CHAPTER 31

OpenDDS Developer’s Guide

31.1 Introduction

OpenDDS is an open source implementation of the OMG Data Distribution Service (DDS) for Real-Time Systems specification (OMG Document formal/07-01-01). OpenDDS is sponsored by Object Computing, Inc. (OCI) and is available via `http://www.opendds.org/`. This documentation is based on the version 1.2 release of OpenDDS.

DDS defines a service for efficiently distributing application data between participants in a distributed application. This service is not specific to CORBA. The specification provides a platform independent model (PIM) as well as a platform specific model (PSM) that maps the PIM onto a CORBA IDL implementation. The service is divided into two levels of interfaces: the Data-Centric Publish-Subscribe (DCPS) layer and an optional Data Local Reconstruction Layer (DLRL). The DCPS layer transports data from publishers to subscribers according to Quality of Service constraints associated with the data topic, publisher, and subscriber. The DLRL allows distributed data to be shared by local objects located remotely from each other as if the data were local. The DLRL is built on top of the DCPS layer.

The DCPS layer provides another publish-subscribe API for applications that is conceptually similar to the OMG Event and Notification Services as well as the TAO Real-Time Event Service. The main difference with DCPS is that it only specifies CORBA IDL interfaces for the set up, control, and configuration of the application and assumes that the data transmission occurs via mechanisms other than CORBA. This enables DDS implementations to achieve higher performance and better quality of service than the CORBA-based alternatives mentioned above.

For additional details about DDS, developers should refer to the DDS specification (OMG Document formal/07-01-01) 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 by OCI. It is available for download from `http://www.opendds.org/downloads.html` and is compatible with recent patch levels of TAO version 1.4a, 1.5a, and 1.6.x.

Note *OpenDDS currently implements a subset of the DCPS layer and is mostly compliant with the OMG DDS version 1.0 specification. None of the DLRL functionality is currently implemented. See the compliance information in 31.3.1 or at* `http://www.opendds.org/` *for more information.*

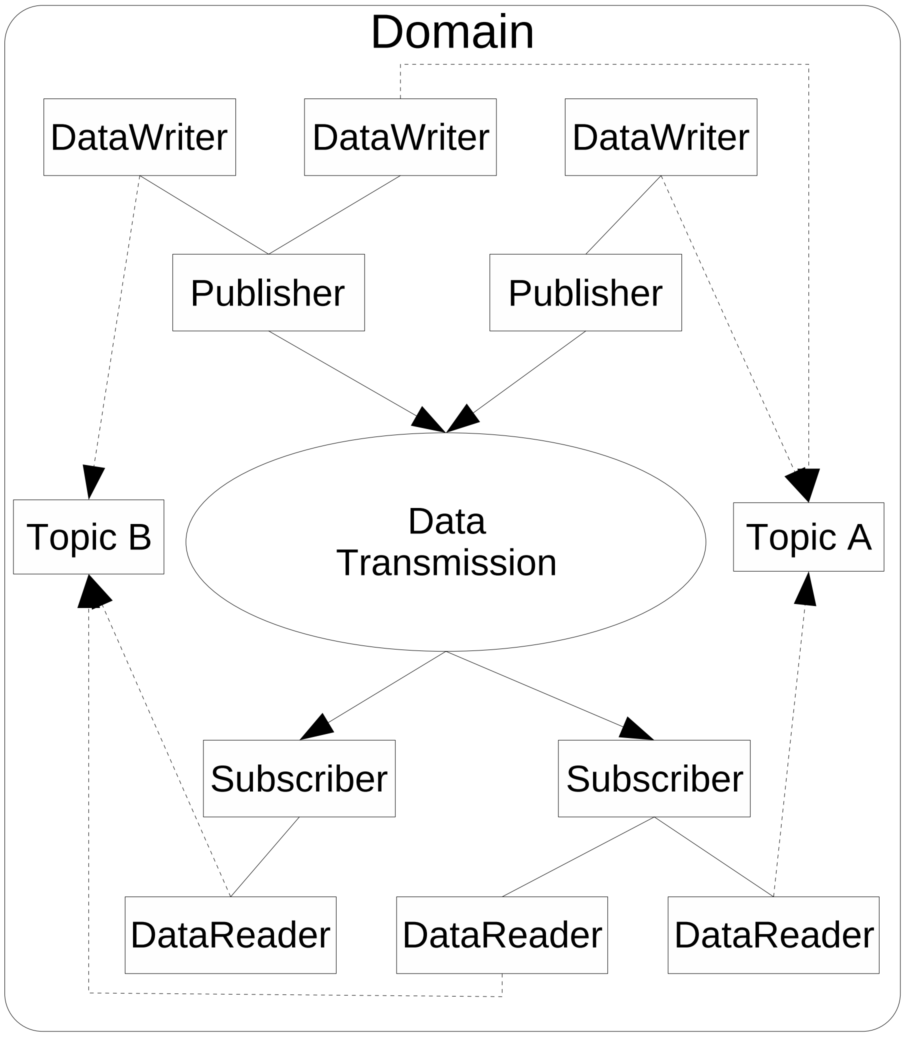
31.2 DCPS Overview

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

31.2.1 Basic Concepts

Figure 31-1 shows an overview of the DDS DCPS layer. The following subsections define the concepts shown in this diagram.

Figure 31-1 DCPS Conceptual Overview



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

31.2.1.2 Domain Participant

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.

31.2.1.3 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.

31.2.1.4 Data Writer

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.

31.2.1.5 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.

31.2.1.6 Subscriber

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

31.2.1.7 Data Reader

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.

31.2.2 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:

Topic Name	**Description**
DCPSParticipant	Each instance represents a domain participant.
DCPSTopic	Each topic is an instance.
DCPSPublication	Each instance represents a data writer
DCPSSubscription	Each instance represents a data reader.

Figure 31-2 Built-In Topics

31.2.3 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 collaborate to specify QoS through an offer-request paradigm. Publishers *offer* a set of QoS policies to all subscribers. Subscribers *request* a set of policies that they require. The DDS implementation then attempts to match the requested policies with the offered policies. If the policies are consistent the subscription is initiated. If the policies are not consistent then the subscription attempt fails.

The QoS policies currently implemented by OpenDDS are discussed in detail in 31.7.

31.2.4 Listeners

The DPCS 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.

31.2.5 Conditions

Note *OpenDDS currently supports only Status and Guard Conditions. Read and Query Conditions are not currently supported.*

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 Status Condition, and attaches it to a Wait Set.
- The application waits on the Wait Set until one or more Conditions become true.
- The application calls operations on the corresponding entity objects to extract the necessary information.

31.3 OpenDDS Implementation

31.3.1 Compliance

Appendix A 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

This section describes OpenDDS's compliance with these profiles in terms of the entities and quality of service policies defined by the DDS specification.

31.3.1.1 Entity Compliance

The DDS specification defines five modules that make up the DCPS PIM:

1. Infrastructure Module
2. Domain Module
3. Topic-Definition Module
4. Publication Module
5. Subscription Module

Various entities are defined within each module. Not all entities pertain to every profile listed in 31.3.1. Table 31-1 through Table 31-5 show which entities are included in each module and to which profiles each entity pertains, as well as whether or not the entity is implemented by OpenDDS.

Table 31-1 Infrastructure Module Entities

Entity Name	Profiles	Impl?
`Entity`	All	Yes
`DomainEntity`	All	Yes
`QosPolicy`	All	Yes
`Listener`	All	Yes
`Status`	All	Yes
`WaitSet`	All	Yes

Table 31-1 Infrastructure Module Entities

Entity Name	Profiles	Impl?
`Condition`	All	Yes
`GuardCondition`	All	Yes
`StatusCondition`	All	Yes

Table 31-2 Domain Module Entities

Entity Name	Profiles	Impl?
`DomainParticipant`	All	Yes
`DomainParticipantFactory`	All	Yes
`DomainParticipantListener`	All	Yes

Table 31-3 Topic-Definition Module Entities

Entity Name	Profiles	Impl?
`TopicDescription`	All	Yes
`Topic`	All	Yes
`ContentFilteredTopic`	Content-Subscription	No
`MultiTopic`	Content-Subscription	No
`TopicListener`	All	Yes
`TypeSupport`	All	Yes

Table 31-4 Publication Module Entities

Entity Name	Profiles	Impl?
`Publisher`	All	Yes
`DataWriter`	All	Yes
`PublisherListener`	All	Yes
`DataWriterListener`	All	Yes

Table 31-5 Subscription Module Entities

Entity Name	Profiles	Impl?
`Subscriber`	All	Yes
`DataReader`	All	Yes
`DataSample`	All	Yes
`SampleInfo`	All	Yes
`SubscriberListener`	All	Yes
`DataReaderListener`	All	Yes
`ReadCondition`	All	No
`QueryCondition`	Content-Subscription	No

31.3.1.2 Quality of Service (QoS) Compliance

The DDS specification defines several QoS policies. Each policy is applicable to certain entities. Not all policies pertain to every profile listed in 31.3.1. Table 31-6 shows the various QoS policies and their possible values, the entities to which the policies apply, the profiles to which each policy/value pertains, as well as whether or not the policy/value is implemented by OpenDDS.

Table 31-6 QoS Policies

Policy Name	Entities	Values	Profiles	Impl?
`USER_DATA`	`DomainParticipant` `DataWriter` `DataReader`	sequence of octets	All	Yes
`TOPIC_DATA`	`Topic`	sequence of octets	All	Yes
`GROUP_DATA`	`Publisher` `Subscriber`	sequence of octets	All	Yes
`DURABILITY`	`Topic` `DataWriter` `DataReader`	`VOLATILE`	All	Yes
		`TRANSIENT_LOCAL`	All	Yes
		`TRANSIENT (includes DURABILITY_SERVICE)`	Persistence	Yes
		`PERSISTENT (includes DURABILITY_SERVICE)`	Persistence	Yes

Table 31-6 QoS Policies

Policy Name	Entities	Values	Profiles	Impl?
PRESENTATION	Publisher Subscriber	INSTANCE scope COHERENT=true	All	No
		INSTANCE scope ORDERED=true	All	No
		TOPIC scope COHERENT=true	All	No
		TOPIC scope ORDERED=true	All	No
		GROUP scope COHERENT=true	Object Model	No
		GROUP scope ORDERED=true	Object Model	No
DEADLINE	Topic DataWriter DataReader	integer (period)	All	Yes
LATENCY_BUDGET	Topic DataWriter DataReader	integer (duration)	All	No
OWNERSHIP	Topic DataWriter DataReader	SHARED	All	Yes
		EXCLUSIVE	Ownership	No
OWNERSHIP_STRENGTH	Topic DataWriter DataReader	integer (value)	Ownership	No
LIVELINESS	Topic DataWriter DataReader	AUTOMATIC	All	Yes
		MANUAL_BY_PARTICIPANT	All	No
		MANUAL_BY_TOPIC	All	No
TIME_BASED_FILTER	DataReader	integer (minimum_separation)	All	No
PARTITION	Publisher Subscriber	sequence of strings	All	Yes[1]
RELIABILITY	Topic DataWriter DataReader	BEST_EFFORT	All	Yes
		RELIABLE	All	Yes[2]
TRANSPORT_PRIORITY	Topic DataWriter	integer	All	No
LIFESPAN	Topic DataWriter	integer (duration)	All	Yes
DESTINATION_ORDER	Topic DataWriter DataReader	BY_RECEPTION_TIMESTAMP	All	Yes
		BY_SOURCE_TIMESTAMP	All	No
HISTORY	Topic DataWriter DataReader	KEEP_LAST integer (depth)	All[3]	Yes
		KEEP_ALL	All	Yes

Table 31-6 QoS Policies

Policy Name	Entities	Values	Profiles	Impl?
RESOURCE_LIMITS	Topic DataWriter DataReader	integer (max_samples) integer (max_instances) integer (max_samples_per_instance)	All	Yes
ENTITY_FACTORY	DomainParticipantFactory DomainParticipant Publisher Subscriber	AUTO_ENABLE=true	All	Yes
		AUTO_ENABLE=false	All	No
WRITER_DATA_ LIFECYCLE	DataWriter	boolean (autodispose_unregistered_ instances)	All	No
READER_DATA_ LIFECYCLE	DataReader	integer (autopurge_nowriter_samples_ delay) integer (autopurge_disposed_samples_ delay)	All	No

1. Only wildcards of ‘*’ and ‘?’ are currently supported.
2. RELIABILITY.kind=RELIABLE supported only if the TCP or Reliable Multicast transport implementation is used.
3. KEEP_LAST.depth > 1 only applies to Ownership profile.

31.3.2 OpenDDS Architecture

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

31.3.2.1 Pluggable Transport Layer

OpenDDS uses the CORBA interfaces defined by the DDS specification to initialize and control service usage. Data transmission is accomplished via a OpenDDS-specific *Pluggable Transport* layer that allows the service to be used with a variety of transport protocols. OpenDDS currently implements simple TCP, UDP, reliable multicast and unreliable multicast transports. Transports are created via a factory object and are associated with publishers and subscribers who use them for their data transmission.

The pluggable transport layer enables application developers to implement their own customized protocols. Implementing your own custom transport involves specializing a number of classes defined in the transport framework directory `$DDS_ROOT/dds/DCPS/transport/framework`. See the simple

TCP implementation in `$DDS_ROOT/dds/DCPS/transport/simpleTCP` for details.

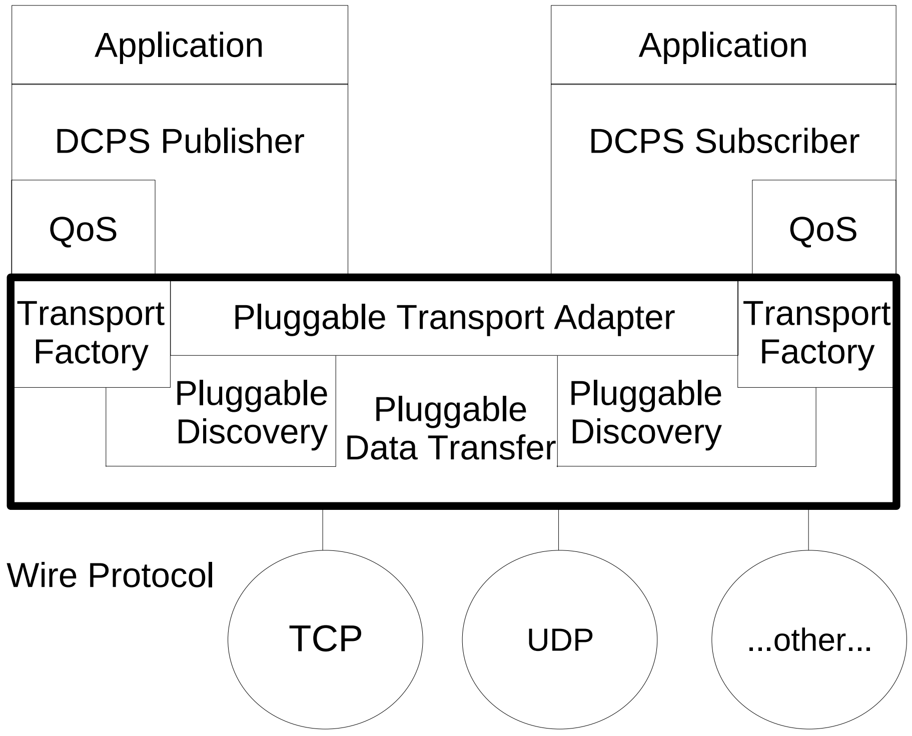


Figure 31-3 OpenDDS Pluggable Transport Framework

31.3.2.2 Custom Marshaling

Because data transmission is not done with CORBA, DDS implementations are free to marshal the data using customized formats. OpenDDS uses a more efficient variation of CORBA's Common Data Representation (CDR). A new IDL compiler switch (`-Gdcps`) causes the TAO IDL compiler to generate the appropriate marshaling and instance key support code for DCPS-enabled types.

31.3.2.3 DCPS Information Repository

The DCPS Information Repository acts as the intermediary or broker between the publisher and subscriber. It is currently 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. This process needs to be running whenever OpenDDS is being used. The InfoRepo is not involved in data propagation, its role is limited in scope to publishers and subscribers discovering one another.

It is possible to operate with more than a single repository providing 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 31.13.1 for further information about repository federations.

31.3.2.4 Threading

OpenDDS creates its own ORB as well as a separate thread upon which to run that ORB. It also uses its own threads to process incoming and outgoing non-CORBA 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 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 31.7.

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

31.3.2.5 Configuration

OpenDDS includes a file-based configuration framework for configuring both global items such as debug level, memory allocation, and `DCPSInfoRepo` locations, as well as transport implementations for publishers and subscribers. The complete set of configuration settings is described in 31.8.

31.4 Using DCPS

This section focuses on an example application using DCPS to distribute data from a publisher process to a subscriber. 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 Simple TCP transport. Full source code for this example is in the OpenDDS source code distribution in the directory `$DDS_ROOT/DevGuideExamples/DCPS/Messenger.` Additional DDS and DCPS features are discussed in later sections.

31.4.1 Defining the Data Types

Each data type used by DDS is defined using IDL. OpenDDS uses `#pragma` statements to identify the data types that DDS transmits and processes. These data types are processed by the TAO IDL compiler and the `dcps_ts.pl` script to generate code necessary for transmitting these types with DDS. Here is the IDL file that defines our `Message` data type:

```
module Messenger {

#pragma DCPS_DATA_TYPE "Messenger::Message"
#pragma DCPS_DATA_KEY "Messenger::Message subject_id"

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

The `DCPS_DATA_TYPE` pragma marks a data type for use with OpenDDS. A fully scoped type name must be used with this pragma. Currently, OpenDDS requires the data type to be a structure. The structure may contain scalar types (short, long, float, etc.), enumerations, strings, sequences, arrays, structures, and unions. This example defines the structure `Message` in the `Messenger` module for use in this OpenDDS example.

The `DCPS_DATA_KEY` pragma identifies a field of the DCPS data type that is used as the key for this type. A data type may have zero or more keys. These keys are used to identify the different instances within a topic that use this type. Each key should be a numeric or enumerated type. The pragma is passed the fully scoped type name and the member name that is the key for that type. Multiple keys are specified via separate `DCPS_DATA_KEY` pragmas with the same data type. In the above example, we identify the `subject_id` member of `Messenger::Message` as the key. Each message published with a unique subject ID value is defined as a different instance within a topic. Subsequent samples with the same subject ID value are treated as replacement values for that instance.

31.4.2 Processing the IDL

The OpenDDS IDL is processed like any other IDL with the exception that we pass the `-Gdcps` option the TAO IDL compiler.

```
tao_idl -Gdcps Messenger.idl
```

This causes the IDL compiler to generate additional serialization and key support code that OpenDDS uses to marshal and demarshal the `Message` structure.

In addition, we need to process the IDL file with the `dcps_ts.pl` script to generate the required type support code for the data readers and writers. This script 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 `dcps_ts.pl` as follows

```
dcps_ts.pl 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 servant implementations for these interfaces. The generated IDL file should itself be compiled to generate stubs and skeletons. These and the implementation file should be linked with your OpenDDS applications that use the `Message` type. This type support generation script has a number of options that specialize the generated code. These options are described in 31.12.

Typically, you do not directly invoke the IDL compiler or `dcps_ts.pl` script as above, but let your build environment do it for you. The entire process is simplified when using MPC, by inheriting from the `dcpsexe_with_tcp` project. Here is the MPC file section common to both the publisher and subscriber

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

  TypeSupport_Files {
    Messenger.idl
  }

  custom_only = 1
}
```

The `dcps` parent project adds the `-Gdcps` IDL compiler option and 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 `dcps_ts.pl` script. Here is the publisher section:

```
project(DDS*Publisher) : dcpsexe_with_tcp, dcps_unreliable_dgram,
                          dcps_reliable_multicast {

  exename   = publisher
  after    += DDS*idl

  TypeSupport_Files {
    Messenger.idl
  }

  Source_Files {
    publisher.cpp
    Writer.cpp
  }
}
```

The `dcpsexe_with_tcp` project links in the DCPS library.

31.4.3 Starting the DCPS Information Repository

The source code for DCPS Information Repository server is found in `$DDS_ROOT/dds/InfoRepo` and the server executable is `$DDS_ROOT/bin/DCPSInfoRepo`. This server process hosts the DCPSInfo CORBA object that is the entry point for all OpenDDS functionality. This object is mapped against the key string ‘`DCPSInfoRepo`’ in the processes IORTable. Thus a `corbaloc` ObjectURL such as:

```
corbaloc:iiop:localhost:12345/DCPSInfoRepo
```

can be used to locate the DCPSInfo object. The server also writes out the DCPSInfo object’s IOR as a string to a file, which can also be used to bootstrap clients. We can alter the file name used for writing this IOR with the `-o` command line option.

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

The full set of command line options for the DCPSInfoRepo server are documented in 31.13.

31.4.4 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`.

31.4.4.1 Participant Initialization

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(411, // domain ID
                              PARTICIPANT_QOS_DEFAULT,
                              DDS::DomainParticipantListener::_nil());
    if (CORBA::is_nil (participant.in())) {
      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 31.12. The `create_participant()` operation uses the domain participant factory to register this process as a participant in the domain specified by the ID of 411. The participant uses the default QoS policies and no listeners.

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

31.4.4.2 Data Type registration and topic creation

First, we create a `MessageTypeSupportImpl` object, then register the type with a type name. In this example, the type is registered with a nil string type name in which case the `MessageTypeSupport` interface repository id is used as the type name. A specific type name such as “Message” can be used as well.

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

Next, we obtain the registered type name from the TypeSupport servant and create the topic with the type name using the participant.

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

    DDS::Topic_var topic =
      participant->create_topic ("Movie Discussion List",
                                 type_name.in (),
                                 TOPIC_QOS_DEFAULT,
                                 DDS::TopicListener::_nil());
    if (CORBA::is_nil(topic.in())) {
      std::cerr << "create_topic failed." << std::endl;
      return 1;
    }
```

This creates a topic named “Movie Discussion List” with the registered type and the default QoS policies.

31.4.4.3 Transport initialization and registration

We now initialize the transport we want to use.

```
// This value must match the value in the publisher’s configuration file.
const OpenDDS::DCPS::TransportIdType TRANSPORT_IMPL_ID = 1;

OpenDDS::DCPS::TransportImpl_rch tcp_impl =
  TheTransportFactory->create_transport_impl (TRANSPORT_IMPL_ID,
                                              OpenDDS::DCPS::AUTO_CONFIG);
```

This code obtains the transport implementation from the singleton transport factory, called `TheTransportFactory`. The `OpenDDS::DCPS::AUTO_CONFIG` argument indicates that we are using a configuration file to configure the transport implementation.The value of the `TRANSPORT_IMPL_ID` identifier must match the transport id value in our configuration file (more on that later). Note that the code itself does not need to know any details about the transport implementation, such as whether it uses TCP or UDP, what its endpoints are, etc.

The `create_transport_impl()` operation can also be used to create a transport implementation with the default configuration:

```
OpenDDS::DCPS::TransportImpl_rch tcp_impl =
  TheTransportFactory->create_transport_impl (
    OpenDDS::DCPS::DEFAULT_SIMPLE_TCP_ID, OpenDDS::DCPS::AUTO_CONFIG);
```

The code above uses the default simple TCP transport identity `DEFAULT_SIMPLE_TCP_ID`. OpenDDS reserves a range`(0xFFFFFF00 ~ 0xFFFFFFFF)`for default transport identities. Currently, only the simple TCP, simple UDP and simple Mcast transport ids are supported. The default transport identities are defined in `TransportDef.h`.

```
const TransportIdType DEFAULT_SIMPLE_TCP_ID = 0xFFFFFF00;
const TransportIdType DEFAULT_SIMPLE_UDP_ID = 0xFFFFFF01;
const TransportIdType DEFAULT_SIMPLE_MCAST_PUB_ID = 0xFFFFFF02;
const TransportIdType DEFAULT_SIMPLE_MCAST_SUB_ID = 0xFFFFFF03;
```

The `TransportFactory` also provides alternate APIs to create a transport implementation.

```
const OpenDDS::DCPS::TransportIdType TRANSPORT_IMPL_ID = 1;
OpenDDS::DCPS::TransportImpl_rch tcp_impl =
  TheTransportFactory->create_transport_impl (
    TRANSPORT_IMPL_ID, "SimpleTcp", OpenDDS::DCPS::AUTO_CONFIG);
```

The code above creates a SimpleTCP transport implementation with default configuration. This API can be used to create multiple transport instances with the default configuration in a single process by passing unique transport IDs. This API can be used with file-based configurations as long as the matching transport configuration (based upon the transport id) also specifies the same transport type (in our example that is “SimpleTCP”).

We can also configure the transport implementation programmatically, eliminating the need for a configuration file. Here is sample code to create and configure a simple TCP transport implementation.

```
const OpenDDS::DCPS::TransportIdType TRANSPORT_IMPL_ID = 1;
OpenDDS::DCPS::TransportImpl_rch  tcp_impl =
  TheTransportFactory->create_transport_impl (
    TRANSPORT_IMPL_ID, "SimpleTcp", OpenDDS::DCPS::DONT_AUTO_CONFIG);

OpenDDS::DCPS::TransportConfiguration_rch config =
  TheTransportFactory->create_configuration (TRANSPORT_IMPL_ID);
OpenDDS::DCPS::SimpleTcpConfiguration* tcp_config =
  static_cast <OpenDDS::DCPS::SimpleTcpConfiguration*> (config.in());

ACE_INET_Addr local_address ("localhost:4444");
tcp_config->local_address_ = local_address;
tcp_config->local_address_str_ = "localhost:4444";

if (tcp_impl->configure(config.in()) != 0)
  {
    ACE_ERROR((LM_ERROR,
               ACE_TEXT(" Failed to configure the transport.\n")));
    return -1;
  }
```

31.4.4.4 Publisher creation

Now we are ready to create the publisher and attach the transport implementation we want it to use.

```
DDS::Publisher_var pub =
  participant->create_publisher(
    PUBLISHER_QOS_DEFAULT, DDS::PublisherListener::_nil());
if (CORBA::is_nil(pub.in())) {
  std::cerr << "create_publisher failed." << std::endl;
  return 1;
}

// Attach the publisher to the transport.
OpenDDS::DCPS::PublisherImpl* pub_impl =
  dynamic_cast<OpenDDS::DCPS::PublisherImpl> (pub.in());
if (0 == pub_impl) {
  std::cerr << "Failed to obtain publisher servant" << std::endl;
  return 1;
}

OpenDDS::DCPS::AttachStatus status =
  pub_impl->attach_transport(transport_impl.in());
```

We need to call `attach_transport()` on the publisher servant and not on the publisher reference because the OMG-defined Publisher interface lacks any OpenDDS-specific pluggable transport functionality. Since OpenDDS uses a CORBA local object to implement the publisher interface, we can just dynamically cast the publisher reference to the OpenDDS publisher implementation type to access the servant’s functionality.

31.4.4.5 DataWriter creation and instance registration

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

```
// Create the datawriter
DDS::DataWriter_var dw =
  pub->create_datawriter(topic.in (),
                         DATAWRITER_QOS_DEFAULT,
                         DDS::DataWriterListener::_nil());
if (CORBA::is_nil(dw.in())) {
  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. Now we can register the instance we wish to publish. We narrow the data writer reference to a `MessageDataWriter` object reference so we can use the type-specific registration and publication operations.

```
::Messenger::MessageDataWriter_var message_dw =
     ::Messenger::MessageDataWriter::_narrow(writer.in());

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

After we populate the Message structure we called the `_cxx_register()` 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 later use the returned instance handle when we publish a sample.

Note *This registration operation is actually* `register()` *in IDL but because register is a C++ keyword, the OMG IDL-to-C++ mapping maps the operation to the* `_cxx_register()` *member function.*

The example code waits for the subscriber to become connected and fully initialized. Once this is completed, the message publication is quite straightforward:

```
//Populate instance
 message.from       = CORBA::string_dup("Comic Book Guy");
 message.subject    = CORBA::string_dup("Review");
 message.text       = CORBA::string_dup("Worst. Movie. Ever.");
 message.count      = 0;
 DDS::ReturnCode_t ret = message_dw->write(message, handle);
```

This message is distributed to all connected subscribers that are registered for our topic. The second argument to `write()` specifies the instance we are publishing the sample upon. It should be passed either a handle returned by `_cxx_register()` 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.

31.4.5 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.

31.4.5.1 Participant Initialization

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(411, // Domain ID
                              PARTICIPANT_QOS_DEFAULT,
                              DDS::DomainParticipantListener::_nil());
    if (CORBA::is_nil (participant.in ())) {
      std::cerr << "create_participant failed." << std::endl;
      return 1 ;
    }
```

31.4.5.2 Data Type registration and topic creation

Then the message type and topic are initialized. 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.

```
    MessageTypeSupport_var mts = new MessageTypeSupportImpl();
    if (DDS::RETCODE_OK != mts->register_type(participant.in (),
                                                     "")) {
      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.in (),
                                TOPIC_QOS_DEFAULT,
                                DDS::TopicListener::_nil());
    if (CORBA::is_nil(topic.in())) {
      std::cerr << "Failed to create_topic." << std::endl;
      return 1;
    }
```

31.4.5.3 Transport initialization and registration

We now initialize the Simple TCP transport the same way as in the publisher, using the file-based configuration mechanism.

```
// This value must match the value in the subscriber’s configuration file.
const OpenDDS::DCPS::TransportIdType TRANSPORT_IMPL_ID = 1;

OpenDDS::DCPS::TransportImpl_rch transport_impl =
  TheTransportFactory->create_transport_impl (TRANSPORT_IMPL_ID,
                                              OpenDDS::DCPS::AUTO_CONFIG);
```

Next, we create the subscriber with the default QoS and then attach the Simple TCP object to the subscriber servant.

```
// Create the subscriber and attach to the corresponding
// transport.
DDS::Subscriber_var sub =
  participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT,
                                 DDS::SubscriberListener::_nil());
if (CORBA::is_nil(sub.in())) {
  std::cerr << "Failed to create_subscriber." << std::endl;
  return 1;
}

// Attach the subscriber to the transport.
OpenDDS::DCPS::SubscriberImpl* sub_impl =
  dynamic_cast<OpenDDS::DCPS::SubscriberImpl> (sub.in());
if (0 == sub_impl) {
  std::cerr << "Failed to obtain subscriber servant" << std::endl;
  return 1;
}
OpenDDS::DCPS::AttachStatus status =
  sub_impl->attach_transport(transport_impl.in());
```

31.4.5.4 DataReader and Listener activation

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

```
// activate the listener
DDS::DataReaderListener_var listener (new DataReaderListenerImpl);
DataReaderListenerImpl* listener_servant =
  dynamic_cast<DataReaderListenerImpl*>(listener.in());

if (CORBA::is_nil(listener.in())) {
  std::cerr << "listener is nil." << std::endl;
  return 1;
}
```

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. See 31.4.8 to see what changes to the code are necessary for the listener to be allocated on the stack.

Note *Previous versions of OpenDDS showed examples of objects being created on the stack and being passed to the library. Such usage is no longer supported and results in crashes on exit, unless the changes mentioned in 31.4.8 are also implemented.*

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 Datareaders
DDS::DataReader_var dr = sub->create_datareader(topic.in (),
                                                DATAREADER_QOS_DEFAULT,
                                                listener.in ());
if (CORBA::is_nil(dr.in())) {
  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.

31.4.6 The Data Reader Listener Servant

Our listener servant 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_match(in DataReader reader,
                               in SubscriptionMatchStatus status);
    void on_sample_lost(in DataReader reader, in SampleLostStatus status);
  };
};
```

Our example servant 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 function of this servant we need to explore.

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

  try {
    ::Messenger::MessageDataReader_var message_dr =
          ::Messenger::MessageDataReader::_narrow(reader);
    if (CORBA::is_nil(message_dr.in())) {
      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 = message_dr->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 31.6.

31.4.7 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.in ());
TheTransportFactory->release();
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. Lastly, we release our transport factory and shutdown the service participant.

31.4.8 Stack Allocated Listeners

Applications typically allocate objects on the heap and pass them to the OpenDDS library. Developers can also allocate objects on the stack. Following the example in 31.4.5.4, we could create the listener servant on the stack and pass its address to the listener.

```
StackDataReaderListenerImpl listener_servant;
DDS::DataReaderListener_var listener (&listener_servant);
```

The class definition for `StackDataReaderListenerImpl` is identical to `DataReaderListenerImpl` except that it inherits directly from `OpenDDS::DCPS::LocalObject_NoRefCount<DataReaderListenerImpl>`, rather than `OpenDDS::DCPS::LocalObject<DDS::DataReaderListener>`.

```
class StackDataReaderListenerImpl : public virtual
  OpenDDS::DCPS::LocalObject_NoRefCount<DDS::DataReaderListener>
```

31.4.9 Configuring the Example

OpenDDS includes a file-based configuration mechanism. Using this mechanism, you can configure a publisher's or subscriber's transport, the location of the `DCPSInfoRepo` process, memory allocation, and many other settings. The syntax of the configuration file is similar to the syntax of a Windows `INI` file. It contains several sections, which in turn contain property-like entries. The basic syntax is as follows:

```
[section1-name]
Attribute1=value1
Attribute2=value2

[section2-name]
Attribute1=value1
Attribute2=value2
```

Our example uses one configuration file, `pub.ini`, for the publisher, and a second configuration file, `sub.ini`, for the subscriber. First, we will examine `pub.ini`:

```
[common]
DCPSDebugLevel=0
DCPSInfoRepo=file://repo.ior

[transport_impl_1]
transport_type=SimpleTcp
local_address=localhost:4444
```

Notice that there are two sections, `[common]` and `[transport_impl_1]`. The `[common]` section contains configuration values that apply to the entire process; in this configuration file, we specify a debug level and an object reference for the `DCPSInfoRepo` object. The `[transport_impl_1]` section contains configuration values for the transport with the id of "1". We have configured the publisher to use the Simple TCP transport and to listen on port 4444 on the loopback network interface.

Recall in the publisher's code, we defined:

```
    const OpenDDS::DCPS::TransportIdType TRANSPORT_IMPL_ID = 1;
```

and configured the transport via

```
    OpenDDS::DCPS::TransportImpl_rch transport_impl =
      TheTransportFactory->create_transport_impl (
        TRANSPORT_IMPL_ID,
        OpenDDS::DCPS::AUTO_CONFIG);
```

The “1” in the transport configuration file matches the “1” defined in code as a transport id. Naturally, a publisher or subscriber process may contain more than one transport, each configured differently.

Next, we will examine the subscriber’s configuration file, `sub.ini`:

```
    [common]
    DCPSDebugLevel=0
    DCPSInfoRepo=file://repo.ior

    [transport_impl_1]
    transport_type=SimpleTcp
    local_address=localhost
```

We have configured the subscriber to also use the Simple TCP transport and to listen on an ephemeral port on the loopback network interface.

See 31.8 for a complete description of the OpenDDS configuration parameters.

31.4.10 Running the Example

Our simple example is now ready to be run. It can be run with the following commands. Running each of these commands in its own window should enable you to most easily understand the output.

```
$DDS_ROOT/bin/DCPSInfoRepo -ORBSvcConf tcp.conf -o repo.ior
./publisher -ORBSvcConf tcp.conf -DCPSConfigFile pub.ini
./subscriber -ORBSvcConf tcp.conf -DCPSConfigFile sub.ini
```

The `-DCPSConfigFile` command-line argument passes the location of the relevant configuration file to OpenDDS.

The `-ORBSvcConf` configuration directive file dynamically loads and configures the SimpleTCP library.

One side effect of using the default QoS properties is that, as we increase the number of samples being published, some of the samples will be dropped as the subscriber falls behind. To avoid dropping samples, we need to either ensure that the subscriber can keep up or change the QoS settings.

31.5 OpenDDS Java Bindings

Starting with release 1.2, OpenDDS provides Java 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.

See the `$DDS_ROOT/java/FAQ` file for information on common issues encountered while developing applications with the Java bindings.

31.5.1 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 31-7 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`	`dcps_ts.pl`: 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}`	`dcps_ts.pl`: DDS type-specific C++ impl.
`Baz/BarTypeSupportImpl.java`	`dcps_ts.pl`: DDS type-specific Java impl.
`Baz/Bar*.java`	`idl2jni`: Java representation of IDL struct
`FooJC.{h,cpp}`	`idl2jni`: JNI native method implementations

31.5.2 Setting up a 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 one line starting with “`#pragma DCPS_DATA_TYPE`”. 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:

   ```
       --- BEGIN Foo.mpc ---
       project: dcps_java {

         idlflags      += -Wb,stub_export_include=Foo_Export.h \
                          -Wb,stub_export_macro=Foo_Export
         dcps_ts_flags += --export=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
         }
       }
       --- END Foo.mpc ---
   ```

   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%•in\mwc.pl -type [CompilerType]
   ```

`CompilerType` can be `vc71`, `vc8`, `vc9`, and `nmake`

Make sure this is running ActiveState 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` or `vcexpress` (Express Edition) so that it inherits the environment variables. Command-line tools for building include `vcbuild` and invoking the IDE (`devenv` or `vcexpress`) 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. 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 `-Djni.nativeDebug=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.

31.5.3 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.

31.5.3.1 Participant Initialization

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;
    }
    DomainParticipant dp = dpf.create_participant(411,
        PARTICIPANT_QOS_DEFAULT.get(), null);
    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.

31.5.3.2 Data Type registration and Topic creation

Next we register our data type with the Domain Participant. We can specify a type name or pass an empty string. An empty string denotes application intent for the middleware to use the idl compiler generated Id.

```
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);
```

Now we have a topic named "Movie Discussion List" with the registered data type and default QoS policies.

31.5.3.3 Transport initialization and registration

We now initialize the transport we want to use.

```
TransportImpl transport_impl =
  TheTransportFactory.create_transport_impl(1,
    TheTransportFactory.AUTO_CONFIG);
```

The `TheTransportFactory.AUTO_CONFIG` argument indicates intent to use a configuration file for transport initialization. The supplied transport Id must have a matching entry in the configuration file. The code itself is independent of the transport implementation details.

31.5.3.4 Publisher creation

Next, we create a publisher:

```
Publisher pub = dp.create_publisher(PUBLISHER_QOS_DEFAULT.get(), null);
```

and attach it to the transport we previously initialized:

```
AttachStatus stat = transport_impl.attach_to_publisher(pub);
```

DataWriters and DataReaders spawned from this publisher will use the attached transport.

31.5.3.5 DataWriter creation and instance registration

With the publisher attached to a transport, we can now create a DataWriter:

```
DataWriter dw = pub.create_datawriter(top, DATAWRITER_QOS_DEFAULT.get(),
                                      null);
```

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):

```
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);
```

31.5.4 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, create the same named topic, and initialize a compatible transport.

```
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(411,
        PARTICIPANT_QOS_DEFAULT.get(), null);
    if (dp == null) {
      System.err.println ("Domain Participant creation failed");
      return;
    }

    MessageTypeSupportImpl servant = new MessageTypeSupportImpl();

    Topic top = dp.create_topic("Movie Discussion List",
                                servant.get_type_name(),
                                TOPIC_QOS_DEFAULT.get(), null);

    TransportImpl transport_impl =
      TheTransportFactory.create_transport_impl(1,
        TheTransportFactory.AUTO_CONFIG);
```

31.5.4.1 Subscriber creation

As with the Publisher, we create a Subscriber and attach it to the transport:

```
Subscriber sub = dp.create_subscriber(SUBSCRIBER_QOS_DEFAULT.get(),
                                      null);
AttachStatus stat = transport_impl.attach_to_subscriber(sub);
```

31.5.4.2 DataReader and Listener activation

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 DataWriter creation parameter:

```
DataReaderListenerImpl listener = new DataReaderListenerImpl();
DataReader dr = sub.create_datareader(top, DATAREADER_QOS_DEFAULT.get(),
                                                    listener);
```

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.

31.5.5 The DataReader Listener

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.

31.5.6 Cleaning up OpenDDS Java clients

An OpenDDS environment can be cleaned up 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.

```
TheTransportFactory.release();
```

Closes down any open Transports.

```
TheServiceParticipant.shutdown();
```

Shuts down the ServiceParticipant. This cleans up all OpenDDS associated resources.

31.5.7 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 a common property for the DCPSInfoRepo objects location:

```
[common]
DCPSInfoRepo=file://repo.ior
```

and a transport type property:

```
[transport_impl_1]
transport_type=SimpleTcp
```

The `[transport_impl_1]` section contains configuration information for the transport with the id of “1”. This id is used for transport creation in both our publisher and subscriber:

```
TransportImpl transport_impl =
  TheTransportFactory.create_transport_impl(1,
    TheTransportFactory.AUTO_CONFIG);
```

See 31.8 for a complete description of all OpenDDS configuration parameters.

31.5.8 Running the example

To run the Messenger Java OpenDDS application, use the following commands:

```
$DDS_ROOT/bin/DCPSInfoRepo -ORBSvcConf tcp.conf -o repo.ior

$JAVA_HOME/bin/java -ea -cp
classes:$DDS_ROOT/lib/i2jrt.jar:$DDS_ROOT/lib/OpenDDS_DCPS.jar:classes
TestPublisher -ORBSvcConf tcp.conf -DCPSConfigFile pub_tcp.ini

$JAVA_HOME/bin/java -ea -cp
classes:$DDS_ROOT/lib/i2jrt.jar:$DDS_ROOT/lib/OpenDDS_DCPS.jar:classes
TestSubscriber -ORBSvcConf tcp.conf -DCPSConfigFile sub_tcp.ini
```

The `-DCPSConfigFile` command-line argument passes the location of the OpenDDS configuration file.

The `-ORBSvcConf` configuration directive file dynamically loads and configures the SimpleTCP transport library.

31.6 Data Handling Optimizations

31.6.1 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 quickly arriving. 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 specialize which samples are returned from the reader. See the DDS specification for details on their usage.

31.6.2 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 constructing a data sequence with `max_len` (the first parameter) of zero as shown by this sample code (from `DevGuideExamples/DCPS/Messenger_ZeroCopyRead/`):

```
CORBA::Long MAX_ELEMS_TO_RETURN = 3;
Messenger::MessageSeq the_data (0, MAX_ELEMS_TO_RETURN);
DDS::SampleInfoSeq    the_info (0, MAX_ELEMS_TO_RETURN);

// get references to the samples  (zero-copy read of the samples)
DDS::ReturnCode_t status = dr->read (the_data
                                     , the_info
                                     , MAX_ELEMS_TO_RETURN
                                     , ::DDS::ANY_SAMPLE_STATE
                                     , ::DDS::ANY_VIEW_STATE
                                     , ::DDS::ANY_INSTANCE_STATE);
```

For both zero-copy reads and single-copy reads, the sample and info sequences' length are set to the number of samples read. For the zero-copy reads, the `max_len` is be 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 (the_data, the_info);
```

Calling `return_loan()` results in the sequences' `max_len` being set to 0 and its `owns` member set to false; therefore the same sequences could 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` > `0`, 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 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.

31.7 Policies

The previous examples use default QoS policies for the various entities. This section discusses which QoS policies are implemented in OpenDDS and the details of their usage.

31.7.1 Supported Policies

Listed below are the QoS policies that are currently supported by OpenDDS. Any policy not listed here uses its default value. The default values of unsupported policies are as described in the DDS specification and some are discussed in 31.7.2.

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 set, modifying the individual policy structures as necessary, and then applying the QoS structure to an entity (usually when it is created).

Applications can change the QoS of any entity via the `set_qos()` operation. If the QoS is changeable, the QoS changes are propagated to DCPSInfoRepo via QoS update invocations on the corresponding entity, such as `update_subscription_qos()`. 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 and addition of new associations.

Most of the currently supported QoS policies are not changeable. Some changeable QoS policies, including USER_DATA, TOPIC_DATA, GROUP_DATA, and LIFESPAN, do not require compatibility and association re-evaluation. The DEADLINE QoS policy requires compatibility re-evaluation, but not for association.The PARTITION QoS policy does not require compatibility re-evaluation, but does require association re-evaluation.

31.7.1.1 LIVELINESS

The LIVELINESS policy applies to the Topic, Data Reader, and Data Writer entities via the `liveliness` member of their respective QoS structures. 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 is restricted to Automatic (`AUTOMATIC_LIVELINESS_QOS`) in OpenDDS. This means that the service periodically polls participants for liveliness. The `lease_duration` member is set to the desired heartbeat interval. The default lease duration is a pre-defined infinite value, which disables any liveliness testing.

Data writers specify their own liveliness criteria and data readers specify the desired liveliness of their writers. Writers that are not heard from within the lease duration cause notification via the Data Reader Listener's `on_liveliness_changed()` operation. Because OpenDDS's Liveliness is always set to Automatic, the `on_liveliness_lost()` callback is never called on the publisher side.

This policy is considered during the creation of associations between DataWriters and DataReaders. The value of both sides of the association must be compatible in order for an association to be created.The liveliness kind is always compatible since OpenDDS just supports `AUTOMATIC_LIVELINESS_QOS` currently. The `lease_duration of` DataReader must be bigger than or equal to the value of DataWriter.

31.7.1.2 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 Simple TCP transport supports the Reliable value for this policy and the Simple UDP transport only supports the Best Effort value. 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 return because of resource limits (due to transport backpressure—see 31.7.1.6 for details). 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 is Best Effort.

This policy is considered during the creation of associations between DataWriters and DataReaders. The value of both sides of the association must be compatible in order for an association to be created. The liveliness kind of DataWriter must be bigger than or equal to the value of DataWriter.

31.7.1.3 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.

31.7.1.4 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;
  Duration_t service_cleanup_delay;
};
```

By default the `kind` is Volatile and `service_cleanup_delay` is zero which means infinite time delay. OpenDDS currently supports all four kinds durability as listed above, however, the `service_cleanup_delay` is not supported.

The Volatile durability means samples are discarded after being sent to all known subscribers. A side effect of this is that subscribers cannot recover samples sent before they connect.

The Transient Local durability 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.

The Transient durability means that samples outlive Data Writer and last as long as the process. The samples are kept in memory but not in permanent storage. The Data Reader subscribed to the same topic within the same domain will be sent of all cached samples that belong to the same topic.

The Persistent durability provides basically the same functionality as Transient Durability except the cached samples will survive process destruction.

This policy is considered during the creation of associations between DataWriters and DataReaders. The value of both sides of the association must be compatible in order for an association to be created. The durability kind of DataReader must be bigger than or equal to the value of DataWriter. The order of durability kinds is as listed in the enumeration definition in the IDL above; `VOLATILE _DURABILITY_QOS` is the least durable setting and `PERSISTENT_DURABILITY_QOS` is the most durable setting.

31.7.1.5 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.

31.7.1.6 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 Reader. All of 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.

31.7.1.7 PARTITION

The PARTITION QoS policy allows the creation of logical partitions within a domain. It only allows DataReaders and DataWriters to be associated if they have matched partition strings. This policy applies to the Publisher and Subscriber 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 string sequence. 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.

Note *According to the DDS specification, PARTITION names can be regular expressions and include wildcards as defined by the POSIX fnmatch API (1003.2-1992 section B.6). In OpenDDS, only the"*" and "?" wildcards are currently supported.*

The establishment of DataReader and DataWriter association depends on matching partition strings on the publication and subscription ends, the 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.

31.7.1.8 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 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 `period` is infinite, which requires no behavior. When this policy is set to a finite value, then the DataWriter 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 DataReader which 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 DataWriters and DataReaders. The value of both sides of the association must be compatible in order for an association to be created. The deadline period of DataReader must be bigger than or equal to the value of DataWriter.

The value of this policy may change after the associated entity is enabled. In the case where the policy of a DataReader or DataWriter is made, the change is successfully applied only if the change remains consistent with the remote end of all associations in which the DataReader or DataWriter is participating. If the policy of a Topic is changed, it will affect only DataReaders and DataWriters that are created after the change has been made. Any existing DataReaders or DataWriters, and any existing associations between them, will not be affected by the Topic policy value change.

31.7.1.9 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 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 `duration` of LifespanQosPolicy 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`. Expired samples are not removed from DataWriter and DataReader caches in the current version.

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

31.7.1.10 USER_DATA

The USER_DATA policy applies to the DomainParticipant, Data Reader and Data Writer entities via the `user_data` member of their respective QoS structures. Below is the IDL related to the UserData 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 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.

31.7.1.11 TOPIC_DATA

This TOPIC_DATA policy applies to the Topic entities via the `topic_data` member of TopicQoS structures. Below is the IDL related to the TopicData 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 TopicData is available in DataWriter, DataReader and Topic built-in-topic data. The remote application can obtain the information via built-in-topic and use it in an application-defined way.

31.7.1.12 GROUP_DATA

This 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 GroupData QoS policy:

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

By default, the `value` member of GroupDataQosPolicy 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-topic. The DataWriter built-in-topic data contains the GROUP_DATA from the publisher and the DataReader 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 31.7.1.7 except the decision could be made based on an application-defined policy.

31.7.2 Unsupported Policies

The unsupported policies cannot be modified with OpenDDS and always take the default value. The following subsections discuss some of the default values that may affect application behavior.

31.7.2.1 ENTITY_FACTORY

The ENTITY_FACTORY policy controls whether created entities are automatically enabled. The default is that all entities are automatically enabled on creation.

31.7.2.2 PRESENTATION

The PRESENTATION policy controls the ordering or grouping of samples in a topic. The default value of Instance means that all samples within an instance are delivered in the order the subscriber receives them. Samples from different instances of the same topic may be arbitrarily reordered by the service.

31.7.3 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.in (),
                             dw_qos,
                             DDS::DataWriterListener::_nil());
```

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.

31.8 Configuration

OpenDDS includes a file-based configuration framework for configuring both global settings as well as transport implementations for publishers and subscribers.This section summarizes the configuration settings in OpenDDS.

We use the `-DCPSConfigFile` command-line argument to pass the location of the configuration file into OpenDDS. For example,

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

causes the OpenDDS service participant to read configuration settings from the `pub.ini` configuration file. More accurately, we pass the publisher’s command-line arguments to the service participant singleton when we initialize the domain participant factory. We did this in the preceding examples by using the `TheParticipantFactoryWithArgs` macro:

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

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

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

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

31.8.1 Common Configuration Settings

The `[common]` section of the OpenDDS configuration file contains settings for attributes such as debugging output, the default object reference of the `DCPSInfoRepo` process, and memory preallocation settings. A sample `[common]` section follows:

```
[common]
DCPSDebugLevel=0
DCPSInfoRepo=corbaloc:iiop:localhost:12345/DCPSInfoRepo
DCPSLivelinessFactor=80
DCPSChunks=20
DCPSChunksAssociationMultiplier=10
DCPSBitTransportPort=
DCPSBitLookupDurationMsec=2000
```

It is not necessary to specify every attribute.

Each `[common]` attribute's value can be overridden by a command-line argument. The command-line argument has the same name as the configuration option with a "-" prepended to the front of it. For example,

```
subscriber -DCPSInfoRepo corbaloc:iiop:localhost:12345/DCPSInfoRepo
```

The following table summarizes the `[common]` configuration attributes:

Table 31-8 Common Configuration Settings

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`

Table 31-8 Common Configuration Settings

Option	Description	Default
`DCPSTransportDebugLevel n`	Integer value for controlling the transport logging granularity. Legal values span from 0 to 5.	`0`
`DCPSInfoRepo` *`objref`*	Object reference for locating the DCPS Information Repository	`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`
`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`

Table 31-8 Common Configuration Settings

Option	Description	Default
`DCPSBit [1|0]`	Toggle Built-In-Topic support.	`1`
`DCPSBitTransportPort` *`port`*	Port used by the Simple TCP transport for Built-In Topics.	none; OS chooses port
`DCPSBitTransportIPAddress`	IP address identifying the local interface to be used by SimpleTcp transport for the Built-In Topics.	empty string; equivalent to INADDR_ANY
`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`

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`).

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.

31.8.2 Transport Configuration Settings

A OpenDDS user may configure one or more transports in a single configuration file. A sample transport configuration is below:

```
[transport_impl_1]
transport_type=SimpleTcp
local_address=localhost:4444
```

```
swap_bytes=0
optimum_packet_size=8192
```

Again, it is not necessary to specify every attribute.

The “1” in the transport_impl_1 marker is the identifier for the transport. That number must match the transport id in our code. You may recall, in both the publisher and subscriber, we identified the transport id as follows:

```
const OpenDDS::DCPS::TransportIdType TRANSPORT_IMPL_ID = 1;
```

and created the transport implementation object as follows:

```
OpenDDS::DCPS::TransportImpl_rch trans_impl =
  TheTransportFactory->create_transport_impl (
    TRANSPORT_IMPL_ID,
    OpenDDS::DCPS::AUTO_CONFIG);
```

Thus, we can see where the transport's identifier of “1” in the configuration file maps to the creation of the transport in the C++ code.

31.8.2.1 Common Transport Configuration Settings

The following table summarizes the transport configuration attributes that are common to all transports:

Table 31-9 Transport Configuration Settings

Option	Description	Default
`transport_type` *`transport`*	Type of the transport; the list of available transports can be extended programmatically via the OpenDDS Pluggable Transport Framework. `SimpleTcp`, `SimpleUdp`, `SimpleMcast,` and `ReliableMulticast` are included with OpenDDS.	none

Table 31-9 Transport Configuration Settings

Option	Description	Default
`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 setting between machines. The purpose of this setting is to allow the developer to decide which side will make the endian adjustment, if necessary.	`0`
`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`
`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`
`thread_per_connection` *`0|1`*	Enable or disable the thread per connection send strategy.	`0(disabled)`
`datalink_release_delay`	The datalink_release_delay is the delay (in seconds) 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.	`10`

Enabling the `thread_per_connection` setting 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.

31.8.2.2 SimpleTcp Transport Configuration Settings

The following table summarizes the transport configuration attributes that are either unique to the Simple TCP transport, or whose default value or description is overridden by the Simple TCP transport:

Table 31-10 SimpleTcp Configuration Settings

Option	Description	Default
`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.	fqdn: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`
`conn_retry_initial_delay` *`n`*	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`

Table 31-10 SimpleTcp Configuration Settings

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`
`conn_retry_attempts` *n*	Number of reconnect attempts before giving up and calling on_publication_lost and on_subscription_lost callback.	`3`
`max_output_pause_period` *n*	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_connect_duration` *n*	Timeout (milliseconds) for initial passive connection establishment. This does NOT affect the reconnect timing.	`0 (wait forever)`
`passive_reconnect_duration` *n*	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`

SimpleTcp Reconnection Options

When a TCP connection gets closed DDS attempts to reconnect. The reconnection process is (a successful reconnect ends this sequence):

- Upon detecting a lost connection immediately attempt reconnect.
- 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.

31.8.2.3 SimpleUdp/SimpleMcast Transport Configuration Settings

While both SimpleUdp and SimpleMcast are unreliable datagram transports, they share a set of common transport configuration attributes. The following table summarizes those common transport configuration attributes that are either unique to both SimpleUdp and SimpleMcast transports, or whose default value or description is overridden by SimpleUdp and SimpleMcast transports:

Table 31-11 SimpleUdp and SimpleMcast Common Configuration Settings

Option	Description	Default
`max_packet_size` *`n`*	Maximum size of a UDP packet. The `SimpleUdp` and `SimpleMcast` transports have a different default value than the other transports.	`62501`
`max_output_pause_period` *`n`*	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 socket will be closed and on_*_lost callbacks will be called. If the value is zero, the default, then this check will not be made.	`0`

The SimpleUdp and SimpleMcast share the `local_address` configuration but its meaning is different for the different transport implementations. Here are the settings unique to the SimpleUdp transport:

Table 31-12 SimpleUdp Configuration Settings

Option	Description	Default
`local_address` *`host:port`*	Address and port at which the transport reads UDP packets. The default value is the FQDN and port 0, which means the OS will choose the port.	fqdn:0

In addition to the common configuration attributes listed above, the SimpleMcast transport specifies a few other configuration attributes. The following table summarizes those configuration attributes that are unique to the SimpleMcast transport.

Table 31-13 SimpleMcast Configuration Settings

Option	Description	Default
`local_address` *`host:port`*	Used on the publisher side to specify which NIC card will be used. This is not available on the subscriber side; it defaults to use the FQDN and port 0, which means the OS will choose the port.	fqdn:0
`multicast_group_address` *`host:port`*	Address at which the publisher sends multicast packets to and subscriber receives multicast packets from. Uses ACE default multicast address as default.	224.9.9.2:20001 (IPv4) ff05:0::ff01:1:20001(IPv6)
`receiver` *`0|1`*	Flag indicates if the transport is receiving side (subscriber) or sending side (publisher). Defaults to 0 (publisher) side.	`0`

31.8.2.4 ReliableMcast Transport Configuration Settings

The ReliableMcast transport builds data reliability upon the multicast protocol. There are some similarities between its options and those of the SimpleMcast transport. Here is the full list of ReliableMcast options:

Table 31-14 ReliableMcast Configuration Settings

Option	Description	Default
`local_address` *`host:port`*	Used on the publisher side to specify which NIC card will be used. This is not available on the subscriber side; it defaults to use the FQDN and port 0, which means the OS will choose the port.	fqdn:0
`multicast_group_address` *`host:port`*	Address at which the publisher sends multicast packets to and subscriber receives multicast packets from. Uses ACE default multicast address as default.	224.9.9.2:20001 (IPv4) ff05:0::ff01:1:20001(IPv6)

Table 31-14 ReliableMcast Configuration Settings

Option	Description	Default
`receiver` *`0|1`*	Flag indicates if the transport is receiving side (subscriber) or sending side (publisher). Defaults to 0 (publisher) side.	`0`
`sender_history_size n`	Specifies the history buffer size for the sender, in units of packets. The history buffer consumes memory but provides recall in the event of dropped packets at any receiver.	1024
`receiver_buffer_size n`	Specifies the buffer size for a receiver, in units of packets. This buffer lets the receiver properly order incoming packets and detect gaps. A larger buffer will consume memory, while a smaller one would reduce the effectiveness of the reliability protocol.	256

31.8.3 Multiple DCPSInfoRepo Configuration

With the release of OpenDDS 1.1, a single OpenDDS process can be associated with multiple DCPS information repositories (DCPSInfoRepo).

The repository information and domain associations can be configured using configuration file, or via application API. Previously used defaults, command line arguments, and configuration file settings will work as is for existing applications that do not want to use multiple DCPSInfoRepo associations.

Domains not explicitly mapped with a repository are automatically associated with the default repository. Individual DCPSInfoRepos can be associated with multiple domains, however domains cannot be shared between multiple DCPSInfoRepos.

Repository and domain association information is contained within individual `[repository]` and `[domain]` subsections within the configuration file. The subsections are specified using a slash separated path syntax. Repository subsection header follow the format `[repository/<NAME>]` where the "`repository/`" is literal and "`<NAME>`" is replaced with an arbitrarily chosen but unique subsection name. Similarly, a domain subsection is specified as `[domain/<NAME>]`. There may be any number of repository or domain sections within a single configuration file.

Each repository section requires the keys RepositoryIor and RepositoryKey to be defined. The RepositoryKey values must be unique for each repository within the configuration file.

Each Domain subsection requires the keys DomainId and DomainRepoKey to be defined. The DomainRepoKey matched to a RepositoryKey maps the domain to that repository. The special value DEFAULT_REPO can be used to associate a domain with the default repository.

Table 31-15 Multiple repository configuration sections

Subsection	Key	Value
[repository/<NAME>]	RepositoryIor	Repository IOR.
	RepositoryKey	Unique key value for the repository.
[domain/<NAME>]	DomainId	Domain being associated with a repository.
	DomainRepoKey	Key value of the mapped repository.

31.9 Pluggable Transports

The previous section gave an overview of currently available configuration options. What follows is a discussion of the specifics of the individual transports and how their behavior can be modified by using these options.

31.9.1 Simple TCP Transport

As observed in the previous section, there are a number of configurable options for SimpleTCP. 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 parameter is used by the peer to establish a connection. By default, the TCP transport selects a random port number on the NIC with 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 an inter-host test, the local_address can not be localhost and should be configured with an externally visible interface(i.e. `192.168.0.2`), or you can leave it unspecified in which case the FQDN and a random port will be used. Note that this parameter also applies to unreliable datagram transports with the same restrictions.

FQDN resolution is dependent upon system configuration. In the absence of a FQDN (e.g. example.ociweb.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. loopback).

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 `passive_connect_duration` parameter 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. When a FQDN is not found, the system will emit a warning.

SimpleTCP exists as an independent library and therefore needs to be linked and configured like the other pluggable transport libraries. The `-ORBSvcConf` option feeds the ACE Service Configuration directive file to configure the SimpleTCP library. The Messenger example from 31.4.4 demonstrates dynamically loading and configuring the SimpleTCP library.

When the SimpleTCP library is built statically, your application must link directly against the SimpleTCP library. To do this, your application must first include the proper header for service initialization, `$DDS_ROOT/dds/DCPS/transport/simpleTCP/SimpleTcp.h`. Then, the static initialization directive

```
static DCPS_SimpleTcpLoader "-type SimpleTcp"
```

will configure the SimpleTCP transport at run-time.

You can also configure the publisher and subscriber transport implementations programatically, as described in 31.4. Configuring subscribers and publishers should be identical, but different addresses/ports should be assigned to each Transport Implementation.

31.9.2 Unreliable Datagram Transports

As mentioned in previous sections, two unreliable datagram transports, SimpleUdp and SimpleMcast, are supported in this release. Both transports exist in the `SimpleUnreliableDgram` library. To use these transports, the `SimpleUnreliableDgram` library needs be dynamically or statically linked via the -ORBSvcConf option. You can dynamically load the `SimpleUnreliableDgram` library with a service configuration directive:

```
dynamic OPENDDS_DCPS_SimpleUnreliableDgramLoader Service_Object *
SimpleUnreliableDgram:_make_OPENDDS_DCPS_SimpleUnreliableDgramLoader()
"-type SimpleUdp"
```

With this service configuration directive, the `SimpleUdp` component is registered with the transport factory as the library is loaded. To apply the SimpleMcast transport, replace `SimpleUdp` in the directive above with `SimpleMcast`. A single process can apply both SimpleUdp and SimpleMcast transports via multiple service configuration directives.

Because the unreliable datagram transports do not support fragmentation of a single sample into multiple packets, they currently limit the size of marshaled samples, including headers, to 64 KB. Attempting to send a sample greater than 64 KB with these unreliable datagram transports will result in an error message and the sample not being delivered.

Using the unreliable datagram transport involves the same steps that we have seen before: creating a Transport Implementation, attaching it to the Publisher and Subscriber servants, and configuring it through one or more configuration files. As observed in the previous section, SimpleUdp and SimpleMcast transport configurations share a common set of attributes. In addition, the SimpleMcast transport has its own specific attributes. The following sections show a transport configuration example for SimpleUdp and SimpleMcast and special notes for the individual attributes.

31.9.2.1 SimpleUDP Transport

Here is a SimpleUDP transport configuration example:

```
# file pub_udp.ini

[common]
DCPSDebugLevel=0
DCPSInfoRepo=file://repo.ior
```

```
[transport_impl_2]
transport_type=SimpleUdp
local_address=localhost:16701
max_output_pause_period=0
```

According to this configuration file, a publisher application will read UDP packets on port 16701 on the loopback network interface.

Note that the `max_output_pause_period` configuration attribute specifies the timeout when the transport is under backpressure. Unlike SimpleTcp and SimpleMcast transports, backpressure has not been observed during internal testing and development; this parameter and its functionality have been included as such a situation may exist in a DDS deployment environment. Backpressure is handled in a similar manner to SimpleTcp and SimpleMcast.

The example above shows the configuration for a publisher, but a subscriber's configuration may just differ in terms of its `local_address` (IP address and port).

31.9.2.2 SimpleMcast Transport

Here is a SimpleMcast transport configuration example for a publisher:

```
# file pub_mcast.ini

[common]
DCPSDebugLevel=0
DCPSInfoRepo=file://repo.ior

[transport_impl_3]
transport_type=SimpleMcast
local_address=192.168.0.2:16701
multicast_group_address=224.0.0.1:29803
receiver=0
max_output_pause_period=0
```

In this example, a publisher sends multicast packets to port 29803 on the 224.0.0.1 multicast group address from port 16701 on the NIC with the IP address of 192.168.0.2.

Note that on Win32 machines, the `local_address` parameter should not be the loopback address (`localhost`, or `127.0.0.1`). It must have an external interface's address or remain blank to let the transport automatically select the default NIC.

Here is the configuration for a SimpleMcast subscriber:

```
# file sub_mcast.ini

[common]
DCPSDebugLevel=0
DCPSInfoRepo=file://repo.ior

[transport_impl_3]
transport_type=SimpleMcast
multicast_group_address=224.0.0.1:29803
receiver=1
max_output_pause_period=0
```

This example configures a subscriber application to listen to the 224.0.0.1 multicast group address, again at port 29803. The same `multicast_group_address` should be used for both publishers and subscribers.

The `receiver` configuration attribute specifies the role of the transport. It must be `0` on the publisher side and `1` on the subscriber side. Unlike SimpleTcp and SimpleUdp transports, the same SimpleMcast transport object cannot be shared by both the publisher and subscriber.

Of particular importance is the missing `local_address` parameter. While it is perfectly acceptable to specify this parameter for a publisher, it is not available for subscribers in this version of DDS.

31.9.3 Reliable Multicast Transport

The reliable multicast transport provides reliable operation on an unreliable multicast channel. Understanding the meaning of "reliable", how this reliability is achieved, and what happens when reliability is compromised are all vital to properly configuring the transport. Of note to developers is that the reliability components of this transport are completely separate from those that handle sending and receiving of data. Thus, the reliability portion could be extracted and reused in another transport, provided the underlying transport allows for bidirectional communication.

In the context of this transport, reliability is defined as in-order, lossless delivery of data. Since multicast is UDP, it exhibits UDP's loss and transmission characteristics. Therefore, to achieve the desired level of reliability, both the sending and receiving side must have special logic:

A sender must:

- Fragment outgoing messages from the transport framework into packets with headers appropriate for reliability and reassembly.
- Maintain a packet history buffer to respond to retransmission requests.
- Send out periodic heartbeat messages to let receivers detect loss at the end of a burst.
- Respond to requests for expired historical data with a “not available” packet.

A receiver must:

- Buffer data received out of order.
- Detect “gaps” in the transmission and request retransmissions.
- Deliver complete messages to the transport framework in the proper order.
- Report disconnection upon receipt of a “not available” packet for data it has requested and not yet received.

Like other transports ReliableMcast needs to be either dynamically or statically configured. Shown below is directive to dynamically load and configure the transport:

```
dynamic OPENDDS_DCPS_ReliableMulticastLoader Service_Object *
ReliableMulticast:_make_OPENDDS_DCPS_ReliableMulticastLoader() ""
```

Static configuration requires the inclusion of the header file:

```
#include "dds/DCPS/transport/ReliableMulticast/ReliableMulticast.h"
```

and the service config directive:

```
static OPENDDS_DCPS_ReliableMulticastLoader ""
```

31.10 Using Built-In Topics

The built-in topics are published by the `DCPSInfoRepo` server whenever the `-NOBITS` option is not specified. Four separate topics are defined for each domain that this server manages. 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 the built-in topics are automatically created for each domain participant. Participants support for BITs can be toggled via configuration option *DCPSBit* (see Table 31-8). To view the data you must simply obtain the built-in Subscriber and then use it to get the Data Reader for the built-in topic of your interest. Then the Data Reader can be used like any other Data Reader.

Note *The Built-In Topics feature is currently dependent upon the SimpleTCP transport library. The DCPSInfoRepo server as well as any participating subscribers and publishers will need to configure the SimpleTCP library to handle Built-In topics.*

Sections 31.10.2 through 31.10.5 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.

31.10.1 Building Without BIT Support

If you are not planning on using Built-in-Topics in your application, you can configure DDS to remove BIT support at build time. Doing so can reduce the footprint of the core DDS library by up to 30%. To remove support for Built-In Topics follow these steps:

1. Regenerate the project files without the Built-In Topic feature. Either use the command line "feature" argument to MPC:

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

   Or alternatively, add the line `built_in_topics=0` to the file `$DDS_ROOT/MPC/config/default.features` and regenerate the project files using MPC.
2. If you are using the gnuace MPC project type (which is the case if you will be using GNU make as your build system), add line "`built_in_topics=0`" to the file `$ACE_ROOT/include/makeinclude/platform_macros.GNU`.
3. Build DDS as usual (see `$DDS_ROOT/docs/INSTALL` for instructions).

31.10.2 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.

31.10.3 DCPSTopic Topic

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.

31.10.4 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.

31.10.5 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.

31.10.6 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.in());

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.

31.11 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 and Transport levels.

31.11.1 DCPS Level Logging

Logging in the DCPS level of OpenDDS is controlled by the `DCPSDebugLevel` configuration setting and command-line option. It can also be set programmatically from 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

31.11.2 Transport Level Logging

OpenDDS transport level logging is controlled via the `DCPSTransportDebugLevel` configuration option. For example, to add transport level logging to any OpenDDS application, add the following option to the command line:

```
-DCPSTransportDebugLevel=level;
```

Transport level logging can also be programmatically configured by appropriately setting the variable:

```
OpenDDS::DCPS::Transport_debug_level = level;
```

Valid transport logging levels range from 0 to 5 with increasing verbosity of output.

Note *Actually, 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.*

31.12 dcps_ts.pl Command Line Options

The `dcps_ts.pl` script is located in `$DDS_ROOT/bin` and parses a single IDL file for DCPS-enabled types then generates type support code for those types. For each IDL file processed, such as `xyz.idl`, it generates three files: `xyzTypeSupport.idl`, `xyzTypeSupportImpl.h`, and `xyzTypeSupportImpl.cpp`. In the typical usage, the script is passed a number of options and the IDL file name as a parameter. For example,

```
$DDS_ROOT/bin/dcps_ts.pl Foo.idl
```

The following table summarizes the entire set of options the script supports. Note that many have terse and verbose variants of the same option.

Table 31-16 dcps_ts.pl Command Line Options

Option	Description	Default
`--verbose` `--noverbose`	Enables/disables verbose execution	Quiet execution
`--debug` `-d`	Enable debug statements in the script	No debug output
`--help` `-h`	Prints a usage message and exits	N/A
`--man`	Prints a man page and exits	N/A

`--dir=dirpath`
`-S` *`dirpath`*
	Subdirectory where IDL file is located	No subdir used

`--export=`*`macro`*
`-X` *`macro`*
	Export macro used for generating C++ implementation code.	No export macro used

`--pch=file`	Pre-compiled header file to include in generated C++ files	No pre-compiled header included
`--timestamp` `-t`	Backup any previously existing generated files with a timestamp suffix.	Old files are not backed up
`--nobackup`	Do not back up the previously generated files	Old files are not backed up
`--idl=`*`file`*	The IDL file to process.	IDL file is assumed to be a parameter

`--output=outdir`
`-o outputdir`
	Output directory where dcps_ts.pl should place the generated files.	The current directory

These options mainly divide into two main categories, those related to the execution of the script and those that control the generated code. In the former category are documentation options like `--help` and `--man` as well as script debugging options like `--verbose` and `--debug`.

The code generation options allow the application developer to use the generated code in a wide variety of environments. The `-dir` option lets you operate on IDL files in other directories and causes the generated IDL code to use the proper paths in the includes. The `--export` 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 4) uses the export keyword. The `--pch` option is required if the generated servant code is to be used in a component that uses precompiled headers. The `--module` option allows you to put the generated code in a C++ namespace and avoid name collisions and pollution of the global name space. The `--timestamp` and `--nobackup` options control whether older versions of the generated files are preserved with timestamp-appended file name or whether they are simply overwritten.

The `--idl` option allows you to specify the IDL file to process with an option instead of with a simple parameter.

31.13 DCPS Information Repository

The table below shows the command line options for the `DCPSInfoRepo` server.

Table 31-17 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

Table 31-17 DCPS Information Repository Options

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.

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 Simple 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 31.13.1 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 31-18 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 InfoRepo 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 InfoRepo can be reincarnated to a prior state.

31.13.1 Repository Federation

Note *Repository federation is a new feature in OpenDDS 1.2 and should be considered experimental at this time.*

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 1.2 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`, has a value of 1382379631 (0x5265706f), and should not be used by applications for data distribution.

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 31.8.3.

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 31-17. 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 31.13.1.1 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.

31.13.1.1 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 `federation` and is located in the `$DDS_ROOT/bin` directory. It has a command format syntax of:

```
federation <cmd> <arguments>
```

Where each individual command has its own format as shown in Table 31-19. 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 31-19 Federation Management Command

Command	Syntax	Description
`join`	`federation 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`	`federation 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`	`federation 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.
`help`	`federation 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:

```
federation join 2112 otherhost:1812
```

This generates a CORBA object reference of `corbaloc:iiop: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:iiop:localhost:2112/Federator`.
A full description of the command arguments are shown in Table 31-20

Table 31-20 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.
`<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.

31.13.1.2 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 31.4 configured to use the federated repositories.

Configuring the Federation Example

There are two configuration files to create for this example one each for the message publisher and subscriber. These are extensions of the configurations from the previous examples with some slight modifications.

The Message Publisher configuration pub.ini for this example is as follows:

```
[common]
DCPSDebugLevel = 0

[domain/information]
DomainId = 411
DomainRepoKey = 1

[repository/primary]
RepositoryKey = 1
RepositoryIor = corbaloc:iiop:localhost:2112/InfoRepo

[repository/secondary]
RepositoryKey = 2
RepositoryIor = file://repo.ior

[transport_impl_1]
transport_type = SimpleTcp
local_address = localhost:4444
```

Note that the DCPSInfo attribute/value pair has been removed from the [common] section. This has been replaced by the [domain/user] section as described in 31.8.3. The user domain is 411, 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 [transport_impl_1] section and programming for the transport is unchanged from the earlier example. No change to the Message Publisher application code is required.

The Subscriber process is configured with the sub.ini file as follows:

```
[common]
DCPSDebugLevel = 0

[domain/information]
DomainId = 411
DomainRepoKey = 1

[repository/primary]
RepositoryKey = 1
RepositoryIor = file://repo.ior

[repository/secondary]
RepositoryKey = 2
RepositoryIor = corbaloc:iiop:localhost:2112/InfoRepo

[transport_impl_1]
transport_type = SimpleTcp
local_address = localhost
```

Note that this is the same as the `pub.ini` file except for the transport specification section, which is the same as the sub.ini file in the previous example of 31.4.9. 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.

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 31.4.10.

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 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 31.4.10.