TAO Developer's Guide Excerpt

This is an excerpt from the comprehensive 2-volume *TAO Developer's Guide* provided by Object Computing Incorporated. Information on how to obtain this guide can be found at

`http://www.theaceorb.com/product/index.html#TAO Developers Guide`.

CHAPTER 31

Data Distribution Service

31.1 Introduction

The OMG Data Distribution Service for Real-Time Systems specification (OMG Document formal/04-12-02) 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 components: 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/04-12-02) as it contains in-depth coverage of all the service's features.

OpenDDS is the open-source C++ implementation of OMG's v1.0 DDS specification developed by OCI. It is available for download from http://download.ociweb.com/OpenDDS/ and is compatible with recent patches of TAO version 1.4a, 1.5a and 1.5.x.

Note *OpenDDS currently implements a subset of the DCPS layer. None of the DLRL functionality are currently implemented.*

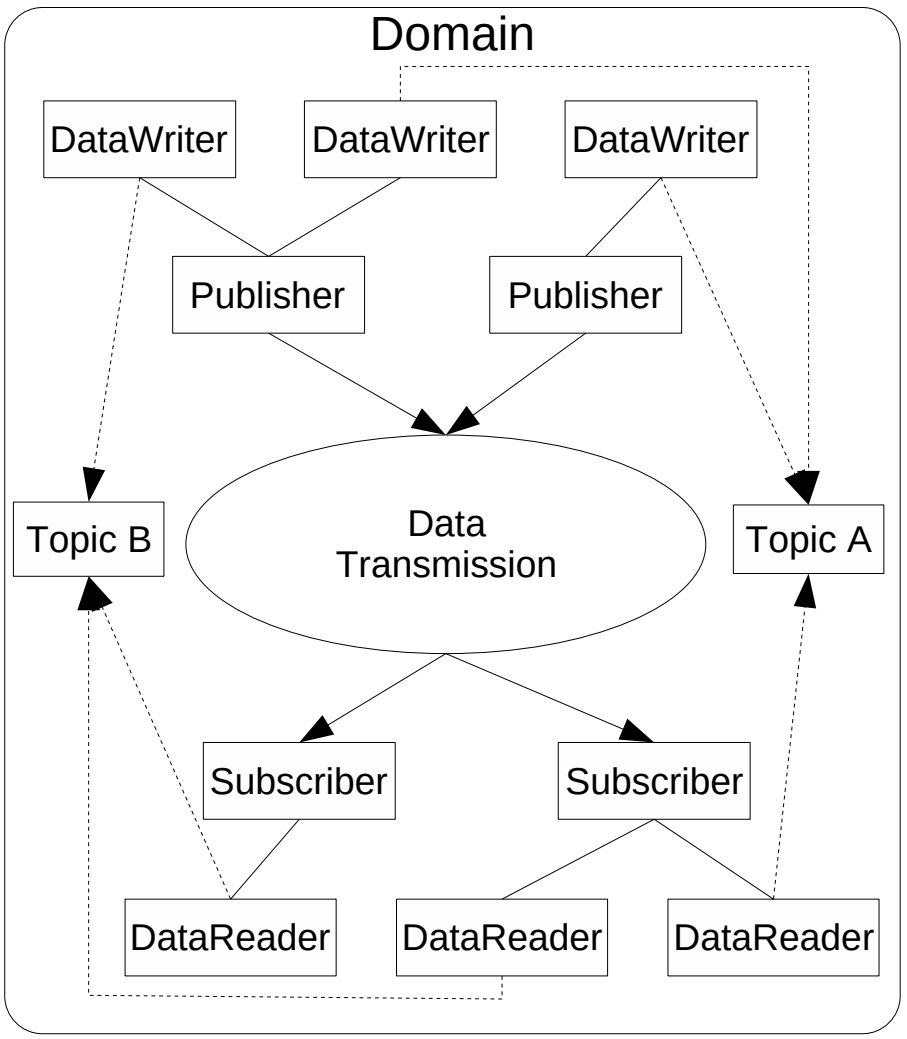
31.2 DCPS Overview

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 classes 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 specify 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 the 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 31.6.

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 does not currently support conditions.*

Conditions and wait-sets allow an alternative to listeners in detecting events of interest in DDS. The general pattern is

- The application creates a specific kind of condition object, such as a Read 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

OpenDDS is working towards the Minimum Compliance 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.

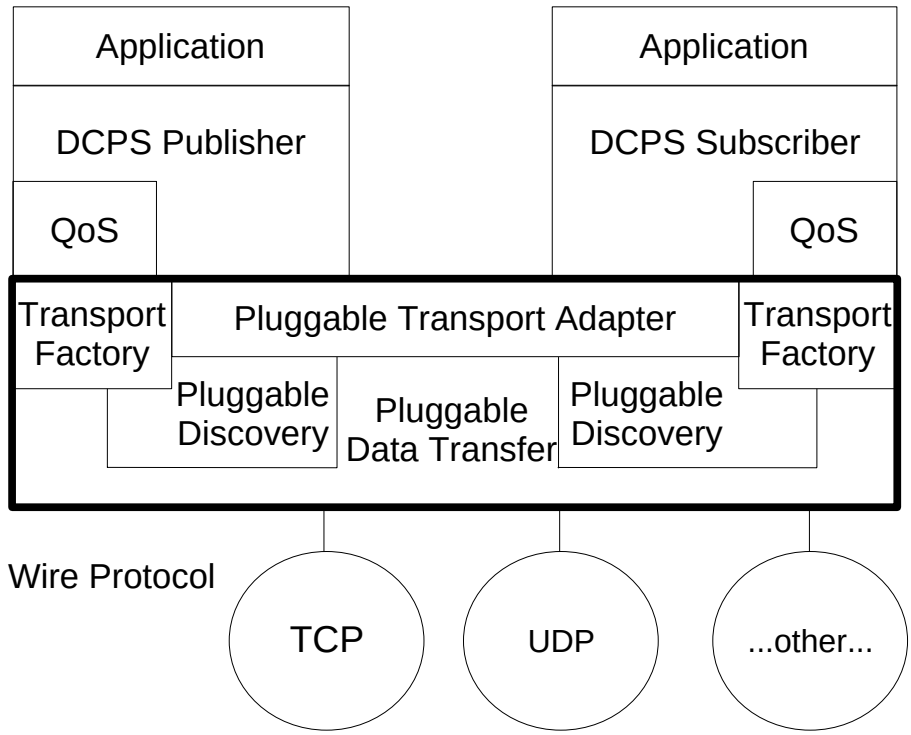


Figure 31-3 OpenDDS Pluggable Transport Framework

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 is a CORBA server that acts as the intermediary or broker between the publisher and subscriber. 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 isn't involved in data propagation, its role being limited in scope to publisher/subscriber association establishment.

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 31.6.

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` location as well as transport implementations for publishers and subscribers. The complete set of configuration settings is described in 31.7.

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 DDS 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 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 DDS. 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 DDS example.

The `DCPS_DATA_KEY` pragma identifies a field of the DDS 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. The 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. 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 DDS 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 DDS 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 `DCPS_DATA_TYPE` pragma encountered. The three files all begin with the data type name (without enclosing scopes) and are appended as follows:

- <type>`TypeSupport.idl`
- <type>`TypeSupportImpl.h`
- <type>`TypeSupportImpl.cpp`

For example, running `dcps_ts.pl` as follows:

```
dcps_ts.pl Messenger.idl
```

generates `MessageTypeSupport.idl`, `MessageTypeSupportImpl.h`, and `MessageTypeSupportImpl.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 DDS 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 31.11.

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 >> MessageTypeSupport.idl MessageTypeSupportImpl.h MessageTyp\
eSupportImpl.cpp
  }

  IDL_Files {
    Messenger.idl
    MessageTypeSupport.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

  IDL_Files {
    Messenger.idl
    MessageTypeSupport.idl
  }

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

The `dcpsexe_with_tcp` project links in the DCPS library.

31.4.3 Starting the DCPS Information Repository

The DCPS Information Repository server is found in `$DDS_ROOT/dds/InfoRepo`. This process hosts the `DCPSInfo` CORBA object that is the entry point for all DDS functionality. This object is mapped

against the key string 'DCPSInfoRepo' in the process IORTable. Thus a corbaloc url e.g.

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

can be used to locate the DCPSInfo object. The server also writes out the object IOR to a file, which can also be used to bootstrap clients. We can alter the file name used with the `-o` option.

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

The full set of command line options for this server are documented in 31.12.

31.4.4 A Simple Message Publisher

In this section we will step through the setup of a simple DDS 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). To see the full code see 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 this process as a DDS 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 DDS service uses as well as the service itself. This allows us to pass `ORB_init()` options on the command line as well as DDS

configuration options of the form `-DCPS*`. These options are fully described in 31.11. 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

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

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. Next, we get 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::TopicQos topic_qos;
participant->get_default_topic_qos(topic_qos);
DDS::Topic_var topic =
  participant->create_topic ("Movie Discussion List",
                             type_name.in (),
                             topic_qos,
                             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 may 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 gets the transport implementation from the singleton transport factory, called `TheTransportFactory`. The `OpenDDS::DCPS::AUTO_CONFIG` argument indicates that we're 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 -- whether it uses TCP, UDP, what its endpoints are, etc.

The function above can also be used to create a transport implementation with default configuration.

```
  OpenDDS::DCPS::TransportImpl_rch tcp_impl =

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

This code 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 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 API's 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);
```

This 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 ID's.

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 example 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;

      if (tcp_impl->configure(tcp_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 =
      OpenDDS::DCPS::reference_to_servant< OpenDDS::DCPS::PublisherImpl> (pub);
    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 do the `attach_transport()` on the servant and not on the reference. The reference implements the OMG-defined Publisher interface and lacks any OpenDDS specific pluggable transport functionality. The `reference_to_servant` function template is a convenience function for navigating from the object reference of an entity to the servant.

31.4.4.5 DataWriter creation and instance registration

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

```
// Create the datawriter
DDS::DataWriterQos dw_qos;
pub->get_default_datawriter_qos (dw_qos);

DDS::DataWriter_var dw =
  pub->create_datawriter(topic.in (),
                         dw_qos,
                         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'll 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

A lot of the subscriber's code is identical or analogous to the publisher that we just finished exploring. We will progress quickly through the similar parts and refer you to the discussion above for details. The beginning of the subscriber is identical to the publisher as we initialize the service and join our domain.

31.4.5.1 Participant Initialization

```
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::TopicQos topic_qos;
participant->get_default_topic_qos(topic_qos);
DDS::Topic_var topic =
  participant->create_topic("Movie Discussion List",
                            type_name.in (),
                            topic_qos,
                            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

Now we 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 can 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 =
      OpenDDS::DCPS::reference_to_servant< OpenDDS::DCPS::SubscriberImpl>
(sub.in());
    if (0 == sub_impl) {
      std::cerr << "Failed to obtain subscriber servant\n" << 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 local object. The `DataReaderListenerImpl` class is shown in the next subsection.

```
    // activate the listener
    DataReaderListenerImpl        listener_servant;
    DDS::DataReaderListener_var listener =
      ::OpenDDS::DCPS::servant_to_reference (&listener_servant);
    if (CORBA::is_nil(listener.in())) {
      std::cerr << "listener is nil." << std::endl;
      return 1;
    }
```

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

```
    // Create the Datareaders
    DDS::DataReaderQos dr_qos;
    sub->get_default_datareader_qos (dr_qos);
    DDS::DataReader_var dr = sub->create_datareader(topic.in (),
```

```
                                                    dr_qos,
                                                    listener.in ());
    if (CORBA::is_nil(dr.in())) {
      std::cerr << "create_datareader failed." << std::endl;
      return 1;
    }
```

Now this thread is free to do any application work we want while our listener object should get called from one of the service's threads when a sample is available.

31.4.6 The Data Reader Listener Servant

Our listener servant implements the `DDS::DataReaderListener` defined by the DDS specification. The DataReaderListener gets wrapped within a DCPS::LocalObject which resolves ambigously-inherited members like _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);
  };
};
```

The servant class stubs out most of these listener operations with simple print statements. The only operation truly 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 and, if successful, prints out each of the messages fields.

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

    if (status == DDS::RETCODE_OK) {
      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;
      std::cout << "SampleInfo.sample_rank = " << si.sample_rank << std::endl;
    } else if (status == DDS::RETCODE_NO_DATA) {
      std::cerr << "ERROR: reader received DDS::RETCODE_NO_DATA!" << std::endl;
    } else {
      std::cerr << "ERROR: read Message: Error: " <<  status << std::endl;
    }
  } catch (CORBA::Exception& e) {
    std::cerr << "Exception caught in read:" << std::endl << e << std::endl;
  }
}
```

If additional samples are available, the service calls this function again. Reading values a single sample at a time is not the most efficient way to process incoming data. The Data Reader interface gives a number of different options for processing data in a more efficient manner. We'll look at some of these operations in 31.5.

31.4.7 Cleaning up in DDS Clients

After we are finished in the publisher and subscriber, we can use the following code to clean up the DDS-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 Configuring the Example

OpenDDS includes a file-based configuration mechanism. With it, a user may 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'll 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've configured publisher to use the Simple TCP transport and to listen on port 4444 on the loopback network interface.

Recall that 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'll examine the subscriber's configuration file, sub.ini:

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

[transport_impl_1]
transport_type=SimpleTcp
local_address=localhost:2222
```

We've configured subscriber to also use the Simple TCP transport and to listen on port 2222 on the loopback network interface.

For a complete description of the OpenDDS configuration parameters, please see 31.7.

31.4.9 Running the Example

This example can be run with the following commands.

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

```
./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.

Running each of these commands in its own window should enable you to most easily understand the output. 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. If we don't want this to happen we need to either ensure that the subscriber can keep up or change the QoS settings.

31.5 Data Handling Optimizations

31.5.1 Reading Multiple Samples

The DDS specification supplies 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.

31.5.2 Zero-Copy Read

The read and take operations that return a sequence of samples provide the user with the option of getting 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 user code specifies its desire to use zero-copy read 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 will be set to the number of samples read. For the zero-copy reads the max_len will be set to a value >= length.

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

```
dr_impl->return_loan (the_data, the_info);
```

This call will result in the sequences' max_len set to 0 and its owns set to false so the same sequences could be used for another zero-copy read call

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 user 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/take's max_samples parameter is larger than this value, then the maximum samples returned by read/take call will be limited by this parameter of the sequence constructor.

Although the user can change the length of a zero-copy sequence, by calling length(len), its use discouraged because this call results in copying the data and creating a single-copy sequence of samples.

31.6 QoS Policies

The above 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.

31.6.1 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 31.6.2.

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).

Note *OpenDDS does not currently support changing the QoS of an existing entity, so all QoS policies must be applied when the entity is created.*

31.6.1.1 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 can't be contacted 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_changed()` callback is never called on the publisher side.

31.6.1.2 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 section 31.6.1.5 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

31.6.1.3 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.

31.6.1.4 Durability

The Durability policy controls whether Data Writers should maintain samples after they have been sent to known subscribers. OpenDDS currently supports Volatile and Transient Local values of the Durability QoS.

By default the Durability is Volatile which 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.

OpenDDS does not currently support the Transient or Persistent values for the Durability QoS.

31.6.1.5 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.

31.6.2 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.

31.6.2.1 Entity Factory

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

31.6.2.2 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.

31.6.3 Policy Example

Here are 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.

31.7 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 `DDS/DCPS/Service_Participant.h` for details.

31.7.1 Common Configuration Settings

The `[common]` section of the OpenDDS configuration file contains settings for attributes such as debugging output, the 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 31-1 Common Configuration Settings

Option	Description	Default
`DCPSDebugLevel` *n*	Integer value that controls the amount of debug information that service prints. Valid values are 0 through 10. See 31.10.1, "DCPS Level Logging" for details.	`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`

Table 31-1 Common Configuration Settings

Option	Description	Default
`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`
`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

Table 31-1 Common Configuration Settings

Option	Description	Default
`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 debug level is useful for diagnosing issues with processes interacting with DDS. A debug level of 10 gives the maximum amount of debug information.

The `DCPSInfoRepo` option's value is passed to `ORB::string_to_object()` and can be of any CORBA 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.

31.7.2 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'll recall that, 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.

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

Table 31-2 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
`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`

Table 31-2 Transport Configuration Settings

Option	Description	Default
`queue_initial_pools` *n*	The initial number of pools for the backpressure queue. The default settings of the two backpressure queue values preallocate space for 50 messages (5 pools of 10 messages).	`5`
`max_packet_size` *n*	The maximum size of a transport packet, including its transport header, sample header, and sample data.	`2147481599`
`max_samples_per_packet` *n*	Maximum number of samples in a transport packet.	`10`
`optimum_packet_size` *n*	Transport packets greater than this size will be sent over the wire even if there are still queued samples to be sent. This value may impact performance depending on your network configuration and application nature.	`4096`
`thread_per_connection` *0|1*	Enable or disable the thread per connection send strategy.	`0(disabled)`
`keep_link` *0|1*	Enable or disable link releasing when last association is removed. This may be useful in cases where data readers/data writers are added and removed frequently. Preventing removal of the connections will reduce the overhead of re-establishment when adding a reader – writer association.	`0(disabled)`

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.

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 31-3 SimpleTcp Configuration Settings

Option	Description	Default
`local_address` *`host:port`*	Hostname and port of the connection acceptor. The default value is the hostname from the hostname() system call and port 0, which means the OS will choose the port.	`host: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 increase latency.	`0`
`conn_retry_initial_delay` *`n`*	Initial delay (milliseconds) for reconnect attempt. The first attempt is at the time lost connection is detected. The second attempt will be after this delay if first retry connect fails.	`500`
`conn_retry_backoff_multiplier` *`n`*	The backoff multiplier for reconnection strategy. The third and so on reconnect will be this value * the previous delay. Hence with conn_retry_initial_delay=500 and conn_retry_backoff_multiplier=1.5, the second reconnect attempt will be at 0.5 seconds after 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. If the value is zero, the default, then this check will not be made.	`0`

Table 31-3 SimpleTcp Configuration Settings

Option	Description	Default
`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`

Reconnection options

When a TCP connection gets closed DDS attempts to reconnect. The reconnection process is:

- Upon detecting a lost connection immediately attempt reconnect.
- If that fails then wait `conn_retry_initial_delay` milliseconds and attempt reconnect.
- while haven't tried more than `conn_retry_attempts`,
- wait (previous wait time * `conn_retry_backoff_multiplier`) milliseconds and attempt to reconnect

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 31-4 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`

Table 31-4 SimpleUdp and SimpleMcast Common Configuration Settings

Option	Description	Default
`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 31-5 SimpleUdp Configuration Settings

Option	Description	Default
`local_address` *host:port*	Address and port at which the transport reads UDP packets.	none

In addition of 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 31-6 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 used the “default” NIC.	none
`multicast_group_address` *host:port*	Address at which the publisher sends multicast packet to and subscriber receives multicast packets from. Use ACE default multicast address as default.	`224.9.9.2:20001`
`receiver` *0|1*	Flag indicates if the transport is receiving side (subscriber) or sending side (publisher). Defaults to be 0 - as publisher side.	`0`

ReliableMcast transport builds data reliability upon the multicast protocol. It supports options similar to the SimpleMcast transport. It accepts the following additional parameters for tuning reliability and performance:

Table 31-7 ReliableMcast Configuration Settings

Option	Description	Default
`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 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.

31.8.1 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 default NIC. Therefore, you may wish to explicitly set the address if you have a 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 default NIC address and a random

port will be used. Note that this parameter also applies to unreliable datagram transports with the same restrictions.

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.

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 section 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 section 31.4. Configuring subscribers and publishers should be identical, but different addresses/ports should be assigned to each Transport Implementation.

Note that you can modify and apply the above configuration technique when using the other available transport libraries.

31.8.2 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 samples to about 64 KB. Attempting to send a sample over 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.

31.8.2.1 SimpleUDP Transport

Let's look at 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).

31.8.2.2 SimpleMcast Transport

Let's look at a SimpleMcast transport configuration example, first 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.

Next, let's examine 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.

31.8.3 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 ""
```

31.9 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 31.1). 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 *Built-In topics 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 31.9.2 on detail the data published for each of the four built-in topics. Following these sections is some example code that shows how to read from a built-in topic.

31.9.1 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 with option "-features built_in_topics=0".
   ```
   mwc.pl -type <yourtype> -features built_in_topics=0
   DDS.mwc
   ```
   This has the same effect as adding line "built_in_topics=0" to the file $DDS_ROOT/MPC/config/default.features
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).

31.9.2 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, making each one its own instance within this topic.

31.9.3 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;
  DeadlineQosPolicy 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.

31.9.4 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.

31.9.5 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.

31.9.6 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.

31.10 Logging

By default, the OpenDDS framework will only log when there is a serious error that is not indicated by a return code. A DDS user may increase the amount of logging by controls at the DCPS and Transport levels.

31.10.1 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 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

31.10.2 Transport Level Logging

DDS has a stratified logging mechanism where the most critical errors will always be output to stderr. Most DDS log messages fall in the range of 0-5 with 0 being the highest priority messages.

Turning on logging is a two stage process where firstly the DDS core needs to be built to the desired logging level. This is done by setting the build macro DDS_BLD_DEBUG_LEVEL between 0-5. For run-time optimization the

default is 1. Once built, the application macro DDS_RUN_DEBUG_LEVEL can be used to tweak the actual debug level between 0-DDS_BLD_DEBUG_LEVEL.

The below code will output the maximum logging currently compiled in.

```
int
main (int argc, ACE_TCHAR *argv[])
{
    DDS_RUN_DEBUG_LEVEL = DDS_BLD_DEBUG_LEVEL;
```

31.11 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 type it finds, such as `xyz`, it generates three files: `xyzTypeSupport.idl`, `xyzTypeSupportImpl.h`, and `xyzTypeSupportImpl.cpp`. Because a single IDL file could find multiple DCPS-enable types, an invocation of the script may generate a multitude of files. 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 31-8 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

Table 31-8 dcps_ts.pl Command Line Options

Option	Description	Default
`--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

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 IDL file to process with an option instead of with the simple parameter.

31.12 DCPS Information Repository

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

Table 31-9 DCPS Information Repository Options

Option	Description	Default
`-o` *`file`*	Write the IOR of the `DCPSInfo` object to the specified file	`repo.ior`
`-d` *`file`*	Load domain IDs from the specified file	`domain_ids`
`-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)
`-?`	Display the command line usage and exit	N/A

DDS clients usually 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.

The domain file is a simple list of integer domain ID values, one per line. Any domain IDs used that do not appear in this file are assumed to be invalid and result in an `INVALID_DOMAIN` exception being thrown.

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.

File persistence is implemented as an ACE Service object and is controlled via service config directives. Currently available configuration options are:

Table 31-10 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.