-thru to transport)
- Unsupported
    - Deadline (still used for reader/writer match)
    - History
    - Resource Limits
    - Durability Service
    - Ownership and Ownership Strength (still used for reader/writer match)
- Affects reader/writer matching and Built-In Topics but otherwise ignored
    - Partition

OpenDDS   Developer's   Guide

- – Reliability (still used by transport negotiation)
- – Destination Order
- – Latency Budget
- – User/Group Data

12.1.3.1 Durability details

On the `Recorder` side, transient local durability works just the same as any normal `DataReader`. Durable data is received from matched `DataWriters`. On the `Replayer` side there are some differences. As opposed to the normal DDS `DataWriter`, `Replayer` is not caching/storing any data samples (they are simply sent to the transport). Because instances are not known, storing data samples according to the usual History and Resource Limits rules is not possible. Instead, transient local durability can be supported with a "pull" model whereby the middleware invokes a method on the `ReplayerListener` when a new remote `DataReader` is discovered. The application can then call a method on the `Replayer` with any data samples that should be sent to that newly-joined `DataReader`. Determining which samples these are is left to the application.

12.2 Observer

To observe the most important events happening within OpenDDS, applications can create classes that derive from the Observer abstract base class (in `dds/DCPS/Observer.h`). The design of Observer is intended to allow applications to have a single Observer object observing many Entities, however this is flexible to allow many different use cases. The following events can be observed:

- • DataWriter/Reader enabled, deleted

- • DataWriter/Reader QoS changed

- • DataWriter/Reader peer associated, disassociated

- • DataWriter sample sent

- • DataReader sample received (enters the cache), read, taken

12.2.1 Attaching Observers To Entities

Entity is the spec-defined base interface of the following types:

- • DataWriter, DataReader

  - ◦ As seen above in 12.2, the Observer events originate in the DataWriter and DataReader Entities

- DomainParticipant, Publisher, Subscriber

    ◦ Among their other roles, these Entities act as containers (either directly or indirectly) for DataWriters and DataReaders.

    ◦ If a smaller-scoped Entity (such as a DataWriter) has no Observer for the event in question, its containing Entity (in this example, a Publisher) is checked for an Observer.

- Topic

    ◦ Although it is an Entity, no Observer events are generated by Topics or Entities they contain (since they don't contain any Entities)

The class EntityImpl (in `dds/DCPS/EntityImpl.h`) is OpenDDS's base class for all Entity types. EntityImpl includes public methods for Observer registration: set_observer and get_observer. These methods are not part of the IDL interfaces, so invoking them the requires a cast to the implementation (Impl) of Entity.

```
DDS::DataWriter_var dw = /* … */;

EntityImpl* entity = dynamic_cast<EntityImpl*>(dw.in());

Observer_rch observer = make_rch<MyObserver>();

entity->set_observer(observer, Observer::e_SAMPLE_SENT);
```

Note that since the Observer class as an internal (not IDL) interface, it uses the "RCH" (Reference Counted Handle) smart pointer classes. Observer itself inherits from RcObject, and uses of Observer-derived classes should use the RcHandle template and its associated functions, as in the example above. See `dds/DCPS/RcHandle_T.h` for details.

12.2.2 Writing Observer-Derived Classes

The virtual methods in the Observer class are divided into 3 groups based on the general category of events they observe:

1. Operations on the observed Entity itself

    ◦ on_enabled, on_deleted, on_qos_changed

    ◦ The only parameter to these methods is the Entity, so the Observer implementation can use the public methods on the Entity.

2. Events relating to associating with remote matched endpoints

    ◦ on_associated, on_disassociated

- In addition to the Entity, the Observer implementation receives a GUID_t structure which is the internal representation of remote Entity identity. The GUID_t values from on_associated could be stored or logged to correlate them with the values from on_disassociated.

3. Events relating to data samples moving through the system

   - on_sample_sent, on_sample_received, on_sample_read, on_sample_taken

   - In addition to the Entity, the Observer implementation receives an instance of the Sample structure. The definition of this structure is nested within Observer. See below for details.

12.2.3 The Observer::Sample Structure

The Observer::Sample structure contains the following fields:

- instance and instance_state

  - Describe the instance that this sample belongs to, using the spec-defined types

- timestamp and sequence_number

  - Attributes of the sample itself: timestamp uses a spec-defined type whereas sequence_number uses the OpenDDS internal type for DDSI-RTPS 64-bit sequence numbers.

- data and data_dispatcher

  - Since Observer is an un-typed interface, the contents of the data sample itself are represented only as a void pointer

  - Implementations that need to process this data can use the data_dispatcher object to interpret it. See the class definition of ValueWriterDispatcher in `dds/DCPS/ValueWriter.h` for more details.

CHAPTER 13

Safety Profile

13.1 Overview

The Safety Profile configuration allows OpenDDS to be used in environments that have a restricted set of operating system and standard library functions available and that require dynamic memory allocation to occur only at system start-up.

OpenDDS Safety Profile (and the corresponding features in ACE) were developed for the Open Group's FACE specification, edition 2.1 (http://www.opengroup.org/face/tech-standard-2.1). It can be used along with the support for FACE Transport Services to create FACE-conformant DDS applications, or it can be used by general DDS applications that are not written to the FACE Transport Services APIs. This latter use-case is described by this section of the developer's guide. For more information on the former use-case see the file FACE/README.txt in the source distribution or contact us at sales@objectcomputing.com (commercial support) or opendds-main@lists.sourceforge.net (community support).

13.2 Safety Profile Subset of OpenDDS

The following features of OpenDDS are not available when it is configured for Safety Profile:

- DCPSInfoRepo and its associated libraries and tools

- Transport types: tcp, udp, multicast, shared memory

    - The rtps_udp transport type is available (uses UDP unicast or multicast)

- OpenDDS Monitor library and monitoring GUI

When developing the Safety Profile, the following DDS Compliance Profiles were disabled:

- content_subscription

- ownership_kind_exclusive

- object_model_profile

- persistence_profile

See Section 1.3.3 for more details on compliance profiles. It is possible that enabling any of these compliance profiles in a Safety Profile build will result in a compile-time or run-time error.

To build OpenDDS Safety Profile, pass the command line argument "--safety-profile" to the configure script along with any other arguments needed for your platform or configuration. When safety profile is enabled in the configure script, the four compliance profiles listed above default to disabled. See section 1.3 and the INSTALL.md file in the source distribution for more information about the configure script.

13.3   Safety Profile Configurations of ACE

OpenDDS uses ACE as its platform abstraction library, and in OpenDDS's Safety Profile configuration, one of the following safety profile configurations must be enabled in ACE:

- FACE Safety Base (always uses the memory pool)

- FACE Safety Extended with Memory Pool

- FACE Safety Extended with Standard C++ Dynamic Allocation

OpenDDS's configure script will automatically configure ACE. Pass the command line argument "--safety-profile=base" to select the Safety Base profile. Otherwise a "--safety-profile" (no equals sign) configuration will default to Safety Extended with Memory Pool.

The Safety Extended with Standard C++ Dynamic Allocation configuration is not automatically generated by the configure script, but the file "build/target/ACE_wrappers/ace/config.h" can be edited after it is generated by configure (and before running make). Remove the macro definition for ACE_HAS_ALLOC_HOOKS to disable the memory pool.

ACE's safety profile configurations have been tested on Linux and on LynxOS-178 version 2.3.2+patches. Other platforms may work too but may require additional configuration.

13.4   Run-time Configurable Options

The memory pool used by OpenDDS can be configured by setting values in the [common] section of the configuration file. See section 7.2 and the pool_size and pool_granularity rows of table Table 7-2.

13.5   Running ACE and OpenDDS Tests

After configuring and building OpenDDS Safety Profile, note that there are two sub-directories of the top level that each contain some binary artifacts:

- build/host has the build-time code generators tao_idl and opendds_idl

- build/target has the run-time libraries for safety profile ACE and OpenDDS and the OpenDDS tests

Therefore, testing needs to be relative to the build/target sub-directory. Source-in the generated file `build/target/setenv.sh` to get all of the needed environment variables.

ACE tests are not built by default, but once this environment is set up all it takes to build them is generating makefiles and running make:

1. `cd $ACE_ROOT/tests`

2. `$ACE_ROOT/bin/mwc.pl -type gnuace`

3. `make`

Run ACE tests by changing to the $ACE_ROOT/tests directory and using run_test.pl. Pass any "-Config XYZ" options required for your configuration (use run_test.pl -h to see the available Config options).

Run OpenDDS tests by changing to the $DDS_ROOT and using bin/auto_run_tests.pl. Pass "-Config OPENDDS_SAFETY_PROFILE", "-Config SAFETY_BASE" (if using safety base), "-Config RTPS", and -Config options corresponding to each disabled compliance profile, by default: "-Config DDS_NO_OBJECT_MODEL_PROFILE -Config DDS_NO_OWNERSHIP_KIND_EXCLUSIVE -Config DDS_NO_PERSISTENCE_PROFILE -Config DDS_NO_CONTENT_SUBSCRIPTION".

Alternatively, an individual test can be run using run_test.pl from that test's directory. Pass the same set of -Config options to run_test.pl.

13.6 Using the Memory Pool in Applications

When the Memory Pool is enabled at build time, all dynamic allocations made by code in OpenDDS or in ACE (methods invoked by OpenDDS) go through the pool. Since the pool is a general purpose dynamic allocator, it may be desirable for application code to use the pool too. Since these APIs are internal to OpenDDS, they may change in future releases.

The class OpenDDS::DCPS::MemoryPool (dds/DCPS/MemoryPool.h) contains the pool implementation. However, most client code shouldn't interact directly with it. The class SafetyProfilePool (dds/DCPS/SafetyProfilePool.h) adapts the pool to the ACE_Allocator interface. PoolAllocator<T> (PoolAllocator.h) adapts the pool to the C++ Allocator concept (C++03). Since the PoolAllocator is stateless, it depends on the ACE_Allocator's singleton. When OpenDDS is configured with the memory pool, ACE_Allocator's singleton instance will point to an object of class SafetyProfilePool.

Application code that makes use of C++ Standard Library classes can either use PoolAllocator directly, or make use of the macros defined in PoolAllocator.h (for example OPENDDS_STRING).

Application code that allocates raw (untyped) buffers of dynamic memory can use SafetyProfilePool either directly or via the ACE_Allocator::instance() singleton.

Application code that allocates objects from the heap can use the PoolAllocator<T> template.

Classes written by the application developer can derive from PoolAllocationBase (see PoolAllocationBase.h) to inherit class-scoped operators new and delete, thus redirecting all dynamic allocation of these classes to the pool.

CHAPTER 14

DDS Security

14.1 Building OpenDDS with Security Enabled

Prior to utilizing DDS Security, OpenDDS must be built to include security elements into the resulting libraries. The following instructions show how this is to be completed on various platforms.

14.1.1 Prerequisites

OpenDDS includes an implementation of the OMG DDS Security 1.1 specification. Building OpenDDS with security enabled requires the following dependencies:

1. Xerces-C++ v3.x

2. OpenSSL v1.0.2+ or v1.1 (1.1 is preferred)

3. Google Test (only required if building OpenDDS tests)

   (A) If you are using OpenDDS from a git repository, Google Test is provided as a git submodule. Make sure to enable submodules with the `--recursive` option to git clone.

4. CMake (required if building OpenDDS tests and building Google Test and other dependencies from source).

General Notes on Using OpenDDS Configure Script with DDS Security

1. DDS Security is disabled by default, enable it with `--security`

2. OpenDDS tests are disabled by default, enable them with `--tests`

   (A) Disabling tests skips the Google Test and CMake dependencies

   (B) If tests are enabled, the configure script can run CMake and build Google Test

14.1.2 Building OpenDDS With Security On Windows Using Microsoft vcpkg

Microsoft vcpkg is a "C++ Library Manager for Windows, Linux, and macOS" which helps developers build/install dependencies. Although it is cross-platform, this guide only discusses vcpkg on Windows.

As of this writing, vcpkg is only supported on Visual Studio 2015 Update 3 and later versions; if using an earlier version of Visual Studio, skip down to the manual setup instructions later in this section.

1) If OpenDDS tests will be built, install CMake or put the one that comes with Visual Studio on the PATH (see Common7\IDE\CommonExtensions\Microsoft\CMake).

2) If you need to obtain and install vcpkg, navigate to https://github.com/Microsoft/vcpkg and follow the instructions to obtain vcpkg by cloning the repository and bootstrapping it.

3) Fetch and build the dependencies; by default, vcpkg targets x86 so be sure to specify the x64 target if required by specifying it when invoking vcpkg install, as shown here:
   ```
   vcpkg install openssl:x64-windows xerces-c:x64-windows
   ```

4) Configure OpenDDS by passing the openssl and xerces3 switches. As a convenience, it can be helpful to set an environment variable to store the path since it is the same location for both dependencies.
   ```
   set VCPKG_INSTALL=c:\path\to\vcpkg\installed\x64-windows
   configure --security --openssl=%VCPKG_INSTALL% --xerces3=%VCPKG_INSTALL%
   ```

5) Compile with msbuild or by launching Visual Studio from this command prompt so it inherits the correct environment variables and building from there.

```
msbuild /m DDS_TAOv2_all.sln
```

Manual Build

Note: for all of the build steps listed here, check that each package targets the same architecture (either 32-bit or 64-bit) by compiling all dependencies within the same type of Developer Command Prompt.

Compiling OpenSSL

(see: https://wiki.openssl.org/index.php/Compilation_and_Installation#Windows)

1. Install Perl and add it to the Path environment variable. For this guide, ActiveState is used.

2. Install Netwide Assembler (NASM). Click through the latest stable release and there is a win32 and win64 directory containing executable installers. The installer does not update the Path environment variable, so a manual entry (`%LOCALAPPDATA%\bin\NASM`) is necessary.

3. Download the required version of OpenSSL by cloning the repository.

4. Open a Developer Command Prompt (32-bit or 64-bit depending on the desired target architecture) and change into the freshly cloned openssl directory.

5. Run the configure script and specify a required architecture (`perl Configure VC-WIN32` or `perl Configure VC-WIN64A`).

6. Run `nmake`.

7. Run `nmake install`.

Note: if the default OpenSSL location is desired, which will be searched by OpenDDS, open the Developer Command Prompt as an administrator before running the install. It will write to "C:\Program Files" or "C:\Program Files (x86)" depending on the architecture.

Compiling Xerces-C++ 3

(see: https://xerces.apache.org/xerces-c/build-3.html)

1. Download/extract the Xerces source files.

2. Create a cmake build directory and change into it (from within the Xerces source tree).

    ```
    mkdir build
    ```

```
cd build
```

3. Run cmake with the appropriate generator. In this case Visual Studio 2017 with 64-bit is being used so:

```
cmake -G "Visual Studio 15 2017 Win64" ..
```

4. Run cmake again with the build switch and install target (this should be done in an administrator command-prompt to install in the default location as mentioned above).

```
cmake --build . --target install
```

Configuring and Building OpenDDS:

1. Change into the OpenDDS root folder and run configure with security enabled.

   A) If the default location was used for OpenSSL and Xerces, configure should automatically find the dependencies:

   ```
   configure --security
   ```

   B) If a different location was used (assuming environment variables NEW_SSL_ROOT and NEW_XERCES_ROOT point to their respective library directories):

   ```
   configure --security --openssl=%NEW_SSL_ROOT%
            --xerces3=%NEW_XERCES_ROOT%
   ```

2. Compile with msbuild (or by opening the solution file in Visual Studio and building from there).

   ```
   msbuild /m DDS_TAOv2_all.sln
   ```

14.1.3 Building OpenDDS With Security On Linux

Xerces-C++ and OpenSSL may be installed using the system package manager, or built from source. If using the system package manager (that is, headers can be found under /usr/include), invoke the configure script with the --security option. If Xerces-C++ and/or OpenSSL are built from source or installed in a custom location, also provide the --xerces3=/foo and --openssl=/bar command line options.

14.1.4 Building OpenDDS With Security On MacOS

Xerces-C++ and OpenSSL may be installed using homebrew or another developer-focused package manager, or built from source. The instructions above for Linux also apply to

macOS but the package manager will not install directly in `/usr` so make sure to specify the library locations to the configure script.

14.1.5 Building OpenDDS With Security For Android

See the `docs/android.md` file included in the OpenDDS source code.

14.2 Architecture of the DDS Security Specification

The DDS Security specification defines plugin APIs for Authentication, Access Control, and Cryptographic operations. These APIs provide a level of abstraction for DDS implementations as well as allowing for future extensibility and version control. Additionally, the specification defines Built-In implementations of each of these plugins, which allows for a baseline of functionality and interoperability between DDS implementations. OpenDDS implements these Built-In plugins, and this document assumes that the Built-In plugins are being used. Developers using OpenDDS may also implement their own custom plugins, but those efforts are well beyond the scope of this document.

14.3 Terms and Background Info

DDS Security uses current industry standards and best-practices in security. As such, this document makes use of several security concepts which may warrant additional research by OpenDDS users.

Term Group	References
Public Key Cryptography (including Private Keys)	• https://en.wikipedia.org/wiki/Public-key_cryptography • RSA – https://en.wikipedia.org/wiki/RSA_(algorithm) • Elliptic Curve Cryptography - https://en.wikipedia.org/wiki/Elliptic_curve_cryptography
Public Key Certificate	• https://en.wikipedia.org/wiki/Public_key_certificate • Certificate Authority – https://en.wikipedia.org/wiki/Certificate_authority • X.509 – https://en.wikipedia.org/wiki/X.509 • PEM - https://en.wikipedia.org/wiki/Privacy-enhanced_Electronic_Mail
Signed Documents	• https://en.wikipedia.org/wiki/Digital_signature

Table 14-1

14.4   Required DDS Security Artifacts

14.4.1   Per-Domain Artifacts

These are shared by all participants within the secured DDS Domain:

- Identity CA Certificate

- Permissions CA Certificate (may be same as Identity CA Certificate)

- Governance Document

    - Signed by Permissions CA using its private key

14.4.2   Per-Participant Artifacts

These are specific to the individual Domain Participants within the DDS Domain:

- Identity Certificate and its Private Key

    - Issued by Identity CA (or a CA that it authorized to act on its behalf)

- Permissions Document

    - Contains a "subject name" which matches the participant certificate's Subject

    - Signed by Permissions CA using its private key

14.5   Required OpenDDS Configuration

The following configuration steps are required to enable OpenDDS Security features:

1.   Select RTPS Discovery and the RTPS-UDP Transport; because DDS Security only works with these configurations, both must be specified for any security-enabled participant.

2.   Enable OpenDDS security-features, which can be done two ways:

    (A)   Via API: "TheServiceParticipant->set_security(true);" or

    (B)   Via config file: "DCPSSecurity=1" in the [common] section.

14.5.1   DDS Security Configuration Via PropertyQosPolicy

When the application creates a DomainParticipant object, the DomainParticipantQos passed to the `create_participant()` method now contains a PropertyQosPolicy object which has a sequence of name-value pairs. The following properties must be included to enable security. Except where noted, these values take the form of a URI starting with either the scheme "file:" followed by a filesystem path (absolute or relative) or the scheme "data:" followed by the literal data.

Name	Value	Notes
`dds.sec.auth.identity_ca`	Certificate PEM file	Can be the same as `permissions_ca`
`dds.sec.access.permissions_ca`	Certificate PEM file	Can be the same as `identity_ca`
`dds.sec.access.governance`	Signed XML (.p7s)	Signed by `permissions_ca`
`dds.sec.auth.identity_certificate`	Certificate PEM file	Signed by `identity_ca`
`dds.sec.auth.private_key`	Private Key PEM file	Private key for `identity_certificate`
`dds.sec.auth.password`	Private Key Password (not a URI)	Optional, Base64 encoded
`dds.sec.access.permissions`	Signed XML (.p7s)	Signed by `permissions_ca`

Table 14-2

14.5.2   PropertyQosPolicy Example Code

Below is an example of code that sets the DDS Participant QoS's PropertyQoSPolicy in order to configure DDS Security.

```
// DDS Security artifact file locations
const char auth_ca_file[] = "file:identity_ca_cert.pem";
const char perm_ca_file[] = "file:permissions_ca_cert.pem";
const char id_cert_file[] = "file:test_participant_01_cert.pem";
const char id_key_file[] = "file:test_participant_01_private_key.pem";
const char governance_file[] = "file:governance_signed.p7s";
const char permissions_file[] = "file:permissions_01_signed.p7s";

// DDS Security property names
const char DDSSEC_PROP_IDENTITY_CA[] = "dds.sec.auth.identity_ca";
const char DDSSEC_PROP_IDENTITY_CERT[] = "dds.sec.auth.identity_certificate";
const char DDSSEC_PROP_IDENTITY_PRIVKEY[] = "dds.sec.auth.private_key";
```

```
const char DDSSEC_PROP_PERM_CA[] = "dds.sec.access.permissions_ca";
const char DDSSEC_PROP_PERM_GOV_DOC[] = "dds.sec.access.governance";
const char DDSSEC_PROP_PERM_DOC[] = "dds.sec.access.permissions";

void append(DDS::PropertySeq& props, const char* name, const char* value)
{
  const DDS::Property_t prop = {name, value, false /*propagate*/};
  const unsigned int len = props.length();
  props.length(len + 1);
  props[len] = prop;
}

int main(int argc, char* argv[])
{
  DDS::DomainParticipantFactory_var dpf =
    TheParticipantFactoryWithArgs(argc, argv);

  // Start with the default Participant QoS
  DDS::DomainParticipantQos part_qos;
  dpf->get_default_participant_qos(part_qos);

  // Add properties required by DDS Security
  DDS::PropertySeq& props = part_qos.property.value;
  append(props, DDSSEC_PROP_IDENTITY_CA, auth_ca_file);
  append(props, DDSSEC_PROP_IDENTITY_CERT, id_cert_file);
  append(props, DDSSEC_PROP_IDENTITY_PRIVKEY, id_key_file);
  append(props, DDSSEC_PROP_PERM_CA, perm_ca_file);
  append(props, DDSSEC_PROP_PERM_GOV_DOC, governance_file);
  append(props, DDSSEC_PROP_PERM_DOC, permissions_file);

  // Create the participant
  participant = dpf->create_participant(4, // DomainID
                                        part_qos,
                                        0, // No listener
                                        OpenDDS::DCPS::DEFAULT_STATUS_MASK);
…
```

14.5.3 Identity Certificates And Certificate Authorities

All certificate inputs to OpenDDS, including self-signed CA certificates, are expected to be an X.509 v3 certificate in PEM format for either a 2048-bit RSA key or a 256-bit Elliptic Curve key (using the prime256v1 curve).

14.5.4 Identity, Permissions, And Subject Names

The "subject_name" element for a signed permissions XML document must match the "Subject:" field provided by the accompanying Identity Certificate which is transmitted during participant discovery, authentication, and authorization. This ensures that the permissions granted by the Permissions CA do, in fact, correspond to the identity provided.

14.5.5 Examples In The OpenDDS Source Code Repository

Examples to demonstrate how the DDS Security features are used with OpenDDS can be found in the OpenDDS GitHub repository found here:

OpenDDS GitHub - https://github.com/objectcomputing/OpenDDS

The following table describes the various examples and where to find them in the source tree.

Example	Source Location
C++ application that configures security QoS policies via command-line parameters	tests/DCPS/Messenger/publisher.cpp
Identity CA Certificate (along with private key)	tests/security/certs/identity/identity_ca_cert.pem
Permissions CA Certificate (along with private key)	tests/security/certs/permissions/permissions_ca_cert.pem
Participant Identity Certificate (along with private key)	tests/security/certs/identity/test_participant_01_cert.pem
Governance XML Document (alongside signed document)	tests/DCPS/Messenger/governance.xml
Permissions XML Document (alongside signed document)	tests/DCPS/Messenger/permissions_1.xml

Table 14-3

OpenDDS Developer's Guide

14.5.6   **Using OpenSSL Utilities For OpenDDS**

To generate certificates using the openssl command, a configuration file "openssl.cnf" is required (see below for example commands). Before proceeding, it may be helpful to review OpenSSL's manpages to get help with the file format. In particular, configuration file format and ca command's documentation and configuration file options.

An example OpenSSL CA-Config file used in OpenDDS testing can be found here:

https://github.com/objectcomputing/OpenDDS/blob/master/tests/security/certs/identity/ identity_ca_openssl.cnf

Creating Self-Signed Certificate Authorities

Generate a self-signed 2048-bit RSA CA:

```
openssl genrsa -out ca_key.pem 2048
openssl req -config openssl.cnf -new -key ca_key.pem -out ca.csr
openssl x509 -req -days 3650 -in ca.csr -signkey ca_key.pem -out ca_cert.pem
```

Generate self-signed 256-bit Elliptic Curve CA:

```
openssl ecparam -name prime256v1 -genkey -out ca_key.pem
openssl req -config openssl.cnf -new -key ca_key.pem -out ca.csr
openssl x509 -req -days 3650 -in ca.csr -signkey ca_key.pem -out ca_cert.pem
```

Creating Signed Certificates with an Existing CA

Generate a signed 2048-bit RSA certificate:

```
openssl genrsa -out cert_1_key.pem 2048
openssl req -new -key cert_1_key.pem -out cert_1.csr
openssl ca -config openssl.cnf -days 3650 -in cert_1.csr -out cert_1.pem
```

Generate a signed 256-bit Elliptic Curve certificate:

```
openssl ecparam -name prime256v1 -genkey -out cert_2_key.pem
openssl req -new -key cert_2_key.pem -out cert_2.csr
openssl ca -config openssl.cnf -days 3650 -in cert_2.csr -out cert_2.pem
```

Signing Documents with SMIME

Sign a document using existing CA & CA private key:

```
openssl smime -sign -in doc.xml -text -out doc_signed.p7s -signer ca_cert.pem
-inkey ca_private_key.pem
```

14.6   Domain Governance Document

The signed governance document is used by the DDS Security built-in access control plugin in order to determine both per-domain and per-topic security configuration options for specific domains. For full details regarding the content of the governance document, see the OMG DDS Security specification section 9.4.1.2.

14.6.1   Global Governance Model

It's worth noting that the DDS Security Model expects the governance document to be globally shared by all participants making use of the relevant domains described within the governance document. Even if this is not the case, the local participant will verify incoming authentication and access control requests as if the remote participant shared the same governance document and accept or reject the requests accordingly.

14.6.2   Key Governance Elements

<u>Domain Id Set</u>

A list of domain ids and/or domain id ranges of domains impacted by the current domain rule. The syntax is the same as the domain id set found in the governance document.

The set is made up of <id> tags or <id_range> tags. An <id> tag simply contains the domain id that are part of the set. An <id_range> tag can be used to add multiple ids at once. It must contain a <min> tag to say where the range starts and may also have a <max> tag to say where the range ends. If the <max> tag is omitted then the set includes all valid domain ids starting at <min>.

If the domain rule or permissions grant should to apply to all domains, use the following:

```
<domains>
  <id_range><min>0</min></id_range>
</domains>
```

If there's a need to be selective about what domains are chosen, here's an annotated example:

```
<domains>
  <id>2</id>
  <id_range><min>4</min><max>6</max></id_range> <!-- 4, 5, 6 -->
  <id_range><min>10</min></id_range> <!-- 10 and onward -->
</domains>
```

<u>Governance Configuration Types</u>

The following types and values are used in configuring both per-domain and per-topic security configuration options. We summarize them here to simplify discussion of the configuration options where they're used, found below.

Boolean

A boolean value indicating whether a configuration option is enabled or not. Recognized values are: TRUE/true/1 or FALSE/false/0.

ProtectionKind

The method used to protect domain data (message signatures or message encryption) along with the ability to include origin authentication for either protection kind. Currently, OpenDDS doesn't implement origin authentication. So while the "_WITH_ORIGIN_AUTHENTICATION" options are recognized, the underlying configuration is unsupported. Recognized values are: {NONE, SIGN, ENCRYPT, SIGN_WITH_ORIGIN_AUTHENTICATION, or ENCRYPT_WITH_ORIGIN_AUTHENTICATION}

BasicProtectionKind

The method used to protect domain data (message signatures or message encryption). Recognized values are: {NONE, SIGN, or ENCRYPT}

FnmatchExpression

A wildcard-capable string used to match topic names. Recognized values will conform to POSIX fnmatch() function as specified in POSIX 1003.2-1992, Section B.6.

14.6.3   Domain Rule Configuration Options

The following XML elements are used to configure domain participant behaviors.

Element	Type	Description
`<allow_unauthenticated_participants>`	Boolean	A boolean value which determines whether to allow unauthenticated participants for the current domain rule
`<enable_join_access_control>`	Boolean	A boolean value which determines whether to enforce domain access controls for authenticated participants
`<discovery_protection_kind>`	ProtectionKind	The discovery protection element specifies the protection kind used for the built-in DataWriter(s) and DataReader(s) used for secure endpoint discovery messages
`<liveliness_protection_kind>`	ProtectionKind	The liveliness protection element specifies the protection kind used for the built-in DataWriter and DataReader used for secure liveliness messages
`<rtps_protection_kind>`	ProtectionKind	Indicate the desired level of protection for the whole RTPS message. Very little RTPS data exists outside the "metadata protection" envelope (see topic rule configuration options), and so for most use cases topic-level "data protection" or "metadata protection" can be combined with discovery protection and/or liveliness protection in order to secure domain data adequately. One item that is not secured by "metadata protection" is the timestamp, since RTPS uses a separate InfoTimestamp submessage for this. The timestamp can be secured by using `<rtps_protection_kind>`

Table 14-4

14.6.4　Topic Rule Configuration Options

The following XML elements are used to configure topic endpoint behaviors:

`<topic_expression>` : **FnmatchExpression**

A wildcard-capable string used to match topic names. See description above. A "default" rule to catch all previously unmatched topics can be made with:

`<topic_expression>*</topic_expression>`

`<enable_discovery_protection>` : **Boolean**

Enables the use of secure discovery protections for matching user topic announcements.

`<enable_read_access_control>` : **Boolean**

Enables the use of access control protections for matching user topic DataReaders.

`<enable_write_access_control>` : **Boolean**

Enables the use of access control protections for matching user topic DataWriters.

`<metadata_protection_kind>` : **ProtectionKind**

Specifies the protection kind used for the RTPS SubMessages sent by any DataWriter and DataReader whose associated Topic name matches the rule's topic expression.

`<data_protection_kind>` : **BasicProtectionKind**

Specifies the basic protection kind used for the RTPS SerializedPayload SubMessage element sent by any DataWriter whose associated Topic name matches the rule's topic expression.

14.6.5   Governance XML Example

```xml
<?xml version="1.0" encoding="utf-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.omg.org/spec/DDS-
Security/20170801/omg_shared_ca_domain_governance.xsd">
  <domain_access_rules>
    <domain_rule>
      <domains>
        <id>0</id>
        <id_range>
          <min>10</min>
          <max>20</max>
        </id_range>
      </domains>
<allow_unauthenticated_participants>FALSE</allow_unauthenticated_participants>
        <enable_join_access_control>TRUE</enable_join_access_control>
```

```
<rtps_protection_kind>SIGN</rtps_protection_kind>
<discovery_protection_kind>ENCRYPT</discovery_protection_kind>
<liveliness_protection_kind>SIGN</liveliness_protection_kind>
<topic_access_rules>
  <topic_rule>
    <topic_expression>Square*</topic_expression>
    <enable_discovery_protection>TRUE</enable_discovery_protection>
    <enable_read_access_control>TRUE</enable_read_access_control>
    <enable_write_access_control>TRUE</enable_write_access_control>
    <metadata_protection_kind>ENCRYPT</metadata_protection_kind>
    <data_protection_kind>ENCRYPT</data_protection_kind>
  </topic_rule>
  <topic_rule>
    <topic_expression>Circle</topic_expression>
    <enable_discovery_protection>TRUE</enable_discovery_protection>
    <enable_read_access_control>FALSE</enable_read_access_control>
    <enable_write_access_control>TRUE</enable_write_access_control>
    <metadata_protection_kind>ENCRYPT</metadata_protection_kind>
    <data_protection_kind>ENCRYPT</data_protection_kind>
  </topic_rule>
  <topic_rule>
    <topic_expression>Triangle</topic_expression>
    <enable_discovery_protection>FALSE</enable_discovery_protection>
    <enable_read_access_control>FALSE</enable_read_access_control>
    <enable_write_access_control>TRUE</enable_write_access_control>
    <metadata_protection_kind>NONE</metadata_protection_kind>
    <data_protection_kind>NONE</data_protection_kind>
  </topic_rule>
  <topic_rule>
    <topic_expression>*</topic_expression>
    <enable_discovery_protection>TRUE</enable_discovery_protection>
    <enable_read_access_control>TRUE</enable_read_access_control>
    <enable_write_access_control>TRUE</enable_write_access_control>
    <metadata_protection_kind>ENCRYPT</metadata_protection_kind>
    <data_protection_kind>ENCRYPT</data_protection_kind>
  </topic_rule>
</topic_access_rules>
```

```
          </domain_rule>
      </domain_access_rules>
  </dds>
```

14.7   Participant Permissions Document

The signed permissions document is used by the DDS Security built-in access control plugin in order to determine participant permissions to join domains and to create endpoints for reading, writing, and relaying domain data. For full details regarding the content of the permissions document, see the OMG DDS Security specification section 9.4.1.3.

14.7.1   Key Permissions Elements

Grants

Each permissions file consists of one or more permissions grants. Each grant bestows access control privileges to a single subject name for a limited validity period.

Subject Name

Each grant's subject name is intended to match against a corresponding identity certificate's "subject" field. In order for permissions checks to successfully validate for both local and remote participants, the supplied identity certificate subject name must match the subject name of one of the grants included in the permissions file.

Validity

Each grant's validity section contains a start date and time (`<not_before>`) and an end date and time (`<not_after>`) to indicate the period of time during which the grant is valid.

The format of the date and time, which is like ISO-8601, must take one of the following forms:

1) `YYYY-MM-DDThh:mm:ss`

   ○ Example: `2020-10-26T22:45:30`

2) `YYYY-MM-DDThh:mm:ssZ`

   ○ Example: `2020-10-26T22:45:30Z`

3) `YYYY-MM-DDThh:mm:ss+hh:mm`

   ○ Example: `2020-10-26T23:45:30+01:00`

4) `YYYY-MM-DDThh:mm:ss-hh:mm`

○ Example: `2020-10-26T16:45:30-06:00`

All fields shown must include leading zeros to fill out their full width, as shown in the examples. YYYY-MM-DD is the date and hh:mm:ss is the time in 24-hour format. The date and time must be able to be represented by the time_t (C standard library) type of the system. The seconds field can also include a variable length fractional part, like 00.0 or 01.234, but it will be ignored because time_t represents a whole number of seconds. Examples #1 and #2 are both interpreted to be using UTC. To put the date and time in a local time, a time zone offset can to be added that says how far the local timezone is ahead of (using '+' as in example #3) or behind (using '-' as in example #4) UTC at that date and time.

Allow / Deny Rules

Grants will contain one or more allow / deny rules to indicate which privileges are being applied. When verifying that a particular operation is allowed by the supplied grant, rules are checked in the order they appear in the file. If the domain, partition, and (when implemented) data tags for an applicable topic rule match the operation being verified, the rule is applied (either allow or deny). Otherwise, the next rule is considered. Special Note: If a grant contains any allow rule that matches a given domain (even one with no publish / subscribe / relay rules), the grant may be used to join a domain with join access controls enabled.

Default Rule

The default rule is the rule applied if none of the grant's allow rules or deny rules match the incoming operation to be verified.

Domain Id Set

Every allow or deny rule must contain a set of domain ids to which it applies. The syntax is the same as the domain id set found in the governance document. See section 14.6.2 for details.

Publish / Subscribe / Relay Rules (PSR rules)

Every allow or deny rule may optionally contain a list of publish, subscribe, or relay rules bestowing privileges to publish, subscribe, or relay data (respectively). Each rule applies to a collection of topics in a set of partitions with a particular set of data tags. As such, each rule must then meet these three conditions (topics, partitions, and (when implemented) data tags) in order to apply to a given operation. These conditions are governed by their relevant subsection, but the exact meaning and default values will vary depending on the both the PSR type (publish, subscribe, relay) as well as whether this is an allow rule or a deny rule. Each condition is summarized below, but please refer to the OMG DDS Security

specification for full details. OpenDDS does not currently support relay-only behavior and consequently ignores allow and deny relay rules for both local and remote entities. Additionally, OpenDDS does not currently support data tags, and so the data tag condition applied is always the "default" behavior described below.

Topic List

The list of topics and/or topic expressions for which a rule applies. Topic names and expressions are matched using POSIX fnmatch() rules and syntax. If the triggering operation matches any of the topics listed, the topic condition is met. The topic section must always be present for a PSR rule, so there there is no default behavior.

Partition List

The partitions list contains the set of partition names for which the parent PSR rule applies. Similarly to topics, partition names and expressions are matched using POSIX fnmatch() rules and syntax. For "allow" PSR rules, the DDS entity of the associated triggering operation must be using a strict subset of the partitions listed for the rule to apply. When no partition list is given for an "allow" PSR rule, the "empty string" partition is used as the default value. For "deny" PSR rules, the rule will apply if the associated DDS entity is using any of the partitions listed. When no partition list is given for a "deny" PSR rule, the wildcard expression "*" is used as the default value.

Data Tags List

Data tags are an optional part of the DDS Security specification and are not currently implemented by OpenDDS. If they were implemented, the condition criteria for data tags would be similar to partitions. For "allow" PSR rules, the DDS entity of the associated triggering operation must be using a strict subset of the data tags listed for the rule to apply. When no data tag list is given for an "allow" PSR rule, the empty set of data tags is used as the default value. For "deny" PSR rules, the rule will apply if the associated DDS entity is using any of the data tags listed. When no data tag list is given for a "deny" PSR rule, the set of "all possible tags" is used as the default value.

14.7.2   Permissions XML Example

```
<?xml version="1.0" encoding="UTF-8"?>

<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.omg.org/spec/DDS-Security/20170801/
omg_shared_ca_permissions.xsd">

  <permissions>

    <grant name="ShapesPermission">
```

```
            <subject_name>emailAddress=cto@acme.com, CN=DDS Shapes Demo, OU=CTO
Office, O=ACME Inc., L=Sunnyvale, ST=CA, C=US</subject_name>
        <validity>
          <not_before>2015-10-26T00:00:00</not_before>
          <not_after>2020-10-26T22:45:30</not_after>
        </validity>
        <allow_rule>
          <domains>
            <id>0</id>
          </domains>
        </allow_rule>
        <deny_rule>
          <domains>
            <id>0</id>
          </domains>
          <publish>
            <topics>
              <topic>Circle1</topic>
            </topics>
          </publish>
          <publish>
            <topics>
              <topic>Square</topic>
            </topics>
            <partitions>
              <partition>A_partition</partition>
            </partitions>
          </publish>
          <subscribe>
            <topics>
              <topic>Square1</topic>
            </topics>
          </subscribe>
          <subscribe>
            <topics>
              <topic>Tr*</topic>
            </topics>
```

```
                    <partitions>
                      <partition>P1*</partition>
                    </partitions>
                 </subscribe>
              </deny_rule>
              <default>DENY</default>
           </grant>
        </permissions>
     </dds>
```

14.8   DDS Security Implementation Status

The following DDS Security features are not implemented in OpenDDS.

1. Optional parts of the DDS Security v1.1 specification

    A) Ability to write a custom plugin in C or in Java (C++ is supported)

    B) Logging Plugin support

    C) Built-in Logging Plugin

    D) Data Tagging

2. Use of RTPS KeyHash for encrypted messages

    A) OpenDDS doesn't use KeyHash, so it meets the spec requirements of not leaking secured data through KeyHash

3. Immutability of Publisher's Partition QoS (see OMG Issue DDSSEC12-49)

4. Use of multiple plugin configurations (with different Domain Participants)

5. CRL (RFC 5280) and OCSP (RFC 2560) support

6. Certain plugin operations not used by built-in plugins may not be invoked by middleware

7. Origin Authentication

8. PKCS#11 for certificates, keys, passwords

9. Relay as a permissions "action" (Publish and Subscribe are supported)

10. Legacy matching behavior of permissions based on Partition QoS (9.4.1.3.2.3.1.4 in spec)

11. 128-bit AES keys (256-bit is supported)

12. Configuration of Built-In Crypto's key reuse (within the DataWriter) and blocks-per-session

13. Signing (without encrypting) at the payload level, see OMG Issue DDSSEC12-59

CHAPTER 15

# Internet-Enabled RTPS

15.1   Overview

Like any specification, standard, or system, RTPS was designed with certain assumptions. Two of these assumptions severely limit the ability to use RTPS in modern network environments. First, RTPS, or more specifically, SPDP uses multicast for discovery. Multicast is not supported on the public Internet which precludes the use of RTPS for Internet of Things (IoT) applications and Industrial Internet of Things (IIoT) applications. Second, SPDP and SEDP advertise locators (IP and port pairs) for endpoints (DDS readers and writer). If the participant is behind a firewall that performs network address translation, then the locators advertised by the participant are useless to participants on the public side of the firewall.

This chapter describes different tools and techniques for getting around these limitations. First, we introduce the *RtpsRelay* as a service for forwarding RTPS messages according to application-defined groups. The RtpsRelay can be used to connect participants that are deployed in environments that don't support multicast and whose packets are subject to NAT. Second, we introduce Interactive Connection Establishment (ICE) for RTPS. Adding ICE to RTPS is an optimization that allows participants that are behind firewalls that perform NAT to exchange messages directly. ICE requires a back channel for distributing discovery information and is typically used with the RtpsRelay.

15.2   **The RtpsRelay**

The RtpsRelay is designed to allow participants to exchange RTPS datagrams when separated by a firewall that performs network address translation (NAT) and/or a network that does not support multicast like the public Internet. The RtpsRelay supports both IPv4 and IPv6. A participant that uses an RtpsRelay Instance is a *client* of that instance. Each RtpsRelay instance contains two participants: the *Application Participant* and the *Relay Participant*. The Application Participant runs in the domain of the application. The RtpsRelay reads the built-in topics to discover Participants, DataReaders, and DataWriters. It then shares this information with other RtpsRelay instances using the Relay Participant. Each RtpsRelay instance maintains a map of associated readers and writers. When a client sends an RTPS datagram to its RtpsRelay instance, the RtpsRelay instance uses the association table to forward the datagram to other clients and other RtpsRelay instances so that they can deliver it to their clients. Clients send RTPS datagrams via unicast which is generally supported and compatible with NAT. The RtpsRelay can be used in lieu of or in addition to conventional RTPS discovery.
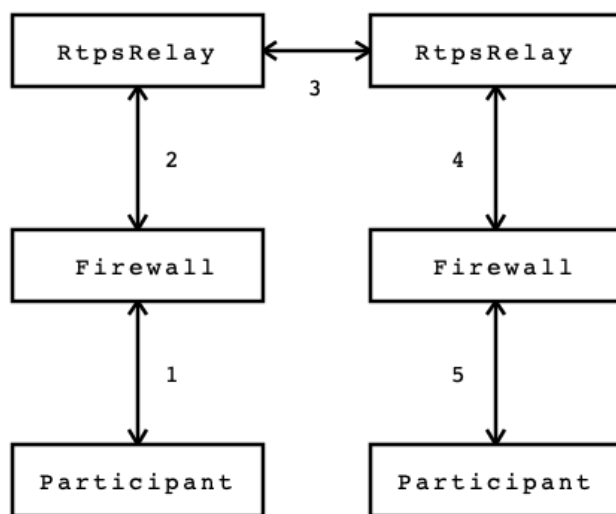
The preceding diagram illustrates how the RtpsRelay can be used to connect participants that are behind firewalls that may be performing NAT. First, a Participant sends an RTPS datagram to its associated RtpsRelay (1). This datagram is intercepted by the firewall, the source address and port are replaced by the external IP address and port of the firewall, and then the datagram is sent to the RtpsRelay (2). The relationship between the source address and external IP address and port selected by the firewall is called a NAT binding. The RtpsRelay instance forwards the datagram to other RtpsRelay instances (3). The RtpsRelays then forward the datagram to all of the destination participants (4). Firewalls

on the path to the participants intercept the packet and replace the destination address (which is the external IP and port of the firewall) with the address of the Participant according to a previously created NAT binding (5).

The RTPS implementation in OpenDDS uses a port for SPDP, a port for SEDP, and a port for conventional RTPS messages. The relay mirrors this idea and exposes three ports to handle each type of traffic.

To keep NAT bindings alive, clients send STUN binding requests and indications periodically to the RtspRelay ports. Participants using ICE may use these ports as a STUN server for determining a server reflexive address. The timing parameters for the periodic messages are controlled via the ICE configuration variables for server reflexive addresses.

15.2.1 Using The RtpsRelay

Support for the RtpsRelay is activated via configuration. See Table 7-5 RTPS Discovery Configuration Options and Table 7-17 RTPS_UDP Configuration Options. As an example:

```
[common]
DCPSGlobalTransportConfig=$file

[domain/4]
DiscoveryConfig=rtps

[rtps_discovery/rtps]
SpdpRtpsRelayAddress=1.2.3.4:4444
SedpRtpsRelayAddress=1.2.3.4:4445
UseRtpsRelay=1

[transport/the_rtps_transport]
transport_type=rtps_udp
DataRtpsRelayAddress=1.2.3.4:4446
UseRtpsRelay=1
```

Each participant should use a single RtpsRelay instance due to the way NAT bindings work. Most firewalls will only forward packets received from the destination address that was originally used to create the NAT binding. That is, if participant A is interacting with relay A and participant B is interacting with relay B, then a message from A to B must go from A to relay A, to relay B, and finally to B. Relay A cannot send directly to B since that packet will not be accepted by the firewall.

15.2.2   **Usage**

The RtpsRelay itself is an OpenDDS application. The source code is located in `tools/rtpsrelay`. Each RtpsRelay process has a set of ports for exchanging RTPS messages with the participants called the "vertical" ports and a set of ports for exchanging RTPS messages with other relays called the "horizontal" ports.

- `-VerticalAddress ADDRESS`
  Determines the base network address used for receiving RTPS messages from the participants. By default, the relay listens on 0.0.0.0:4444 for SPDP messages, 0.0.0.0:4445 for SEDP messages, and 0.0.0.0.4446 for data messages.

- `-HorizontalAddress ADDRESS`
  Determines the base network address used for receiving RTPS message from other relays. By default, the relay listens on the first IP network and uses port 11444 for SPDP messages, 11445 for SEDP messages, and 11446 for data messages.

- `-RelayDomain DOMAIN`
  Sets the DDS domain used by the Relay Participant. The default is 0.

- `-ApplicationDomain DOMAIN`
  Sets the DDS domain used by the Application Participant. The default is 1.

- `-Lifespan SECONDS`
  RtpsRelay will only forward a datagram to a client if it has received a datagram from the client in this amount of time. The default is 60 seconds.

- `-UserData STRING`
  Set the contents of the Application Participant's UserData QoS policy to the provided string.

- `-StaticLimit INTEGER`
  Limit the number of concurrent active clients served by this relay instance. Defaults to 0 (unlimited).

- `-MaxPending INTEGER`
  Prevent new clients from connecting when there are a given number of pending clients. Defaults to 0 (unlimited). Pending clients are those that have sent data to one of the relay addresses but have not yet completed discovery with the Application Participant.

- `-IdentityCA PATH`
  `-PermissionsCA PATH`
  `-IdentityCertificate PATH`
  `-IdentityKey PATH`

```
-Governance PATH
-Permissions PATH
```
Provide paths to the DDS Security documents. Requires a security-enabled build.

- ```
  -LogWarnings 0|1
  -LogEntries 0|1
  -LogDiscovery 0|1
  -LogActivity 0|1
  ```
  Enable/disable logging of the various event types.

- ```
  -LogRelayStatistics SECONDS
  -LogHandlerStatistics SECONDS
  -LogParticipantStatistics SECONDS
  -LogDomainStatistics SECONDS
  ```
  Write statistics for the various event types to the log at the given interval, defaults to 0 (disabled).

- ```
  -PublishRelayStatistics SECONDS
  -PublishHandlerStatistics SECONDS
  -PublishParticipantStatistics SECONDS
  -PublishDomainStatistics SECONDS
  ```
  Configure the relay to publish usage statistics on DDS topics at the given interval, defaults to 0 (disabled).

15.2.3 Deployment Considerations

Running an RtpsRelay relay cluster with RTPS in the cloud leads to a bootstrapping problem since multicast is not supported in the cloud. One option is to not use RTPS for discovery. Another option is to run a single well-known relay that allows the other relays to discover each other. A third option is to use a program translates multicast to unicast.

RTPS uses UDP which typically cannot be load balanced effectively due to the way NAT bindings work. Consequently, each RtpsRelay server must have a public IP address. Load balancing can be achieved by having the participants choose a relay according to a load balancing policy. To illustrate, each relay could also run an HTTP server which does nothing but serve the public IP address of the relay. These simple web servers would be exposed via a centralized load balancer. A participant, then, could access the HTTP load balancer to select a relay.

15.3 Interactive Connectivity Establishment (ICE) for RTPS

Interactive Connectivity Establishment (ICE) is protocol for establishing connectivity between a pair of hosts that are separated by at least one firewall that performs network address translation. ICE can be thought of as an optimization for situations that require an RtpsRelay. The success of ICE depends on the firewall(s) that separate the hosts.

The ICE protocol has three steps. First, a host determines its public IP address by sending a STUN binding request to a public STUN server. The STUN server sends a binding success response that contains the source address of the request. If the host has a public IP address, then the address returned by STUN will match the IP address of the host. Otherwise, the address will be the public address of the outermost firewall. Second, the hosts generate and exchange candidates (which includes the public IP address determined in the first step) using a side channel. A candidate is an IP and port that responds to STUN messages and sends datagrams. Third, the hosts send STUN binding requests to the candidates in an attempt to generate the necessary NAT bindings and establish connectivity.

For OpenDDS, ICE can be used to potentially establish connectivity between SPDP endpoints, SEDP endpoints, and ordinary RTPS endpoints. SPDP is used as the side channel for SEDP and SEDP is used as the side channel for the ordinary RTPS endpoints. To this, we added two parameters to the RTPS protocol for sending general ICE information and ICE candidates and added the ability to execute the ICE protocol and process STUN messages to the RTPS transports.

ICE is defined in IETF RFC 8445. ICE utilizes the STUN protocol that is defined in IETF RFC 5389. The ICE implementation in OpenDDS does not use TURN servers.

ICE is enabled through configuration. The minimum configuration involves setting the UseIce flag and providing addresses for the STUN servers. See Table 7-5 RTPS Discovery Configuration Options and Table 7-17 RTPS_UDP Configuration Options for details.

```
[common]
DCPSGlobalTransportConfig=$file
DCPSDefaultDiscovery=DEFAULT_RTPS

[transport/the_rtps_transport]
transport_type=rtps_udp
DataRtpsRelayAddress=5.6.7.8:4446
UseIce=1
```

```
DataStunServerAddress=1.2.3.4:3478

[domain/42]
DiscoveryConfig=DiscoveryConfig1

[rtps_discovery/DiscoveryConfig1]
SpdpRtpsRelayAddress=5.6.7.8:4444
SedpRtpsRelayAddress=5.6.7.8:4445
UseIce=1
SedpStunServerAddress=1.2.3.4:3478
```

OpenDDS Developer's Guide

CHAPTER 16

XTypes

16.1 Overview

The DDS specification defines a way to build distributed applications using a data-centric publish and subscribe model. In this model, publishing and subscribing applications communicate via Topics and each Topic has a data type. An assumption built into this model is that all applications agree on data type definitions for each Topic that they use. This assumption is not practical as systems must be able to evolve while remaining compatible and interoperable.

The DDS XTypes (Extensible and Dynamic Topic Types) specification loosens the requirement on applications to have a common notion of data types. Using XTypes, the application developer adds IDL annotations that indicate where the types may vary between publisher and subscriber and how those variations are handled by the middleware.

This release of OpenDDS implements the XTypes specification version 1.3 at the Basic Conformance level, except for the Dynamic Types API. Some features described by the specification are not yet implemented in OpenDDS — those are noted below in section 16.7. This includes IDL annotations that are not yet implemented. The "Specification Differences" section (16.8) describes situations where the implementation of XTypes in OpenDDS departs from or infers something about the specification. Specification issues have been raised for these situations.

16.2   Features

16.2.1   Extensibility

There are 3 kinds of extensibility for types: Appendable, Mutable, and Final

- *Appendable* denotes a constructed type which may have additional members added onto or removed from the end.

- *Mutable* denotes a constructed type that allows for members to be added, removed, and reordered so long as the keys and the required members of the sender and receiver remain. Mutable extensibility is accomplished by assigning a stable identifier to each member.

- *Final* denotes a constructed type that can not add, remove, or reorder members,. This can be considered a non-extensible constructed type, with behavior similar to that of a type created before XTypes.

Extensibility is set by the user in the IDL with the annotations: @appendable, @mutable, @final

The default extensibility is Appendable. This default extensibility can be changed with the IDL compiler command line option --default-extensibility EXTENSIBILITY Where EXTENSIBILITY is "final", "appendable" or "mutable".

Structs and unions are the only types which can use any of the extensibilities.

16.2.2   Assignability

Assignability describes the ability of values of one type to be coerced to values of a possibility different type.

Assignability between the type of a writer and reader is checked as part of discovery. If the types are assignable but not identical, then the "Try Construct" mechanism will be used to coerce values of the writer's type to values of the reader's type.

In order for two constructed types to be assignable they must

- Have the same extensibility.

- Have the same set of keys.

Each member of a constructed type has an identifier. This identifier may be assigned automatically or explicitly.

Union assignability depends on two dimensions. First, unions are only assignable if their discriminators are assignable. Second, for any branch label or default that exists in both unions, the members selected by that branch label must be assignable.

16.2.3 Interoperability With Non-XTypes Implementations

Communication with a non-XTypes DDS (either an older OpenDDS or another DDS implementation which has RTPS but not XTypes 1.2+) requires compatible IDL types and the use of RTPS Discovery. Compatible IDL types means that the types are structurally equivalent and serialize to the same bytes using XCDR version 1.

Additionally, the XTypes-enabled participant needs to be set up as follows:

- Types cannot use Mutable extensibility

- Data Writers must have their Data Representation QoS policy set to DDS::XCDR_DATA_REPRESENTATION

- Data Readers must include DDS::XCDR_DATA_REPRESENTATION in the list of data representations in their Data Representation QoS (This is true by default)

The "Data Representation" section below shows how to change the data representation.

16.3 Examples and Explanation

Suppose you are in charge of deploying a set of weather stations that publish temperature, pressure, and humidity. The following examples show how various features of XTypes may be applied to address changes in the schema published by the weather station. Specifically, without XTypes, one would either need to create a new type with its own DataWriters/DataReaders or update all applications simultaneously. With proper planning and XTypes, one can simply modify the existing type (within limits) and writers and readers using earlier versions of the topic type will remain compatible with each other and be compatible with writers and readers using new versions of the topic type.

16.3.1 Mutable Extensibility

The type published by the weather stations can be made extensible with the @mutable annotation:

```
// Version 1
@topic
@mutable
struct StationData {
```

```
  short temperature;
  double pressure;
  double humidity;
};
```

Suppose that some time in the future, a subset of the weather stations are upgraded to monitor wind speed and direction:

```
enum WindDir {N, NE, NW, S, SE, SW, W, E};
// Version 2
@topic
@mutable
struct StationData {
  short temperature;
  double pressure;
  double humidity;
  short wind_speed;
  WindDir wind_direction;
};
```

When a Version 2 writer interacts with a Version 1 reader, the additional fields will be ignored by the reader. When a Version 1 writer interacts with a Version 2 reader, the additional fields will be initialized to a "logical zero" value for its type (empty string, FALSE boolean) - see Table 9 of the XTypes specification for details.

16.3.2   Assignability

The first and second versions of the StationData type are *assignable* meaning that it is possible to construct a version 2 value from a version 1 value and vice-versa. The assignability of non-constructed types (e.g., integers, enums, strings) is based on the types being identical or identical up to parameterization, i.e., bounds of strings and sequences may differ. The assignability of constructed types like structs and unions is based on finding corresponding members with assignable types. Corresponding members are those that have the same id.

A type marked as @mutable allows for members to be added, removed, or reordered so long as member ids are preserved through all of the mutations.

16.3.3   Member IDs

Member ids are assigned using various annotations. A policy for a type can be set with either @autoid(SEQUENTIAL) or @autoid(HASH):

```
// Version 3
@topic
@mutable
```

```
@autoid(SEQUENTIAL)
struct StationData {
  short temperature;
  double pressure;
  double humidity;
};


// Version 4
@topic
@mutable
@autoid(HASH)
struct StationData {
  short temperature;
  double pressure;
  double humidity;
};
```

SEQUENTIAL causes ids to be assigned based on the position in the type. HASH causes ids to be computed by hashing the name of the member. If no @autoid annotation is specified, the policy is SEQUENTIAL.

Suppose that Version 3 was used in the initial deployment of the weather stations and the decision was made to switch to @autoid(HASH) when adding the new fields for wind speed and direction. In this case, the ids of the pre-existing members can be set with @id:

```
enum WindDir {N, NE, NW, S, SE, SW, W, E};

// Version 5
@topic
@mutable
@autoid(HASH)
struct StationData {
  @id(0) short temperature;
  @id(1) double pressure;
  @id(2) double humidity;
  short wind_speed;
  WindDir wind_direction;
};
```

See the "Member ID Annotations" section for more details.

16.3.4   Appendable Extensibility

Mutable extensibility requires a certain amount of overhead both in terms of processing and network traffic. A more efficient but less flexible form of extensibility is @appendable. Extensibility with @appendable is limited in that members can only be added to or removed from the end of the type. With @appendable, the initial version of the weather station IDL would be:

```
// Version 6
@topic
```

```
@appendable
struct StationData {
  short temperature;
  double pressure;
  double humidity;
};
```

And the subsequent addition of the wind speed and direction members would be:

```
enum WindDir {N, NE, NW, S, SE, SW, W, E};

// Version 7
@topic
@appendable
struct StationData {
  short temperature;
  double pressure;
  double humidity;
  short wind_speed;
  WindDir wind_direction;
};
```

As with @mutable, when a Version 7 Writer interacts with a Version 6 Reader, the additional fields will be ignored by the reader. When a Version 6 Writer interacts with a Version 7 Reader, the additional fields will be initialized to default values based on Table 9 of the XTypes specification.

Appendable is the default extensibility.

16.3.5   Final Extensibility

The third kind of extensibility is @final. Annotating a type with @final means that it will not be compatible with (assignable to/from) a type that's structurally different. The @final annotation can be used to define types for pre-XTypes compatibility or in situations where the overhead of @mutable or @appendable is unacceptable.

16.3.6   Try Construct

From a reader's perspective, there are three possible scenarios when attempting to initialize a member. First, the member type is identical to the member type of the reader. This is the trivial case the value from the writer is copied to the value for the reader. Second, the writer does not have the member. In this case, the value for the reader is initialized to a default value based on Table 9 of the XTypes specification (this is the "logical zero" value for the type). Third, the type offered by the writer is assignable but not identical to the type required by the reader. In this case, the reader must try to construct its value from the corresponding value provided by the writer.

Suppose that the weather stations also publish a topic containing station information:

```
typedef string<8> StationID;
typedef string<256> StationName;

// Version 1
@topic
@mutable
struct StationInfo {
  @try_construct(TRIM) StationID station_id;
  StationName station_name;
};
```

Eventually, the pool of station IDs is exhausted so the IDL must be refined as follows:

```
typedef string<16> StationID;
typedef string<256> StationName;

// Version 2
@topic
@mutable
struct StationInfo {
  @try_construct(TRIM) StationID station_id;
  StationName station_name;
};
```

If a Version 2 writer interacts with a Version 1 reader, the station ID will be truncated to 8 characters. While perhaps not ideal, it will still allow the systems to interoperate.

There are two other forms of try-construct behavior. Fields marked as @try_construct(USE_DEFAULT) will receive a default value if value construction fails. In the previous example, this means the reader would receive an empty string for the station ID if it exceeds 8 characters. Fields marked as @try_construct(DISCARD) cause the entire sample to be discarded. In the previous example, the Version 1 reader will never see a sample from a Version 2 writer where the original station ID contains more than 8 characters. @try_construct(DISCARD) is the default behavior.

16.4   Data Representation

Data representation is the way a data sample can be encoded for transmission. Data representation can be XML, XCDR1, or XCDR2.

- XML is unsupported and should not be used

- XCDR1 with appendable extensibility should not be used

- XCDR2 is completely supported and preferred

XCDR2 is a more robust version of XCDR1 and should be used in preference to XCDR1 unless there is a reason to do otherwise.

Data representation is a QoS policy alongside the other QoS options. Its listed values represent allowed serialized forms of the data sample. The DataWriter and DataReader need to have at least one matching data representation for communication between them to be possible.

The default value of the DataRepresentationQoS policy is an empty sequence. This is interpreted by the middleware as XCDR2 for DataWriters and the alternatives XCDR1 | XCDR2 for DataReaders. A writer or reader without an explicitly-set DataRepresentationQoS will therefore be able to communicate with another reader or writer which is compatible with XCDR2. The example below shows a possible configuration for an XCDR1 DataWriter.

```
DDS::DataWriterQos qos;
pub->get_default_datawriter_qos(qos);
qos.representation.value.length(1);
qos.representation.value[0] = DDS::XCDR_DATA_REPRESENTATION;
DDS::DataWriter_var dw = pub->create_datawriter(topic, qos, 0, 0);
```

Note that the IDL constant used for XCDR1 is `XCDR_DATA_REPRESENTATION` (without the digit).

In addition to a DataWriter/DataReader QoS setting for data representation, each type defined in IDL can have its own data representation specified via an annotation. This value restricts which data representations can be used for that type. A DataWriter/DataReader must have at least one data representation in common with the type it uses.

The default value for an unspecified data representation annotation is to allow all forms of serialization.

The type's set of allowed data representations can be specified by the user in IDL with the notation: `@OpenDDS::data_representation(XCDR2)` where XCDR2 is replaced with the specific data representation.

16.5   Type Consistency Enforcement

The Type Consistency Enforcement QoS policy lets the application fine-tune details of how types may differ between writers and readers. The policy is only applies to data readers. This means that each reader can set its own policy for how its type may vary from the types of the writers that it may match.

There are six members of the `TypeConsistencyEnforcementQosPolicy` struct defined by XTypes, but OpenDDS only supports setting one of them: `ignore_member_names`. All other members should be kept at their default values.

`ignore_member_names` defaults to `FALSE` so member names (along with Member IDs, see 16.6.5) are significant for type compatibility. Changing this to TRUE means that only Member IDs are used for type compatibility.

16.6 IDL Annotations

16.6.1 Indicating Which Types Can Be Topic Types

16.6.1.1 @topic

Applies To: struct or union type declarations

The topic annotation marks a topic type for samples to be transmitted from a publisher or received by a subscriber. A topic type may contain other topic and non-topic types. See section 2.1.1 for more details.

16.6.1.2 @nested

Applies To: struct or union type declarations

The @nested annotation marks a type that will always be contained within another. This can be used to prevent a type from being used as a topic. One reason to do so is to reduce the amount of code generated for that type.

16.6.1.3 @default_nested

Applies To: modules

The @default_nested(TRUE) or @default_nested(FALSE) sets the default nesting behavior for a module. Types within a module marked with @default_nested(FALSE) can still set their own behavior with @nested.

16.6.2 Specifying Allowed Data Representations

Data Representation annotations mark the formats in which data samples of this type can be represented in a serialized form. The Data Representation annotations listed on the type will be compared to those in the QoS policies of the reader or writer that is trying to use the type. If a data representation is shared between the type and entity, then they can be used together. OpenDDS's default data representation for entities is XCDR2. If no data representation is specified for a type, there are no restrictions on which data representations that a QoS can use with the type.

16.6.2.1   @OpenDDS::data_representation(XML)

Applies To: topic types

Limitations: XML is not currently supported

16.6.2.2   @OpenDDS::data_representation(XCDR1)

Applies To: topic types

Limitations: XCDR1 is not recommended. See section 16.4 for details

16.6.2.3   @OpenDDS::data_representation(XCDR2)

Applies To: topic types

XCDR2 is currently the recommended data representation.

16.6.3   Determining Extensibility

The extensibility annotations determine how a type may be changed and still be compatible. If no extensibility annotation is set, the type will default to appendable. The default can be changed with the command line option --default-extensibility *type*, where *type* can be final, appendable, or mutable.

16.6.3.1   @mutable

Alias: @extensibility(MUTABLE)

Applies To: type declarations

This annotation indicates a type may have non-key or non-must-understand members removed. It may also have additional members added.

16.6.3.2   @appendable

Alias: @extensibility(APPENDABLE)

Applies To: type declarations

This annotation indicates a type may have additional members added or members at the end of the type removed.

Limitations: Appendable is not currently supported when XCDR1 is used as the data representation.

            OpenDDS Developer's Guide

16.6.3.3 @final

Alias: @extensibility(FINAL)

Applies To: type declarations

This annotation marks a type that cannot be changed and still be compatible. Final is most similar to pre-XTypes.

16.6.4 Customizing XTypes Per-member

Try Construct annotations dictate how members of one object should be converted from members of a different but assignable object. If no try construct annotation is added, it will default to discard.

16.6.4.1 @try_construct(USE_DEFAULT)

Applies to: structure and union members, sequence and array elements

The use_default try construct annotation will set the member whose deserialization failed to a default value which is determined by the XTypes specification. Sequences will be of length 0, with the same type as the original sequence. Primitives will be set equal to 0. Strings will be replaced with the empty string. Arrays will be of the same length but have each element set to the default value. Enums will be set to the first enumerator defined.

16.6.4.2 @try_construct(TRIM)

Applies to: structure and union members, sequence and array elements

The trim try construct annotation will, if possible, shorten a received value to one fitting the receiver's bound. As such, trim only makes logical sense on bounded strings and bounded sequences.

16.6.4.3 @try_construct(DISCARD)

Applies to: structure and union members, sequence and array elements

The discard try construct annotation will "throw away" the sample if an element fails to deserialize.

16.6.5 Member ID Assignment

If no explicit id annotation is used, then Member IDs will automatically be assigned sequentially.

16.6.5.1 @id(value)

Applies to: structure and union members

The *value* is a 32-bit integer which assigns that member's ID.

16.6.5.2 @autoid(value)

Applies to: module declarations, structure declarations, union declarations

The autoid annotation can take two *value*s, HASH or SEQUENTIAL. SEQUENTIAL states that the identifier shall be computed by incrementing the preceding one. HASH states that the identifier should be calculated with a hashing algorithm – the input to this hash is the member's name. HASH is the default value of autoid.

16.6.5.3 @hashid(value)

Applies to: structure and union members

The @hashid sets the identifier to the hash of the *value* parameter, if one is specified. If the *value* parameter is omitted or is the empty string, the member's name is used as if it was the *value*.

16.6.6 Determining The Key Fields Of A Type

16.6.6.1 @key

Applies to: structure members, union discriminator

The @key annotation marks a member used to determine the Instances of a topic type. See section 2.1.1.2 for more details on the general concept of a Key. For XTypes specifically, two types can only be compatible if each contains the members that are keys within the other.

16.7 Unsupported Features

OpenDDS implements the XTypes specification version 1.3 at the Basic Conformance level, except for the Dynamic Types API and the specific features listed below. The two optional profiles, XTypes 1.1 Interoperability (XCDR1) and XML, are not implemented.

16.7.1 Type System

- IDL map type

- IDL bitmask type

- Struct and union inheritance

16.7.2   Annotations

IDL4 defines many standardized annotations and XTypes uses some of them. The Annotations recognized by XTypes are in Table 21 in XTypes 1.3. Of those listed in that table, the following are not supported in OpenDDS. They are listed in groups defined by the rows of that table. Some annotations in that table, and not listed here, can only be used with new capabilities of the Type System (see 16.7.1).

- Struct members

  - @optional

  - @must_understand

  - @non_serialized

- Struct or union members

  - @external

- Enums

  - @bit_bound

  - @default_literal

  - @value

- @verbatim

16.8   Differences from the specification

Spec issues tracked in OMG's Jira database can be viewed at
https://issues.omg.org/issues/lists/dds-xtypes-rtf

- Inconsistent topic status isn't set for reader/reader or writer/writer in non-XTypes use cases

- DDSXTY14-29: Define the encoding and extensibility used by Type Lookup Service

- DDSXTY14-33: Enums must have the same "bit bound" to be assignable

- DDSXTY14-27: Default data representation is XCDR2

- DDSSEC12-86: Type Lookup Service when using DDS Security

- DDSXTY14-35: Anonymous types in Strongly Connected Components

- DDSXTY14-40: Meaning of ignore_member_names in TypeConsistencyEnforcement