Data Distribution Service for Real-Time Systems Specification (DDS)

This OMG document accompanies the the RTF Recommendation and Report for this specification.
The base document for the revised specification is formal/07-01-01~~pte/2014-05-05~~.
This document is the result of applying to the adopted specification the issues resolved by the RTF.

Note: All changes in this document relative to the base document are the result of applying the resolution of issue 19366.

Contents

Preface

Object Management Group

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling, and vertical domain frameworks. A catalog of all OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML

- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Suite 100
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- Objective Interface Systems, Inc.
- Real-Time Innovations, Inc.
- THALES
- The Mitre Corporation
- University of Toronto

Overview 1

Contents

This chapter contains the following sections.

1.1 Introduction

~~This~~The DDS specification describes ~~two levels of interfaces:~~ a Data-Centric Publish-Subscribe (DCPS) model for distributed application communication and integration. This specification defines both the Application Interfaces (APIs) and the Communication Semantics (behaviour and quality of service) that enable the efficient delivery of information from information producers to matching consumers.

The purpose of the DDS specification can be summarized as enabling the “Efficient and Robust Delivery of the Right Information to the Right Place at the Right Time.”

- ~~A lower DCPS (Data-Centric Publish-Subscribe) level that is targeted towards the efficient delivery of the proper information to the proper recipients.~~
- ~~An optional higher DLRL (Data Local Reconstruction Layer) level, which allows for a simple integration of the Service into the application layer.~~

The expected application domains require DCPS to be high-performance and predictable as well as efficient in its use of resources. To meet these requirements it is important that the interfaces are designed in such a way that they:

- A~~a~~llow the middleware to pre-allocate resources so that dynamic resource allocation can be reduced to the minimum,

- Aavoid properties that may require the use of unbounded or hard-to-predict resources, and
- Mminimize the need to make copies of the data.

EvenDDS uses at the DCPS level, typed interfaces (i.e., interfaces that take into account the actual data types) are preferred to the extent possible. Typed interfaces offer the following advantages:

- They are simpler to use: the programmer directly manipulates constructs that naturally represent the data.
- They are safer to use: verifications can be performed at compile time.
- They can be more efficient: the execution code can rely on the knowledge of the exact data type it has in advance, to e.g., pre-allocate resources.

It should be noted that the decision to use typed interfaces implies the need for a generation tool to translate type descriptions into appropriate interfaces and implementations that fill the gap between the typed interfaces and the generic middleware.

QoS (Quality of Service) is a general concept that is used to specify the behavior of a service. Programming service behavior by means of QoS settings offers the advantage that the application developer only indicates ‘what’ is wanted rather than ‘how’ this QoS should be achieved. Generally speaking, QoS is comprised of several QoS policies. Each QoS policy is then an independent description that associates a name with a value. Describing QoS by means of a list of independent QoS policies gives rise to more flexibility.

This specification is designed to allow a clear separation between the publish and the subscribe sides, so that an application process that only participates as a publisher can embed just what strictly relates to publication. Similarly, an application process that participates only as a subscriber can embed only what strictly relates to subscription.

1.2 Purpose

Many real-time applications have a requirement to model some of their communication patterns as a pure data-centric exchange, where applications publish (supply or stream) “data” which is then available to the remote applications that are interested in it. Relevant real-time applications can be found in C4I, industrial automation, distributed control and simulation, telecom equipment control, sensor networks, and network management systems. More generally, any application requiring (selective) information dissemination is a candidate for a data-driven network architecture.

Predictable distribution of data with minimal overhead is of primary concern to these real-time applications. Since it is not feasible to infinitely extend the needed resources, it is important to be able to specify the available resources and provide policies that allow the middleware to align the resources to the most critical requirements. This necessity translates into the ability to control Quality of Service (QoS) properties that affect predictability, overhead, and resource utilization.

The need to scale to hundreds or thousands of publishers and subscribers in a robust manner is also an important requirement. This is actually not only a requirement of scalability but also a requirement of flexibility: on many of these systems, applications are added with no need/possibility to reconstruct the whole system. Data-centric communications decouples senders from receivers; the less coupled the publishers and the subscribers are, the easier these extensions become.

Distributed shared memory is a classic model that provides data-centric exchanges. However, this model is difficult to implement efficiently over a network and does not offer the required scalability and flexibility. Therefore, another model, the ***Data-Centric Publish-Subscribe (DCPS)*** model, has become popular in many real-time applications. This model builds on the concept of a "global data space" that is accessible to all interested applications. Applications that want to contribute information to this data space declare their intent to become "Publishers." Similarly, applications that want to access portions of this data space declare their intent to become "Subscribers." Each time a Publisher posts new data into this "global data space," the middleware propagates the information to all interested Subscribers.

Underlying any data-centric publish subscribe system is a *data model*. This model defines the "global data space" and specifies how Publishers and Subscribers refer to portions of this space. The data-model can be as simple as a set of unrelated *data-structures*, each identified by a *topic* and a *type*. The topic provides an identifier that uniquely identifies some data items within the global data space[1]. The type provides structural information needed to tell the middleware how to manipulate the data and also allows the middleware to provide a level of type safety. However, the target applications often require a higher-level data model that allows expression of aggregation and coherence relationships among data elements.

~~Another common need is a ***Data-Local Reconstruction Layer (DLRL)*** that automatically reconstructs the data locally from the updates and allows the application to access the data 'as if' it were local. In that case, the middleware not only propagates the information to all interested subscribers but also updates a local copy of the information.~~

Prior to the adoption of the DDS specification t~~T~~here ~~are~~were commercially~~-~~ available products that implemented ~~DCPS~~many of these features ~~fully and the DLRL partially~~ (among them, NDDS from Real-Time Innovations and Splice from THALES Naval Nederland); however, these products ~~are~~were proprietary and d~~o~~id not offer standardized interfaces and behaviors. ~~that would allow portability of the applications built upon them.~~ ***The purpose of ~~this~~the DDS specification is to define the~~offer those~~ standardized interfaces and behaviors~~:~~ that enable applicaton portability***. Since DDS' adoption, at least ten compliant implementations have been developed. See http://portals.omg.org/dds/category/web-links/vendors.

1. In addition to topic and type, it is sometimes desirable for subscriptions to further refine the data they are interested in based on the content of the data itself. These so called content-based subscriptions are gaining popularity in large-scale systems.

This specification focuses on the portability of applications using the Data-Distribution Service. ~~This is consistent with the requirements expressed in the RFP.~~ Wire-protocol interoperability between vendor implementations is ~~planned as an extension.~~covered in a different OMG specification: The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol."

Data-Centric Publish-Subscribe (DCPS) 2

Contents

This chapter contains the following sections.

This chapter describes ~~the mandatory~~ DCPS ~~layer~~. ~~The~~ DCPS ~~layer provides~~defines the functionality ~~required for~~used by an application to publish and subscribe to the values of data objects.

It allows:

- Publishing applications to identify the data objects they intend to publish, and then provide values for these objects.
- Subscribing applications to identify which data objects they are interested in, and then access their data values.
- Applications to define topics, to attach type information to the topics, to create publisher and subscriber entities, to attach QoS policies to all these entities and, in summary, to make all these entities operate.

The description is organized into two subsections:

- The Platform Independent Model (PIM).
- The Platform Specific Model (PSM) for the OMG IDL platform based on the PIM.

2.1 Platform Independent Model (PIM)

2.1.1 Overview and Design Rationale

2.1.1.1 Format and conventions

The purpose of this subsection is to provide an operational overview of the DCPS PIM. To do so, it introduces many terms. Some of them are common terms whose meaning, in the context of publish-subscribe, is different from common usage. In cases where it is deemed appropriate, such terms will be *italicized*. Other terms are unique to publish-subscribe and/or to this specification, and are incorporated as key elements of the Class Model. The first time such terms are used, they will be formatted with ***Bold-italics***[1]. Subsequent occurrences may not be highlighted in any way.

In addition to the UML diagrams, all the classes that constitute the Service are documented using tables. The format used to document these classes is shown below:

<class name>		
attributes		
<attribute name>	<attribute type>	
...	...	
operations		
<operation name>		<return type>
	<parameter>	<parameter type>
	...	...
...		...

The operation <parameter> can contain the modifier “in,” “out,” or “inout” ahead of the parameter name. If this modifier is omitted, it is implied that the parameter is an “in” parameter.

In some cases the operation parameters or return value(s) are a collection with elements of a given <type>. This is indicated with the notation “<type> [ ].” This notation does not imply that it will be implemented as an array. The actual implementation is defined by the PSM: it may end up being mapped to a sequence, a list, or other kind of collection.

For example, the class named ‘MyClass’ below has a single attribute, named ‘my_attribute’ of type ‘long’ and a single operation ‘my_operation’ that returns a long. The operation takes four parameters. The first, ‘param1,’ is an output parameter of type

1. In this case, the written name is exactly the one of the corresponding class, which forbids the use of the plural. In case this would lead to ambiguity, it has been followed by 'objects' to state that there may not be only one of these.

long; the second, ‘param2,’ an input-output parameter of type long; the third, ‘param3,’ is an input parameter (the “in” modifier is implied by omission) of type long; and the fourth, ‘param4,’ is also an input parameter of type collection of longs[2].

MyClass		
attributes		
my_attribute	long	
operations		
my_operation		long
	out: param1	long
	inout: param2	long
	param3	long
	in: param4	long []

At the PIM level we have modeled errors as operation return codes typed ***ReturnCode_t***. Each PSM may map these to either return codes or exceptions. The complete list of return codes is indicated below.

Return codes	
OK	Successful return.
ERROR	Generic, unspecified error.
BAD_PARAMETER	Illegal parameter value.
UNSUPPORTED	Unsupported operation. Can only be returned by operations that are optional.
ALREADY_DELETED	The object target of this operation has already been deleted.

2. That is, a collection where the type of each element is ‘long’.

Return codes	
OUT_OF_RESOURCES	Service ran out of the resources needed to complete the operation.
NOT_ENABLED	Operation invoked on an ***Entity*** that is not yet enabled.
IMMUTABLE_POLICY	Application attempted to modify an immutable ***QosPolicy***.
INCONSISTENT_POLICY	Application specified a set of policies that are not consistent with each other.
PRECONDITION_NOT_MET	A pre-condition for the operation was not met.
TIMEOUT	The operation timed out.
ILLEGAL_OPERATION	An operation was invoked on an inappropriate object or at an inappropriate time (as determined by policies set by the specification or the Service implementation). There is no precondition that could be changed to make the operation succeed.
NO_DATA	Indicates a transient situation where the operation did not return any data but there is no inherent error.

Any operation with return type ***ReturnCode_t*** may return OK, ERROR, or ILLEGAL_OPERATION. Any operation that takes an input parameter may additionally return BAD_PARAMETER. Any operation on an object created from any of the factories may additionally return ALREADY_DELETED. Any operation that is stated as optional may additionally return UNSUPPORTED. The return codes OK, ERROR, ILLEGAL_OPERATION, ALREADY_DELETED, UNSUPPORTED, and BAD_PARAMETER are the standard return codes and the specification won't mention them explicitly for each operation. Operations that may return any of the additional (non-standard) error codes above will state so explicitly.

It is an error for an application to use an Entity that has already been deleted by means of the corresponding delete operation on the factory. If an application does this, the result is unspecified and will depend on the implementation and the PSM. In the cases where the implementation can detect the use of a deleted entity, the operation should fail and return ALREADY_DELETED.

2.1.1.2 Conceptual Outline

2.1.1.2.1 Overview

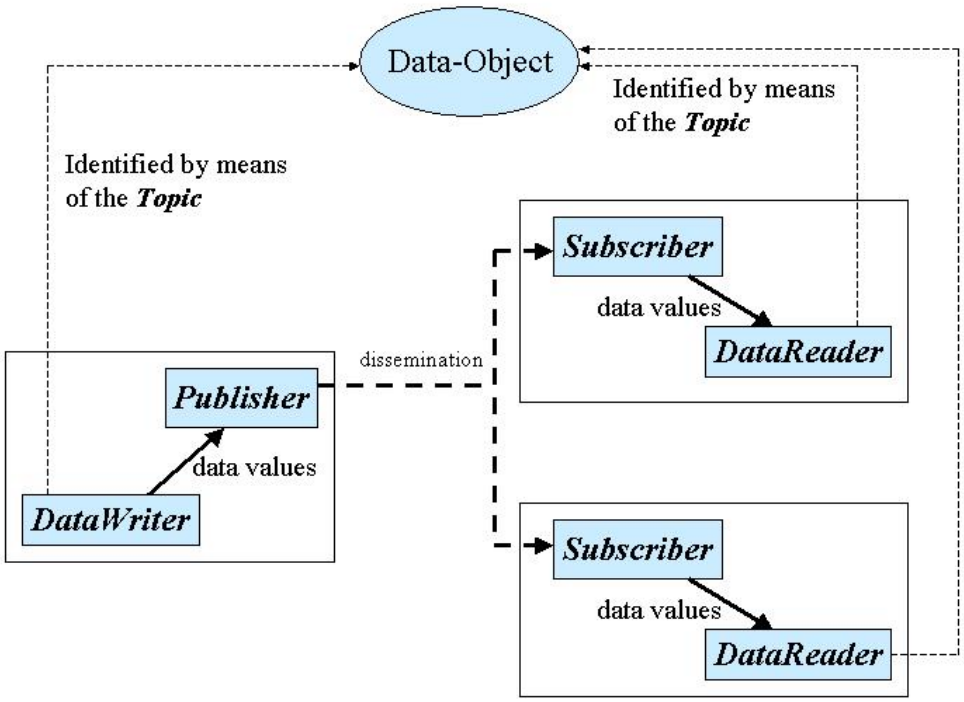

Figure 2-1 Overview

Information flows with the aid of the following constructs[3]: ***Publisher*** and ***DataWriter*** on the sending side, ***Subscriber***, and ***DataReader*** on the receiving side.

- A ***Publisher*** is an object responsible for data distribution. It may publish data of different data types. A ***DataWriter*** acts as a typed[4] accessor to a publisher. The ***DataWriter*** is the object the application must use to communicate to a publisher the existence and value of data-objects of a given type. When data-object values have been communicated to the publisher through the appropriate data-writer, it is the publisher's responsibility to perform the distribution (the publisher will do this according to its own QoS, or the QoS attached to the corresponding data-writer). A

3. All those constructs are local to the application part. Actually they play the role of *proxies* to the service.

4. 'typed' means that each ***DataWriter*** object is dedicated to one application data-type.

publication is defined by the association of a data-writer to a publisher. This association expresses the intent of the application to publish the data described by the data-writer in the context provided by the publisher.

- A ***Subscriber*** is an object responsible for receiving published data and making it available (according to the Subscriber's QoS) to the receiving application. It may receive and dispatch data of different specified types. To access the received data, the application must use a typed ***DataReader*** attached to the subscriber. Thus, a *subscription* is defined by the association of a data-reader with a subscriber. This association expresses the intent of the application to subscribe to the data described by the data-reader in the context provided by the subscriber.

Topic objects conceptually fit between publications and subscriptions. Publications must be known in such a way that subscriptions can refer to them unambiguously. A ***Topic*** is meant to fulfill that purpose: it associates a name (unique in the domain[5]), a data-type, and QoS related to the data itself. In addition to the topic QoS, the QoS of the ***DataWriter*** associated with that ***Topic*** and the QoS of the ***Publisher*** associated to the ***DataWriter*** control the behavior on the publisher's side, while the corresponding ***Topic***, ***DataReader***, and ***Subscriber*** QoS control the behavior on the subscriber's side.

When an application wishes to publish data of a given type, it must create a ***Publisher*** (or reuse an already created one) and a ***DataWriter*** with all the characteristics of the desired publication. Similarly, when an application wishes to receive data, it must create a ***Subscriber*** (or reuse an already created one) and a ***DataReader*** to define the subscription.

2.1.1.2.2 Overall Conceptual Model

The overall conceptual model is shown in Figure 2-2 on page 2-8. Notice that all the main communication objects (the specializations of ***Entity***) follow unified patterns of:

- Supporting QoS (made up of several ***QosPolicy***); QoS provides a generic mechanism for the application to control the behavior of the Service and tailor it to its needs. Each ***Entity*** supports its own specialized kind of QoS policies. The complete list of QoS policies and their meaning is described in Section 2.1.3, "Supported QoS," on page 2-102.
- Accepting a ***Listener***[6]; listeners provide a generic mechanism for the middleware to notify the application of relevant asynchronous events, such as arrival of data corresponding to a subscription, violation of a QoS setting, etc. Each DCPS entity

5. Broadly speaking, a domain represents the set of applications that are communicating with each other. This concept is defined more precisely in Section 2.1.1.2.2, "Overall Conceptual Model," on page 2-6 and Section 2.1.2.2.1, "DomainParticipant Class," on page 2-21.

6. This specification made the choice of allowing the attachment of only one Listener per entity (instead of a list of them). The reason for that choice is that this allows a much simpler (and, thus, more efficient) implementation as far as the middleware is concerned. Moreover, if it were required, implementing a listener that, when triggered, triggers in return attached 'sub-listeners,' can be easily done by the application.

supports its own specialized kind of listener. Listeners are related to changes in status conditions. This relationship is described in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

- Accepting a ***StatusCondition*** (and a set of ***ReadCondition*** objects for the ***DataReader***); conditions (in conjunction with ***WaitSet*** objects) provide support for an alternate communication style between the middleware and the application (i.e., wait-based rather than notification-based). The complete set of status conditions is described in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

All these DCPS entities are attached to a ***DomainParticipant***. A domain participant represents the local membership of the application in a domain. A *domain* is a distributed concept that links all the applications able to communicate with each other. It represents a communication plane: only the publishers and the subscribers attached to the same *domain* may interact.

DomainEntity is an intermediate object whose only purpose is to state that a ***DomainParticipant*** cannot contain other domain participants.

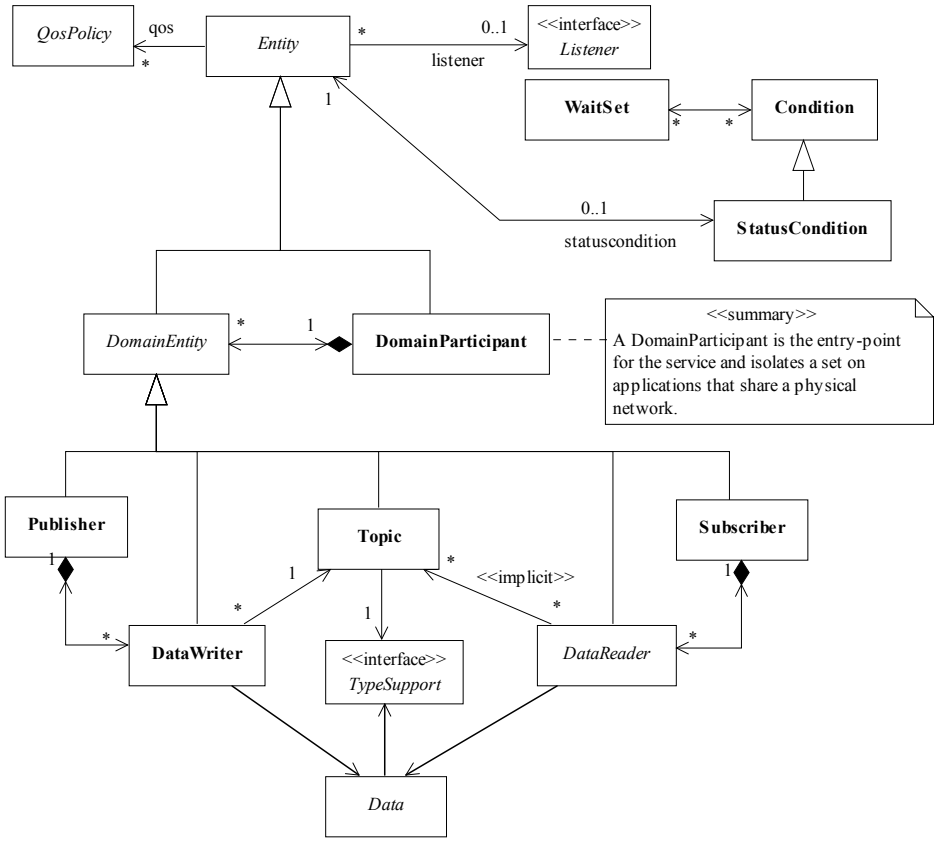

Figure 2-2 DCPS conceptual model

At the DCPS level, data types represent information that is sent atomically[7].

By default, each data modification is propagated individually, independently, and uncorrelated with other modifications. However, an application may request that several modifications be sent as a whole and interpreted as such at the recipient side. This functionality is offered on a ***Publisher***/***Subscriber*** basis. That is, these relationships can only be specified among ***DataWriter*** objects attached to the same ***Publisher*** and retrieved among ***DataReader*** objects attached to the same ***Subscriber***.

By definition, a ***Topic*** corresponds to a single data type. However, several topics may refer to the same data type. Therefore, a ***Topic*** identifies data of a single type, ranging from one single instance to a whole collection of instances of that given type. This is shown in Figure 2-3 for the hypothetical data-type “Foo.”

7. Note that the optional DLRL layer provides the means to break data-objects into separate elements, each sent atomically.

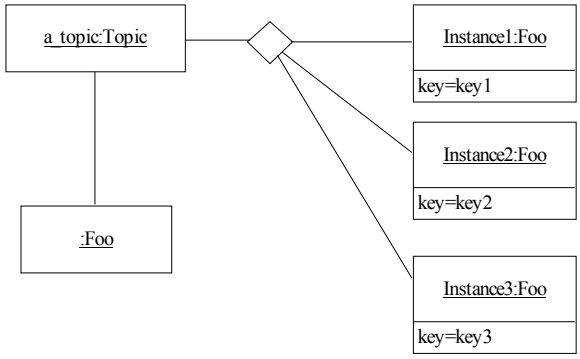

Figure 2-3 A topic can identify a collection of data-object instances

In case a set of instances is gathered under the same topic, different instances must be distinguishable. This is achieved by means of the values of some data fields that form the ***key*** to that data set. The ***key*** description (i.e., the list of data fields whose value forms the key) has to be indicated to the middleware. The rule is simple: *different data values with the same key value represent successive values for the same instance, while different data values with different key values represent different instances*. If no key is provided, the data set associated with the ***Topic*** is restricted to a single instance.

Topics need to be known by the middleware and potentially propagated. ***Topic*** objects are created using the create operations provided by ***DomainParticipant***.

The interaction style is straightforward on the publisher's side: when the application decides that it wants to make data available for publication, it calls the appropriate operation on the related ***DataWriter*** (this, in turn, will trigger its ***Publisher***).

On the subscriber's side however, there are more choices: relevant information may arrive when the application is busy doing something else or when the application is just waiting for that information. Therefore, depending on the way the application is designed, asynchronous notifications or synchronous access may be more appropriate. Both interaction modes are allowed, a ***Listener*** is used to provide a callback for synchronous access and a ***WaitSet*** associated with one or several ***Condition*** objects provides asynchronous data access.

The same synchronous and asynchronous interaction modes can also be used to access changes that affect the middleware communication status. For instance, this may occur when the middleware asynchronously detects an inconsistency. In addition, other middleware information that may be relevant to the application (such as the list of the existing topics) is made available by means of built-in topics that the application can access as plain application data, using built-in[8] data-readers.

8. These built-in data-readers should be provided with every implementation of the service. They are further described in Section 2.1.5, "Built-in Topics," on page 2-144.

2.1.2 PIM Description

The DCPS is comprised of five modules:

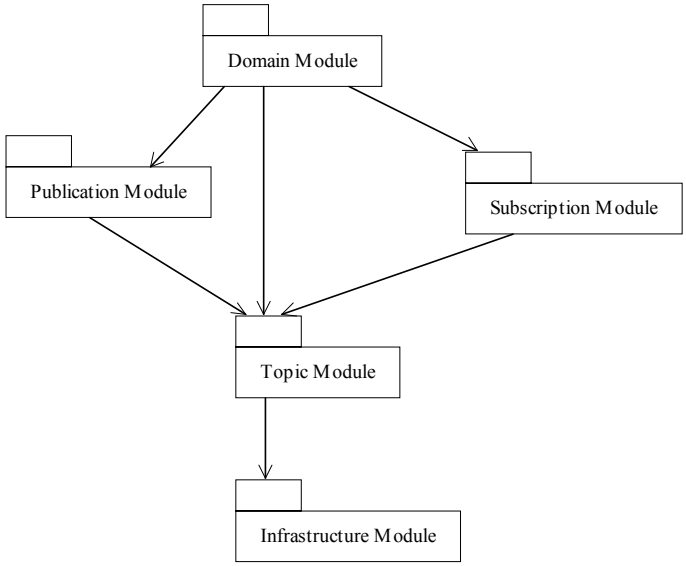

Figure 2-4 DCPS module breakdown

- The Infrastructure Module defines the abstract classes and the interfaces that are refined by the other modules. It also provides support for the two interaction styles (notification- and wait- based) with the middleware.
- The Domain Module contains the ***DomainParticipant*** class that acts as an entry-point of the Service and acts as a factory for many of the classes. The ***DomainParticipant*** also acts as a container for the other objects that make up the Service.
- The Topic-Definition Module contains the ***Topic***, ***ContentFilteredTopic***, and ***MultiTopic*** classes, the ***TopicListener*** interface, and more generally, all that is needed by the application to define ***Topic*** objects and attach QoS policies to them.
- The Publication Module contains the ***Publisher*** and ***DataWriter*** classes as well as the ***PublisherListener*** and ***DataWriterListener*** interfaces, and more generally, all that is needed on the publication side.
- The Subscription Module contains the ***Subscriber, DataReader***, ***ReadCondition***, and ***QueryCondition*** classes, as well as the ***SubscriberListener*** and ***DataReaderListener*** interfaces, and more generally, all that is needed on the subscription side.

At the PIM level, we have chosen to model any entity as a class or interface. It should be noted, however, that this does not mean that any of them will be translated into an IDL interface. In general, we have chosen to model as interfaces the entities that the application will have to extend to interact with the Service. The remaining entities have been modeled as classes.

2.1.2.1 Infrastructure Module

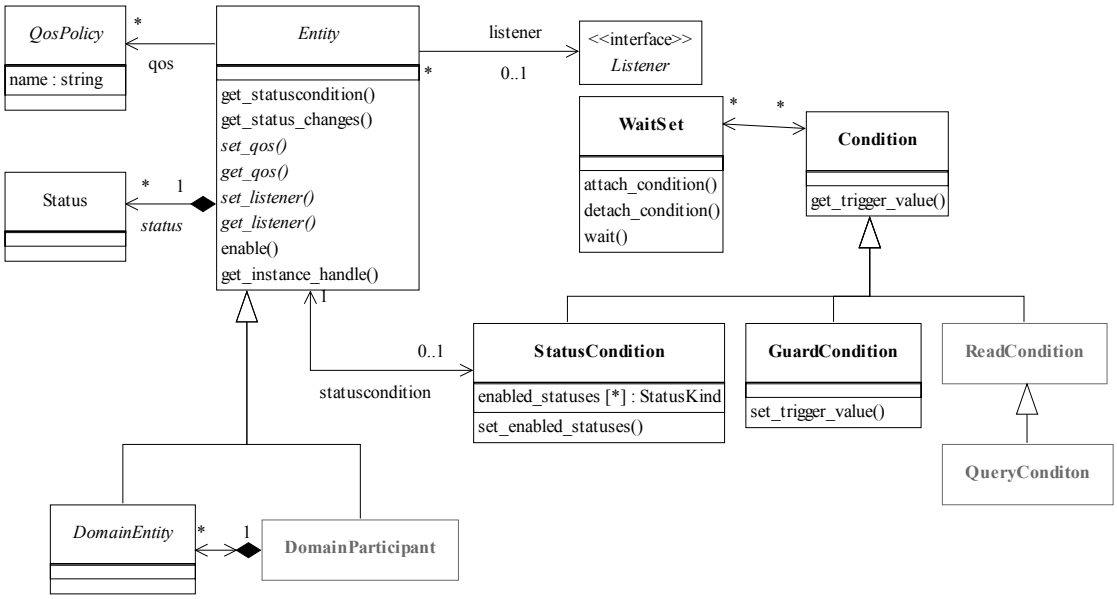

Figure 2-5 Class model of the DCPS Infrastructure Module

The DCPS Infrastructure Module is comprised of the following classifiers:

- Entity
- DomainEntity
- QosPolicy
- Listener
- Status
- WaitSet
- Condition
- GuardCondition
- StatusCondition

2.1.2.1.1 Entity Class

This class is the abstract base class for all the DCPS objects that support QoS policies, a listener and a status condition.

Entity		
no attributes		
operations		
abstract set_qos		ReturnCode_t
	qos_list	QosPolicy []
abstract get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
abstract set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
abstract get_listener		Listener
get_statuscondition		StatusCondition
get_status_changes		StatusKind []
enable		ReturnCode_t
get_instance_handle		InstanceHandle_t

StatusKind is an enumerated type that identifies each concrete ***Status*** type.

The following sections explain all the operations in detail.

2.1.2.1.1.1 set_qos (abstract)

This operation is used to set the QoS policies of the ***Entity***. This operation must be provided by each of the derived ***Entity*** classes (***DomainParticipant***, ***Topic***, ***Publisher***, ***DataWriter***, ***Subscriber***, ***DataReader***) so that the policies that are meaningful to each ***Entity*** can be set.

The set of policies specified as the ***qos_list*** parameter are applied on top of the existing QoS, replacing the values of any policies previously set.

As described in Section 2.1.3, “Supported QoS,” on page 2-102, certain policies are “immutable;” they can only be set at ***Entity*** creation time, or before the entity is made enabled. If ***set_qos*** is invoked after the ***Entity*** is enabled and it attempts to change the value of an “immutable” policy, the operation will fail and it returns IMMUTABLE_POLICY.

Section 2.1.3, “Supported QoS,” on page 2-102 also describes that certain values of QoS policies can be incompatible with the settings of the other policies. The ***set_qos*** operation will also fail if it specifies a set of values that once combined with the existing values would result in an inconsistent set of policies. In this case, the return value is INCONSISTENT_POLICY.

If the application supplies a non-default value for a QoS policy that is not supported by the implementation of the service, the **set_qos** operation will fail and return UNSUPPORTED.

The existing set of policies are only changed if the ***set_qos*** operation succeeds. This is indicated by the OK return value. In all other cases, none of the policies is modified.

Each derived ***Entity*** class (***DomainParticipant***, ***Topic***, ***Publisher***, ***DataWriter***, ***Subscriber***, ***DataReader***) has a corresponding special value of the QoS (PARTICIPANT_QOS_DEFAULT, PUBLISHER_QOS_DEFAULT, SUBSCRIBER_QOS_DEFAULT, TOPIC_QOS_DEFAULT, DATAWRITER_QOS_DEFAULT, DATAREADER_QOS_DEFAULT). This special value may be used as a parameter to the **set_qos** operation to indicate that the QoS of the ***Entity*** should be changed to match the current default QoS set in the ***Entity***'s factory. The operation set_qos cannot modify the immutable QoS so a successful return of the operation indicates that the mutable QoS for the ***Entity*** has been modified to match the current default for the ***Entity***'s factory.

Possible error codes returned in addition to the standard ones: INCONSISTENT_POLICY, IMMUTABLE_POLICY.

2.1.2.1.1.2 get_qos (abstract)

This operation allows access to the existing set of QoS policies for the ***Entity***. This operation must be provided by each of the derived ***Entity*** classes (***DomainParticipant***, ***Topic***, ***Publisher***, ***DataWriter***, ***Subscriber***, ***DataReader***) so that the policies meaningful to the particular ***Entity*** are retrieved.

2.1.2.1.1.3 set_listener (abstract)

This operation installs a ***Listener*** on the ***Entity***. The listener will only be invoked on the changes of communication status indicated by the specified ***mask***.

It is permitted to use 'nil' as the value of the *listener*. The 'nil' listener behaves as a ***Listener*** whose operations perform no action.

Only one listener can be attached to each ***Entity***. If a listener was already set, the operation ***set_listener*** will replace it with the new one. Consequently if the value 'nil' is passed for the listener parameter to the ***set_listener*** operation, any existing listener will be removed.

This operation must be provided by each of the derived ***Entity*** classes (***DomainParticipant***, ***Topic***, ***Publisher***, ***DataWriter***, ***Subscriber***, ***DataReader***) so that the listener is of the concrete type suitable to the particular ***Entity***.

2.1.2.1.1.4 get_listener (abstract)

This operation allows access to the existing ***Listener*** attached to the ***Entity***.

This operation must be provided by each of the derived ***Entity*** classes (***DomainParticipant***, ***Topic***, ***Publisher***, ***DataWriter***, ***Subscriber***, ***DataReader***) so that the listener is of the concrete type suitable to the particular ***Entity***.

2.1.2.1.1.5 get_statuscondition

This operation allows access to the ***StatusCondition*** (Section 2.1.2.1.9, ”StatusCondition Class”) associated with the ***Entity***. The returned condition can then be added to a ***WaitSet*** (Section 2.1.2.1.6, ”WaitSet Class”) so that the application can wait for specific status changes that affect the ***Entity***.

2.1.2.1.1.6 get_status_changes

This operation retrieves the list of communication statuses in the ***Entity*** that are ‘triggered.’ That is, the list of statuses whose value has changed since the last time the application read the status. The precise definition of the ‘triggered’ state of communication statuses is given in Section 2.1.4.2, “Changes in Status,” on page 2-134.

When the entity is first created or if the entity is not enabled, all communication statuses are in the “untriggered” state so the list returned by the ***get_status_changes*** operation will be empty.

The list of statuses returned by the ***get_status_changes*** operation refers to the status that are triggered on the ***Entity*** itself and does not include statuses that apply to contained entities.

2.1.2.1.1.7 enable

This operation enables the ***Entity***. ***Entity*** objects can be created either enabled or disabled. This is controlled by the value of the ENTITY_FACTORY Qos policy (Section 2.1.3.20, ”ENTITY_FACTORY”) on the corresponding factory for the ***Entity***.

The default setting of ENTITY_FACTORY is such that, by default, it is not necessary to explicitly call ***enable*** on newly created entities (see Section 2.1.3.20, ”ENTITY_FACTORY”).

The ***enable*** operation is idempotent. Calling ***enable*** on an already enabled ***Entity*** returns OK and has no effect.

If an ***Entity*** has not yet been enabled, the following kinds of operations may be invoked on itl:

- Operations to set or get an ***Entity's*** QoS policies (including default QoS policies) and listener
- get_statuscondition
- ‘factory’ operations
- get_status_changes and other get status operations (although the status of a disabled entity never changes)
- ‘lookup’ operations

Other operations may explicitly state that they may be called on disabled entities; those that do not will return the error NOT_ENABLED.

It is legal to delete an ***Entity*** that has not been enabled by calling the proper operation on its factory.

Entities created from a factory that is disabled, are created disabled regardless of the setting of the ENTITY_FACTORY Qos policy.

Calling enable on an ***Entity*** whose factory is not enabled will fail and return PRECONDITION_NOT_MET.

If the ENTITY_FACTORY Qos policy has *autoenable_created_entities* set to TRUE, the ***enable*** operation on the factory will automatically enable all entities created from the factory.

The ***Listeners*** associated with an entity are not called until the entity is enabled. Conditions associated with an entity that is not enabled are "inactive," that is, have a ***trigger_value***==FALSE (see Section 2.1.4.4, "Conditions and Wait-sets," on page 2-140).

2.1.2.1.1.8 get_instance_handle

This operation returns the ***InstanceHandle_t*** that represents the ***Entity***.

2.1.2.1.2 DomainEntity Class

DomainEntity is the abstract base class for all DCPS entities, except for the ***DomainParticipant***. Its sole purpose is to express that ***DomainParticipant*** is a special kind of ***Entity***, which acts as a container of all other ***Entity,*** but itself cannot contain other ***DomainParticipant***.

DomainEntity
no attributes
no operations

2.1.2.1.3 QosPolicy Class

This class is the abstract root for all the QoS policies.

QosPolicy	
attributes	
`name`	`string`
no operations	

It provides the basic mechanism for an application to specify quality of service parameters. It has an attribute ***name*** that is used to identify uniquely each QoS policy. All concrete ***QosPolicy*** classes derive from this root and include a ***value*** whose type depends on the concrete QoS policy.

The type of a ***QosPolicy*** value may be atomic, such as an integer or float, or compound (a structure). Compound types are used whenever multiple parameters must be set coherently to define a consistent value for a ***QosPolicy***.

Each ***Entity*** can be configured with a list of ***QosPolicy***. However, any ***Entity*** cannot support any ***QosPolicy***. For instance, a ***DomainParticipant*** supports different ***QosPolicy*** than a ***Topic*** or a ***Publisher***.

QosPolicy can be set when the ***Entity*** is created, or modified with the ***set_qos*** method. Each ***QosPolicy*** in the list is treated independently from the others. This approach has the advantage of being very extensible. However, there may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified via the ***set_qos*** operation.

When a policy is changed after being set to a given value, it is not required that the new value be applied instantaneously; the Service is allowed to apply it after a transition phase. In addition, some ***QosPolicy*** have "immutable" semantics meaning that they can only be specified either at ***Entity*** creation time or else prior to calling the ***enable*** operation on the ***Entity***.

Section 2.1.3, "Supported QoS," on page 2-102 provides the list of all ***QosPolicy***, their meaning, characteristics and possible values, as well as the concrete ***Entity*** to which they apply.

2.1.2.1.4 Listener Interface

Listener is the abstract root for all ***Listener*** interfaces. All the supported kinds of concrete ***Listener*** interfaces (one per concrete ***Entity***: ***DomainParticipant***, ***Topic***, ***Publisher***, ***DataWriter***, ***Subscriber***, and ***DataReader***) derive from this root and add methods whose prototype depends on the concrete ***Listener***.

Listener
no attributes
no operations

See Section 2.1.4.3, "Access through Listeners," on page 2-137 for the list of defined listener interfaces. Listener interfaces provide a mechanism for the Service to asynchronously inform the application of relevant changes in the communication status.

2.1.2.1.5 Status Class

Status is the abstract root class for all communication status objects. All concrete kinds of ***Status*** classes specialize this class.

Status
no attributes
no operations

Each concrete ***Entity*** is associated with a set of ***Status*** objects whose value represents the "communication status" of that entity. These status values can be accessed with corresponding methods on the ***Entity***. The changes on these status values are the ones that both cause activation of the corresponding ***StatusCondition*** objects and trigger invocation of the proper ***Listener*** objects to asynchronously inform the application.

Status objects and their relationship to ***Listener*** and ***Condition*** objects are detailed in Section 2.1.4.1, "Communication Status," on page 2-129.

2.1.2.1.6 WaitSet Class

A ***WaitSet*** object allows an application to wait until one or more of the attached ***Condition*** objects has a ***trigger_value*** of ***TRUE*** or else until the timeout expires.

WaitSet		
no attributes		
operations		
attach_condition		ReturnCode_t
	a_condition	Condition
detach_condition		ReturnCode_t
	a_condition	Condition
wait		ReturnCode_t
	inout: active_conditions	Condition []
	timeout	Duration_t
get_conditions		ReturnCode_t
	inout: attached_conditions	Condition []

WaitSet has no factory. It is created as an object directly by the natural means in each language binding (e.g., using "new" in C++ or Java). This is because it is not necessarily associated with a single ***DomainParticipant*** and could be used to wait on ***Condition*** objects associated with different ***DomainParticipant*** objects.

The following sections explain all the operations in detail.

2.1.2.1.6.1 attach_condition

Attaches a ***Condition*** to the ***WaitSet***.

It is possible to attach a ***Condition*** on a ***WaitSet*** that is currently being waited upon (via the ***wait*** operation). In this case, if the ***Condition*** has a ***trigger_value*** of TRUE, then attaching the condition will unblock the ***WaitSet***.

Adding a ***Condition*** that is already attached to the ***WaitSet*** has no effect.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.

2.1.2.1.6.2 detach_condition

Detaches a ***Condition*** from the ***WaitSet***.

If the ***Condition*** was not attached to the ***WaitSet*** the operation will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.1.6.3 wait

This operation allows an application thread to wait for the occurrence of certain conditions. If none of the conditions attached to the ***WaitSet*** have a ***trigger_value*** of TRUE, the wait operation will block suspending the calling thread.

The result of the ***wait*** operation is the list of all the attached conditions that have a ***trigger_value*** of TRUE (i.e., the conditions that unblocked the wait).

The wait operation takes a ***timeout*** argument that specifies the maximum duration for the wait. It this duration is exceeded and none of the attached ***Condition*** objects is true, ***wait*** will return with the return code TIMEOUT.

It is not allowed for more than one application thread to be waiting on the same ***WaitSet***. If the ***wait*** operation is invoked on a ***WaitSet*** that already has a thread blocking on it, the operation will return immediately with the value PRECONDITION_NOT_MET.

2.1.2.1.6.4 get_conditions

This operation retrieves the list of attached conditions.

2.1.2.1.7 Condition Class

A ***Condition*** is a root class for all the conditions that may be attached to a ***WaitSet***. This basic class is specialized in three classes that are known by the middleware: ***GuardCondition*** (Section 2.1.2.1.8), ***StatusCondition*** (Section 2.1.2.1.9), and ***ReadCondition*** (Section 2.1.2.5.8).

Condition		
no attributes		
operations		
`get_trigger_value`		`boolean`

A ***Condition*** has a ***trigger_value*** that can be TRUE or FALSE and is set automatically by the Service.

2.1.2.1.7.1 get_trigger_value

This operation retrieves the ***trigger_value*** of the ***Condition***.

2.1.2.1.8 GuardCondition Class

A ***GuardCondition*** object is a specific ***Condition*** whose ***trigger_value*** is completely under the control of the application.

GuardCondition		
no attributes		
operations		
set_trigger_value		ReturnCode_t
	value	boolean

GuardCondition has no factory. It is created as an object directly by the natural means in each language binding (e.g., using "new" in C++ or Java. When first created the ***trigger_value*** is set to FALSE.

The purpose of the ***GuardCondition*** is to provide the means for the application to manually wakeup a ***WaitSet***. This is accomplished by attaching the ***GuardCondition*** to the ***WaitSet*** and then setting the ***trigger_value*** by means of the ***set_trigger_value*** operation.

2.1.2.1.8.1 set_trigger_value

This operation sets the ***trigger_value*** of the ***GuardCondition***.

WaitSet objects behavior depends on the changes of the ***trigger_value*** of their attached conditions. Therefore, any ***WaitSet*** to which is attached the ***GuardCondition*** is potentially affected by this operation.

2.1.2.1.9 StatusCondition Class

A ***StatusCondition*** object is a specific ***Condition*** that is associated with each ***Entity***.

StatusCondition		
no attributes		
operations		
set_enabled_statuses		ReturnCode_t
	mask	StatusKind []
get_enabled_statuses		StatusKind []
get_entity		Entity

The ***trigger_value*** of the ***StatusCondition*** depends on the communication status of that entity (e.g., arrival of data, loss of information, etc.), 'filtered' by the set of ***enabled_statuses*** on the ***StatusCondition***.

The ***enabled_statuses*** and its relation to ***Listener*** and ***WaitSet*** is detailed in Trigger State of the StatusCondition.

2.1.2.1.9.1 set_enabled_statuses

This operation defines the list of communication statuses that are taken into account to determine the ***trigger_value*** of the ***StatusCondition***. This operation may change the ***trigger_value*** of the ***StatusCondition***.

WaitSet objects behavior depend on the changes of the ***trigger_value*** of their attached conditions. Therefore, any ***WaitSet*** to which the ***StatusCondition*** is attached is potentially affected by this operation.

If this function is not invoked, the default list of enabled statuses includes all the statuses.

2.1.2.1.9.2 get_enabled_statuses

This operation retrieves the list of communication statuses that are taken into account to determine the ***trigger_value*** of the ***StatusCondition***. This operation returns the statuses that were explicitly set on the last call to ***set_enabled_statuses*** or, if ***set_enabled_statuses*** was never called, the default list (see Section 2.1.2.1.9.1).

2.1.2.1.9.3 get_entity

This operation returns the ***Entity*** associated with the ***StatusCondition***. Note that there is exactly one ***Entity*** associated with each ***StatusCondition***.

2.1.2.2 Domain Module

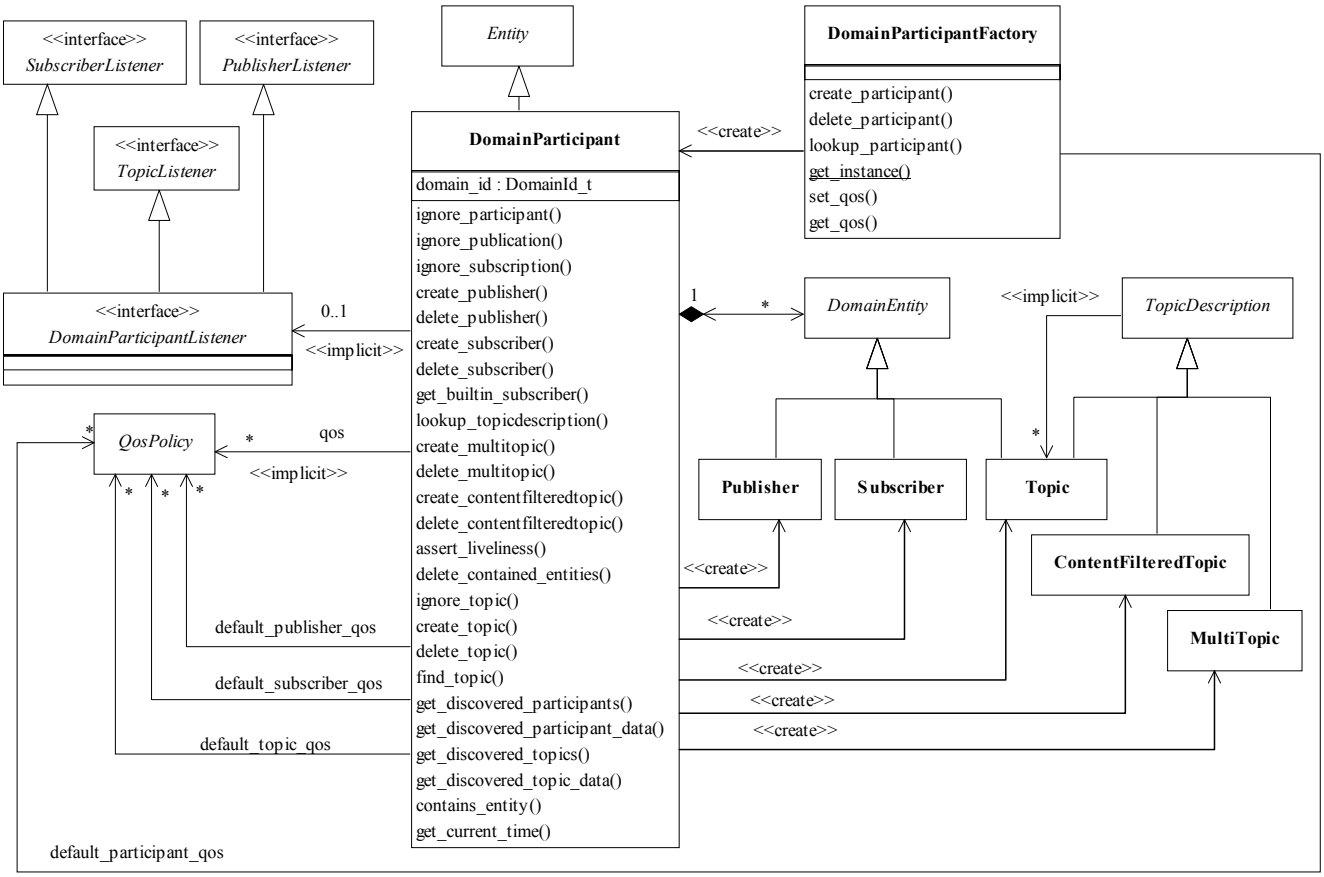

Figure 2-6 Class model of the DCPS Domain Module

The DCPS Domain Module is comprised of the following classes:

- DomainParticipant
- DomainParticipantFactory
- DomainParticipantListener

2.1.2.2.1 DomainParticipant Class

The ***DomainParticipant*** object plays several roles:

- It acts as a container for all other ***Entity*** objects.
- It acts as factory for the ***Publisher, Subscriber, Topic*** and ***MultiTopic Entity*** objects.
- It represents the participation of the application on a communication plane that isolates applications running on the same set of physical computers from each other. A domain establishes a “virtual network” linking all applications that share the

same ***domainId***[9] and isolating them from applications running on different domains. In this way, several independent distributed applications can coexist in the same physical network without interfering, or even being aware of each other.

- It provides administration services in the domain, offering operations that allow the application to 'ignore' locally any information about a given participant (***ignore_participant***), publication (***ignore_publication***), subscription (***ignore_subscription***) or topic (***ignore_topic***).

DomainParticipant		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
create_publisher		Publisher
	qos_list	QosPolicy []
	a_listener	PublisherListener
	mask	StatusKind []
delete_publisher		ReturnCode_t
	a_publisher	Publisher
create_subscriber		Subscriber
	qos_list	QosPolicy []
	a_listener	SubscriberListener
	mask	StatusKind []
delete_subscriber		ReturnCode_t
	a_subscriber	Subscriber
create_topic		Topic
	topic_name	string
	type_name	string
	qos_list	QosPolicy []
	a_listener	TopicListener
	mask	StatusKind []
delete_topic		ReturnCode_t

9. The actual format of the ***domainId*** is middleware specific. From the application point of view, it is a configuration parameter that appears only when the ***DomainParticipant*** is created.

	a_topic	Topic
create_contentfilteredtopic		ContentFilteredTopic
	name	string
	related_topic	Topic
	filter_expression	string
	expression_parameters	string []
delete_contentfilteredtopic		ReturnCode_t
	a_contentfilteredtopic	ContentFilteredTopic
create_multitopic		MultiTopic
	name	string
	type_name	string
	subscription_expression	string
	expression_parameters	string []
delete_multitopic		ReturnCode_t
	a_multitopic	MultiTopic
find_topic		Topic
	topic_name	string
	timeout	Duration_t
lookup_topicdescription		TopicDescription
	name	string
get_builtin_subscriber		Subscriber
ignore_participant		ReturnCode_t
	handle	InstanceHandle_t
ignore_topic		ReturnCode_t
	handle	InstanceHandle_t
ignore_publication		ReturnCode_t
	handle	InstanceHandle_t
ignore_subscription		ReturnCode_t
	handle	InstanceHandle_t
get_domain_id		DomainId_t
delete_contained_entities		ReturnCode_t
assert_liveliness		ReturnCode_t
set_default_publisher_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_publisher_qos		ReturnCode_t
	out: qos_list	QosPolicy []
set_default_subscriber_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_subscriber_qos		ReturnCode_t
	out: qos_list	QosPolicy []
set_default_topic_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_topic_qos		ReturnCode_t

	out: qos_list	QosPolicy []
get_discovered_participants		ReturnCode_t
	inout: participant_handles	InstanceHandle_t []
get_discovered_participant_data		ReturnCode_t
	inout: participant_data	ParticipantBuiltin-TopicData
	participant_handle	InstanceHandle_t
get_discovered_topics		ReturnCode_t
	inout: topic_handles	InstanceHandle_t []
get_discovered_topic_data		ReturnCode_t
	inout: topic_data	TopicBuiltinTopicData
	topic_handle	InstanceHandle_t
contains_entity		boolean
	a_handle	InstanceHandle_t
get_current_time		ReturnCode_t
	inout: current_time	Time_t

The following sections explain all the operations in detail.

The following operations may be called even if the ***DomainParticipant*** is not enabled. Other operations will have the value NOT_ENABLED if called on a disabled ***DomainParticipant***:

- Operations defined at the base-class level namely, ***set_qos***, ***get_qos***, ***set_listener***, ***get_listener*** and ***enable***.
- Factory methods: **create_topic, create_publisher, create_subscriber, delete_topic, delete_publisher, delete_subscriber**
- Operations that access the status: **get_statuscondition**

2.1.2.2.1.1 create_publisher

This operation creates a ***Publisher*** with the desired QoS policies and attaches to it the specified ***PublisherListener***.

If the specified QoS policies are not consistent, the operation will fail and no ***Publisher*** will be created.

The special value PUBLISHER_QOS_DEFAULT can be used to indicate that the ***Publisher*** should be created with the default ***Publisher*** QoS set in the factory. The use of this value is equivalent to the application obtaining the default ***Publisher*** QoS by means of the operation ***get_default_publisher_qos*** (Section 2.1.2.2.1.21, ”get_default_publisher_qos”) and using the resulting QoS to create the ***Publisher***.

The created ***Publisher*** belongs to the ***DomainParticipant*** that is its factory.

In case of failure, the operation will return a ‘nil’ value (as specified by the platform).

2.1.2.2.1.2 delete_publisher

This operation deletes an existing ***Publisher***.

A ***Publisher*** cannot be deleted if it has any attached ***DataWriter*** objects. If ***delete_publisher*** is called on a ***Publisher*** with existing ***DataWriter*** object, it will return PRECONDITION_NOT_MET.

The ***delete_publisher*** operation must be called on the same ***DomainParticipant*** object used to create the ***Publisher***. If ***delete_publisher*** is called on a different ***DomainParticipant***, the operation will have no effect and it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.2.1.3 create_subscriber

This operation creates a ***Subscriber*** with the desired QoS policies and attaches to it the specified ***SubscriberListener***.

If the specified QoS policies are not consistent, the operation will fail and no ***Subscriber*** will be created.

The special value SUBSCRIBER_QOS_DEFAULT can be used to indicate that the ***Subscriber*** should be created with the default ***Subscriber*** QoS set in the factory. The use of this value is equivalent to the application obtaining the default ***Subscriber*** QoS by means of the operation ***get_default_subscriber_qos*** (Section 2.1.2.2.1.21) and using the resulting QoS to create the ***Subscriber***.

The created ***Subscriber*** belongs to the ***DomainParticipant*** that is its factory.

In case of failure, the operation will return a ‘nil’ value (as specified by the platform).

2.1.2.2.1.4 delete_subscriber

This operation deletes an existing ***Subscriber***.

A ***Subscriber*** cannot be deleted if it has any attached ***DataReader*** objects. If the ***delete_subscriber*** operation is called on a ***Subscriber*** with existing ***DataReader*** objects, it will return PRECONDITION_NOT_MET.

The ***delete_subscriber*** operation must be called on the same ***DomainParticipant*** object used to create the ***Subscriber***. If ***delete_subscriber*** is called on a different ***DomainParticipant***, the operation will have no effect and it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.2.1.5 create_topic

This operation creates a ***Topic*** with the desired QoS policies and attaches to it the specified ***TopicListener***.

If the specified QoS policies are not consistent, the operation will fail and no ***Topic*** will be created.

The special value TOPIC_QOS_DEFAULT can be used to indicate that the ***Topic*** should be created with the default ***Topic*** QoS set in the factory. The use of this value is equivalent to the application obtaining the default ***Topic*** QoS by means of the operation ***get_default_topic_qos*** (Section 2.1.2.2.1.21, ”get_default_publisher_qos”) and using the resulting QoS to create the ***Topic***.

The created ***Topic*** belongs to the ***DomainParticipant*** that is its factory.

The ***Topic*** is bound to a type described by the ***type_name*** argument. Prior to creating a ***Topic*** the type must have been registered with the Service. This is done using the ***register_type*** operation on a derived class of the ***TypeSupport*** interface as described in Section 2.1.2.3.6, “TypeSupport Interface,” on page 2-43.

In case of failure, the operation will return a ‘nil’ value (as specified by the platform).

2.1.2.2.1.6 delete_topic

This operation deletes a ***Topic***.

The deletion of a ***Topic*** is not allowed if there are any existing ***DataReader***, ***DataWriter***, ***ContentFilteredTopic***, or ***MultiTopic*** objects that are using the ***Topic***. If the ***delete_topic*** operation is called on a ***Topic*** with any of these existing objects attached to it, it will return PRECONDITION_NOT_MET.

The ***delete_topic*** operation must be called on the same ***DomainParticipant*** object used to create the ***Topic***. If ***delete_topic*** is called on a different ***DomainParticipant***, the operation will have no effect and it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.2.1.7 create_contentfilteredtopic

This operation creates a ***ContentFilteredTopic***. As described in Section 2.1.2.3, “Topic-Definition Module,” on page 2-38, a ***ContentFilteredTopic*** can be used to do content-based subscriptions.

The related ***Topic*** being subscribed to is specified by means of the ***related_topic*** parameter. The ***ContentFilteredTopic*** only relates to samples published under that ***Topic,*** filtered according to their content. The filtering is done by means of evaluating a logical expression that involves the values of some of the data-fields in the sample. The logical expression is derived from the ***filter_expression*** and ***expression_parameters*** arguments.

The syntax of the filter expression and parameters is described in Appendix B.

In case of failure, the operation will return a ‘nil’ value (as specified by the platform).

2.1.2.2.1.8 delete_ contentfilteredtopic

This operation deletes a ***ContentFilteredTopic***.

The deletion of a ***ContentFilteredTopic*** is not allowed if there are any existing ***DataReader*** objects that are using the ***ContentFilteredTopic***. If the ***delete_contentfilteredtopic*** operation is called on a ***ContentFilteredTopic*** with existing ***DataReader*** objects attached to it, it will return PRECONDITION_NOT_MET.

The ***delete_contentfilteredtopic*** operation must be called on the same ***DomainParticipant*** object used to create the ***ContentFilteredTopic***. If ***delete_contentfilteredtopic*** is called on a different ***DomainParticipant***, the operation will have no effect and it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.2.1.9 create_multitopic

This operation creates a ***MultiTopic***. As described in Section 2.1.2.3, “Topic-Definition Module,” on page 2-38 a ***MultiTopic*** can be used to subscribe to multiple topics and combine/filter the received data into a resulting type. In particular, ***MultiTopic*** provides a content-based subscription mechanism.

The resulting type is specified by the ***type_name*** argument. Prior to creating a ***MultiTopic*** the type must have been registered with the Service. This is done using the ***register_type*** operation on a derived class of the ***TypeSupport*** interface as described in Section 2.1.2.3.6, “TypeSupport Interface,” on page 2-43.

The list of topics and the logic used to combine filter and re-arrange the information from each ***Topic*** are specified using the ***subscription_expression*** and ***expression_parameters*** arguments.

The syntax of the expression and parameters is described in Appendix B.

In case of failure, the operation will return a ‘nil’ value (as specified by the platform).

2.1.2.2.1.10 delete_multitopic

This operation deletes a ***MultiTopic***.

The deletion of a ***MultiTopic*** is not allowed if there are any existing ***DataReader*** objects that are using the ***MultiTopic***. If the ***delete_multitopic*** operation is called on a ***MultiTopic*** with existing ***DataReader*** objects attached to it, it will return PRECONDITION_NOT_MET.

The ***delete_multitopic*** operation must be called on the same ***DomainParticipant*** object used to create the ***MultiTopic***. If ***delete_multitopic*** is called on a different ***DomainParticipant***, the operation will have no effect and it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.2.1.11 find_topic

The operation ***find_topic*** gives access to an existing (or ready to exist) enabled ***Topic***, based on its name. The operation takes as arguments the ***name*** of the ***Topic*** and a ***timeout***.

If a ***Topic*** of the same name already exists, it gives access to it, otherwise it waits (blocks the caller) until another mechanism creates it (or the specified timeout occurs). This other mechanism can be another thread, a configuration tool, or some other middleware service. Note that the ***Topic*** is a local object[10] that acts as a 'proxy' to designate the global concept of topic. Middleware implementations could choose to propagate topics and make remotely created topics locally available.

A ***Topic*** obtained by means of ***find_topic***, must also be deleted by means of ***delete_topic*** so that the local resources can be released. If a ***Topic*** is obtained multiple times by means of ***find_topic*** or ***create_topic***, it must also be deleted that same number of times using ***delete_topic***.

Regardless of whether the middleware chooses to propagate topics, the ***delete_topic*** operation deletes only the local proxy.

If the operation times-out, a ‘nil’ value (as specified by the platform) is returned.

2.1.2.2.1.12 lookup_topicdescription

The operation ***lookup_topicdescription*** gives access to an existing locally-created ***TopicDescription***, based on its name. The operation takes as argument the ***name*** of the ***TopicDescription.***

If a ***TopicDescription*** of the same name already exists, it gives access to it, otherwise it returns a ‘nil’ value. The operation never blocks.

The operation ***lookup_topicdescription*** may be used to locate any locally-created ***Topic***, ***ContentFilteredTopic***, and ***MultiTopic*** object.

Unlike ***find_topic***, the operation ***lookup_topicdescription*** searches only among the locally created topics. Therefore, it should never create a new ***TopicDescription***. The ***TopicDescription*** returned by ***lookup_topicdescription*** does not require any extra deletion. It is still possible to delete the ***TopicDescription*** returned by ***lookup_topicdescription***, provided it has no readers or writers, but then it is really deleted and subsequent lookups will fail.

If the operation fails to locate a ***TopicDescription***, a ‘nil’ value (as specified by the platform) is returned.

10. All the objects that make up this specification are local objects that are actually proxies to the service to be used by the application.

2.1.2.2.1.13 get_builtin_subscriber

This operation allows access to the built-in ***Subscriber***. Each ***DomainParticipant*** contains several built-in ***Topic*** objects as well as corresponding ***DataReader*** objects to access them. All these ***DataReader*** objects belong to a single built-in ***Subscriber***.

The built-in Topics are used to communicate information about other ***DomainParticipant***, ***Topic***, ***DataReader***, and ***DataWriter*** objects. These built-in objects are described in Section 2.1.5, “Built-in Topics,” on page 2-144.

2.1.2.2.1.14 ignore_participant

This operation allows an application to instruct the Service to locally ignore a remote domain participant. From that point onwards the Service will locally behave as if the remote participant did not exist. This means it will ignore any ***Topic***, publication, or subscription that originates on that domain participant.

This operation can be used, in conjunction with the discovery of remote participants offered by means of the “DCPSParticipant” built-in ***Topic,*** to provide, for example, access control. Application data can be associated with a ***DomainParticipant*** by means of the USER_DATA QoS policy. This application data is propagated as a field in the built-in topic and can be used by an application to implement its own access control policy. See Section 2.1.5, “Built-in Topics,” on page 2-144 for more details on the built-in topics.

The domain participant to ignore is identified by the ***handle*** argument. This handle is the one that appears in the ***SampleInfo*** retrieved when reading the data-samples available for the built-in ***DataReader*** to the “DCPSParticipant” topic. The built-in ***DataReader*** is read with the same ***read***/***take*** operations used for any ***DataReader***. These data-accessing operations are described in Section 2.1.2.5, “Subscription Module,” on page 2-64.

The **ignore*_participant*** operation is not required to be reversible. The Service offers no means to reverse it.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.

2.1.2.2.1.15 ignore_topic

This operation allows an application to instruct the Service to locally ignore a ***Topic***. This means it will locally ignore any publication or subscription to the ***Topic***.

This operation can be used to save local resources when the application knows that it will never publish or subscribe to data under certain topics.

The ***Topic*** to ignore is identified by the ***handle*** argument. This handle is the one that appears in the ***SampleInfo*** retrieved when reading the data-samples from the built-in ***DataReader*** to the “DCPSTopic” topic.

The ***ignore_topic*** operation is not required to be reversible. The Service offers no means to reverse it.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.

2.1.2.2.1.16 ignore_publication

This operation allows an application to instruct the Service to locally ignore a remote publication; a publication is defined by the association of a topic name, and user data and partition set on the ***Publisher*** (see the "DCPSPublication" built-in ***Topic*** in Section 2.1.5, "Built-in Topics," on page 2-144). After this call, any data written related to that publication will be ignored.

The ***DataWriter*** to ignore is identified by the ***handle*** argument. This handle is the one that appears in the ***SampleInfo*** retrieved when reading the data-samples from the built-in ***DataReader*** to the "DCPSPublication" topic.

The ***ignore_publication*** operation is not required to be reversible. The Service offers no means to reverse it.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.

2.1.2.2.1.17 ignore_subscription

This operation allows an application to instruct the Service to locally ignore a remote subscription; a subscription is defined by the association of a topic name, and user data and partition set on the ***Subscriber*** (see the "DCPSSubscription" built-in ***Topic*** in Section 2.1.5, "Built-in Topics," on page 2-144). After this call, any data received related to that subscription will be ignored.

The ***DataReader*** to ignore is identified by the ***handle*** argument. This handle is the one that appears in the ***SampleInfo*** retrieved when reading the data-samples from the built-in ***DataReader*** to the "DCPSSubscription" topic.

The ***ignore_subscription*** operation is not required to be reversible. The Service offers no means to reverse it.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.

2.1.2.2.1.18 delete_contained_entities

This operation deletes all the entities that were created by means of the "create" operations on the ***DomainParticipant***. That is, it deletes all contained ***Publisher***, ***Subscriber***, ***Topic***, ***ContentFilteredTopic***, and ***MultiTopic***.

Prior to deleting each contained entity, this operation will recursively call the corresponding ***delete_contained_entities*** operation on each contained entity (if applicable). This pattern is applied recursively. In this manner the operation ***delete_contained_entities*** on the ***DomainParticipant*** will end up deleting all the entities recursively contained in the ***DomainParticipant***, that is also the ***DataWriter***, ***DataReader***, as well as the ***QueryCondition*** and ***ReadCondition*** objects belonging to the contained ***DataReaders***.

The operation will return PRECONDITION_NOT_MET if the any of the contained entities is in a state where it cannot be deleted.

Once ***delete_contained_entities*** returns successfully, the application may delete the ***DomainParticipant*** knowing that it has no contained entities.

2.1.2.2.1.19 assert_liveliness

This operation manually asserts the liveliness of the ***DomainParticipant***. This is used in combination with the LIVELINESS QoS policy (cf. Section 2.1.3, “Supported QoS,” on page 2-102) to indicate to the Service that the entity remains active.

This operation needs to only be used if the ***DomainParticipant*** contains ***DataWriter*** entities with the LIVELINESS set to MANUAL_BY_PARTICIPANT and it only affects the liveliness of those ***DataWriter*** entities. Otherwise, it has no effect.

Note – Writing data via the ***write*** operation on a ***DataWriter*** asserts liveliness on the ***DataWriter*** itself and its ***DomainParticipant***. Consequently the use of ***assert_liveliness*** is only needed if the application is not writing data regularly.

Complete details are provided in Section 2.1.3.11, “LIVELINESS,” on page 2-119.

2.1.2.2.1.20 set_default_publisher_qos

This operation sets a default value of the ***Publisher*** QoS policies which will be used for newly created ***Publisher*** entities in the case where the QoS policies are defaulted in the ***create_publisher*** operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

The special value PUBLISHER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the **set_default_publisher_qos** operation had never been called.

2.1.2.2.1.21 get_default_publisher_qos

This operation retrieves the default value of the ***Publisher*** QoS, that is, the QoS policies which will be used for newly created ***Publisher*** entities in the case where the QoS policies are defaulted in the ***create_publisher*** operation.

The values retrieved ***get_default_publisher_qos*** will match the set of values specified on the last successful call to ***set_default_publisher_qos,*** or else, if the call was never made, the default values listed in the QoS table in Section 2.1.3, “Supported QoS,” on page 2-102.

2.1.2.2.1.22 set_default_subscriber_qos

This operation sets a default value of the ***Subscriber*** QoS policies that will be used for newly created ***Subscriber*** entities in the case where the QoS policies are defaulted in the ***create_subscriber*** operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

The special value SUBSCRIBER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the **set_default_subscriber_qos** operation had never been called.

2.1.2.2.1.23 get_default_subscriber_qos

This operation retrieves the default value of the ***Subscriber*** QoS, that is, the QoS policies which will be used for newly created ***Subscriber*** entities in the case where the QoS policies are defaulted in the ***create_subscriber*** operation.

The values retrieved ***get_default_subscriber_qos*** will match the set of values specified on the last successful call to ***set_default_subscriber_qos,*** or else, if the call was never made, the default values listed in the QoS table in Section 2.1.3, “Supported QoS,” on page 2-102.

2.1.2.2.1.24 set_default_topic_qos

This operation sets a default value of the ***Topic*** QoS policies which will be used for newly created ***Topic*** entities in the case where the QoS policies are defaulted in the ***create_topic*** operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

The special value TOPIC_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the **set_default_topic_qos** operation had never been called.

2.1.2.2.1.25 get_default_topic_qos

This operation retrieves the default value of the ***Topic*** QoS, that is, the QoS policies which will be used for newly created ***Topic*** entities in the case where the QoS policies are defaulted in the ***create_topic*** operation.

The values retrieved ***get_default_topic_qos*** will match the set of values specified on the last successful call to ***set_default_topic_qos,*** or else, if the call was never made, the default values listed in the QoS table in Section 2.1.3, “Supported QoS,” on page 2-102.

2.1.2.2.1.26 get_domain_id

This operation retrieves the *domain_id* used to create the ***DomainParticipant***. The ***domain_id*** identifies the DDS domain to which the ***DomainParticipant*** belongs. As described in the introduction to Section 2.1.2.2.1 each DDS domain represents a separate data "communication plane" isolated from other domains.

2.1.2.2.1.27 get_discovered_participants

This operation retrieves the list of DomainParticipants that have been discovered in the domain and that the application has not indicated should be "ignored" by means of the DomainParticipant ignore_participant operation.

The operation may fail if the infrastructure does not locally maintain the connectivity information. In this case the operation will return UNSUPPORTED.

2.1.2.2.1.28 get_discovered_participant_data

This operation retrieves information on a ***DomainParticipant*** that has been discovered on the network. The participant must be in the same domain as the participant on which this operation is invoked and must not have been "ignored" by means of the ***DomainParticipant ignore_participant*** operation.

The *participant_handle* must correspond to such a ***DomainParticipant***. Otherwise, the operation will fail and return PRECONDITION_NOT_MET.

Use the operation ***get_discovered_participants*** to find the DomainParticipants that are currently discovered.

The operation may also fail if the infrastructure does not hold the information necessary to fill in the *participant_data*. In this case the operation will return UNSUPPORTED.

2.1.2.2.1.29 get_discovered_topics

This operation retrieves the list of Topics that have been discovered in the domain and that the application has not indicated should be "ignored" by means of the ***DomainParticipant ignore_topic*** operation.

2.1.2.2.1.30 get_discovered_topic_data

This operation retrieves information on a Topic that has been discovered on the network. The topic must have been created by a participant in the same domain as the participant on which this operation is invoked and must not have been "ignored" by means of the ***DomainParticipant ignore_topic*** operation.

The topic_handle must correspond to such a topic. Otherwise, the operation will fail and return PRECONDITION_NOT_MET.

Use the operation ***get_discovered_topics*** to find the topics that are currently discovered.

The operation may also fail if the infrastructure does not hold the information necessary to fill in the *topic_data*. In this case the operation will return UNSUPPORTED.

The operation may fail if the infrastructure does not locally maintain the connectivity information. In this case the operation will return UNSUPPORTED.

2.1.2.2.1.31 contains_entity

This operation checks whether or not the given *a_handle* represents an ***Entity*** that was created from the ***DomainParticipant***. The containment applies recursively. That is, it applies both to entities (***TopicDescription***, ***Publisher***, or ***Subscriber***) created directly using the ***DomainParticipant*** as well as entities created using a contained ***Publisher***, or ***Subscriber*** as the factory, and so forth.

The instance handle for an ***Entity*** may be obtained from built-in topic data, from various statuses, or from the Entity operation ***get_instance_handle***.

2.1.2.2.1.32 get_current_time

This operation returns the current value of the time that the service uses to time-stamp data-writes and to set the reception-timestamp for the data-updates it receives.

2.1.2.2.2 DomainParticipantFactory Class

The sole purpose of this class is to allow the creation and destruction of ***DomainParticipant*** objects. ***DomainParticipantFactory*** itself has no factory. It is a pre-existing singleton object that can be accessed by means of the ***get_instance*** class operation on the ***DomainParticipantFactory***.

DomainParticipantFactory		
no attributes		
operations		
create_participant		DomainParticipant
	domain_id	DomainId_t
	qos_list	QosPolicy []
	a_listener	DomainParticipantLis-tener
	mask	StatusKind []
delete_participant		ReturnCode_t
	a_participant	DomainParticipant
(static) get_instance		DomainParticipantFactory
lookup_participant		DomainParticipant
	domain_id	DomainId_t
set_default_participant_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_participant_qos		ReturnCode_t
	out: qos_list	QosPolicy []
get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
set_qos		ReturnCode_t
	qos_list	QosPolicy []

The following sections give details about the operations.

2.1.2.2.2.1 create_participant

This operation creates a new ***DomainParticipant*** object. The ***DomainParticipant*** signifies that the calling application intends to join the Domain identified by the ***domain_id*** argument.

If the specified QoS policies are not consistent, the operation will fail and no ***DomainParticipant*** will be created.

The special value PARTICIPANT_QOS_DEFAULT can be used to indicate that the ***DomainParticipant*** should be created with the default ***DomainParticipant*** QoS set in the factory. The use of this value is equivalent to the application obtaining the default ***DomainParticipant*** QoS by means of the operation ***get_default_participant_qos*** (2.1.2.2.2.6) and using the resulting QoS to create the ***DomainParticipant***.

In case of failure, the operation will return a 'nil' value (as specified by the platform).

2.1.2.2.2.2 delete_participant

This operation deletes an existing ***DomainParticipant***. This operation can only be invoked if all domain entities belonging to the participant have already been deleted. Otherwise the error PRECONDITION_NOT_MET is returned.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.2.2.3 get_instance

This operation returns the ***DomainParticipantFactory*** singleton. The operation is idempotent, that is, it can be called multiple times without side-effects and it will return the same ***DomainParticipantFactory*** instance.

The ***get_instance*** operation is a static operation implemented using the syntax of the native language and can therefore not be expressed in the IDL PSM.

The pre-defined value ***TheParticipantFactory*** can also be used as an alias for the singleton factory returned by the operation ***get_instance***.

2.1.2.2.2.4 lookup_participant

This operation retrieves a previously created ***DomainParticipant*** belonging to specified domain_id. If no such ***DomainParticipant*** exists, the operation will return a 'nil' value.

If multiple ***DomainParticipant*** entities belonging to that *domain_id* exist, then the operation will return one of them. It is not specified which one.

2.1.2.2.2.5 set_default_participant_qos

This operation sets a default value of the ***DomainParticipant*** QoS policies which will be used for newly created ***DomainParticipant*** entities in the case where the QoS policies are defaulted in the ***create_participant*** operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

2.1.2.2.2.6 get_default_participant_qos

This operation retrieves the default value of the ***DomainParticipant*** QoS, that is, the QoS policies which will be used for newly created ***DomainParticipant*** entities in the case where the QoS policies are defaulted in the ***create_participant*** operation.

The values retrieved ***get_default_participant_qos*** will match the set of values specified on the last successful call to ***set_default_participant_qos,*** or else, if the call was never made, the default values listed in the QoS table in Section 2.1.3, “Supported QoS,” on page 2-102.

2.1.2.2.2.7 set_qos

This operation sets the value of the ***DomainParticipantFactory*** QoS policies. These policies control the behavior of the object a factory for entities.

Note that despite having QoS, the ***DomainParticipantFactory*** is not an ***Entity***.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

2.1.2.2.2.8 get_qos

This operation returns the value of the ***DomainParticipantFactory*** QoS policies.

2.1.2.2.3 DomainParticipantListener Interface

This is the interface that can be implemented by an application-provided class and then registered with the ***DomainParticipant*** such that the application can be notified by the DCPS Service of relevant status changes.

The ***DomainParticipantListener*** interface extends all other ***Listener*** interfaces and has no additional operation beyond the ones defined by the more general listeners.

DomainParticipantListener		
no attributes		
operations		
on_inconsistent_topic		void
	the_topic	Topic
	status	InconsistentTopicStatus
on_liveliness_lost		void
	the_writer	DataWriter
	status	LivelinessLostStatus
on_offered_deadline_missed	the_writer	DataWriter
	status	OfferedDeadlineMissedStatus
on_offered_incompatible_qos	the_writer	DataWriter
	status	OfferedIncompatibleQosStatus
on_data_on_readers		void
	the_subscriber	Subscriber
on_sample_lost		void
	the_reader	DataReader
	status	SampleLostStatus
on_data_available		void
	the_reader	DataReader

on_sample_rejected		void
	the_reader	DataReader
	status	SampleRejectedStatus
on_liveliness_changed		void
	the_reader	DataReader
	status	LivelinessChangedStatus
on_requested_deadline_missed		void
	the_reader	DataReader
	status	RequestedDeadlineMissedStatus
on_requested_incompatible_qos		void
	the_reader	DataReader
	status	RequestedIncompatibleQosStatus
on_publication_matched		void
	the_writer	DataWriter
	status	PublicationMatchedStatus
on_subscription_matched		void
	the_reader	DataReader
	status	SubscriptionMatchedStatus

The purpose of the ***DomainParticipantListener*** is to be the listener of last resort that is notified of all status changes not captured by more specific listeners attached to the ***DomainEntity*** objects. When a relevant status change occurs, the DCPS Service will first attempt to notify the listener attached to the concerned ***DomainEntity*** if one is installed. Otherwise, the DCPS Service will notify the ***Listener*** attached to the ***DomainParticipant***. The relationship between listeners is described in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

2.1.2.3 Topic-Definition Module

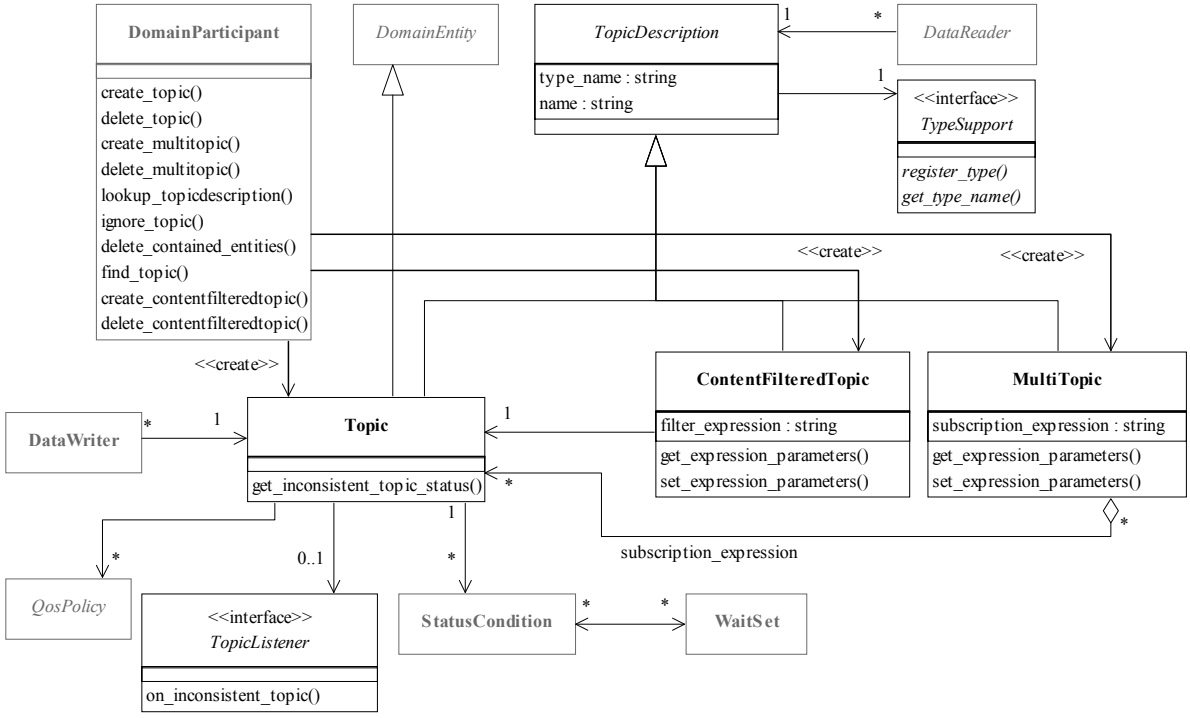

Figure 2-7 Class model of the DCPS Topic-definition Module

The Topic-Definition Module is comprised of the following classes:

- TopicDescription
- Topic
- ContentFilteredTopic
- MultiTopic
- TopicListener
- TypeSupport

2.1.2.3.1 TopicDescription Class

This class is an abstract class. It is the base class for ***Topic, ContentFilteredTopic***, and ***MultiTopic***.

TopicDescription		
no attributes		
readonly: name	string	
readonly: type_name	string	
operations		
get_participant		DomainParticipant
get_type_name		string
get_name		string

TopicDescription represents the fact that both publications and subscriptions are tied to a single data-type. Its attribute ***type_name*** defines a unique resulting type for the publication or the subscription and therefore creates an implicit association with a ***TypeSupport***. ***TopicDescription*** has also a ***name*** that allows it to be retrieved locally.

2.1.2.3.1.1 get_participant

This operation returns the ***DomainParticipant*** to which the ***TopicDescription*** belongs.

2.1.2.3.1.2 type_name

The ***type_name*** used to create the ***TopicDescription***.

2.1.2.3.1.3 name

The ***name*** used to create the ***TopicDescription***.

2.1.2.3.2 Topic Class

Topic is the most basic description of the data to be published and subscribed.

Topic		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []

(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
get_inconsistent_topic_status		ReturnCode_t
	out: status	InconsistentTopicStatus

A ***Topic*** is identified by its ***name****, which must be unique in the whole Domain*. In addition (by virtue of extending ***TopicDescription***) it fully specifies the type of the data that can be communicated when publishing or subscribing to the ***Topic***.

Topic is the only ***TopicDescription*** that can be used for publications and therefore associated to a ***DataWriter***.

All operations except for the base-class operations ***set_qos***, ***get_qos***, ***set_listener***, ***get_listener***, ***enable*** and ***get_status_condition*** may return the value NOT_ENABLED.

The following sections describe its operations.

2.1.2.3.2.1 get_inconsistent_topic_status

This method allows the application to retrieve the INCONSISTENT_TOPIC status of the ***Topic***.

Each ***DomainEntity*** has a set of relevant communication statuses. A change of status causes the corresponding ***Listener*** to be invoked and can also be monitored by means of the associated ***StatusCondition***.

The complete list of communication status, their values, and the ***DomainEntities*** they apply to is provided in Section 2.1.4.1, “Communication Status,” on page 2-129.

2.1.2.3.3 ContentFilteredTopic Class

ContentFilteredTopic is a specialization of ***TopicDescription*** that allows for content-based subscriptions.

ContentFilteredTopic

attributes		
readonly: filter_expression	string	
operations		
get_related_topic		Topic
get_expression_parameters		ReturnCode_t
	out: expression_parameters	string []
set_expression_parameters		ReturnCode_t
	expression_parameters	string []

ContentFilteredTopic describes a more sophisticated subscription that indicates the subscriber does not want to necessarily see all values of each instance published under the ***Topic***. Rather, it wants to see only the values whose contents satisfy certain criteria. This class therefore can be used to request content-based subscriptions.

The selection of the content is done using the ***filter_expression*** with parameters ***expression_parameters***.

- The ***filter_expression*** attribute is a string that specifies the criteria to select the data samples of interest. It is similar to the WHERE part of an SQL clause.
- The ***expression_parameters*** attribute is a sequence of strings that give values to the 'parameters' (i.e., "%n" tokens) in the ***filter_expression***. The number of supplied parameters must fit with the requested values in the ***filter_expression*** (i.e., the number of %n tokens).

Appendix B describes the syntax of ***filter_expression*** and ***expression_parameters***.

2.1.2.3.3.1 get_related_topic

This operation returns the ***Topic*** associated with the ***ContentFilteredTopic***. That is, the ***Topic*** specified when the ***ContentFilteredTopic*** was created.

2.1.2.3.3.2 filter_expression

The ***filter_expression*** associated with the ***ContentFilteredTopic***. That is, the expression specified when the ***ContentFilteredTopic*** was created.

2.1.2.3.3.3 get_expression_parameters

This operation returns the ***expression_parameters*** associated with the ***ContentFilteredTopic***. That is, the parameters specified on the last successful call to ***set_expression_parameters***, or if ***set_expression_parameters*** was never called, the parameters specified when the ***ContentFilteredTopic*** was created.

2.1.2.3.3.4 set_expression_parameters

This operation changes the ***expression_parameters*** associated with the ***ContentFilteredTopic***.

2.1.2.3.4 MultiTopic Class [optional]

MultiTopic is a specialization of ***TopicDescription*** that allows subscriptions to combine/filter/rearrange data coming from several topics.

MultiTopic		
attributes		
readonly: subscription_expression	string	
operations		
get_expression_parameters		ReturnCode_t
	out: expression_parameters	string []
set_expression_parameters		ReturnCode_t
	expression_parameters	string []

MultiTopic allows a more sophisticated subscription that can select and combine data received from multiple topics into a single resulting type (specified by the inherited ***type_name***). The data will then be filtered (selection) and possibly re-arranged (aggregation/projection) according to a ***subscription_expression*** with parameters ***expression_parameters***.

- The ***subscription_expression*** is a string that identifies the selection and re-arrangement of data from the associated topics. It is similar to an SQL clause where the SELECT part provides the fields to be kept, the FROM part provides the names of the topics that are searched for those fields[11], and the WHERE clause gives the content filter. The Topics combined may have different types but they are restricted in that the type of the fields used for the NATURAL JOIN operation must be the same.
- The ***expression_parameters*** attribute is a sequence of strings that give values to the 'parameters' (i.e., "%n" tokens) in the ***subscription_expression***. The number of supplied parameters must fit with the requested values in the ***subscription_expression*** (i.e., the number of %n tokens).
- ***DataReader*** entities associated with a ***MultiTopic*** are alerted of data modifications by the usual ***Listener*** or ***Condition*** mechanisms (see Section 2.1.4, "Listeners, Conditions, and Wait-sets," on page 2-129) whenever modifications occur to the data associated with any of the topics relevant to the ***MultiTopic***.
- ***DataReader*** entities associated with a ***MultiTopic*** access instances that are "constructed" at the ***DataReader*** side from the instances written by multiple ***DataWriter*** entities. The ***MultiTopic*** access instance will begin to exist as soon as all the constituting ***Topic*** instances are in existence. The ***view_state*** and ***instance_state*** is computed from the corresponding states of the constituting instances:

11.It should be noted that in that case, the source for data may not be restricted to a single topic.

- The ***view_state*** (see Section 2.1.2.5.1) of the ***MultiTopic*** instance is NEW if at least one of the constituting instances has ***view_state*** = NEW, otherwise it will be NOT_NEW.
- The ***instance_state*** (see Section 2.1.2.5.1) of the ***MultiTopic*** instance is "ALIVE" if the ***instance_state*** of all the constituting ***Topic*** instances is ALIVE. It is "NOT_ALIVE_DISPOSED" if at least one of the constituting ***Topic*** instances is NOT_ALIVE_DISPOSED. Otherwise it is NOT_ALIVE_NO_WRITERS.

Appendix B describes the syntax of ***subscription_expression*** and ***expression_parameters***.

2.1.2.3.4.1 subscription_expression

The ***subscription_expression*** associated with the ***MultiTopic***. That is, the expression specified when the ***MultiTopic*** was created.

2.1.2.3.4.2 get_expression_parameters

This operation returns the ***expression_parameters*** associated with the ***MultiTopic***. That is, the parameters specified on the last successful call to ***set_expression_parameters***, or if ***set_expression_parameters*** was never called, the parameters specified when the ***MultiTopic*** was created.

2.1.2.3.4.3 set_expression_parameters

This operation changes the ***expression_parameters*** associated with the ***MultiTopic***.

2.1.2.3.5 TopicListener Interface

Since ***Topic*** is a kind of ***Entity***, it has the ability to have an associated listener. In this case, the associated listener should be of concrete type ***TopicListener***.

TopicListener		
no attributes		
operations		
`on_inconsistent_topic`		`void`
	`the_topic`	`Topic`
	`status`	`InconsistentTopicStatus`

2.1.2.3.6 TypeSupport Interface

The ***TypeSupport*** interface is an abstract interface that has to be specialized for each concrete type that will be used by the application.

It is required that each implementation of the Service provides an automatic means to generate this type-specific class from a description of the type (using IDL for example in the OMG IDL mapping). A ***TypeSupport*** must be registered using the ***register_type*** operation on this type-specific class before it can be used to create ***Topic*** objects.

TypeSupport		
no attributes		
operations		
`register_type`		`ReturnCode_t`
	`participant`	`DomainParticipant`
	`type_name`	`string`
`get_type_name`		`string`

2.1.2.3.6.1 register_type

This operation allows an application to communicate to the Service the existence of a data type. The generated implementation of that operation embeds all the knowledge that has to be communicated to the middleware in order to make it able to manage the contents of data of that data type. This includes in particular the ***key*** definition that will allow the Service to distinguish different instances of the same type.

It is a pre-condition error to use the same ***type_name*** to register two different ***TypeSupport*** with the same ***DomainParticipant***. If an application attempts this, the operation will fail and return PRECONDITION_NOT_MET. However, it is allowed to register the same ***TypeSupport*** multiple times with a ***DomainParticipant*** using the same or different values for the ***type_name***. If ***register_type*** is called multiple times on the same ***TypeSupport*** with the same ***DomainParticipant*** and ***type_name*** the second (and subsequent) registrations are ignored but the operation returns OK.

The application may pass nil as the value for the *type_name*. In this case the default type-name as defined by the ***TypeSupport*** (i.e., the value returned by the **get_type_name** operation) will be used.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET and OUT_OF_RESOURCES.

2.1.2.3.6.2 get_type_name

This operation returns the default name for the data-type represented by the ***TypeSupport***.

2.1.2.3.7 Derived Classes for Each Application Class

For each data class defined by the application, there is a number of specialized classes that are required to facilitate the type-safe interaction of the application with the Service.

It is required that each implementation of the Service provides an automatic means to generate all these type-specific classes. ***TypeSupport*** is one of the interfaces that these automatically-generated classes must implement. The complete set of automatic classes created for a hypothetical application data-type named “Foo” are shown in Figure 2-8..

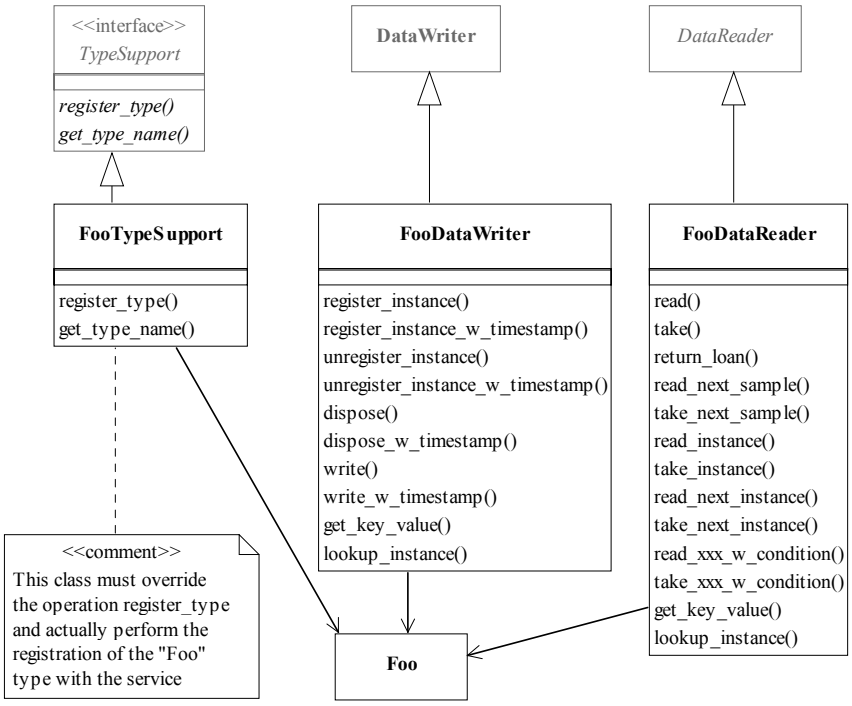

Figure 2-8 Classes auto-created for an application data type named Foo

2.1.2.4 Publication Module

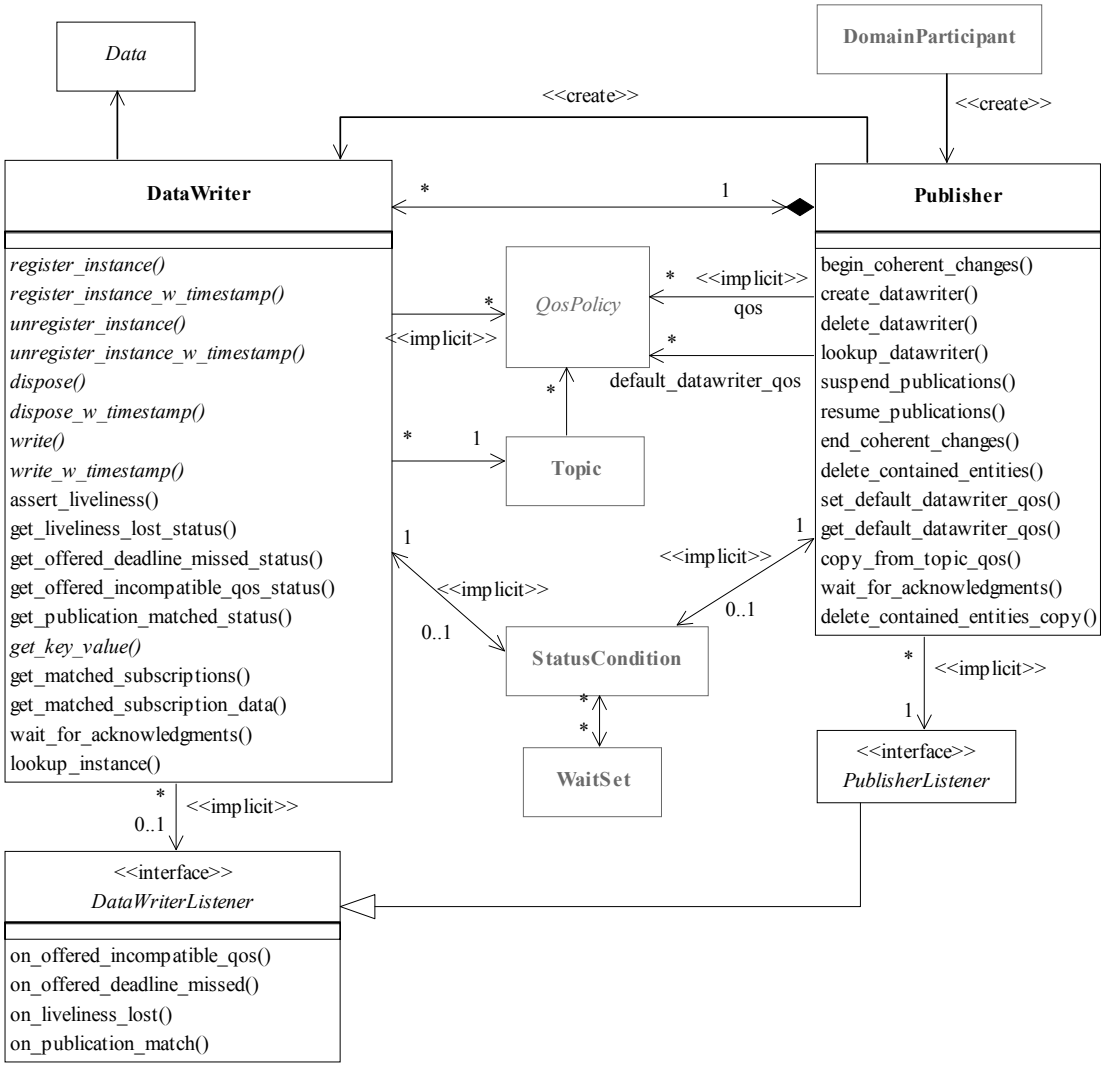

Figure 2-9 Class model of the DCPS Publication Module

The DCPS Publication Module is comprised of the following classifiers:

- Publisher
- DataWriter
- PublisherListener
- DataWriterListener

2.1.2.4.1 Publisher Class

A ***Publisher*** is the object responsible for the actual dissemination of publications.

Publisher		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
create_datawriter		DataWriter
	a_topic	Topic
	qos	QosPolicy []
	a_listener	DataWriterListener
	mask	StatusKind []
delete_datawriter		ReturnCode_t
	a_datawriter	DataWriter
lookup_datawriter		DataWriter
	topic_name	string
suspend_publications		ReturnCode_t
resume_publications		ReturnCode_t
begin_coherent_changes		ReturnCode_t
end_coherent_changes		ReturnCode_t
wait_for_acknowledgments		ReturnCode_t
	max_wait	Duration_t
get_participant		DomainParticipant
delete_contained_entities		ReturnCode_t
set_default_datawriter_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_datawriter_qos		ReturnCode_t
	out: qos_list	QosPolicy []
copy_from_topic_qos		ReturnCode_t
	inout: a_datawriter_qos	QosPolicy []
	a_topic_qos	QosPolicy []

The ***Publisher*** acts on the behalf of one or several ***DataWriter*** objects that belong to it. When it is informed of a change to the data associated with one of its ***DataWriter*** objects, it decides when it is appropriate to actually send the data-update message. In making this decision, it considers any extra information that goes with the data (timestamp, writer, etc.) as well as the QoS of the ***Publisher*** and the ***DataWriter***.

All operations except for the base-class operations ***set_qos***, ***get_qos***, ***set_listener***, ***get_listener***, ***enable, get_statuscondition***, ***create_datawriter***, and ***delete_datawriter*** may return the value NOT_ENABLED.

2.1.2.4.1.1 set_listener (from Entity)

By virtue of extending ***Entity***, a ***Publisher*** can be attached to a ***Listener*** at creation time or later by using the ***set_listener*** operation. The ***Listener*** attached must extend ***PublisherListener***. Listeners are described in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

2.1.2.4.1.2 get_listener (from Entity)

Retrieves the attached **PublisherListener**.

2.1.2.4.1.3 set_qos (from Entity)

By virtue of extending ***Entity***, a ***Publisher*** can be given QoS at creation time or later by using the ***set_qos*** operation. See Section 2.1.3, “Supported QoS,” on page 2-102 for the QoS policies that may be set on a ***Publisher***.

Possible error codes returned in addition to the standard ones: IMMUTABLE_POLICY, INCONSISTENT_POLICY.

2.1.2.4.1.4 get_qos (from Entity)

Allows access to the values of the QoS.

2.1.2.4.1.5 create_ datawriter

This operation creates a ***DataWriter***. The returned ***DataWriter*** will be attached and belongs to the ***Publisher***.

The ***DataWriter*** returned by the ***create_datawriter*** operation will in fact be a derived class, specific to the data-type associated with the Topic. As described in Section 2.1.2.3.7, “Derived Classes for Each Application Class,” on page 2-44, for each application-defined type “Foo” there is an implied, auto-generated class ***FooDataWriter*** that extends ***DataWriter*** and contains the operations to write data of type “Foo.”

In case of failure, the operation will return a ‘nil’ value (as specified by the platform).

Note that a common application pattern to construct the QoS for the ***DataWriter*** is to:

- Retrieve the QoS policies on the associated ***Topic*** by means of the ***get_qos*** operation on the ***Topic***.
- Retrieve the default ***DataWriter*** qos by means of the ***get_default_datawriter_qos*** operation on the ***Publisher***.
- Combine those two QoS policies and selectively modify policies as desired.

- Use the resulting QoS policies to construct the ***DataWriter***.

The special value DATAWRITER_QOS_DEFAULT can be used to indicate that the ***DataWriter*** should be created with the default ***DataWriter*** QoS set in the factory. The use of this value is equivalent to the application obtaining the default ***DataWriter*** QoS by means of the operation ***get_default_datawriter_qos*** (2.1.2.4.1.15) and using the resulting QoS to create the ***DataWriter***.

The special value DATAWRITER_QOS_USE_TOPIC_QOS can be used to indicate that the ***DataWriter*** should be created with a combination of the default ***DataWriter*** QoS and the ***Topic*** QoS. The use of this value is equivalent to the application obtaining the default ***DataWriter*** QoS and the ***Topic*** QoS (by means of the operation ***Topic::get_qos***) and then combining these two QoS using the operation ***copy_from_topic_qos*** whereby any policy that is set on the ***Topic*** QoS "overrides" the corresponding policy on the default QoS. The resulting QoS is then applied to the creation of the ***DataWriter***.

The ***Topic*** passed to this operation must have been created from the same ***DomainParticipant*** that was used to create this ***Publisher***. If the ***Topic*** was created from a different ***DomainParticipant***, the operation will fail and return a nil result.

2.1.2.4.1.6 delete_datawriter

This operation deletes a ***DataWriter*** that belongs to the ***Publisher***.

The ***delete_datawriter*** operation must be called on the same ***Publisher*** object used to create the ***DataWriter***. If ***delete_datawriter*** is called on a different ***Publisher***, the operation will have no effect and it will return PRECONDITION_NOT_MET.

The deletion of the ***DataWriter*** will automatically unregister all instances. Depending on the settings of the WRITER_DATA_LIFECYCLE QosPolicy, the deletion of the ***DataWriter*** may also dispose all instances. Refer to Section 2.1.3.21, "WRITER_DATA_LIFECYCLE," on page 2-125for details.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.4.1.7 lookup_datawriter

This operation retrieves a previously created ***DataWriter*** belonging to the ***Publisher*** that is attached to a ***Topic*** with a matching ***topic_name***. If no such ***DataWriter*** exists, the operation will return 'nil.'

If multiple ***DataWriter*** attached to the ***Publisher*** satisfy this condition, then the operation will return one of them. It is not specified which one.

2.1.2.4.1.8 suspend_publications

This operation indicates to the Service that the application is about to make multiple modifications using ***DataWriter*** objects belonging to the ***Publisher***.

It is a hint to the Service so it can optimize its performance by e.g., holding the dissemination of the modifications and then batching them.

It is not required that the Service use this hint in any way.

The use of this operation must be matched by a corresponding call to ***resume_publications*** indicating that the set of modifications has completed. If the ***Publisher*** is deleted before **resume_publications** is called, any suspended updates yet to be published will be discarded.

2.1.2.4.1.9 resume_publications

This operation indicates to the Service that the application has completed the multiple changes initiated by the previous ***suspend_publications***. This is a hint to the Service that can be used by a Service implementation to e.g., batch all the modifications made since the ***suspend_publications***.

The call to ***resume_publications*** must match a previous call to ***suspend_publications***. Otherwise the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.4.1.10 begin_coherent_changes

This operation requests that the application will begin a 'coherent set' of modifications using ***DataWriter*** objects attached to the ***Publisher***. The 'coherent set' will be completed by a matching call to ***end_coherent_changes***.

A 'coherent set' is a set of modifications that must be propagated in such a way that they are interpreted at the receivers' side as a consistent set of modifications; that is, the receiver will only be able to access the data after all the modifications in the set are available at the receiver end[12].

A connectivity change may occur in the middle of a set of coherent changes; for example, the set of partitions used by the ***Publisher*** or one of its ***Subscribers*** may change, a late-joining ***DataReader*** may appear on the network, or a communication failure may occur. In the event that such a change prevents an entity from receiving the entire set of coherent changes, that entity must behave as if it had received none of the set.

These calls can be nested. In that case, the coherent set terminates only with the last call to ***end_coherent_ changes***.

The support for 'coherent changes' enables a publishing application to change the value of several data-instances that could belong to the same or different topics and have those changes be seen 'atomically' by the readers. This is useful in cases where the values are inter-related (for example, if there are two data-instances representing the 'altitude' and 'velocity vector' of the same aircraft and both are changed, it may be useful to communicate those values in a way the reader can see both together; otherwise, it may e.g., erroneously interpret that the aircraft is on a collision course).

12. This does not imply that the middleware has to encapsulate all the modifications in a single message; it only implies that the receiving applications will behave as if this was the case.

2.1.2.4.1.11 end_coherent_changes

This operation terminates the 'coherent set' initiated by the matching call to ***begin_coherent_ changes***. If there is no matching call to ***begin_coherent_ changes***, the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET

2.1.2.4.1.12 wait_for_acknowledgments

This operation blocks the calling thread until either all data written by the reliable ***DataWriter*** entities is acknowledged by all matched reliable ***DataReader*** entities, or else the duration specified by the ***max_wait*** parameter elapses, whichever happens first. A return value of OK indicates that all the samples written have been acknowledged by all reliable matched data readers; a return value of TIMEOUT indicates that ***max_wait*** elapsed before all the data was acknowledged.

2.1.2.4.1.13 get_participant

This operation returns the ***DomainParticipant*** to which the ***Publisher*** belongs.

2.1.2.4.1.14 delete_contained_entities

This operation deletes all the entities that were created by means of the "create" operations on the ***Publisher***. That is, it deletes all contained ***DataWriter*** objects.

The operation will return PRECONDITION_NOT_MET if the any of the contained entities is in a state where it cannot be deleted.

Once ***delete_contained_entities*** returns successfully, the application may delete the ***Publisher*** knowing that it has no contained ***DataWriter*** objects.

2.1.2.4.1.15 set_default_datawriter_qos

This operation sets a default value of the ***DataWriter*** QoS policies which will be used for newly created ***DataWriter*** entities in the case where the QoS policies are defaulted in the ***create_datawriter*** operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

The special value DATAWRITER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the **set_default_datawriter_qos** operation had never been called.

2.1.2.4.1.16 get_default_datawriter_qos

This operation retrieves the default value of the ***DataWriter*** QoS, that is, the QoS policies which will be used for newly created ***DataWriter*** entities in the case where the QoS policies are defaulted in the ***create_datawriter*** operation.

The values retrieved by ***get_default_datawriter_qos*** will match the set of values specified on the last successful call to ***set_default_datawriter_qos,*** or else, if the call was never made, the default values listed in the QoS table in Section 2.1.3, “Supported QoS,” on page 2-102.

2.1.2.4.1.17 copy_from_topic_qos

This operation copies the policies in the *a_topic_qos* to the corresponding policies in the *a_datawriter_qos* (replacing values in the *a_datawriter_qos*, if present).

This is a “convenience” operation most useful in combination with the operations ***get_default_datawriter_qos*** and ***Topic::get_qos***. The operation *copy_from_topic_qos* can be used to merge the ***DataWriter*** default QoS policies with the corresponding ones on the ***Topic.*** The resulting QoS can then be used to create a new ***DataWriter***, or set its QoS.

This operation does not check the resulting *a_datawriter_qos* for consistency. This is because the ‘merged’ *a_datawriter_qos* may not be the final one, as the application can still modify some policies prior to applying the policies to the ***DataWriter***.

2.1.2.4.2 DataWriter Class

DataWriter allows the application to set the value of the data to be published under a given ***Topic***.

DataWriter		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
register_instance		InstanceHandle_t
	instance	Data
register_instance_w_timestamp		InstanceHandle_t
	instance	Data
	timestamp	Time_t
unregister_instance		ReturnCode_t
	instance	Data
	handle	InstanceHandle_t
unregister_instance_w_timestamp		ReturnCode_t
	instance	Data
	handle	InstanceHandle_t

	timestamp	Time_t
get_key_value		ReturnCode_t
	inout: key_holder	Data
	handle	InstanceHandle_t
lookup_instance		InstanceHandle_t
	instance	Data
write		ReturnCode_t
	data	Data
	handle	InstanceHandle_t
write_w_timestamp		ReturnCode_t
	data	Data
	handle	InstanceHandle_t
	timestamp	Time_t
dispose		ReturnCode_t
	data	Data
	handle	InstanceHandle_t
dispose_w_timestamp		ReturnCode_t
	data	Data
	handle	InstanceHandle_t
	timestamp	Time_t
wait_for_acknowledgments		ReturnCode_t
	max_wait	Duration_t
get_liveliness_lost_status		ReturnCode_t
	out: status	LivelinessLostStatus
get_offered_deadline_missed_status		ReturnCode_t
	out: status	OfferedDeadlineMissedStatus
get_offered_incompatible_qos_status		ReturnCode_t
	out: status	OfferedIncompatibleQosStatus
get_publication_matched_status		ReturnCode_t
	out: status	PublicationMatchedStatus
get_topic		Topic
get_publisher		Publisher
assert_liveliness		ReturnCode_t
get_matched_subscription_data		ReturnCode_t
	inout: subscription_data	SubscriptionBuiltinTopicData
	subscription_handle	InstanceHandle_t
get_matched_subscriptions		ReturnCode_t
	inout: subscription_handles	InstanceHandle_t []

A ***DataWriter*** is attached to exactly one ***Publisher*** that acts as a factory for it.

A ***DataWriter*** is bound to exactly one ***Topic*** and therefore to exactly one data type. The ***Topic*** must exist prior to the ***DataWriter***'s creation.

DataWriter is an abstract class. It must be specialized for each particular application data-type as shown in Figure 2-8. The additional methods that must be defined in the auto-generated class for a hypothetical application type “Foo” are shown in the table below:

FooDataWriter		
no attributes		
operations		
register_instance		InstanceHandle_t
	instance	Foo
register_instance_w_timestamp		InstanceHandle_t
	instance	Foo
	timestamp	Time_t
unregister_instance		ReturnCode_t
	instance	Foo
	handle	InstanceHandle_t
unregister_instance_w_timestamp		ReturnCode_t
	instance	Foo
	handle	InstanceHandle_t
	timestamp	Time_t
get_key_value		ReturnCode_t
	inout: key_holder	Foo
	handle	InstanceHandle_t
lookup_instance		InstanceHandle_t
	instance	Foo
write		ReturnCode_t
	instance_data	Foo
	handle	InstanceHandle_t
write_w_timestamp		ReturnCode_t
	instance_data	Foo
	handle	InstanceHandle_t
	timestamp	Time_t
dispose		ReturnCode_t
	instance	Foo
	handle	InstanceHandle_t
dispose_w_timestamp		ReturnCode_t
	instance	Foo
	handle	InstanceHandle_t
	timestamp	Time_t

All operations except for the base-class operations ***set_qos***, ***get_qos***, ***set_listener***, ***get_listener***, ***enable***, and ***get_statuscondition*** may return the value NOT_ENABLED.

The following sections provide details on the methods.

2.1.2.4.2.1 set_listener (from Entity)

By virtue of extending ***Entity***, a ***DataWriter*** can be attached to a ***Listener*** at creation time or later by using the ***set_listener*** operation. The attached ***Listener*** must extend ***DataWriterListener***. Listeners are described in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

2.1.2.4.2.2 get_listener (from Entity)

Allows access to the attached ***DataWriterListener***.

2.1.2.4.2.3 set_qos (from Entity)

By virtue of extending ***Entity***, a ***DataWriter*** can be given QoS at creation time or later by using the ***set_qos*** operation. See Section 2.1.3, “Supported QoS,” on page 2-102 for the QoS policies that may be set on a ***DataWriter***.

Possible error codes returned in addition to the standard ones: IMMUTABLE_POLICY, INCONSISTENT_POLICY.

2.1.2.4.2.4 get_qos (from Entity)

Allows access to the values of the QoS.

2.1.2.4.2.5 register_instance

This operation informs the Service that the application will be modifying a particular instance. It gives an opportunity to the Service to pre-configure itself to improve performance.

It takes as a parameter an instance (to get the key value) and returns a handle that can be used in successive write or dispose operations.

This operation should be invoked prior to calling any operation that modifies the instance, such as ***write***, ***write_w_timestamp***, ***dispose*** and ***dispose_w_timestamp***.

The special value HANDLE_NIL may be returned by the Service if it does not want to allocate any handle for that instance.

This operation may block and return TIMEOUT under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

This operation may return OUT_OF_RESOURCES under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

The operation ***register_instance*** is idempotent. If it is called for an already registered instance, it just returns the already allocated handle. This may be used to lookup and retrieve the handle allocated to a given instance. The explicit use of this operation is optional as the application may call directly the ***write*** operation and specify a HANDLE_NIL to indicate that the 'key' should be examined to identify the instance.

2.1.2.4.2.6 register_instance_w_timestamp

This operation performs the same function as ***register_instance*** and can be used instead of ***register_instance*** in the cases where the application desires to specify the value for the ***source_timestamp***. The ***source_timestamp*** potentially affects the relative order in which readers observe events from multiple writers. For details see Section 2.1.3.17, "DESTINATION_ORDER," on page 2-123 for the QoS policy DESTINATION_ORDER).

This operation may block and return TIMEOUT under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

This operation may return OUT_OF_RESOURCES under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

2.1.2.4.2.7 unregister_instance

This operation reverses the action of ***register_instance***. It should only be called on an instance that is currently registered.

The operation ***unregister_instance*** should be called just once per instance, regardless of how many times ***register_instance*** was called for that instance.

This operation informs the Service that the ***DataWriter*** is not intending to modify any more of that data instance. This operation also indicates that the Service can locally remove all information regarding that instance. The application should not attempt to use the handle previously allocated to that instance after calling ***unregister_instance***.

The special value HANDLE_NIL can be used for the parameter ***handle.*** This indicates that the identity of the instance should be automatically deduced from the ***instance*** (by means of the ***key***).

If ***handle*** is any value other than HANDLE_NIL, then it must correspond to the value returned by ***register_instance*** when the instance (identified by its ***key***) was registered. Otherwise the behavior is as follows:

- If the ***handle*** corresponds to an existing instance but does not correspond to the same instance referred by the '***instance***' parameter, the behavior is in general unspecified, but if detectable by the Service implementation, the operation shall fail and return the error-code PRECONDITION_NOT_MET.'
- If the ***handle*** does not correspond to an existing instance the behavior is in general unspecified, but if detectable by the Service implementation, the operation shall fail and return the error-code 'BAD_PARAMETER.'

If after that, the application wants to modify (write or dispose) the instance, it has to register it again, or else use the special ***handle*** value HANDLE_NIL.

This operation does not indicate that the instance is deleted (that is the purpose of ***dispose***). The operation ***unregister_instance*** just indicates that the ***DataWriter*** no longer has 'anything to say' about the instance. ***DataReader*** entities that are reading the instance will eventually receive a sample with a NOT_ALIVE_NO_WRITERS instance state if no other ***DataWriter*** entities are writing the instance.

This operation can affect the ownership of the data instance (as described in Section 2.1.3.9 and Section 2.1.3.23.1). If the ***DataWriter*** was the exclusive owner of the instance, then calling ***unregister_instance*** will relinquish that ownership.

This operation may block and return TIMEOUT under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

Possible error codes returned in addition to the standard ones: TIMEOUT, PRECONDITION_NOT_MET.

2.1.2.4.2.8 unregister_instance_w_timestamp

This operation performs the same function as ***unregister_instance*** and can be used instead of ***unregister_instance*** in the cases where the application desires to specify the value for the ***source_timestamp***. The ***source_timestamp*** potentially affects the relative order in which readers observe events from multiple writers. For details see Section 2.1.3.17, "DESTINATION_ORDER," on page 2-123 for the QoS policy DESTINATION_ORDER).

The constraints on the values of the ***handle*** parameter and the corresponding error behavior are the same specified for the ***unregister_instance*** operation (Section 2.1.2.4.2.7).

This operation may block and return TIMEOUT under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

2.1.2.4.2.9 get_key_value

This operation can be used to retrieve the instance key that corresponds to an ***instance_handle***. The operation will only fill the fields that form the key inside the ***key_holder*** instance.

This operation may return BAD_PARAMETER if the ***InstanceHandle_t*** *a_handle* does not correspond to an existing data-object known to the ***DataWriter***. If the implementation is not able to check invalid handles, then the result in this situation is unspecified.

2.1.2.4.2.10 lookup_instance

This operation takes as a parameter an *instance* and returns a handle that can be used in subsequent operations that accept an instance handle as an argument. The *instance* parameter is only used for the purpose of examining the fields that define the key.

This operation does not register the instance in question. If the instance has not been previously registered, or if for any other reason the Service is unable to provide an instance handle, the Service will return the special value HANDLE_NIL.

2.1.2.4.2.11 write

This operation modifies the value of a data instance. When this operation is used, the Service will automatically supply the value of the ***source_timestamp*** that is made available to ***DataReader*** objects by means of the ***source_timestamp*** attribute inside the ***SampleInfo***. See Section 2.1.2.5, “Subscription Module,” on page 2-64 for more details on data timestamps at reader side and Section 2.1.3.17, “DESTINATION_ORDER,” on page 2-123 for the QoS policy DESTINATION_ORDER.

This operation must be provided on the specialized class that is generated for the particular application data-type that is being written. That way the ***data*** argument holding the data has the proper application-defined type (e.g., ‘Foo’).

As a side effect, this operation asserts liveliness on the ***DataWriter*** itself, the ***Publisher*** and the ***DomainParticipant***.

The special value HANDLE_NIL can be used for the parameter ***handle***. This indicates that the identity of the instance should be automatically deduced from the ***instance_data*** (by means of the key).

If ***handle*** is any value other than HANDLE_NIL, then it must correspond to the value returned by ***register_instance*** when the instance (identified by its ***key***) was registered. Otherwise the behavior is as follows:

- If the ***handle*** corresponds to an existing instance but does not correspond to the same instance referred by the '***data***' parameter, the behavior is in general unspecified, but if detectable by the Service implementation, the operation shall fail and return the error-code PRECONDITION_NOT_MET.'
- If the ***handle*** does not correspond to an existing instance the behavior is in general unspecified, but if detectable by the Service implementation, the operation shall fail and return the error-code 'BAD_PARAMETER.'

If the RELIABILITY ***kind*** is set to RELIABLE, the **write** operation may block if the modification would cause data to be lost or else cause one of the limits specified in the RESOURCE_LIMITS to be exceeded. Under these circumstances, the RELIABILITY ***max_blocking_time*** configures the maximum time the **write** operation may block waiting for space to become available. If max_blocking_time elapses before the DataWriter is able to store the modification without exceeding the limits, the write operation will fail and return TIMEOUT.

Specifically, the ***DataWriter*** **write** operation may block in the following situations (note that the list may not be exhaustive), even if its HISTORY kind is KEEP_LAST.

- If (RESOURCE_LIMITS ***max_samples*** < RESOURCE_LIMITS ***max_instances*** * HISTORY ***depth***), then in the situation where the ***max_samples*** resource limit is exhausted the Service is allowed to discard samples of some other instance as long as at least one sample remains for such an instance. If it is still not possible to make space available to store the modification, the writer is allowed to block.
- If (RESOURCE_LIMITS ***max_samples*** < RESOURCE_LIMITS ***max_instances***), then the ***DataWriter*** may block regardless of the HISTORY ***depth***.

Instead of blocking, the write operation is allowed to return immediately with the error code OUT_OF_RESOURCES provided the following two conditions are met:

1. The reason for blocking would be that the RESOURCE_LIMITS are exceeded.
2. The service determines that waiting the '***max_waiting_time***' has no chance of freeing the necessary resources. For example, if the only way to gain the necessary resources would be for the user to unregister an instance.

In case the provided *handle* is valid, i.e. corresponds to an existing instance, but does not correspond to same instance referred by the 'data' parameter, the behavior is in general unspecified, but if detectable by the Service implementation, the return error-code will be 'PRECONDITION_NOT_MET'. In case the *handle* is invalid, the behavior is in general unspecified, but if detectable the returned error-code will be 'BAD_PARAMETER'.

2.1.2.4.2.12 write_w_timestamp

This operation performs the same function as ***write*** except that it also provides the value for the ***source_timestamp*** that is made available to ***DataReader*** objects by means of the ***source_timestamp*** attribute inside the ***SampleInfo***. See Section 2.1.2.5, “Subscription Module,” on page 2-64 for more details on data timestamps at reader side and Section 2.1.3.17, “DESTINATION_ORDER,” on page 2-123 for the QoS policy DESTINATION_ORDER.

The constraints on the values of the ***handle*** parameter and the corresponding error behavior are the same specified for the ***write*** operation (Section 2.1.2.4.2.11).

This operation may block and return TIMEOUT under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

This operation may return OUT_OF_RESOURCES under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

This operation may return PRECONDITION_NOT_MET under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

This operation may return BAD_PARAMETER under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

Similar to ***write***, this operation must also be provided on the specialized class that is generated for the particular application data-type that is being written.

2.1.2.4.2.13 dispose

This operation requests the middleware to delete the data (the actual deletion is postponed until there is no more use for that data in the whole system). In general, applications are made aware of the deletion by means of operations on the ***DataReader*** objects that already knew that instance[13] (see Section 2.1.2.5, “Subscription Module,” on page 2-64 for more details).

13.DataReader objects that didn't know the instance will never see it.

This operation does not modify the value of the instance. The ***instance*** parameter is passed just for the purposes of identifying the instance.

When this operation is used, the Service will automatically supply the value of the ***source_timestamp*** that is made available to ***DataReader*** objects by means of the ***source_timestamp*** attribute inside the ***SampleInfo***.

The constraints on the values of the ***handle*** parameter and the corresponding error behavior are the same specified for the ***unregister_instance*** operation (Section 2.1.2.4.2.7).

This operation may block and return TIMEOUT under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

This operation may return OUT_OF_RESOURCES under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

2.1.2.4.2.14 dispose_w_timestamp

This operation performs the same functions as ***dispose*** except that the application provides the value for the ***source_timestamp*** that is made available to ***DataReader*** objects by means of the ***source_timestamp*** attribute inside the ***SampleInfo*** (see Section 2.1.2.5, “Subscription Module,” on page 2-64).

The constraints on the values of the ***handle*** parameter and the corresponding error behavior are the same specified for the ***dispose*** operation (Section 2.1.2.4.2.13).

This operation may return PRECONDITION_NOT_MET under the same circumstances described for the ***dispose*** operation (Section 2.1.2.4.2.13).

This operation may return BAD_PARAMETER under the same circumstances described for the ***dispose*** operation (Section 2.1.2.4.2.13).

This operation may block and return TIMEOUT under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

This operation may return OUT_OF_RESOURCES under the same circumstances described for the ***write*** operation (Section 2.1.2.4.2.11).

Possible error codes returned in addition to the standard ones: TIMEOUT, PRECONDITION_NOT_MET.

2.1.2.4.2.15 wait_for_acknowledgments

This operation is intended to be used only if the ***DataWriter*** has RELIABILITY QoS kind set to RELIABLE. Otherwise the operation will return immediately with RETCODE_OK.

The operation ***wait_for_acknowledgments*** blocks the calling thread until either all data written by the ***DataWriter*** is acknowledged by all matched ***DataReader*** entities that have RELIABILITY QoS kind RELIABLE, or else the duration specified by the ***max_wait*** parameter elapses, whichever happens first. A return value of OK indicates that all the

samples written have been acknowledged by all reliable matched data readers; a return value of TIMEOUT indicates that ***max_wait*** elapsed before all the data was acknowledged.

2.1.2.4.2.16 get_liveliness_lost_status

This operation allows access to the LIVELINESS_LOST communication status. Communication statuses are described in Section 2.1.4.1, “Communication Status,” on page 2-129.

2.1.2.4.2.17 get_offered_deadline_missed_status

This operation allows access to the OFFERED_DEADLINE_MISSED communication status. Communication statuses are described in section 2.1.4.1.

2.1.2.4.2.18 get_offered_incompatible_qos_status

This operation allows access to the OFFERED_INCOMPATIBLE_QOS communication status. Communication statuses are described in section 2.1.4.1.

2.1.2.4.2.19 get_publication_matched_status

This operation allows access to the PUBLICATION_MATCHED communication status. Communication statuses are described in section 2.1.4.1.

2.1.2.4.2.20 get_topic

This operation returns the ***Topic*** associated with the ***DataWriter***. This is the same ***Topic*** that was used to create the ***DataWriter***.

2.1.2.4.2.21 get_publisher

This operation returns the ***Publisher*** to which the ***DataWriter*** belongs.

2.1.2.4.2.22 assert_liveliness

This operation manually asserts the liveliness of the ***DataWriter***. This is used in combination with the LIVELINESS QoS policy (see Section 2.1.3, “Supported QoS,” on page 2-102) to indicate to the Service that the entity remains active.

This operation need only be used if the LIVELINESS setting is either MANUAL_BY_PARTICIPANT or MANUAL_BY_TOPIC. Otherwise, it has no effect.

Note – Writing data via the ***write*** operation on a ***DataWriter*** asserts liveliness on the ***DataWriter*** itself and its ***DomainParticipant***. Consequently the use of ***assert_liveliness*** is only needed if the application is not writing data regularly.

Complete details are provided in Section 2.1.3.11, “LIVELINESS,” on page 2-119.

2.1.2.4.2.23 get_matched_subscription_data

This operation retrieves information on a subscription that is currently “associated” with the ***DataWriter***; that is, a subscription with a matching ***Topic*** and compatible QoS that the application has not indicated should be “ignored” by means of the ***DomainParticipant ignore_subscription*** operation.

The *subscription_handle* must correspond to a subscription currently associated with the ***DataWriter***, otherwise the operation will fail and return BAD_PARAMETER. The operation ***get_matched_subscriptions*** can be used to find the subscriptions that are currently matched with the ***DataWriter***.

The operation may also fail if the infrastructure does not hold the information necessary to fill in the *subscription_data*. In this case the operation will return UNSUPPORTED.

2.1.2.4.2.24 get_matched_subscriptions

This operation retrieves the list of subscriptions currently "associated" with the ***DataWriter***; that is, subscriptions that have a matching ***Topic*** and compatible QoS that the application has not indicated should be “ignored” by means of the ***DomainParticipant ignore_subscription*** operation.

The handles returned in the 'subscription_handles' list are the ones that are used by the DDS implementation to locally identify the corresponding matched ***DataReader*** entities. These handles match the ones that appear in the 'instance_handle' field of the ***SampleInfo*** when reading the "DCPSSubscriptions" builtin topic.

The operation may fail if the infrastructure does not locally maintain the connectivity information.

2.1.2.4.3 PublisherListener Interface

PublisherListener
no attributes
no operations

Since a ***Publisher*** is a kind of ***Entity***, it has the ability to have a listener associated with it. In this case, the associated listener should be of concrete type ***PublisherListener***. The use of this listener and its relationship to changes in the communication status of the ***Publisher*** is described in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

2.1.2.4.4 DataWriterListener Interface

DataWriterListener		
no attributes		
operations		
on_liveliness_lost		void
	the_writer	DataWriter
	status	LivelinessLostStatus
on_offered_deadline_missed	the_writer	DataWriter
	status	OfferedDeadlineMissedStatus
on_offered_incompatible_qos	the_writer	DataWriter
	status	OfferedIncompatibleQosStatus
on_publication_matched		
	the_writer	DataWriter
	status	PublicationMatchedStatus

Since a ***DataWriter*** is a kind of ***Entity***, it has the ability to have a listener associated with it. In this case, the associated listener should be of concrete type ***DataWriterListener***. The use of this listener and its relationship to changes in the communication status of the ***DataWriter*** is described in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

2.1.2.4.5 Concurrency Behavior

This specification makes no assumption about the way the publishing application is designed. In particular, several ***DataWriter*** may operate in different threads. If they share the same ***Publisher***, the middleware guarantees that its operations are thread-safe. However, it is not required that each requesting thread be treated in isolation from the others (leading e.g., to several isolated sets of coherent changes). If this is the desired behavior, the proper design is to create a ***Publisher*** for each thread.

2.1.2.5 Subscription Module

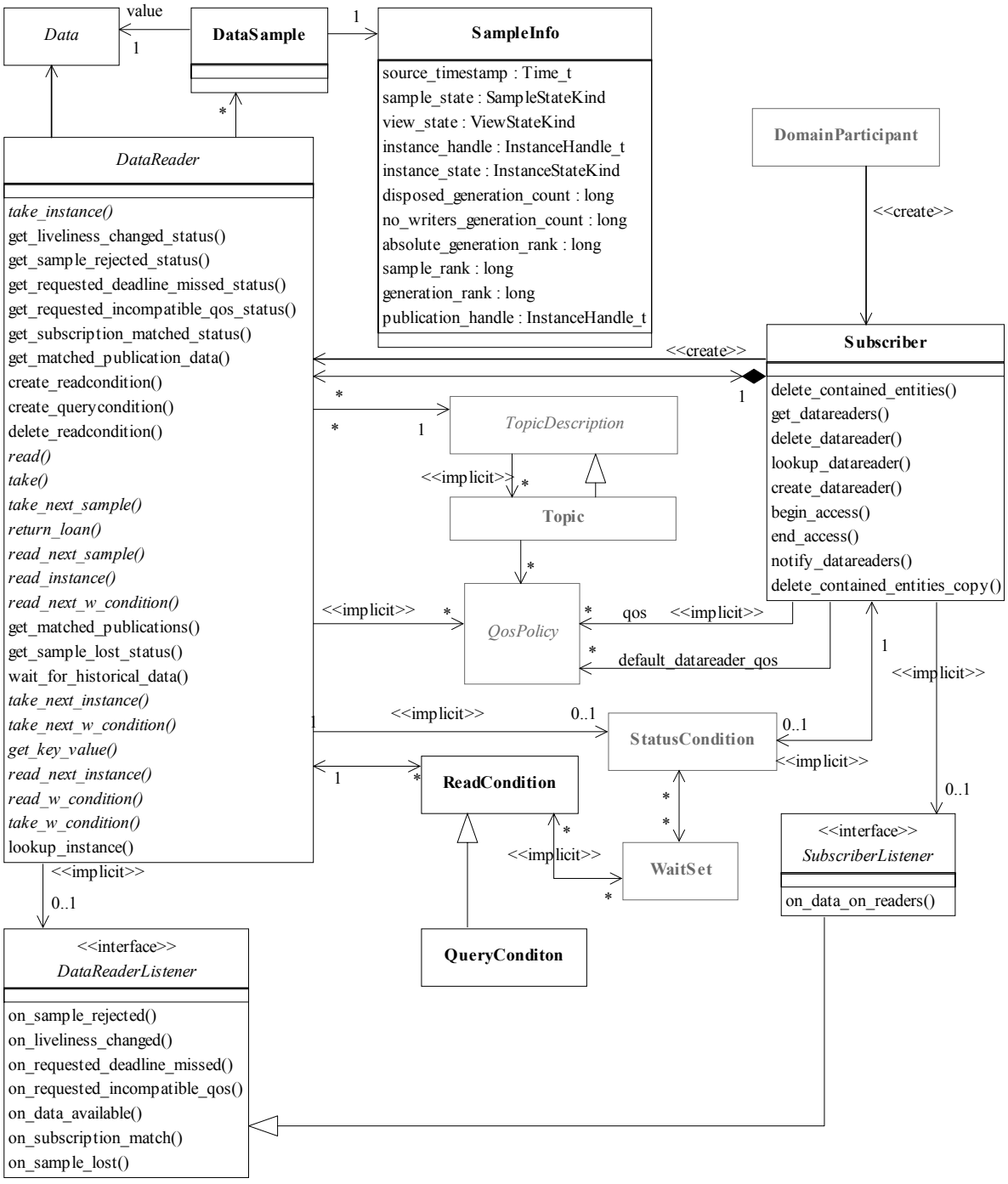

Figure 2-10 Class model of the DCPS Subscription Module

The Subscription Module is comprised of the following classifiers:

- Subscriber
- DataReader
- DataSample
- SampleInfo
- SubscriberListener
- DataReaderListener
- ReadCondition
- QueryCondition

The following section presents how the data can be accessed and introduces the ***sample_state***, ***view_state,*** and ***instance_state***. Section 2.1.2.5.2 (Subscriber Class) through Section 2.1.2.5.9 (QueryCondition Class) provide details on each class belonging to this module.

2.1.2.5.1 Access to the data

Data is made available to the application by the following operations on ***DataReader*** objects: ***read***, ***read_w_condition***, ***take,*** and ***take_w_condition***. The general semantics of the "***read***" operations is that the application only gets access to the corresponding data[14]; the data remains the middleware's responsibility and can be read again. The semantics of the "***take***" operation is that the application takes full responsibility for the data; that data will no longer be accessible to the ***DataReader***. Consequently, it is possible for a ***DataReader*** to access the same sample multiple times but only if all previous accesses were ***read*** operations.

Each of these operations returns an ordered collection of ***Data*** values and associated ***SampleInfo*** objects. Each data value represents an atom of data information (i.e., a value for one instance). This collection may contain samples related to the same or different instances (identified by the ***key***). Multiple samples can refer to the same instance if the settings of the HISTORY QoS (Section 2.1.3.18) allow for it.

2.1.2.5.1.1 Interpretation of the SampleInfo

The *SampleInfo* contains information pertaining to the associated ***Data*** value:

- The ***sample_state*** of the ***Data*** value (i.e., if the sample has already been READ or NOT_READ by that same ***DataReader***).
- The ***view_state*** of the related instance (i.e., if the instance is NEW, or NOT_NEW for that ***DataReader***) – see below.

14.Meaning a precise instance value.

- The ***instance_state*** of the related instance (i.e., if the instance is ALIVE, NOT_ALIVE_DISPOSED, or NOT_ALIVE_NO_WRITERS) – see below.
- The ***valid_data*** flag. This flag indicates whether there is data associated with the sample. Some samples do not contain data indicating only a change on the ***instance_state*** of the corresponding instance – see below.
- The values of ***disposed_generation_count*** and ***no_writers_generation_count*** for the related instance at the time the sample was received. These counters indicate the number of times the instance had become ALIVE (with ***instance_state***= ALIVE) at the time the sample was received – see below.
- The ***sample_rank*** and ***generation_rank*** of the sample within the returned sequence. These ranks provide a preview of the samples that follow within the sequence returned by the ***read*** or ***take*** operations.
- The ***absolute_generation_rank*** of the sample within the ***DataReader***. This rank provides a preview of what is available within the ***DataReader***.
- The ***source_timestamp*** of the sample. This is the timestamp provided by the ***DataWriter*** at the time the sample was produced.

2.1.2.5.1.2 Interpretation of the SampleInfo sample_state

For each sample received, the middleware internally maintains a ***sample_state*** relative to each ***DataReader***. The ***sample_state*** can either be READ or NOT_READ.

- READ indicates that the ***DataReader*** has already accessed that sample by means of ***read***[15].
- NOT_READ indicates that the ***DataReader*** has not accessed that sample before.

The ***sample_state*** will, in general, be different for each sample in the collection returned by ***read*** or ***take***.

2.1.2.5.1.3 Interpretation of the SampleInfo instance_state

For each instance the middleware internally maintains an ***instance_state.*** The ***instance_state*** can be ALIVE, NOT_ALIVE_DISPOSED or NOT_ALIVE_NO_WRITERS.

- ALIVE indicates that (a) samples have been received for the instance, (b) there are live ***DataWriter*** entities writing the instance, and (c) the instance has not been explicitly disposed (or else more samples have been received after it was disposed).
- NOT_ALIVE_DISPOSED indicates the instance was explicitly disposed by a ***DataWriter*** by means of the ***dispose*** operation.

15. Had the sample been accessed by ***take*** it would no longer be available to the ***DataReader***.

- NOT_ALIVE_NO_WRITERS indicates the instance has been declared as not-alive by the ***DataReader*** because it detected that there are no live ***DataWriter*** entities writing that instance.

The precise behavior events that cause the ***instance_state*** to change depends on the setting of the OWNERSHIP QoS:

- If OWNERSHIP is set to EXCLUSIVE, then the ***instance_state*** becomes NOT_ALIVE_DISPOSED only if the ***DataWriter*** that "owns" the instance[16] explicitly disposes it. The ***instance_state*** becomes ALIVE again only if the ***DataWriter*** that owns the instance writes it.
- If OWNERSHIP is set to SHARED, then the ***instance_state*** becomes NOT_ALIVE_DISPOSED if any ***DataWriter*** explicitly disposes the instance. The ***instance_state*** becomes ALIVE as soon as any ***DataWriter*** writes the instance again.

The ***instance_state*** available in the ***SampleInfo*** is a snapshot of the ***instance_state*** of the instance at the time the collection was obtained (i.e., at the time ***read*** or ***take*** was called). The ***instance_state*** is therefore be the same for all samples in the returned collection that refer to the same instance.

2.1.2.5.1.4 Interpretation of the SampleInfo valid_data

Normally each ***DataSample*** contains both a ***SampleInfo*** and some Data. However there are situations where a ***DataSample*** contains only the ***SampleInfo*** and does not have any associated data. This occurs when the Service notifies the application of a change of state for an intance that was caused by some internal mechanism (such as a timeout) for which there is no associated data. An example of this situation is when the Service detects that an instance has no writers and changes the coresponding instance_state to NOT_ALIVE_NO_WRITERS.

The actual set of scenarios under which the middleware returns DataSamples containing no Data is implementation dependent. The application can distinguish wether a particular ***DataSample*** has data by examining the value of the ***valid_data*** flag. If this flag is set to TRUE, then the ***DataSample*** contains valid Data, if the flag is set to FALSE the ***DataSample*** contains no Data.

To ensure corerctness and portability, the ***valid_data*** flag must be examined by the application prior to accessing the Data associated with the ***DataSample*** and if the flag is set to FALSE, the application should not access the Data associated with the ***DataSample***, that is, the application should access only the ***SampleInfo***.

2.1.2.5.1.5 Interpretation of the SampleInfo disposed_generation_count and no_writers_generation_count

For each instance the middleware internally maintains two counts: the ***disposed_generation_count*** and ***no_writers_generation_count,*** relative to each ***DataReader***:

16. The concept of "ownership" is described in Section 2.1.3.9, "OWNERSHIP," on page 2-118.

- The ***disposed_generation_count*** and ***no_writers_generation_count*** are initialized to zero when the ***DataReader*** first detects the presence of a never-seen-before instance.
- The ***disposed_generation_count*** is incremented each time the ***instance_state*** of the corresponding instance changes from NOT_ALIVE_DISPOSED to ALIVE.
- The ***no_writers_generation_count*** is incremented each time the ***instance_state*** of the corresponding instance changes from NOT_ALIVE_NO_WRITERS to ALIVE.

The ***disposed_generation_count*** and ***no_writers_generation_count*** available in the ***SampleInfo*** capture a snapshot of the corresponding counters at the time the sample was received.

2.1.2.5.1.6 Interpretation of the SampleInfo sample_rank, generation_rank, and absolute_generation_rank

The ***sample_rank*** and ***generation_rank*** available in the ***SampleInfo*** are computed based solely on the actual samples in the ordered collection returned by ***read*** or ***take***.

- The ***sample_rank*** indicates the number of samples of the same instance that follow the current one in the collection.
- The ***generation_rank*** available in the ***SampleInfo*** indicates the difference in 'generations' between the sample (S) and the **M**ost **R**ecent **S**ample of the same instance that appears **I**n the returned **C**ollection (**MRSIC**). That is, it counts the number of times the instance transitioned from not-alive to alive in the time from the reception of the S to the reception of MRSIC.

The ***generation_rank*** is computed using the formula:

generation_rank =
(MRSIC.disposed_generation_count + MRSIC.no_writers_generation_count)
- (S.disposed_generation_count + S.no_writers_generation_count)

The ***absolute_generation_rank*** available in the ***SampleInfo*** indicates the difference in 'generations' between the sample (S) and the **M**ost **R**ecent **S**ample of the same instance that the middleware has received (**MRS**). That is, it counts the number of times the instance transitioned from not-alive to alive in the time from the reception of the S to the time when the ***read*** or ***take*** was called.

absolute_generation_rank =
(MRS.disposed_generation_count + MRS.no_writers_generation_count)
- (S.disposed_generation_count + S.no_writers_generation_count)

2.1.2.5.1.7 Interpretation of the SampleInfo counters and ranks

These counters and ranks allow the application to distinguish samples belonging to different 'generations' of the instance. Note that it is possible for an instance to transition from not-alive to alive (and back) several times before the application accesses the data by means of ***read*** or ***take***. In this case the returned collection may contain samples that cross generations (i.e., some samples were received before the instance became not-alive, other after the instance re-appeared again). Using the information in the ***SampleInfo*** the application can anticipate what other information regarding the same instance appears in

the returned collection, as well as, in the infrastructure and thus make appropriate decisions. For example, an application desiring to only consider the most current sample for each instance would only look at samples with ***sample_rank***==0. Similarly an application desiring to only consider samples that correspond to the latest generation in the collection will only look at samples with ***generation_rank***==0. An application desiring only samples pertaining to the latest generation available will ignore samples for which ***absolute_generation_rank*** != 0. Other application-defined criteria may also be used.

2.1.2.5.1.8 Interpretation of the SampleInfo view_state

For each instance (identified by the ***key***), the middleware internally maintains a ***view_state*** relative to each ***DataReader***. The ***view_state*** can either be NEW or NOT_NEW.

- NEW indicates that either this is the first time that the ***DataReader*** has ever accessed samples of that instance, or else that the ***DataReader*** has accessed previous samples of the instance, but the instance has since been reborn (i.e., become not-alive and then alive again). These two cases are distinguished by examining the ***disposed_generation_count*** and the ***no_writers_generation_count***.
- NOT_NEW indicates that the ***DataReader*** has already accessed samples of the same instance and that the instance has not been reborn since.

The ***view_state*** available in the ***SampleInfo*** is a snapshot of the ***view_state*** of the instance relative to the ***DataReader*** used to access the samples at the time the collection was obtained (i.e., at the time ***read*** or ***take*** was called). The ***view_state*** is therefore the same for all samples in the returned collection that refer to the same instance.

Once an instance has been detected as not having any “live” writers and all the samples associated with the instance are ‘taken’ from the ***DataReader***, the middleware can reclaim all local resources regarding the instance. Future samples will be treated as ‘never seen’.

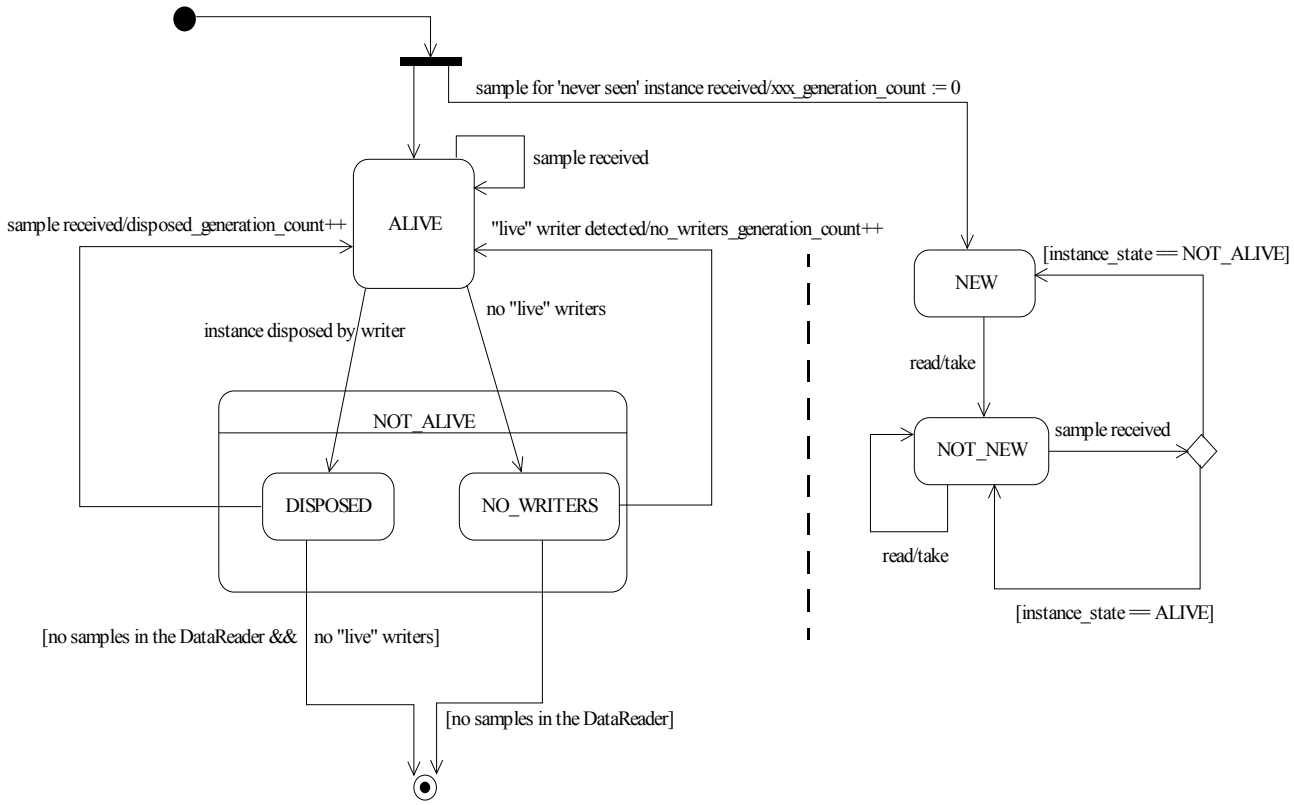

Figure 2-11 Statechart of the ***instance_state*** and ***view_state*** for a single instance.

2.1.2.5.1.9 Data access patterns

The application accesses data by means of the operations ***read*** or ***take*** on the ***DataReader***. These operations return an ordered collection of ***DataSamples*** consisting of a ***SampleInfo*** part and a ***Data*** part. The way the middleware builds this collection[17] depends on QoS policies set on the ***DataReader*** and ***Subscriber***, as well as the source timestamp of the samples, and the parameters passed to the ***read***/***take*** operations, namely:

- The desired sample states (i.e., READ, NOT_READ, or both).
- The desired view states (i.e., NEW, NOT_NEW, or both).

17. i.e., the data-samples that are parts of the list as well as their order.

- The desired instance states (ALIVE, NOT_ALIVE_DISPOSED, NOT_ALIVE_NO_WRITERS, or a combination of these).

The ***read*** and ***take*** operations are non-blocking and just deliver what is currently available that matches the specified states.

The ***read_w_condition*** and ***take_w_condition*** operations take a ***ReadCondition*** object as a parameter instead of sample, view, and instance states. The behavior is that the samples returned will only be those for which the condition is TRUE. These operations, in conjunction with ***ReadCondition*** objects and a ***WaitSet***, allow performing waiting reads (see below).

Once the data samples are available to the data readers, they can be read or taken by the application. The basic rule is that the application may do this in any order it wishes. This approach is very flexible and allows the application ultimate control. However, the application must use a specific access pattern in case it needs to retrieve samples in the proper order received, or it wants to access a complete set of coherent changes.

To access data coherently, or in order, the PRESENTATION QoS (explained in Section 2.1.3.6, "PRESENTATION," on page 2-115) must be set properly and the application must conform to the access pattern described below. Otherwise, the application will still access the data but will not necessarily see all coherent changes together, nor will it see the changes in the proper order.

There is a general pattern that will provide both ordered and coherent access across multiple ***DataReaders***. This pattern will work for any settings of the PRESENTATION QoS. Simpler patterns may also work for specific settings of the QoS as described below.

1. **General pattern to access samples as a coherent set and/or in order across *DataReader* entities**. This case applies when PRESENTATION QoS specifies "***access_scope***=GROUP."
   - Upon notification to the ***SubscriberListener*** or following the similar ***StatusCondition***[18] enabled, the application uses ***begin_access*** on the ***Subscriber*** to indicate it will be accessing data through the ***Subscriber***.
   - Then it calls get ***get_datareaders*** on the ***Subscriber*** to get the list of ***DataReader*** objects where data samples are available.
   - Following this it calls ***read*** or ***take*** on each ***DataReader*** in the same order returned to access all the relevant changes in the ***DataReader***.
   - Once it has called ***read*** or ***take*** on all the readers, it calls ***end_access***.

   Note that if the PRESENTATION QoS policy specifies ***ordered_access***=TRUE, then the list of ***DataReaders*** may return the same reader several times. In this manner the correct sample order can be maintained among samples in different ***DataReader*** objects.

2. **Specialized pattern if no order or coherence needs to be maintained across *DataReader* entities**. This case applies if PRESENTATION QoS policy specifies ***access_scope*** something other than GROUP.

18.For instance, on ***Subscriber*** with mask referring to DATA_ON_READERS.

- In this case, it is not required for the application to call ***begin_access*** and ***end_access***. However, doing so is not an error and it will have no effect.
- The application accesses the data by calling ***read*** or ***take***[19] on each ***DataReader*** in any order it wishes.
- The application can still call ***get_datareaders*** to determine which readers have data to be read, but it does not need to read all of them, nor read them in a particular order. Furthermore, the return of ***get_datareaders*** will be logically a "set" in the sense that the same reader will not appear twice, and the order of the readers returned is not specified.

3. **Specialized pattern if the application accesses the data within the *SubscriberListener*.** This case applies regardless of the PRESENTATION QoS policy when the application accesses the data inside the listener's implementation of the ***on_data_on_readers*** operation.
   - Similar to the previous case (2 above), it is not required for the application to call ***begin_access*** and ***end_access,*** but doing so has no effect.
   - The application can access data by calling ***read*** or ***take***[20] on each ***DataReader*** in any order it wishes.
   - The application can also delegate the accessing of the data to the ***DataReaderListener*** objects installed on each ***DataReader*** by calling ***notify_datareaders***.
   - Similar to the previous case (2 above), the application can still call ***get_datareaders*** to determine which readers have data to be read, but it does not need to read all of them, nor read them in a particular order. Furthermore, the return of ***get_datareaders*** will be logically a 'set.'

2.1.2.5.2 Subscriber Class

A ***Subscriber*** is the object responsible for the actual reception of the data resulting from its subscriptions.

Subscriber		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener

19.Or the variants ***read_w_condition*** and ***take_w_condition***.

20.Or the variants ***read_w_condition*** and ***take_w_condition***.

	mask	StatusKind []
create_datareader		DataReader
	a_topic	TopicDescription
	qos	QosPolicy []
	a_listener	DataReaderListener
	mask	StatusKind []
delete_datareader		ReturnCode_t
	a_datareader	DataReader
lookup_datareader		DataReader
	topic_name	string
begin_access		ReturnCode_t
end_access		ReturnCode_t
get_datareaders		ReturnCode_t
	out: readers	DataReader []
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
notify_datareaders		ReturnCode_t
get_participant		DomainParticipant
delete_contained_entities		ReturnCode_t
set_default_datareader_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_datareader_qos		ReturnCode_t
	out: qos_list	QosPolicy []
copy_from_topic_qos		ReturnCode_t
	inout: a_datareader_qos	QosPolicy []
	a_topic_qos	QosPolicy []

A ***Subscriber*** acts on the behalf of one or several ***DataReader*** objects that are related to it. When it receives data (from the other parts of the system), it builds the list of concerned ***DataReader*** objects, and then indicates to the application that data is available, through its listener or by enabling related conditions. The application can access the list of concerned ***DataReader*** objects through the operation ***get_datareaders*** and then access the data available though operations on the ***DataReader***.

All operations except for the base-class operations ***set_qos***, ***get_qos***, ***set_listener***, ***get_listener***, ***enable***, ***get_statuscondition***, and ***create_datareader*** may return the value NOT_ENABLED.

2.1.2.5.2.1 set_listener (from Entity)

By virtue of extending ***Entity***, a ***Subscriber*** can be attached to a ***Listener*** at creation time or later by using the ***set_listener*** operation. The ***Listener*** attached must extend ***SubscriberListener***. Listeners are described in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

2.1.2.5.2.2 get_listener (from Entity)

Allows access to the attached ***SubscriberListener***.

2.1.2.5.2.3 set_qos (from Entity)

By virtue of extending ***Entity***, a ***Subscriber*** can be given QoS at creation time or later by using the ***set_qos*** operation. See Section 2.1.3, “Supported QoS,” on page 2-102 for the list of QoS policies that may be set on a ***Subscriber***.

Possible error codes returned in addition to the standard ones: IMMUTABLE_POLICY, INCONSISTENT_POLICY.

2.1.2.5.2.4 get_qos (from Entity)

Allows access to the values of the QoS.

2.1.2.5.2.5 create_datareader

This operation creates a ***DataReader***. The returned ***DataReader*** will be attached and belong to the ***Subscriber***.

The ***DataReader*** returned by the ***create_datareader*** operation will in fact be a derived class, specific to the data-type associated with the Topic. As described in Section 2.1.2.3.7, for each application-defined type “Foo” there is an implied auto-generated class ***FooDataReader*** that extends ***DataReader*** and contains the operations to read data of type “Foo.”

In case of failure, the operation will return a ‘nil’ value (as specified by the platform).

Note that a common application pattern to construct the QoS for the ***DataReader*** is to:

- Retrieve the QoS policies on the associated ***Topic*** by means of the ***get_qos*** operation on the ***Topic***.
- Retrieve the default ***DataReader*** qos by means of the ***get_default_datareader_qos*** operation on the ***Subscriber***.
- Combine those two QoS policies and selectively modify policies as desired.
- Use the resulting QoS policies to construct the ***DataReader***.

The special value DATAREADER_QOS_DEFAULT can be used to indicate that the ***DataReader*** should be created with the default ***DataReader*** QoS set in the factory. The use of this value is equivalent to the application obtaining the default ***DataReader*** QoS by means of the operation ***get_default_datareader_qos*** (Section 2.1.2.4.1.15) and using the resulting QoS to create the ***DataReader***.

Provided that the ***TopicDescription*** passed to this method is a ***Topic*** or a ***ContentFilteredTopic***, the special value DATAREADER_QOS_USE_TOPIC_QOS can be used to indicate that the ***DataReader*** should be created with a combination of the default ***DataReader*** QoS and the ***Topic*** QoS. (In the case of a ContentFilteredTopic, the Topic in question is the ContentFilteredTopic’s “related Topic.”) The use of this value is equivalent to the application obtaining the default ***DataReader*** QoS and the ***Topic*** QoS (by means of the operation ***Topic::get_qos***) and then combining these two QoS using the operation ***copy_from_topic_qos*** whereby any policy that is set on the ***Topic*** QoS “overrides” the corresponding policy on the default QoS. The resulting QoS is then

applied to the creation of the ***DataReader***. It is an error to use DATAREADER_QOS_USE_TOPIC_QOS when creating a ***DataReader*** with a ***MultiTopic***; this method will return a 'nil' value in that case.

The ***TopicDescription*** passed to this operation must have been created from the same ***DomainParticipant*** that was used to create this ***Subscriber***. If the ***TopicDescription*** was created from a different ***DomainParticipant***, the operation will fail and return a nil result.

2.1.2.5.2.6 delete_datareader

This operation deletes a ***DataReader*** that belongs to the ***Subscriber***. If the ***DataReader*** does not belong to the ***Subscriber***, the operation returns the error PRECONDITION_NOT_MET.

The deletion of a ***DataReader*** is not allowed if there are any existing ***ReadCondition*** or ***QueryCondition*** objects that are attached to the ***DataReader***. If the ***delete_datareader*** operation is called on a ***DataReader*** with any of these existing objects attached to it, it will return PRECONDITION_NOT_MET.

The deletion of a ***DataReader*** is not allowed if it has any outstanding loans as a result of a call to **read**, **take**, or one of the variants thereof. If the **delete_datareader** operation is called on a ***DataReader*** with one or more outstanding loans, it will return PRECONDITION_NOT_MET.

The ***delete_datareader*** operation must be called on the same ***Subscriber*** object used to create the ***DataReader***. If ***delete_datareader*** is called on a different ***Subscriber***, the operation will have no effect and it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.5.2.7 lookup_datareader

This operation retrieves a previously-created ***DataReader*** belonging to the ***Subscriber*** that is attached to a ***Topic*** with a matching ***topic_name***. If no such ***DataReader*** exists, the operation will return 'nil.'

If multiple ***DataReaders*** attached to the ***Subscriber*** satisfy this condition, then the operation will return one of them. It is not specified which one.

The use of this operation on the built-in ***Subscriber*** allows access to the built-in ***DataReader*** entities for the built-in topics[21].

2.1.2.5.2.8 begin_access

This operation indicates that the application is about to access the data samples in any of the ***DataReader*** objects attached to the ***Subscriber***.

The application is required to use this operation only if PRESENTATION ***QosPolicy*** of the ***Subscriber*** to which the ***DataReader*** belongs has the ***access_scope*** set to 'GROUP.'

21.See Section 2.1.5 for more details on built-in topics.

In the aforementioned case, the operation ***begin_access*** must be called prior to calling any of the sample-accessing operations, namely: ***get_datareaders*** on the **Subscriber** and ***read***, ***take***, ***read_w_condition***, ***take_w_condition*** on any ***DataReader.*** Otherwise the sample-accessing operations will return the error PRECONDITION_NOT_MET. Once the application has finished accessing the data samples it must call ***end_access***.

It is not required for the application to call ***begin_access***/***end_access*** if the PRESENTATION ***QosPolicy*** has the ***access_scope*** set to something other than 'GROUP.' Calling ***begin_access***/***end_access*** in this case is not considered an error and has no effect.

The calls to ***begin_access***/***end_access*** may be nested. In that case, the application must call ***end_access*** as many times as it called ***begin_access***.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.5.2.9 end_access

Indicates that the application has finished accessing the data samples in ***DataReader*** objects managed by the ***Subscriber***.

This operation must be used to 'close' a corresponding ***begin_access***.

After calling ***end_access*** the application should no longer access any of the ***Data*** or ***SampleInfo*** elements returned from the sample-accessing operations. This call must close a previous call to ***begin_access*** otherwise the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.5.2.10 get_datareaders

This operation allows the application to access the ***DataReader*** objects that contain samples with the specified ***sample_states***, ***view_states,*** and ***instance_states***.

If the PRESENTATION ***QosPolicy*** of the ***Subscriber*** to which the ***DataReader*** belongs has the ***access_scope*** set to 'GROUP.' This operation should only be invoked inside a ***begin_access***/***end_access*** block. Otherwise it will return the error PRECONDITION_NOT_MET.

Depending on the setting of the PRESENTATION QoS policy (see Section 2.1.3.6, "PRESENTATION," on page 2-115), the returned collection of ***DataReader*** objects may be a 'set' containing each ***DataReader*** at most once in no specified order, or a 'list' containing each ***DataReader*** one or more times in a specific order.

1. If PRESENTATION ***access_scope*** is INSTANCE or TOPIC the returned collection is a 'set.'
2. If PRESENTATION ***access_scope*** is GROUP and ***ordered_access*** is set to TRUE, then the returned collection is a 'list.'

This difference is due to the fact that, in the second situation it is required to access samples belonging to different ***DataReader*** objects in a particular order. In this case, the application should process each ***DataReader*** in the same order it appears in the 'list' and read or take exactly one sample from each ***DataReader***. The patterns that an application should use to access data is fully described in section 2.1.2.5.1 "Access to the data."

2.1.2.5.2.11 notify_datareaders

This operation invokes the operation ***on_data_available*** on the ***DataReaderListener*** objects attached to contained ***DataReader*** entities with a DATA_AVAILABLE status that is considered changed as described in Section 2.1.4.2.2, "Changes in Read Communication Statuses" .

This operation is typically invoked from the ***on_data_on_readers*** operation in the ***SubscriberListener***. That way the ***SubscriberListener*** can delegate to the ***DataReaderListener*** objects the handling of the data.

2.1.2.5.2.12 get_sample_lost_status

This operation allows access to the SAMPLE_LOST communication status. Communication statuses are described in Section 2.1.4.1, "Communication Status," on page 2-129.

2.1.2.5.2.13 get_participant

This operation returns the ***DomainParticipant*** to which the ***Subscriber*** belongs.

2.1.2.5.2.14 delete_contained_entities

This operation deletes all the entities that were created by means of the "create" operations on the ***Subscriber***. That is, it deletes all contained ***DataReader*** objects. This pattern is applied recursively. In this manner the operation ***delete_contained_entities*** on the ***Subscriber*** will end up deleting all the entities recursively contained in the ***Subscriber***, that is also the ***QueryCondition*** and ***ReadCondition*** objects belonging to the contained ***DataReaders***.

The operation will return PRECONDITION_NOT_MET if the any of the contained entities is in a state where it cannot be deleted. This will occur, for example, if a contained ***DataReader*** cannot be deleted because the application has called a ***read*** or ***take*** operation and has not called the corresponding ***return_loan*** operation to return the loaned samples.

Once ***delete_contained_entities*** returns successfully, the application may delete the ***Subscriber*** knowing that it has no contained ***DataReader*** objects.

2.1.2.5.2.15 set_default_datareader_qos

This operation sets a default value of the ***DataReader*** QoS policies which will be used for newly created ***DataReader*** entities in the case where the QoS policies are defaulted in the ***create_datareader*** operation.

This operation will check that the resulting policies are self consistent; if they are not the operation will have no effect and return INCONSISTENT_POLICY.

The special value DATAREADER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the **set_default_datareader_qos** operation had never been called.

2.1.2.5.2.16 get_default_datareader_qos

This operation retrieves the default value of the ***DataReader*** QoS, that is, the QoS policies which will be used for newly created ***DataReader*** entities in the case where the QoS policies are defaulted in the ***create_datareader*** operation.

The values retrieved ***get_default_datareader_qos*** will match the set of values specified on the last successful call to ***get_default_datareader_qos,*** or else, if the call was never made, the default values listed in the QoS table in Section 2.1.3, “Supported QoS,” on page 2-102.

2.1.2.5.2.17 copy_from_topic_qos

This operation copies the policies in the *a_topic_qos* to the corresponding policies in the *a_datareader_qos* (replacing values in the *a_datareader_qos*, if present).

This is a “convenience” operation most useful in combination with the operations ***get_default_datareader_qos*** and ***Topic::get_qos***. The operation *copy_from_topic_qos* can be used to merge the ***DataReader*** default QoS policies with the corresponding ones on the ***Topic***. The resulting QoS can then be used to create a new ***DataReader***, or set its QoS.

This operation does not check the resulting *a_datareader_qos* for consistency. This is because the ‘merged’ *a_datareader_qos* may not be the final one, as the application can still modify some policies prior to applying the policies to the ***DataReader***.

2.1.2.5.3 DataReader Class

A ***DataReader*** allows the application (1) to declare the data it wishes to receive (i.e., make a subscription) and (2) to access the data received by the attached ***Subscriber***.

DataReader		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
read		ReturnCode_t
	inout: data_values	Data []

	inout: sample_infos	SampleInfo []
	max_samples	long
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_w_condition		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_condition	ReadCondition
take_w_condition		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_condition	ReadCondition
read_next_sample		ReturnCode_t
	inout: data_value	Data
	inout: sample_info	SampleInfo
take_next_sample		ReturnCode_t
	inout: data_value	Data
	inout: sample_info	SampleInfo
read_instance		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take_instance		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []

	instance_states	InstanceStateKind []
read_next_instance		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take_next_instance		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_next_instance_w_condition		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	a_condition	ReadCondition
take_next_instance_w_condition		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	a_condition	ReadCondition
return_loan		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
get_key_value		ReturnCode_t
	inout: key_holder	Data
	handle	InstanceHandle_t
lookup_instance		InstanceHandle_t
	instance	Data
create_readcondition		ReadCondition
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
create_querycondition		QueryCondition
	sample_states	SampleStateKind []

	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
	query_expression	string
	query_parameters	string []
delete_readcondition		ReturnCode_t
	a_condition	ReadCondition
get_liveliness_changed_status		ReturnCode_t
	out: status	LivelinessChangedStatus
get_requested_deadline_missed_status		ReturnCode_t
	out: status	RequestedDeadlineMissedStatus
get_requested_incompatible_qos_status		ReturnCode_t
	out: status	RequestedIncompatibleQosStatus
get_sample_lost_status		ReturnCode_t
	out: status	SampleLostStatus
get_sample_rejected_status		ReturnCode_t
	out: status	SampleRejectedStatus
get_subscription_matched_status		ReturnCode_t
	out: status	SubscriptionMatchedStatus
get_topicdescription		TopicDescription
get_subscriber		Subscriber
delete_contained_entities		ReturnCode_t
wait_for_historical_data		ReturnCode_t
	max_wait	Duration_t
get_matched_publication_data		ReturnCode_t
	inout: publication_data	PublicationBuiltinTopicData
	publication_handle	InstanceHandle_t
get_matched_publications		ReturnCode_t
	inout: publication_handles	InstanceHandle_t []

A ***DataReader*** refers to exactly one ***TopicDescription*** (either a ***Topic***, a ***ContentFilteredTopic***, or a ***MultiTopic***) that identifies the data to be read. The subscription has a unique resulting type. The data-reader may give access to several instances of the resulting type, which can be distinguished from each other by their *key* (as described in Section 2.1.1.2.2, “Overall Conceptual Model,” on page 2-6).

DataReader is an abstract class. It must be specialized for each particular application data-type as shown in Figure 2-8 on page 2-45. The additional methods that must be defined in the auto-generated class for a hypothetical application type “Foo” are shown in the table below:

FooDataReader		
no attributes		
operations		
read		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_w_condition		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_info	SampleInfo []
	max_samples	long
	a_condition	ReadCondition
take_w_condition		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_condition	ReadCondition
read_next_sample		ReturnCode_t
	inout: data_value	Foo
	inout: sample_info	SampleInfo
take_next_sample		ReturnCode_t
	inout: data_value	Foo
	inout: sample_info	SampleInfo
read_instance		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_handle	InstanceHandle_t

	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take_instance		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_next_instance		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take_next_instance		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_next_instance_w_ condition		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	a_condition	ReadCondition
take_next_instance_w_ condition		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	a_condition	ReadCondition

return_loan		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_info	SampleInfo []
get_key_value		ReturnCode_t
	inout: key_holder	Foo
	handle	InstanceHandle_t
lookup_instance		InstanceHandle_t
	instance	Foo

All operations except for the base-class operations ***set_qos***, ***get_qos***, ***set_listener***, ***get_listener***, ***enable***, and ***get_statuscondition*** may return the error NOT_ENABLED.

All sample-accessing operations, namely all variants of ***read***, ***take*** may return the error PRECONDITION_NOT_MET. The circumstances that result on this are described in Section 2.1.2.5.2.8.

The following sections give details on all the operations.

2.1.2.5.3.1 set_listener (from Entity)

By virtue of extending ***Entity***, a ***DataReader*** can be attached to a ***Listener*** at creation time or later by using the ***set_listener*** operation. The ***Listener*** attached must extend ***DataReaderListener***. Listeners are described in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

2.1.2.5.3.2 get_listener (from Entity)

Allows access to the attached ***DataReaderListener***.

2.1.2.5.3.3 set_qos (from Entity)

By virtue of extending ***Entity***, a ***DataReader*** can be given QoS at creation time or later by using the ***set_qos*** operation. See Section 2.1.3, “Supported QoS,” on page 2-102 for the list of QoS policies that may set on a ***DataReader***.

Possible error codes returned in addition to the standard ones: IMMUTABLE_POLICY, INCONSISTENT_POLICY.

2.1.2.5.3.4 get_qos (from Entity)

Allows access to the values of the QoS.

2.1.2.5.3.5 create_readcondition

This operation creates a ***ReadCondition***. The returned ***ReadCondition*** will be attached and belong to the ***DataReader***.

In case of failure, the operation will return a ‘nil’ value (as specified by the platform).

2.1.2.5.3.6 create_querycondition

This operation creates a ***QueryCondition***. The returned ***QueryCondition*** will be attached and belong to the ***DataReader***.

The syntax of the ***query_expression*** and ***query_parameters*** parameters is described in Appendix B.

In case of failure, the operation will return a 'nil' value (as specified by the platform).

2.1.2.5.3.7 delete_readcondition

This operation deletes a ***ReadCondition*** attached to the DataReader. Since ***QueryCondition*** specializes ***ReadCondition*** it can also be used to delete a ***QueryCondition***. If the ***ReadCondition*** is not attached to the ***DataReader***, the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.5.3.8 read

This operation accesses a collection of ***Data*** values from the ***DataReader***. The size of the returned collection will be limited to the specified *max_samples*. The properties of the *data_values* collection and the setting of the PRESENTATION QoS policy (see Section 2.1.3.6, "PRESENTATION," on page 2-115) may impose further limits on the size of the returned 'list.'

1. If PRESENTATION ***access_scope*** is INSTANCE, then the returned collection is a 'list' where samples belonging to the same data-instance are consecutive.
2. If PRESENTATION ***access_scope*** is TOPIC and ***ordered_access*** is set to FALSE, then the returned collection is a 'list' where samples belonging to the same data-instance are consecutive.
3. If PRESENTATION ***access_scope*** is TOPIC and ***ordered_access*** is set to TRUE, then the returned collection is a 'list' were samples belonging to the same instance may or may not be consecutive. This is because to preserve order it may be necessary to mix samples from different instances.
4. If PRESENTATION ***access_scope*** is GROUP and ***ordered_access*** is set to FALSE, then the returned collection is a 'list' where samples belonging to the same data instance are consecutive.
5. If PRESENTATION ***access_scope*** is GROUP and ***ordered_access*** is set to TRUE, then the returned collection contains at most one sample. The difference in this case is due to the fact that it is required that the application is able to read samples belonging to different ***DataReader*** objects in a specific order.

In any case, the relative order between the samples of one instance is consistent with the DESTINATION_ORDER ***QosPolicy***:

- If DESTINATION_ORDER is BY_RECEPTION_TIMESTAMP, samples belonging to the same instances will appear in the relative order in which there were received (FIFO, earlier samples ahead of the later samples).
- If DESTINATION_ORDER is BY_SOURCE_TIMESTAMP, samples belonging to the same instances will appear in the relative order implied by the *source_timestamp* (FIFO, smaller values of *source_timestamp* ahead of the larger values).

In addition to the collection of samples, the ***read*** operation also uses a collection of ***SampleInfo*** structures (*sample_infos*), see Section 2.1.2.5.5, “SampleInfo Class,” on page 2-97.

The initial (input) properties of the *data_values* and *sample_infos* collections will determine the precise behavior of ***read*** operation. For the purposes of this description the collections are modeled as having three properties: the current-length (*len*), the maximum length (*max_len*), and whether the collection container owns the memory of the elements within (*owns*). PSM mappings that do not provide these facilities may need to change the signature of the ***read*** operation slightly to compensate for it.

The initial (input) values of the *len*, *max_len*, and *owns* properties for the *data_values* and *sample_infos* collections govern the behavior of the ***read*** operation as specified by the following rules:

1. The values of *len*, *max_len*, and *owns* for the two collections must be identical. Otherwise ***read*** will and return PRECONDITION_NOT_MET.

2. On successful output, the values of *len*, *max_len*, and *owns* will be the same for both collections.

3. If the input *max_len*==0, then the *data_values* and *sample_infos* collections will be filled with elements that are ‘loaned’ by the ***DataReader***. On output, *owns* will be FALSE, *len* will be set to the number of values returned, and *max_len* will be set to a value verifying *max_len* >= *len*. The use of this variant allows for zero-copy[22] access to the data and the application will need to “return the loan” to the ***DataWriter*** using the ***return_loan*** operation (see Section 2.1.2.5.3.20).

4. If the input *max_len*>0 and the input *owns*==FALSE, then the ***read*** operation will fail and return PRECONDITION_NOT_MET. This avoids the potential hard-to-detect memory leaks caused by an application forgetting to “return the loan.”

5. If input *max_len*>0 and the input *owns*==TRUE, then the ***read*** operation will copy the ***Data*** values and ***SampleInfo*** values into the elements already inside the collections. On output, *owns* will be TRUE, *len* will be set to the number of values copied, and *max_len* will remain unchanged. The use of this variant forces a copy

22.Assuming the implementation supports it.

but the application can control where the copy is placed and the application will not need to "return the loan." The number of samples copied depends on the relative values of *max_len* and *max_samples*:

- If *max_samples* = LENGTH_UNLIMITED, then at most *max_len* values will be copied. The use of this variant lets the application limit the number of samples returned to what the sequence can accommodate.
- If *max_samples* <= *max_len*, then at most *max_samples* values will be copied. The use of this variant lets the application limit the number of samples returned to fewer that what the sequence can accommodate.
- If *max_samples* > *max_len*, then the ***read*** operation will fail and return PRECONDITION_NOT_MET. This avoids the potential confusion where the application expects to be able to access up to *max_samples*, but that number can never be returned, even if they are available in the ***DataReader***, because the output sequence cannot accommodate them.

As described above, upon return the *data_values* and *sample_infos* collections may contain elements "loaned" from the ***DataReader***. If this is the case, the application will need to use the ***return_loan*** operation (see Section 2.1.2.5.3.20) to return the "loan" once it is no longer using the ***Data*** in the collection. Upon return from return_loan, the collection will have max_len=0 and owns=FALSE.

The application can determine whether it is necessary to "return the loan" or not based on how the state of the collections when the read operation was called, or by accessing the 'owns' property. However, in many cases it may be simpler to always call ***return_loan***, as this operation is harmless (i.e., leaves all elements unchanged) if the collection does not have a loan.

To avoid potential memory leaks, the implementation of the ***Data*** and ***SampleInfo*** collections should disallow changing the length of a collection for which owns==FALSE. Furthermore, deleting a collection for which owns==FALSE should be considered an error.

On output, the collection of ***Data*** values and the collection of ***SampleInfo*** structures are of the same length and are in a one-to-one correspondence. Each ***SampleInfo*** provides information, such as the ***source_timestamp***, the ***sample_state***, ***view_state***, and ***instance_state***, etc., about the corresponding sample.

Some elements in the returned collection may not have valid data. If the ***instance_state*** in the ***SampleInfo*** is NOT_ALIVE_DISPOSED or NOT_ALIVE_NO_WRITERS, then the last sample for that instance in the collection, that is, the one whose ***SampleInfo*** has ***sample_rank***==0 does not contain valid data. Samples that contain no data do not count towards the limits imposed by the RESOURCE_LIMITS QoS policy.

The act of reading a sample sets its ***sample_state*** to READ. If the sample belongs to the most recent generation of the instance, it will also set the ***view_state*** of the instance to NOT_NEW. It will not affect the ***instance_state*** of the instance.

This operation must be provided on the specialized class that is generated for the particular application data-type that is being read.

If the ***DataReader*** has no samples that meet the constraints, the return value will be NO_DATA.

2.1.2.5.3.9 take

This operation accesses a collection of data-samples from the ***DataReader*** and a corresponding collection of ***SampleInfo*** structures. The operation will return either a 'list' of samples or else a single sample. This is controlled by the PRESENTATION ***QosPolicy*** using the same logic as for the ***read*** operation (see Section 2.1.2.5.3.8).

The act of taking a sample removes it from the ***DataReader*** so it cannot be 'read' or 'taken' again. If the sample belongs to the most recent generation of the instance, it will also set the ***view_state*** of the instance to NOT_NEW. It will not affect the ***instance_state*** of the instance.

The behavior of the ***take*** operation follows the same rules than the ***read*** operation regarding the pre-conditions and post-conditions for the *data_values* and *sample_infos* collections. Similar to ***read***, the ***take*** operation may 'loan' elements to the output collections which must then be returned by means of ***return_loan***. The only difference with ***read*** is that, as stated, the sampled returned by take will no longer be accessible to successive calls to ***read*** or ***take***.

Similar to ***read***, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the ***DataReader*** has no samples that meet the constraints, the return value will be NO_DATA.

2.1.2.5.3.10 read_w_condition

This operation accesses via 'read' the samples that match the criteria specified in the ***ReadCondition***. This operation is especially useful in combination with ***QueryCondition*** to filter data samples based on the content.

The specified ***ReadCondition*** must be attached to the ***DataReader***; otherwise the operation will fail and return PRECONDITION_NOT_MET.

In case the ***ReadCondition*** is a 'plain' ***ReadCondition*** and not the specialized ***QueryCondition***, the operation is equivalent to calling read and passing as ***sample_states, view_states*** and ***instance_states*** the value of the corresponding attributes in the ***read_condition***. Using this operation the application can avoid repeating the same parameters specified when creating the ***ReadCondition***.

The samples are accessed with the same semantics as the ***read*** operation.

Similar to ***read***, this operation must be provided on the specialized class that is generated for the particular application data-type that is being read.

If the ***DataReader*** has no samples that meet the constraints, the return value will be NO_DATA.

2.1.2.5.3.11 take_w_condition

This operation is analogous to ***read_w_condition*** except it accesses samples via the 'take' operation.

The specified ***ReadCondition*** must be attached to the ***DataReader***; otherwise the operation will fail and return PRECONDITION_NOT_MET.

The samples are accessed with the same semantics as the ***take*** operation.

This operation is especially useful in combination with ***QueryCondition*** to filter data samples based on the content.

Similar to ***take***, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the ***DataReader*** has no samples that meet the constraints, the return value will be NO_DATA.

2.1.2.5.3.12 read_next_sample

This operation copies the next, non-previously accessed ***Data*** value from the ***DataReader***; the operation also copies the corresponding ***SampleInfo***. The implied order among the samples stored in the ***DataReader*** is the same as for the ***read*** operation (section 2.1.2.5.3.8).

The ***read_next_sample*** operation is semantically equivalent to the ***read*** operation where the input ***Data*** sequence has *max_len*=1, the *sample_states*=NOT_READ, the *view_states*=ANY_VIEW_STATE, and the *instance_states*=ANY_INSTANCE_STATE.

The ***read_next_sample*** operation provides a simplified API to 'read' samples avoiding the need for the application to manage sequences and specify states.

If there is no unread data in the ***DataReader***, the operation will return NO_DATA and nothing is copied.

2.1.2.5.3.13 take_next_sample

This operation copies the next, non-previously accessed ***Data*** value from the ***DataReader*** and 'removes' it from the ***DataReader*** so it is no longer accessible. The operation also copies the corresponding ***SampleInfo***. This operation is analogous to the ***read_next_sample*** except for the fact that the sample is 'removed' from the ***DataReader***.

The ***take_next_sample*** operation is semantically equivalent to the ***take*** operation where the input sequence has *max_len*=1, the *sample_states*=NOT_READ, the *view_states*=ANY_VIEW_STATE, and the *instance_states*=ANY_INSTANCE_STATE.

This operation provides a simplified API to 'take' samples avoiding the need for the application to manage sequences and specify states.

If there is no unread data in the ***DataReader***, the operation will return NO_DATA and nothing is copied.

2.1.2.5.3.14 read_instance

This operation accesses a collection of ***Data*** value*s* from the ***DataReader***. The behavior is identical to ***read*** except that all samples returned belong to the single specified instance whose handle is *a_handle*.

Upon successful return, the ***Data*** collection will contain samples all belonging to the same instance. The corresponding ***SampleInfo*** verifies ***instance_handle*** == *a_handle*.

The semantics are the same as for the ***read*** operation, except in building the collection the ***DataReader*** will check that the sample belongs to the specified instance and otherwise it will not place the sample in the returned collection.

The behavior of the ***read_instance*** operation follows the same rules as the ***read*** operation regarding the pre-conditions and post-conditions for the *data_values* and *sample_infos* collections. Similar to ***read***, the ***read_instance*** operation may 'loan' elements to the output collections which must then be returned by means of ***return_loan***.

Similar to ***read***, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the ***DataReader*** has no samples that meet the constraints, the return value will be NO_DATA.

This operation may return BAD_PARAMETER if the ***InstanceHandle_t*** *a_handle* does not correspond to an existing data-object known to the ***DataReader***. If the implementation is not able to check invalid handles, then the result in this situation is unspecified.

2.1.2.5.3.15 take_instance

This operation accesses a collection of ***Data*** value*s* from the ***DataReader***. The behavior is identical to ***take*** except for that all samples returned belong to the single specified instance whose handle is *a_handle*.

The semantics are the same as for the ***take*** operation, except in building the collection the ***DataReader*** will check that the sample belongs to the specified instance and otherwise it will not place the sample in the returned collection.

The behavior of the ***take_instance*** operation follows the same rules as the ***read*** operation regarding the pre-conditions and post-conditions for the *data_values* and *sample_infos* collections. Similar to ***read***, the ***take_instance*** operation may 'loan' elements to the output collections which must then be returned by means of ***return_loan***.

Similar to ***read***, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the ***DataReader*** has no samples that meet the constraints, the return value will be NO_DATA.

This operation may return BAD_PARAMETER if the ***InstanceHandle_t*** *a_handle* does not correspond to an existing data-object known to the ***DataReader***. If the implementation is not able to check invalid handles then the result in this situation is unspecified.

2.1.2.5.3.16 read_next_instance

This operation accesses a collection of ***Data*** value*s* from the ***DataReader*** where all the samples belong to a single instance. The behavior is similar to ***read_instance*** except that the actual instance is not directly specified. Rather the samples will all belong to the 'next' instance with ***instance_handle*** 'greater[23]' than the specified *previous_handle* that has available samples.

This operation implies the existence of a total order 'greater-than' relationship between the instance handles. The specifics of this relationship are not all important and are implementation specific. The important thing is that, according to the middleware, all instances are ordered relative to each other. This ordering is between the instance handles: It should not depend on the state of the instance (e.g. whether it has data or not) and must be defined even for instance handles that do not correspond to instances currently managed by the ***DataReader***. For the purposes of the ordering it should be 'as if' each instance handle was represented as a unique integer.

The behavior of ***read_next_instance*** is 'as if' the ***DataReader*** invoked ***read_instance*** passing the smallest ***instance_handle*** among all the ones that (a) are greater than *previous_handle* and (b) have available samples (i.e., samples that meet the constraints imposed by the specified states).

The special value HANDLE_NIL is guaranteed to be 'less than' any valid ***instance_handle***. So the use of the parameter value *previous_handle*==HANDLE_NIL will return the samples for the instance which has the smallest ***instance_handle*** among all the instances that contain available samples.

The operation ***read_next_instance*** is intended to be used in an application-driven iteration where the application starts by passing *previous_handle*==HANDLE_NIL, examines the samples returned, and then uses the ***instance_handle*** returned in the ***SampleInfo*** as the value of the *previous_handle* argument to the next call to ***read_next_instance***. The iteration continues until ***read_next_instance*** returns the value NO_DATA.

Note that it is possible to call the '***read_next_instance***' operation with a *previous_handle* that does not correspond to an instance currently managed by the ***DataReader***. This is because as stated earlier the 'greater-than' relationship is defined even for handles not managed by the ***DataReader***. One practical situation where this may occur is when an application is iterating though all the instances, takes all the samples of a NOT_ALIVE_NO_WRITERS instance, returns the loan (at which point the instance information may be removed, and thus the handle becomes invalid), and tries to read the next instance.

23.according to some service-defined order.

The behavior of the ***read_next_instance*** operation follows the same rules than the ***read*** operation regarding the pre-conditions and post-conditions for the *data_values* and *sample_infos* collections. Similar to ***read***, the ***read_next_instance*** operation may 'loan' elements to the output collections which must then be returned by means of ***return_loan***.

Similar to ***read***, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the ***DataReader*** has no samples that meet the constraints, the return value will be NO_DATA.

2.1.2.5.3.17 take_next_instance

This operation accesses a collection of ***Data*** value*s* from the ***DataReader*** and 'removes' them from the ***DataReader***.

This operation has the same behavior as ***read_next_instance*** except that the samples are 'taken' from the ***DataReader*** such that they are no longer accessible via subsequent 'read' or 'take' operations.

Similar to the operation ***read_next_instance*** (see Section 2.1.2.5.3.16) it is possible to call ***take_next_instance*** with a *previous_handle* that does not correspond to an instance currently managed by the ***DataReader***.

The behavior of the ***take_next_instance*** operation follows the same rules as the ***read*** operation regarding the pre-conditions and post-conditions for the *data_values* and *sample_infos* collections. Similar to ***read***, the ***take_next_instance*** operation may 'loan' elements to the output collections which must then be returned by means of ***return_loan***.

Similar to ***read***, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the ***DataReader*** has no samples that meet the constraints, the return value will be NO_DATA.

2.1.2.5.3.18 read_next_instance_w_condition

This operation accesses a collection of ***Data*** value*s* from the ***DataReader***. The behavior is identical to ***read_next_instance*** except that all samples returned satisfy the specified condition. In other words, on success all returned samples belong to the same instance, and the instance is the instance with 'smallest' ***instance_handle*** among the ones that verify (a) ***instance_handle*** >= *previous_handle* and (b) have samples for which the specified ***ReadCondition*** evaluates to TRUE.

Similar to the operation ***read_next_instance*** (see Section 2.1.2.5.3.16) it is possible to call ***read_next_instance_w_condition*** with a *previous_handle* that does not correspond to an instance currently managed by the ***DataReader***.

The behavior of the ***read_next_instance_w_condition*** operation follows the same rules than the ***read*** operation regarding the pre-conditions and post-conditions for the *data_values* and *sample_infos* collections. Similar to ***read***, the ***read_next_instance_w_condition*** operation may 'loan' elements to the output collections which must then be returned by means of ***return_loan***.

Similar to ***read***, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the ***DataReader*** has no samples that meet the constraints, the return value will be NO_DATA.

2.1.2.5.3.19 take_next_instance_w_condition

This operation accesses a collection of ***Data*** value*s* from the ***DataReader*** and 'removes' them from the ***DataReader***.

This operation has the same behavior as ***read_next_instance_w_condition*** except that the samples are 'taken' from the ***DataReader*** such that they are no longer accessible via subsequent 'read' or 'take' operations.

Similar to the operation ***read_next_instance*** (see Section 2.1.2.5.3.16) it is possible to call ***take_next_instance_w_condition*** with a *previous_handle* that does not correspond to an instance currently managed by the ***DataReader***.

The behavior of the ***take_next_instance_w_condition*** operation follows the same rules as the ***read*** operation regarding the pre-conditions and post-conditions for the *data_values* and *sample_infos* collections. Similar to ***read***, the ***take_next_instance_w_condition*** operation may 'loan' elements to the output collections which must then be returned by means of ***return_loan***.

Similar to ***read***, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

If the ***DataReader*** has no samples that meet the constraints, the return value will be NO_DATA.

2.1.2.5.3.20 return_loan

This operation indicates to the ***DataReader*** that the application is done accessing the collection of *data_values* and *sample_infos* obtained by some earlier invocation of ***read*** or ***take*** on the ***DataReader***.

The *data_values* and *sample_infos* must belong to a single related 'pair;' that is, they should correspond to a pair returned from a single call to ***read*** or ***take***. The *data_values* and *sample_infos* must also have been obtained from the same ***DataReader*** to which they are returned. If either of these conditions is not met, the operation will fail and return PRECONDITION_NOT_MET.

The operation ***return_loan*** allows implementations of the ***read*** and ***take*** operations to "loan" buffers from the ***DataReader*** to the application and in this manner provide "zero-copy" access to the data. During the loan, the ***DataReader*** will guarantee that the data and sample-information are not modified.

It is not necessary for an application to return the loans immediately after the ***read*** or ***take*** calls. However, as these buffers correspond to internal resources inside the ***DataReader***, the application should not retain them indefinitely.

The use of the ***return_loan*** operation is only necessary if the ***read*** or ***take*** calls "loaned" buffers to the application. As described in Section 2.1.2.5.3.8 this only occurs if the *data_values* and *sample_infos* collections had *max_len*=0 at the time ***read*** or ***take*** was called. The application may also examine the 'owns' property of the collection to determine where there is an outstanding loan. However, calling ***return_loan*** on a collection that does not have a loan is safe and has no side effects.

If the collections had a loan, upon return from ***return_loan*** the collections will have *max_len*=0.

Similar to ***read***, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

2.1.2.5.3.21 get_liveliness_changed_status

This operation allows access to the LIVELINESS_CHANGED communication status. Communication statuses are described in Section 2.1.4.1, "Communication Status," on page 2-129.

2.1.2.5.3.22 get_requested_deadline_missed_status

This operation allows access to the REQUESTED_DEADLINE_MISSED communication status. Communication statuses are described in Section 2.1.4.1, "Communication Status," on page 2-129.

2.1.2.5.3.23 get_requested_incompatible_qos_status

This operation allows access to the REQUESTED_INCOMPATIBLE_QOS communication status. Communication statuses are described in Section 2.1.4.1, "Communication Status," on page 2-129.

2.1.2.5.3.24 get_sample_lost_status

This operation allows access to the SAMPLE_LOST communication status. Communication statuses are described in Section 2.1.4.1, "Communication Status," on page 2-129.

2.1.2.5.3.25 get_sample_rejected_status

This operation allows access to the SAMPLE_REJECTED communication status. Communication statuses are described in Section 2.1.4.1, "Communication Status," on page 2-129.

2.1.2.5.3.26 get_subscription_matched_status

This operation allows access to the SUBSCRIPTION_MATCHED communication status. Communication statuses are described in Section 2.1.4.1, "Communication Status," on page 2-129.

2.1.2.5.3.27 get_topicdescription

This operation returns the ***TopicDescription*** associated with the ***DataReader***. This is the same ***TopicDescription*** that was used to create the ***DataReader***.

2.1.2.5.3.28 get_subscriber

This operation returns the ***Subscriber*** to which the ***DataReader*** belongs.

2.1.2.5.3.29 get_key_value

This operation can be used to retrieve the instance key that corresponds to an ***instance_handle***. The operation will only fill the fields that form the key inside the ***key_holder*** instance.

This operation may return BAD_PARAMETER if the ***InstanceHandle_t*** *a_handle* does not correspond to an existing data-object known to the ***DataReader***. If the implementation is not able to check invalid handles then the result in this situation is unspecified.

2.1.2.5.3.30 lookup_instance

This operation takes as a parameter an *instance* and returns a handle that can be used in subsequent operations that accept an instance handle as an argument. The *instance* parameter is only used for the purpose of examining the fields that define the key.

This operation does not register the instance in question. If the instance has not been previously registered, or if for any other reason the Service is unable to provide an instance handle, the Service will return the special value HANDLE_NIL.

2.1.2.5.3.31 delete_contained_entities

This operation deletes all the entities that were created by means of the “create” operations on the ***DataReader***. That is, it deletes all contained ***ReadCondition*** and ***QueryCondition*** objects.

The operation will return PRECONDITION_NOT_MET if the any of the contained entities is in a state where it cannot be deleted.

Once ***delete_contained_entities*** returns successfully, the application may delete the ***DataReader*** knowing that it has no contained ***ReadCondition*** and ***QueryCondition*** objects.

2.1.2.5.3.32 wait_for_historical_data

This operation is intended only for ***DataReader*** entities that have a non-VOLATILE PERSISTENCE QoS kind.

As soon as an application enables a non-VOLATILE ***DataReader*** it will start receiving both “historical” data, i.e., the data that was written prior to the time the ***DataReader*** joined the domain, as well as any new data written by the ***DataWriter*** entities. There are situations where the application logic may require the application to wait until all “historical” data is received. This is the purpose of the ***wait_for_historical_data*** operation.

The operation ***wait_for_historical_data*** blocks the calling thread until either all "historical" data is received, or else the duration specified by the *max_wait* parameter elapses, whichever happens first. A return value of OK indicates that all the "historical" data was received; a return value of TIMEOUT indicates that *max_wait* elapsed before all the data was received.

2.1.2.5.3.33 get_matched_publication_data

This operation retrieves information on a publication that is currently "associated" with the ***DataReader***; that is, a publication with a matching ***Topic*** and compatible QoS that the application has not indicated should be "ignored" by means of the ***DomainParticipant ignore_publication*** operation.

The *publication_handle* must correspond to a publication currently associated with the ***DataReader*** otherwise the operation will fail and return BAD_PARAMETER. The operation ***get_matched_publications*** can be used to find the publications that are currently matched with the ***DataReader***.

The operation may also fail if the infrastructure does not hold the information necessary to fill in the *publication_data*. In this case the operation will return UNSUPPORTED.

2.1.2.5.3.34 get_matched_publications

This operation retrieves the list of publications currently "associated" with the ***DataReader***; that is, publications that have a matching ***Topic*** and compatible QoS that the application has not indicated should be "ignored" by means of the ***DomainParticipant ignore_ publication*** operation.

The handles returned in the 'publication_handles' list are the ones that are used by the DDS implementation to locally identify the corresponding matched ***DataWriter*** entities. These handles match the ones that appear in the 'instance_handle' field of the ***SampleInfo*** when reading the "DCPSPublications" builtin topic

The operation may fail if the infrastructure does not locally maintain the connectivity information.

2.1.2.5.4 DataSample Class

A ***DataSample*** represents an atom of data information (i.e., one value for one instance) as returned by ***DataReader's*** read/take operations. It consists of two parts: A ***SampleInfo*** and the ***Data***.

2.1.2.5.5 SampleInfo Class

SampleInfo	
attributes	
sample_state	SampleStateKind
view_state	ViewStateKind
instance_state	InstanceStateKind
disposed_generation_count	long
no_writers_generation_count	long
sample_rank	long
generation_rank	long
absolute_generation_rank	long
source_timestamp	Time_t
instance_handle	InstanceHandle_t
publication_handle	InstanceHandle_t
valid_data	boolean
No operations	

SampleInfo is the information that accompanies each sample that is 'read' or 'taken.' It contains the following information:

- The ***sample_state*** (READ or NOT_READ) - indicates whether or not the corresponding data sample has already been read.
- The ***view_state***, (NEW, or NOT_NEW) - indicates whether the ***DataReader*** has already seen samples for the most-current generation of the related instance.
- The ***instance_state*** (ALIVE, NOT_ALIVE_DISPOSED, or NOT_ALIVE_NO_WRITERS) - indicates whether the instance is currently in existence or, if it has been disposed, the reason why it was disposed.
  - ALIVE if this instance is currently in existence.
  - NOT_ALIVE_DISPOSED if this instance was disposed by the a ***DataWriter***.
  - NOT_ALIVE_NO_WRITERS if the instance has been disposed by the ***DataReader*** because none of the ***DataWriter*** objects currently "alive" (according to the LIVELINESS QoS) are writing the instance.
- The ***disposed_generation_count*** that indicates the number of times the instance had become alive after it was disposed explicitly by a ***DataWriter,*** at the time the sample was received.
- The ***no_writers_generation_count*** that indicates the number of times the instance had become alive after it was disposed because there were no writers, at the time the sample was received.
- The ***sample_rank*** that indicates the number of samples related to the same instance that follow in the collection returned by ***read*** or ***take***.

- The ***generation_rank*** that indicates the generation difference (number of times the instance was disposed and become alive again) between the time the sample was received, and the time the most recent sample in the collection related to the same instance was received.
- The ***absolute_generation_rank*** that indicates the generation difference (number of times the instance was disposed and become alive again) between the time the sample was received, and the time the most recent sample (which may not be in the returned collection) related to the same instance was received.
- the ***source_timestamp*** that indicates the time provided by the DataWriter when the sample was written.
- the ***instance_handle*** that identifies locally the corresponding instance.
- the ***publication_handle*** that identifies locally the ***DataWriter*** that modified the instance. The ***publication_handle*** is the same ***InstanceHandle_t*** that is returned by the operation ***get_matched_publications*** on the ***DataReader*** and can also be used as a parameter to the ***DataReader*** operation ***get_matched_publication_data***.
- "the ***valid_data*** flag that indicates whether the ***DataSample*** contains data or else it is only used to communicate of a change in the ***instance_state*** of the instance.

Refer to Section 2.1.2.5.1, ”Access to the data” for a detailed explanation of these states and ranks.

2.1.2.5.6 SubscriberListener Interface

SubscriberListener		
no attributes		
operations		
`on_data_on_readers`		`void`
	`the_subscriber`	`Subscriber`

Since a ***Subscriber*** is a kind of ***Entity***, it has the ability to have an associated listener. In this case, the associated listener should be of concrete type ***SubscriberListener***. Its definition can be found in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

2.1.2.5.7 DataReaderListener Interface

DataReaderListener		
no attributes		
operations		
on_data_available		void
	the_reader	DataReader
on_sample_rejected		void
	the_reader	DataReader
	status	SampleRejectedStatus
on_liveliness_changed		
	the_reader	DataReader
	status	LivelinessChangedStatus
on_requested_deadline_missed		void
	the_reader	DataReader
	status	RequestedDeadlineMissedStatus
on_requested_incompatible_qos		void
	the_reader	DataReader
	status	RequestedIncompatibleQosStatus
on_subscription_matched		
	the_reader	DataReader
	status	SubscriptionMatchedStatus
on_sample_lost		void
	the_reader	DataReader
	status	SampleLostStatus

Since a ***DataReader*** is a kind of ***Entity***, it has the ability to have an associated listener. In this case, the associated listener should be of concrete type ***DataReaderListener***. Its definition can be found in Section 2.1.4, “Listeners, Conditions, and Wait-sets,” on page 2-129.

The operation ***on_subscription_matched*** is intended to inform the application of the discovery of ***DataWriter*** entities that match the ***DataReader***. Some implementations of the service may not propagate this information. In that case the DDS specification does not require this listener operation to be called.

2.1.2.5.8 ReadCondition Class

ReadCondition objects are conditions specifically dedicated to read operations and attached to one ***DataReader***.

ReadCondition	
no attributes	
operations	
`get_datareader`	`DataReader`
`get_sample_state_mask`	`SampleStateKind []`
`get_view_state_mask`	`ViewStateKind []`
`get_instance_state_mask`	`InstanceStateKind []`

ReadCondition objects allow an application to specify the data samples it is interested in (by specifying the desired sample-states, view-states, and instance-states). See the parameter definitions for ***DataReader***'s ***read***/***take*** operations.) This allows the middleware to enable the condition only when suitable information is available[24]. They are to be used in conjunction with a ***WaitSet*** as normal conditions. More than one ***ReadCondition*** may be attached to the same ***DataReader***.

2.1.2.5.8.1 get_datareader

This operation returns the ***DataReader*** associated with the ***ReadCondition***. Note that there is exactly one ***DataReader*** associated with each ***ReadCondition***.

2.1.2.5.8.2 get_sample_state_mask

This operation returns the set of sample-states that are taken into account to determine the ***trigger_value*** of the ***ReadCondition***. These are the sample-states specified when the ***ReadCondition*** was created.

2.1.2.5.8.3 get_view_state_mask

This operation returns the set of view-states that are taken into account to determine the ***trigger_value*** of the ***ReadCondition***. These are the view-states specified when the ***ReadCondition*** was created.

2.1.2.5.8.4 get_instance_state_mask

This operation returns the set of instance-states that are taken into account to determine the ***trigger_value*** of the ***ReadCondition***. These are the instance-states specified when the ***ReadCondition*** was created.

24. For example, the application can specify that the condition must only be enabled when new instances are received by using the NEW view state.

2.1.2.5.9 QueryCondition Class

QueryCondition objects are specialized ***ReadCondition*** objects that allow the application to also specify a filter on the locally available data.

QueryCondition		
no attributes		
operations		
`get_query_expression`		`string`
`get_query_parameters`		`ReturnCode_t`
	`out: query_parameters`	`string []`
`set_query_parameters`		`ReturnCode_t`
	`query_parameters`	`string []`

The query (***query_expression***) is similar to an SQL WHERE clause and can be parameterized by arguments that are dynamically changeable by the ***set_query_parameters*** operation.

Precise syntax for the query expression can be found in Appendix B.

This feature is optional. In the cases where it is not supported, the ***DataReader::create_querycondition*** will return a 'nil' value (as specified by the platform).

2.1.2.5.9.1 get_query_expression

This operation returns the ***query_expression*** associated with the ***QueryCondition***. That is, the expression specified when the ***QueryCondition*** was created.

2.1.2.5.9.2 get_query_parameters

This operation returns the ***query_parameters*** associated with the ***QueryCondition***. That is, the parameters specified on the last successful call to ***set_query_parameters***, or if ***set_query_parameters*** was never called, the arguments specified when the ***QueryCondition*** was created.

2.1.2.5.9.3 set_query_parameters

This operation changes the ***query_parameters*** associated with the ***QueryCondition***.

2.1.3 Supported QoS

The Data-Distribution Service (DDS) relies on the use of QoS. A QoS (Quality of Service) is a set of characteristics that controls some aspect of the behavior of the DDS Service. QoS is comprised of individual QoS policies (objects of type deriving from ***QosPolicy***).

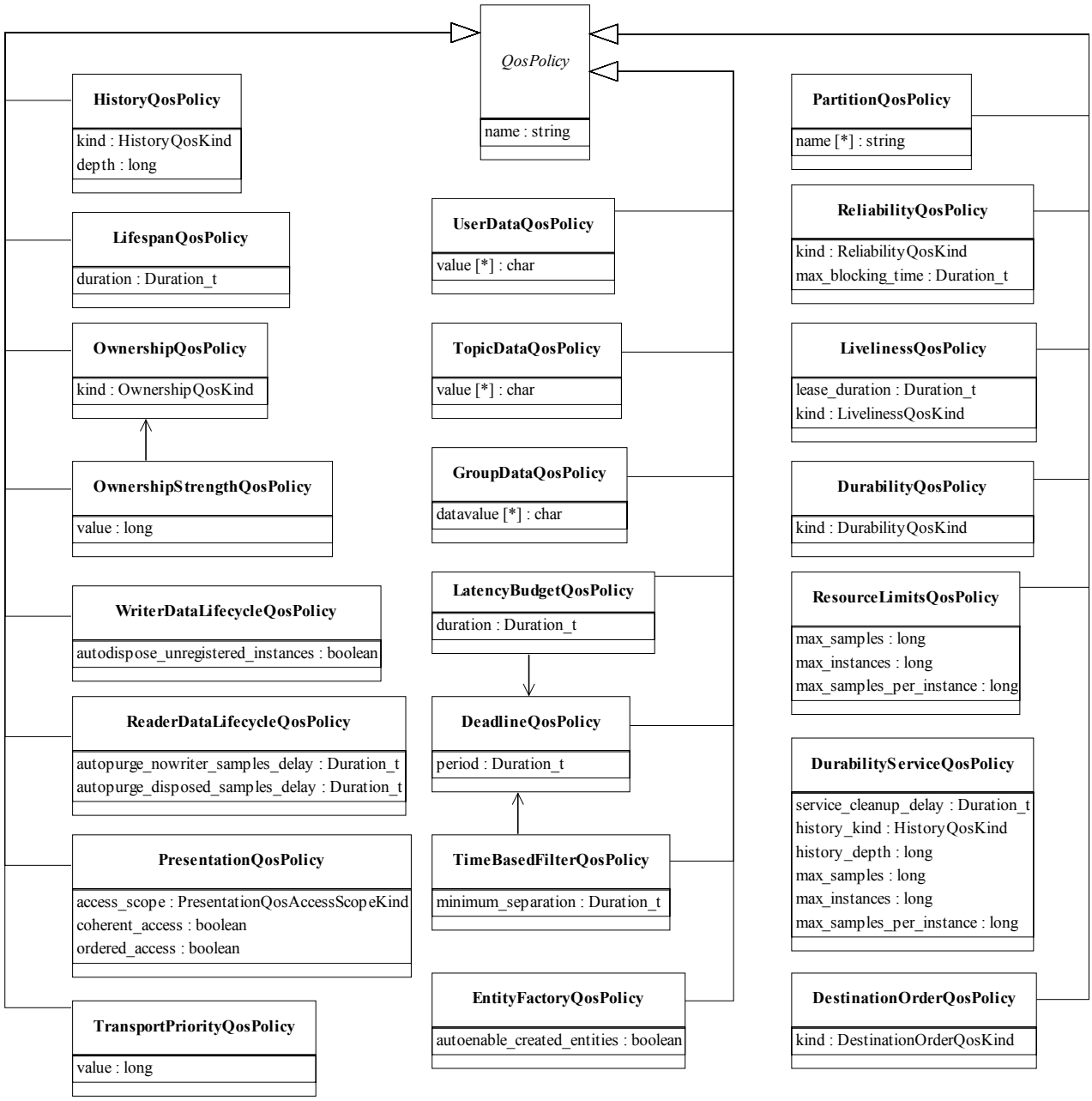

Figure 2-12 Supported QoS policies

QoS (i.e., a list of ***QosPolicy*** objects) may be associated with all ***Entity*** objects in the system such as ***Topic***, ***DataWriter***, ***DataReader***, ***Publisher, Subscriber***, and ***DomainParticipant***.

Some ***QosPolicy*** values may not be consistent with other ones. These cases are described in the table below. When a set of ***QosPolicy*** is passed (***set_qos*** operations), the set resulting from adding the new policies on top of the previous is checked for consistency. If the resulting QoS is inconsistent, the change of QoS operation fails and the previous values are retained.

In several cases, for communications to occur properly (or efficiently), a ***QosPolicy*** on the publisher side must be compatible with a corresponding policy on the subscriber side. For example, if a ***Subscriber*** requests to receive data reliably while the corresponding ***Publisher*** defines a best-effort policy, communication will not happen as requested. To address this issue and maintain the desirable de-coupling of publication and subscription as much as possible, the specification for ***QosPolicy*** follows the subscriber-requested, publisher-offered pattern. In this pattern, the subscriber side can specify a "requested" value for a particular ***QosPolicy***. The Publisher side specifies an "offered" value for that ***QosPolicy***. The Service will then determine whether the value requested by the subscriber side is compatible with what is offered by the publisher side. If the two policies are compatible, then communication will be established. If the two policies are not compatible, the Service will not establish communications between the two ***Entity*** objects and will record this fact by means of the OFFERED_INCOMPATIBLE_QOS on the publisher end and REQUESTED_INCOMPATIBLE_QOS on the subscriber end (see Section 2.1.4.1, "Communication Status," on page 2-129). The application can detect this fact by means of a listener or conditions (see Section 2.1.4, "Listeners, Conditions, and Wait-sets," on page 2-129).

The ***QosPolicy*** objects that need to be set in a compatible manner between the publisher and subscriber ends are indicated by the setting of the 'RxO[25]' property:

- An 'RxO' setting of "Yes" indicates that the policy can be set both at the publishing and subscribing ends and the values must be set in a compatible manner. In this case the compatible values are explicitly defined.
- An 'RxO' setting of "No" indicates that the policy can be set both at the publishing and subscribing ends but the two settings are independent. That is, all combinations of values are compatible.
- An 'RxO' setting of "N/A" indicates that the policy can only be specified at either the publishing or the subscribing end, but not at both ends. So compatibility does not apply.

The 'changeable' property determines whether the ***QosPolicy*** can be changed after the ***Entity*** is enabled. In other words, a policy with 'changeable' setting of 'NO" is considered "immutable" and can only be specified either at ***Entity*** creation time or else prior to calling the ***enable*** operation on the ***Entity***.

25.Requested / Offered

The following tables give the list of supported ***QosPolicy***: their name, semantics, possible values and the ***Entity*** they apply to.

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
USER_DATA	A sequence of octets: “value”	User data not known by the middleware, but distributed by means of built-in topics (cf. Section 2.1.5). The default value is an empty (zero-sized) sequence.	Domain Participant, DataReader, DataWriter	No	Yes
TOPIC_DATA	A sequence of octets: “value”	User data not known by the middleware, but distributed by means of built-in topics (see Section 2.1.5). The default value is an empty (zero-sized) sequence.	Topic	No	Yes
GROUP_DATA	A sequence of octets: “value”	User data not known by the middleware, but distributed by means of built-in topics (see Section 2.1.5). The default value is an empty (zero-sized) sequence.	Publisher, Subscriber	No	Yes
DURABILITY	A “kind”: VOLATILE, TRANSIENT_ LOCAL, TRANSIENT, or PERSISTENT	This policy expresses if the data should 'outlive' their writing time.	Topic, DataReader, DataWriter	Yes	No
	VOLATILE	The Service does not need to keep any samples of data-instances on behalf of any ***DataReader*** that is not known by the ***DataWriter*** at the time the instance is written. In other words the Service will only attempt to provide the data to existing subscribers. This is the default kind.			
	TRANSIENT_ LOCAL, TRANSIENT	The Service will attempt to keep some samples so that they can be delivered to any potential late-joining ***DataReader***. Which particular samples are kept depends on other QoS such as HISTORY and RESOURCE_LIMITS. For TRANSIENT_LOCAL, the service is only required to keep the data in the memory of the *DataWriter* that wrote the data and the data is not required to survive the *DataWriter*. For TRANSIENT, the service is only required to keep the data in memory and not in permanent storage; but the data is not tied to the lifecycle of the *DataWriter* and will, in general, survive it. Support for TRANSIENT kind is optional.			

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
	PERSISTENT	[optional] Data is kept on permanent storage, so that they can outlive a system session.			
DURABILITY_SERVICE	A duration "service_cleanup_ delay" A HistoryQosPolicyKind "history_kind" And three integers: history_depth, max_samples, max_instances, max_samples_per_instance	Specifies the configuration of the durability service. That is, the service that implements the DURABILITY kind of TRANSIENT and PERSISTENT	Topic, DataWriter	No	No
	service_cleanup_delay	Control when the service is able to remove all information regarding a data-instance. By default, zero			
	history_kind, history_depth	Controls the HISTORY QoS of the fictitious DataReader that stores the data within the durability service (see section 2.1.3.4). The default settings are history_kind=KEEP_LAST history_depth=1			
	max_samples, max_instances, max_samples_per_instance	Control the RESOURCE_ LIMITS QoS of the implied ***DataReader*** that stores the data within the durability service. By default they are all LENGTH_UNLIMITED.			
PRESENTATION	An "access_scope": INSTANCE, TOPIC, GROUP And two booleans: "coherent_access" "ordered_access"	Specifies how the samples representing changes to data instances are presented to the subscribing application. This policy affects the application's ability to specify and receive coherent changes and to see the relative order of changes. ***access_scope*** determines the largest scope spanning the entities for which the order and coherency of changes can be preserved. The two booleans control whether coherent access and ordered access are supported within the scope ***access_scope***.	Publisher, Subscriber	Yes	No

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
	INSTANCE	Scope spans only a single instance. Indicates that changes to one instance need not be coherent nor ordered with respect to changes to any other instance. In other words, order and coherent changes apply to each instance separately. This is the default *access_scope*.			
	TOPIC	Scope spans to all instances within the same ***DataWriter*** (or ***DataReader***), but not across instances in different ***DataWriter*** (or ***DataReader***).			
	GROUP	[optional] Scope spans to all instances belonging to ***DataWriter*** (or ***DataReader***) entities within the same ***Publisher*** (or ***Subscriber***).			
	coherent_access	Specifies support ***coherent access***. That is, the ability to group a set of changes as a unit on the publishing end such that they are received as a unit at the subscribing end. The default setting of *coherent_access* is FALSE.			
	ordered_access	Specifies support for ***ordered access*** to the samples received at the subscription end. That is, the ability of the subscriber to see changes in the same order as they occurred on the publishing end. The default setting of *ordered_access* is FALSE.			
DEADLINE	A duration "period"	***DataReader*** expects a new sample updating the value of each instance at least once every deadline ***period***. ***DataWriter*** indicates that the application commits to write a new value (using the ***DataWriter***) for each instance managed by the ***DataWriter*** at least once every deadline ***period***. It is inconsistent for a ***DataReader*** to have a DEADLINE *period* less than its TIME_BASED_FILTER's *minimum_separation*. The default value of the deadline ***period*** is infinite.	Topic, DataReader, DataWriter	Yes	Yes

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
LATENCY_ BUDGET	A duration “duration”	Specifies the maximum acceptable delay from the time the data is written until the data is inserted in the receiver's application-cache and the receiving application is notified of the fact. This policy is a hint to the Service, not something that must be monitored or enforced. The Service is not required to track or alert the user of any violation. The default value of the *duration* is zero indicating that the delay should be minimized.	Topic, DataReader, DataWriter	Yes	Yes
OWNERSHIP	A “kind” SHARED EXCLUSIVE	[optional] Specifies whether it is allowed for multiple ***DataWriters*** to write the same instance of the data and if so, how these modifications should be arbitrated	Topic DataReader, DataWriter	Yes	No
	SHARED	Indicates shared ownership for each instance. Multiple writers are allowed to update the same instance and all the updates are made available to the readers. In other words there is no concept of an “owner” for the instances. This is the default behavior if the OWNERSHIP QoS policy is not specified or supported.			
	EXCLUSIVE	Indicates each instance can only be owned by one ***DataWriter***, but the owner of an instance can change dynamically. The selection of the owner is controlled by the setting of the OWNERSHIP_STRENGTH QoS policy. The owner is always set to be the highest-strength ***DataWriter*** object among the ones currently “active” (as determined by the LIVELINESS QoS).			
OWNERSHIP_ STRENGTH	An integer “value”	[optional] Specifies the value of the “strength” used to arbitrate among multiple ***DataWriter*** objects that attempt to modify the same instance of a data-object (identified by ***Topic*** + ***key***). This policy only applies if the OWNERSHIP QoS policy is of ***kind*** EXCLUSIVE. The default value of the *ownership_strength* is zero.	DataWriter	N/A	Yes

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
LIVELINESS	A “kind”: AUTOMATIC, MANUAL_BY_PARTICIPANT, MANUAL_BY_TOPIC and a duration “lease_duration”	Determines the mechanism and parameters used by the application to determine whether an ***Entity*** is “active” (alive). The “liveliness” status of an Entity is used to maintain instance ownership in combination with the setting of the OWNERSHIP QoS policy. The application is also informed via listener when an ***Entity*** is no longer alive. The ***DataReader*** requests that liveliness of the writers is maintained by the requested means and loss of liveliness is detected with delay not to exceed the ***lease_duration***. The ***DataWriter*** commits to signalling its liveliness using the stated means at intervals not to exceed the ***lease_duration***. Listeners are used to notify the ***DataReader*** of loss of liveliness and ***DataWriter*** of violations to the liveliness contract. The default ***kind*** is AUTOMATIC and the default value of the ***lease_duration*** is infinite.	Topic, DataReader, DataWriter	Yes	No
	AUTOMATIC	The infrastructure will automatically signal liveliness for the ***DataWriters*** at least as often as required by the ***lease_duration***			
	MANUAL modes	The user application takes responsibility to signal liveliness to the Service using one of the mechanisms described in Section 2.1.3.11, “LIVELINESS,” on page 2-119. Liveliness must be asserted at least once every ***lease_duration*** otherwise the Service will assume the corresponding ***Entity*** is no longer “active/alive.”			
	MANUAL_BY_PARTICIPANT	The Service will assume that as long as at least one ***Entity*** within the *DomainParticipant* has asserted its liveliness the other ***Entities*** in that same *DomainParticipant* are also alive.			
	MANUAL_BY_TOPIC	The Service will only assume liveliness of the ***DataWriter*** if the application has asserted liveliness of that ***DataWriter*** itself.			

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
TIME_BASED_ FILTER	A duration "minimum_ separation"	Filter that allows a ***DataReader*** to specify that it is interested only in (potentially) a subset of the values of the data. The filter states that the ***DataReader*** does not want to receive more than one value each ***minimum_separation***, regardless of how fast the changes occur. It is inconsistent for a ***DataReader*** to have a *minimum_separation* longer than its DEADLINE *period.* By default ***minimum_separation***=0 indicating ***DataReader*** is potentially interested in all values.	DataReader	N/A	Yes
PARTITION	A list of strings “name”	Set of strings that introduces a logical partition among the topics visible by the ***Publisher*** and ***Subscriber***. A ***DataWriter*** within a ***Publisher*** only communicates with a ***DataReader*** in a ***Subscriber*** if (in addition to matching the ***Topic*** and having compatible QoS) the ***Publisher*** and ***Subscriber*** have a common partition name string. The empty string ("") is considered a valid partition that is matched with other partition names using the same rules of string matching and regular-expression matching used for any other partition name (see Section 2.1.3.13) The default value for the PARTITION QoS is a zero-length sequence. The zero-length sequence is treated as a special value equivalent to a sequence containing a single element consisting of the empty string.	Publisher, Subscriber	No	Yes
RELIABILITY	A “kind”: RELIABLE, BEST_EFFORT and a duration “max_blocking_ time”	Indicates the level of reliability offered/requested by the Service.	Topic, DataReader, DataWriter	Yes	No

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
	RELIABLE	Specifies the Service will attempt to deliver all samples in its history. Missed samples may be retried. In steady-state (no modifications communicated via the ***DataWriter***) the middleware guarantees that all samples in the ***DataWriter*** history will eventually be delivered to all the ***DataReader***[1] objects. Outside steady state the HISTORY and RESOURCE_LIMITS policies will determine how samples become part of the history and whether samples can be discarded from it. This is the default value for ***DataWriters***.			
	BEST_EFFORT	Indicates that it is acceptable to not retry propagation of any samples. Presumably new values for the samples are generated often enough that it is not necessary to re-send or acknowledge any samples. This is the default value for ***DataReaders*** and ***Topics***.			
	max_blocking_ time	The value of the *max_blocking_time* indicates the maximum time the operation *DataWriter::write* is allowed to block if the *DataWriter* does not have space to store the value written. The default max_blocking_time=100ms.			
TRANSPORT_P RIORITY	An integer “value”	This policy is a hint to the infrastructure as to how to set the priority of the underlying transport used to send the data. The default value of the *transport_priority* is zero.	Topic, DataWriter	N/A	Yes
LIFESPAN	A duration “duration”	Specifies the maximum duration of validity of the data written by the DataWriter The default value of the lifespan *duration* is infinite.	Topic, DataWriter	N/A	Yes
DESTINATION_ ORDER	A “kind”: BY_ RECEPTION_ TIMESTAMP, BY_SOURCE_ TIMESTAMP	Controls the criteria used to determine the logical order among changes made by ***Publisher*** entities to the same instance of data (i.e., matching ***Topic*** and ***key***). The default kind is BY_RECEPTION_TIMESTAMP.	Topic, DataReader, DataWriter	Yes	No
	BY_ RECEPTION_ TIMESTAMP	Indicates that data is ordered based on the reception time at each ***Subscriber***. Since each subscriber may receive the data at different times there is no guaranteed that the changes will be seen in the same order. Consequently, it is possible for each subscriber to end up with a different final value for the data.			

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
	BY_SOURCE_ TIMESTAMP	Indicates that data is ordered based on a timestamp placed at the source (by the Service or by the application). In any case this guarantees a consistent final value for the data in all subscribers.			
HISTORY	A “kind”: KEEP_LAST, KEEP_ALL And an optional integer “depth”	Specifies the behavior of the Service in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing subscribers. This QoS policy controls whether the Service should deliver only the most recent value, attempt to deliver all intermediate values, or do something in between. On the publishing side this policy controls the samples that should be maintained by the ***DataWriter*** on behalf of existing ***DataReader*** entities. The behavior with regards to a ***DataReader*** entities discovered after a sample is written is controlled by the DURABILITY QoS policy. On the subscribing side it controls the samples that should be maintained until the application “takes” them from the Service.	Topic, DataReader, DataWriter	No	No
	KEEP_LAST and optional integer “depth”	On the publishing side, the Service will only attempt to keep the most recent “depth” samples of each instance of data (identified by its key) managed by the ***DataWriter***. On the subscribing side, the ***DataReader*** will only attempt to keep the most recent “depth” samples received for each instance (identified by its ***key***) until the application “takes” them via the ***DataReader’s take*** operation. KEEP_LAST is the default ***kind***. The default value of ***depth*** is 1. If a value other than 1 is specified, it should be consistent with the settings of the RESOURCE_LIMITS QoS policy.			

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
	KEEP_ALL	On the publishing side, the Service will attempt to keep all samples (representing each value written) of each instance of data (identified by its ***key***) managed by the ***DataWriter*** until they can be delivered to all subscribers. On the subscribing side, the Service will attempt to keep all samples of each instance of data (identified by its ***key***) managed by the ***DataReader***. These samples are kept until the application "takes" them from the Service via the ***take*** operation. The setting of ***depth*** has no effect. Its implied value is LENGTH_UNLIMITED[2].			
RESOURCE_ LIMITS	Three integers: max_samples, max_instances, max_samples_ per_instance	Specifies the resources that the Service can consume in order to meet the requested QoS.	Topic, DataReader, DataWriter	No	No
	max_samples	Specifies the maximum number of data-samples the ***DataWriter*** (or ***DataReader***) can manage across all the instances associated with it. Represents the maximum samples the middleware can store for any one ***DataWriter*** (or ***DataReader***). It is inconsistent for this value to be less than *max_samples_per_instance*. By default, LENGTH_UNLIMITED.			
	max_instances	Represents the maximum number of instances ***DataWriter*** (or ***DataReader***) can manage. By default, LENGTH_UNLIMITED[3].			
	max_samples_ per_instance	Represents the maximum number of samples of any one instance a ***DataWriter*** (or ***DataReader***) can manage. It is inconsistent for this value to be greater than *max_samples*. By default, LENGTH_UNLIMITED[4].			
ENTITY_FACTO RY	A boolean: "autoenable_ created_entities"	Controls the behavior of the entity when acting as a factory for other entities. In other words, configures the side-effects of the create_* and delete_* operations.	DomainParticipantFactory, DomainParticipant, Publisher, Subscriber,	No	Yes
	autoenable_ created_entities	Specifies whether the entity acting as a factory automatically enables the instances it creates. If *autoenable_created_entities*==TRUE the factory will automatically enable each created Entity otherwise it will not. By default, TRUE.			

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
WRITER_DATA _LIFECYCLE	A boolean: "autodispose_ unregistered_ instances"	Specifies the behavior of the *DataWriter* with regards to the lifecycle of the data-instances it manages.	DataWriter	N/A	Yes
	autodispose_ unregistered_ instances	Controls whether a *DataWriter* will automatically dispose instances each time they are unregistered. The setting *autodispose_unregistered_instances* = TRUE indicates that unregistered instances will also be considered disposed. By default, TRUE.			
READER_DATA _LIFECYCLE	Two durations "autopurge_ nowriter_samples _delay" and "autopurge_ disposed_ samples_delay"	Specifies the behavior of the *DataReader* with regards to the lifecycle of the data-instances it manages.	DataReader	N/A	Yes
	autopurge_ nowriter_samples _delay	Indicates the duration the *DataReader* must retain information regarding instances that have the *instance_state* NOT_ALIVE_NO_WRITERS. By default, infinite.			
	autopurge_ disposed_ samples_delay	Indicates the duration the *DataReader* must retain information regarding instances that have the *instance_state* NOT_ALIVE_DISPOSED. By default, infinite.			

1. Subject to timeouts that indicate loss of communication with a particular subscriber.

2. In practice this will be limited by the settings of the RESOURCE_LIMITS QoS.

3. Actually, the limit will then be set by the max_samples

4. Actually, the limit will then be set by the max_samples

2.1.3.1 USER_DATA

The purpose of this QoS is to allow the application to attach additional information to the created ***Entity*** objects such that when a remote application discovers their existence it can access that information and use it for its own purposes. One possible use of this QoS is to attach security credentials or some other information that can be used by the remote application to authenticate the source. In combination with operations such as ***ignore_participant***, ***ignore_publication***, ***ignore_subscription***, and ***ignore_topic*** these QoS can assist an application to define and enforce its own security policies. The use of this QoS is not limited to security, rather it offers a simple, yet flexible extensibility mechanism.

2.1.3.2 TOPIC_DATA

The purpose of this QoS is to allow the application to attach additional information to the created ***Topic*** such that when a remote application discovers their existence it can examine the information and use it in an application-defined way. In combination with the listeners on the ***DataReader*** and ***DataWriter*** as well as by means of operations such as ***ignore_topic,*** these QoS can assist an application to extend the provided QoS.

2.1.3.3 GROUP_DATA

The purpose of this QoS is to allow the application to attach additional information to the created ***Publisher*** or ***Subscriber***. The value of the GROUP_DATA is available to the application on the ***DataReader*** and ***DataWriter*** entities and is propagated by means of the built-in topics.

This QoS can be used by an application combination with the ***DataReaderListener*** and ***DataWriterListener*** to implement matching policies similar to those of the PARTITION QoS except the decision can be made based on an application-defined policy.

2.1.3.4 DURABILITY

The decoupling between ***DataReader*** and ***DataWriter*** offered by the Publish/Subscribe paradigm allows an application to write data even if there are no current readers on the network. Moreover, a ***DataReader*** that joins the network after some data has been written could potentially be interested in accessing the most current values of the data as well as potentially some history. This QoS policy controls whether the Service will actually make data available to late-joining readers. Note that although related, this does not strictly control what data the Service will maintain internally. That is, the Service may choose to maintain some data for its own purposes (e.g., flow control) and yet not make it available to late-joining readers if the DURABILITY QoS policy is set to VOLATILE.

The value offered is considered compatible with the value requested if and only if the inequality “offered ***kind*** >= requested ***kind***” evaluates to ‘TRUE.’ For the purposes of this inequality, the values of DURABILITY ***kind*** are considered ordered such that VOLATILE < TRANSIENT_LOCAL < TRANSIENT < PERSISTENT.

For the purpose of implementing the DURABILITY QoS kind TRANSIENT or PERSISTENT, the service behaves “as if” for each ***Topic*** that has TRANSIENT or PERSISTENT DURABILITY kind there was a corresponding "built-in” ***DataReader*** and ***DataWriter*** configured to have the same DURABILITY kind. In other words, it is “as if” somewhere in the system (possibly on a remote node) there was a “built-in durability ***DataReader***” that subscribed to that ***Topic*** and a “built-in durability ***DataWriter***” that published that ***Topic*** as needed for the new subscribers that join the system.

For each ***Topic***, the built-in fictitious “persistence service” ***DataReader*** and ***DataWriter*** has its QoS configured from the ***Topic*** QoS of the corresponding ***Topic***. In other words, it is “as-if” the service first did ***find_topic*** to access the ***Topic***, and then used the QoS from the ***Topic*** to configure the fictitious built-in entities.

A consequence of this model is that the transient or persistence serviced can be configured by means of setting the proper QoS on the ***Topic***.

For a given ***Topic***, the usual request/offered semantics apply to the matching between any ***DataWriter*** in the system that writes the ***Topic*** and the built-in transient/persistent ***DataReader*** for that ***Topic***; similarly for the built-in transient/persistent ***DataWriter*** for a ***Topic*** and any ***DataReader*** for the ***Topic***. As a consequence, a ***DataWriter*** that has an incompatible QoS with respect to what the ***Topic*** specified will not send its data to the transient/persistent service, and a ***DataReader*** that has an incompatible QoS with respect to the specified in the ***Topic*** will not get data from it.

Incompatibilities between local ***DataReader***/***DataWriter*** entities and the corresponding fictitious "built-in transient/persistent entities" cause the REQUESTED_INCOMPATIBLE_QOS/OFFERED_INCOMPATIBLE_QOS status to change and the corresponding ***Listener*** invocations and/or signaling of ***Condition*** and ***WaitSet*** objects as they would with non-fictitious entities.

The setting of the *service_cleanup_delay* controls when the TRANSIENT or PERSISTENT service is able to remove all information regarding a data-instances. Information on a data-instances is maintained until the following conditions are met:

1. the instance has been explicitly disposed (*instance_state* = NOT_ALIVE_DISPOSED),
2. and while in the NOT_ALIVE_DISPOSED state the system detects that there are no more "live" ***DataWriter*** entities writing the instance, that is, all existing writers either unregister the instance (call unregister) or lose their liveliness,
3. and a time interval longer that *service_cleanup_delay* has elapsed since the moment the service detected that the previous two conditions were met.

The utility of the *service_cleanup_delay* is apparent in the situation where an application disposes an instance and it crashes before it has a chance to complete additional tasks related to the disposition. Upon re-start the application may ask for initial data to regain its state and the delay introduced by the *service_cleanup_delay* will allow the restarted application to receive the information on the disposed instance and complete the interrupted tasks.

2.1.3.5 DURABILITY_SERVICE

This policy is used to configure the HISTORY QoS and the RESOURCE_LIMITS QoS used by the fictitious ***DataReader*** and ***DataWriter*** used by the "persistence service." The "persistence service" is the one responsible for implementing the DURABILITY kinds TRANSIENT and PERSISTENCE. See Section 2.1.3.4, "DURABILITY," on page 2-114.

2.1.3.6 PRESENTATION

This QoS policy controls the extent to which changes to data-instances can be made dependent on each other and also the kind of dependencies that can be propagated and maintained by the Service.

The setting of ***coherent_access*** controls whether the Service will preserve the groupings of changes made by the publishing application by means of the operations ***begin_coherent_change*** and ***end_coherent_change***.

The setting of ***ordered_access*** controls whether the Service will preserve the order of changes.

The granularity is controlled by the setting of the ***access_scope***.

If ***coherent_access*** is set, then the ***access_scope*** controls the maximum extent of coherent changes. The behavior is as follows:

- If ***access_scope*** is set to INSTANCE, the use of ***begin_coherent_change*** and ***end_coherent_change*** has no effect on how the subscriber can access the data because with the scope limited to each instance, changes to separate instances are considered independent and thus cannot be grouped by a coherent change.
- If ***access_scope*** is set to TOPIC, then coherent changes (indicated by their enclosure within calls to ***begin_coherent_change*** and ***end_coherent_change***) will be made available as such to each remote ***DataReader*** independently. That is, changes made to instances within each individual ***DataWriter*** will be available as coherent with respect to other changes to instances in that same ***DataWriter***, but will not be grouped with changes made to instances belonging to a different ***DataWriter***.
- If ***access_scope*** is set to GROUP, then coherent changes made to instances through a ***DataWriter*** attached to a common ***Publisher*** are made available as a unit to remote subscribers.

If ***ordered_access*** is set, then the ***access_scope*** controls the maximum extent for which order will be preserved by the Service.

- If ***access_scope*** is set to INSTANCE (the lowest level), then changes to each instance are considered unordered relative to changes to any other instance. That means that changes (creations, deletions, modifications) made to two instances are not necessarily seen in the order they occur. This is the case even if it is the same application thread making the changes using the same ***DataWriter***.
- If ***access_scope*** is set to TOPIC, changes (creations, deletions, modifications) made by a single ***DataWriter*** are made available to subscribers in the same order they occur. Changes made to instances through different ***DataWriter*** entities are not necessarily seen in the order they occur. This is the case, even if the changes are made by a single application thread using ***DataWriter*** objects attached to the same ***Publisher***.
- Finally, if ***access_scope*** is set to GROUP, changes made to instances via ***DataWriter*** entities attached to the same ***Publisher*** object are made available to subscribers on the same order they occur.

Note that this QoS policy controls the scope at which related changes are made available to the subscriber. This means the subscriber can access the changes in a coherent manner and in the proper order; however, it does not necessarily imply that the *Subscriber* will

indeed access the changes in the correct order. For that to occur, the application at the subscriber end must use the proper logic in reading the *DataReader* objects, as described in "Access to the data."

The value offered is considered compatible with the value requested if and only if the following conditions are met:

1. The inequality "offered ***access_scope*** >= requested ***access_scope***" evaluates to 'TRUE.' For the purposes of this inequality, the values of PRESENTATION ***access_scope*** are considered ordered such that INSTANCE < TOPIC < GROUP.
2. Requested ***coherent_access*** is FALSE, or else both offered and requested ***coherent_access*** are TRUE.
3. Requested ***ordered_access*** is FALSE, or else both offered and requested ***ordered_access*** are TRUE.

2.1.3.7 DEADLINE

This policy is useful for cases where a ***Topic*** is expected to have each instance updated periodically. On the publishing side this setting establishes a contract that the application must meet. On the subscribing side the setting establishes a minimum requirement for the remote publishers that are expected to supply the data values.

When the Service 'matches' a ***DataWriter*** and a ***DataReader*** it checks whether the settings are compatible (i.e., offered deadline ***period***<= requested deadline ***period***) if they are not, the two entities are informed (via the listener or condition mechanism) of the incompatibility of the QoS settings and communication will not occur.

Assuming that the reader and writer ends have compatible settings, the fulfilment of this contract is monitored by the Service and the application is informed of any violations by means of the proper listener or condition.

The value offered is considered compatible with the value requested if and only if the inequality "offered deadline ***period*** <= requested deadline ***period***" evaluates to 'TRUE.'

The setting of the DEADLINE policy must be set consistently with that of the TIME_BASED_FILTER. For these two policies to be consistent the settings must be such that "deadline ***period***>= ***minimum_separation***."

2.1.3.8 LATENCY_BUDGET

This policy provides a means for the application to indicate to the middleware the "urgency" of the data-communication. By having a non-zero ***duration*** the Service can optimize its internal operation.

This policy is considered a hint. There is no specified mechanism as to how the service should take advantage of this hint.

The value offered is considered compatible with the value requested if and only if the inequality "offered ***duration*** <= requested ***duration***" evaluates to 'TRUE.'

2.1.3.9 OWNERSHIP

This policy controls whether the Service allows multiple ***DataWriter*** objects to update the same instance (identified by ***Topic*** + ***key***) of a data-object.

There are two kinds of OWNERSHIP selected by the setting of the ***kind***: SHARED and EXCLUSIVE.

2.1.3.9.1 SHARED kind

This setting indicates that the Service does not enforce unique ownership for each instance. In this case, multiple writers can update the same data-object instance. The subscriber to the ***Topic*** will be able to access modifications from all ***DataWriter*** objects, subject to the settings of other QoS that may filter particular samples (e.g., the TIME_BASED_FILTER or HISTORY QoS policy). In any case there is no "filtering" of modifications made based on the identity of the ***DataWriter*** that causes the modification.

2.1.3.9.2 EXCLUSIVE kind

This setting indicates that each instance of a data-object can only be modified by one ***DataWriter***. In other words, at any point in time a single ***DataWriter*** "owns" each instance and is the only one whose modifications will be visible to the ***DataReader*** objects. The owner is determined by selecting the ***DataWriter*** with the highest value of the ***strength***[26] that is both "alive" as defined by the LIVELINESS QoS policy and has not violated its DEADLINE contract with regards to the data-instance. Ownership can therefore change as a result of (a) a ***DataWriter*** in the system with a higher value of the ***strength*** that modifies the instance, (b) a change in the ***strength*** value of the ***DataWriter*** that owns the instance, (c) a change in the liveliness of the ***DataWriter*** that owns the instance, and (d) a deadline with regards to the instance that is missed by the DataWriter that owns the instance.

The behavior of the system is as if the determination was made independently by each ***DataReader***. Each ***DataReader*** may detect the change of ownership at a different time. It is not a requirement that at a particular point in time all the ***DataReader*** objects for that ***Topic*** have a consistent picture of who owns each instance.

It is also not a requirement that the ***DataWriter*** objects are aware of whether they own a particular instance. There is no error or notification given to a ***DataWriter*** that modifies an instance it does not currently own.

The requirements are chosen to (a) preserve the decoupling of publishers and subscriber, and (b) allow the policy to be implemented efficiently.

It is possible that multiple ***DataWriter*** objects with the same strength modify the same instance. If this occurs the Service will pick one of the ***DataWriter*** objects as the "owner." It is not specified how the owner is selected. However, it is required that the policy used to select the owner is such that all ***DataReader*** objects will make the same choice of the particular ***DataWriter*** that is the owner. It is also required that the owner

26.The "strength" of a DataWriter is the value of its OWNERSHIP_STRENGTH QoS.

remains the same until there is a change in ***strength***, ***liveliness***, the owner misses a ***deadline*** on the instance, a new ***DataWriter*** with higher ***strength*** modifies the instance, or another ***DataWriter*** with the same strength that is deemed by the Service to be the new owner modifies the instance.

Exclusive ownership is on an instance-by-instance basis. That is, a subscriber can receive values written by a lower strength ***DataWriter*** as long as they affect instances whose values have not been set by the higher-strength ***DataWriter***.

The value of the OWNERSHIP ***kind*** offered must exactly match the one requested or else they are considered incompatible.

2.1.3.10 OWNERSHIP_STRENGTH

This QoS policy should be used in combination with the OWNERSHIP policy. It only applies to the situation case where OWNERSHIP *kind* is set to EXCLUSIVE.

The value of the OWNERSHIP_STRENGTH is used to determine the ownership of a data-instance (identified by the key). The arbitration is performed by the ***DataReader***. The rules used to perform the arbitration are described in Section 2.1.3.9.2, “EXCLUSIVE kind,” on page 2-118.

2.1.3.11 LIVELINESS

This policy controls the mechanism and parameters used by the Service to ensure that particular entities on the network are still “alive.” The liveliness can also affect the ownership of a particular instance, as determined by the OWNERSHIP QoS policy.

This policy has several settings to support both data-objects that are updated periodically as well as those that are changed sporadically. It also allows customizing for different application requirements in terms of the kinds of failures that will be detected by the liveliness mechanism.

The AUTOMATIC liveliness setting is most appropriate for applications that only need to detect failures at the process-level[27], but not application-logic failures within a process. The Service takes responsibility for renewing the leases at the required rates and thus, as long as the local process where a ***DomainParticipant*** is running and the link connecting it to remote participants remains connected, the entities within the ***DomainParticipant*** will be considered alive. This requires the lowest overhead.

The MANUAL settings (MANUAL_BY_PARTICIPANT, MANUAL_BY_TOPIC), require the application on the publishing side to periodically assert the liveliness before the lease expires to indicate the corresponding ***Entity*** is still alive. The action can be explicit by calling the ***assert_liveliness*** operations, or implicit by writing some data.

27. Process here is used to mean an operating system-process as in an address space providing the context where a number of threads execute.

The two possible manual settings control the granularity at which the application must assert liveliness.

- The setting MANUAL_BY_PARTICIPANT requires only that one ***Entity*** within the publisher is asserted to be alive to deduce all other ***Entity*** objects within the same ***DomainParticipant*** are also alive.
- The setting MANUAL_BY_TOPIC requires that at least one instance within the ***DataWriter*** is asserted.

The value offered is considered compatible with the value requested if and only if the following conditions are met:

1. the inequality "offered ***kind*** >= requested ***kind***" evaluates to 'TRUE'. For the purposes of this inequality, the values of LIVELINESS ***kind*** are considered ordered such that:
   AUTOMATIC < MANUAL_BY_PARTICIPANT < MANUAL_BY_TOPIC.
2. the inequality "offered ***lease_duration*** <= requested ***lease_duration***" evaluates to TRUE.

Changes in LIVELINESS must be detected by the Service with a time-granularity greater or equal to the ***lease_duration***. This ensures that the value of the ***LivelinessChangedStatus*** is updated at least once during each ***lease_duration*** and the related Listeners and ***WaitSets*** are notified within a ***lease_duration*** from the time the LIVELINESS changed.

2.1.3.12 TIME_BASED_FILTER

This policy allows a ***DataReader*** to indicate that it does not necessarily want to see all values of each instance published under the ***Topic***. Rather, it wants to see at most one change every ***minimum_separation*** period.

The TIME_BASED_FILTER applies to each instance separately, that is, the constraint is that the ***DataReader*** does not want to see more than one sample of each instance per ***minumum_separation*** period.

This setting allows a ***DataReader*** to further decouple itself from the ***DataWriter*** objects. It can be used to protect applications that are running on a heterogeneous network where some nodes are capable of generating data much faster than others can consume it. It also accommodates the fact that for fast-changing data different subscribers may have different requirements as to how frequently they need to be notified of the most current values.

The setting of a TIME_BASED_FILTER, that is, the selection of a ***minimum_separation*** with a value greater than zero is compatible with all settings of the HISTORY and RELIABILITY QoS. The TIME_BASED_FILTER specifies the samples that are of interest to the ***DataReader***. The HISTORY and RELIABILITY QoS affect the behavior of the middleware with respect to the samples that have been determined to be of interest to the ***DataReader***, that is, they apply after the TIME_BASED_FILTER has been applied.

In the case where the reliability QoS kind is RELIABLE then in steady-state, defined as the situation where the ***DataWriter*** does not write new samples for a period “long” compared to the ***minimum_separation,*** the system should guarantee delivery the last sample to the ***DataReader***.

The setting of the TIME_BASED_FILTER ***minimum_separation*** must be consistent with the DEADLINE ***period***. For these two QoS policies to be consistent they must verify that "***period*** >= ***minimum_separation***." An attempt to set these policies in an inconsistent manner when an entity is created of via a ***set_qos*** operation will cause the operation to fail.

2.1.3.13 PARTITION

This policy allows the introduction of a logical partition concept inside the ‘physical’ partition induced by a domain.

For a ***DataReader*** to see the changes made to an instance by a ***DataWriter***, not only the ***Topic*** must match, but also they must share a common partition. Each string in the list that defines this QoS policy defines a partition name. A partition name may contain wildcards. Sharing a common partition means that one of the partition names matches.

Failure to match partitions is not considered an “incompatible” QoS and does not trigger any listeners nor conditions.

This policy is changeable. A change of this policy can potentially modify the “match” of existing DataReader and DataWriter entities. It may establish new “matchs” that did not exist before, or break existing matchs.

PARTITION names can be regular expressions and include wildcards as defined by the POSIX fnmatch API (1003.2-1992 section B.6). Either ***Publisher*** or ***Subscriber*** may include regular expressions in partition names, but no two names that both contain wildcards will ever be considered to match. This means that although regular expressions may be used both at publisher as well as subscriber side, the service will not try to match two regular expressions (between publishers and subscribers).

Partitions are different from creating ***Entity*** objects in different domains in several ways. First, entities belonging to different domains are completely isolated from each other; there is no traffic, meta-traffic or any other way for an application or the Service itself to see entities in a domain it does not belong to. Second, an ***Entity*** can only belong to one domain whereas an ***Entity*** can be in multiple partitions. Finally, as far as the DDS Service is concerned, each unique data instance is identified by the tuple (***domainId***, ***Topic***, ***key***). Therefore two ***Entity*** objects in different domains cannot refer to the same data instance. On the other hand, the same data-instance can be made available (published) or requested (subscribed) on one or more partitions.

2.1.3.14 RELIABILITY

This policy indicates the level of reliability requested by a ***DataReader*** or offered by a ***DataWriter***. These levels are ordered, BEST_EFFORT being lower than RELIABLE. A ***DataWriter*** offering a level is implicitly offering all levels below.

The setting of this policy has a dependency on the setting of the RESOURCE_LIMITS policy. In case the RELIABILITY ***kind*** is set to RELIABLE the ***write*** operation on the ***DataWriter*** may block if the modification would cause data to be lost or else cause one of the limits in specified in the RESOURCE_LIMITS to be exceeded. Under these circumstances, the RELIABILITY ***max_blocking_time*** configures the maximum duration the ***write*** operation may block.

If the RELIABILITY ***kind*** is set to RELIABLE, data-samples originating from a single ***DataWriter*** cannot be made available to the ***DataReader*** if there are previous data-samples that have not been received yet due to a communication error. In other words, the service will repair the error and re-transmit data-samples as needed in order to re-construct a correct snapshot of the ***DataWriter*** history before it is accessible by the ***DataReader***.

If the RELIABILITY ***kind*** is set to BEST_EFFORT, the service will not re-transmit missing data-samples. However for data-samples originating from any one ***DataWriter*** the service will ensure they are stored in the ***DataReader*** history in the same order they originated in the ***DataWriter***. In other words, the ***DataReader*** may miss some data-samples but it will never see the value of a data-object change from a newer value to an order value.

The value offered is considered compatible with the value requested if and only if the inequality “offered ***kind*** >= requested ***kind***” evaluates to ‘TRUE.’ For the purposes of this inequality, the values of RELIABILITY kind are considered ordered such that BEST_EFFORT < RELIABLE.

2.1.3.15 TRANSPORT_PRIORITY

The purpose of this QoS is to allow the application to take advantage of transports capable of sending messages with different priorities.

This policy is considered a hint. The policy depends on the ability of the underlying transports to set a priority on the messages they send. Any value within the range of a 32-bit signed integer may be chosen; higher values indicate higher priority. However, any further interpretation of this policy is specific to a particular transport and a particular implementation of the Service. For example, a particular transport is permitted to treat a range of priority values as equivalent to one another. It is expected that during transport configuration the application would provide a mapping between the values of the TRANSPORT_PRIORITY set on ***DataWriter*** and the values meaningful to each transport. This mapping would then be used by the infrastructure when propagating the data written by the ***DataWriter***.

2.1.3.16 LIFESPAN

The purpose of this QoS is to avoid delivering “stale” data to the application.

Each data sample written by the ***DataWriter*** has an associated ‘expiration time’ beyond which the data should not be delivered to any application. Once the sample expires, the data will be removed from the ***DataReader*** caches as well as from the transient and persistent information caches.

The ‘expiration time’ of each sample is computed by adding the duration specified by the LIFESPAN QoS to the *source timestamp*. As described in Section 2.1.2.4.2.11, ”write” and Section 2.1.2.4.2.12, ”write_w_timestamp” the *source timestamp* is either automatically computed by the Service each time the ***DataWriter write*** operation is called, or else supplied by the application by means of the ***write_w_timestamp*** operation.

This QoS relies on the sender and receiving applications having their clocks sufficiently synchronized. If this is not the case and the Service can detect it, the ***DataReader*** is allowed to use the reception timestamp instead of the source timestamp in its computation of the ‘expiration time.’

2.1.3.17 DESTINATION_ORDER

This policy controls how each subscriber resolves the final value of a data instance that is written by multiple ***DataWriter*** objects (which may be associated with different ***Publisher*** objects) running on different nodes.

The setting BY_RECEPTION_TIMESTAMP indicates that, assuming the OWNERSHIP policy allows it, the latest received value for the instance should be the one whose value is kept. This is the default value.

The setting BY_SOURCE_TIMESTAMP indicates that, assuming the OWNERSHIP policy allows it, a timestamp placed at the source should be used. This is the only setting that, in the case of concurrent same-strength *DataWriter* objects updating the same instance, ensures all subscribers will end up with the same final value for the instance. The mechanism to set the source timestamp is middleware dependent.

The value offered is considered compatible with the value requested if and only if the inequality “offered ***kind*** >= requested ***kind***” evaluates to ‘TRUE.’ For the purposes of this inequality, the values of DESTINATION_ORDER ***kind*** are considered ordered such that BY_RECEPTION_TIMESTAMP < BY_SOURCE_TIMESTAMP.

2.1.3.18 HISTORY

1. This policy controls the behavior of the Service when the value of an instance changes before it is finally communicated to some of its existing ***DataReader*** entities.

2. If the ***kind*** is set to KEEP_LAST, then the Service will only attempt to keep the latest values of the instance and discard the older ones. In this case, the value of ***depth*** regulates the maximum number of values (up to and including the most current one) the Service will maintain and deliver. The default (and most common setting) for ***depth*** is one, indicating that only the most recent value should be delivered.

3. If the ***kind*** is set to KEEP_ALL, then the Service will attempt to maintain and deliver all the values of the instance to existing subscribers. The resources that the Service can use to keep this history are limited by the settings of the RESOURCE_LIMITS QoS. If the limit is reached, then the behavior of the Service will depend on the RELIABILITY QoS. If the reliability kind is BEST_EFFORT,

then the old values will be discarded. If reliability is RELIABLE, then the Service will block the ***DataWriter*** until it can deliver the necessary old values to all subscribers.

The setting of HISTORY ***depth*** must be consistent with the RESOURCE_LIMITS ***max_samples_per_instance***. For these two QoS to be consistent, they must verify that ***depth*** <= ***max_samples_per_instance***.

2.1.3.19 RESOURCE_LIMITS

This policy controls the resources that the Service can use in order to meet the requirements imposed by the application and other QoS settings.

If the ***DataWriter*** objects are communicating samples faster than they are ultimately taken by the ***DataReader*** objects, the middleware will eventually hit against some of the QoS-imposed resource limits. Note that this may occur when just a single ***DataReader*** cannot keep up with its corresponding ***DataWriter***. The behavior in this case depends on the setting for the RELIABILITY QoS. If reliability is BEST_EFFORT then the Service is allowed to drop samples. If the reliability is RELIABLE, the Service will block the ***DataWriter*** or discard the sample at the ***DataReader*** [28]in order not to lose existing samples.

The constant LENGTH_UNLIMITED may be used to indicate the absence of a particular limit. For example setting ***max_samples_per_instance*** to LENGH_UNLIMITED will cause the middleware to not enforce this particular limit.

The setting of RESOURCE_LIMITS ***max_samples*** must be consistent with the ***max_samples_per_instance***. For these two values to be consistent they must verify that "***max_samples*** >= ***max_samples_per_instance***."

The setting of RESOURCE_LIMITS ***max_samples_per_instance*** must be consistent with the HISTORY ***depth***. For these two QoS to be consistent, they must verify that "***depth*** <= ***max_samples_per_instance***."

An attempt to set this policy to inconsistent values when an entity is created of via a ***set_qos*** operation will cause the operation to fail.

2.1.3.20 ENTITY_FACTORY

This policy controls the behavior of the ***Entity*** as a factory for other entities.

This policy concerns only ***DomainParticipant*** (as factory for ***Publisher***, ***Subscriber***, and ***Topic***), ***Publisher*** (as factory for ***DataWriter***), and ***Subscriber*** (as factory for ***DataReader***).

This policy is mutable. A change in the policy affects only the entities created after the change; not the previously created entities.

28.So that the sample can be re-sent at a later time.

The setting of ***autoenable_created_entities*** to TRUE indicates that the factory ***create_<entity>*** operation will automatically invoke the ***enable*** operation each time a new ***Entity*** is created. Therefore, the ***Entity*** returned by ***create_<entity>*** will already be enabled. A setting of FALSE indicates that the ***Entity*** will not be automatically enabled. The application will need to enable it explicitly by means of the ***enable*** operation (see Section 2.1.2.1.1.7, ”enable”).

The default setting of ***autoenable_created_entities*** = TRUE means that, by default, it is not necessary to explicitly call ***enable*** on newly created entities.

2.1.3.21 WRITER_DATA_LIFECYCLE

This policy controls the behavior of the ***DataWriter*** with regards to the lifecycle of the data-instances it manages, that is, the data-instances that have been either explicitly registered with the ***DataWriter*** using the *register* operations (see Section 2.1.2.4.2.5 and Section 2.1.2.4.2.6) or implicitly by directly writing the data (see Section 2.1.2.4.2.11 and Section 2.1.2.4.2.12).

The ***autodispose_unregistered_instances*** flag controls the behavior when the ***DataWriter*** unregisters an instance by means of the *unregister* operations (see Section 2.1.2.4.2.7, ”unregister_instance” and Section 2.1.2.4.2.8, ”unregister_instance_w_timestamp”):

- The setting ‘***autodispose_unregistered_instances*** = TRUE’ causes the ***DataWriter*** to dispose the instance each time it is unregistered. The behavior is identical to explicitly calling one of the *dispose* operations (Section 2.1.2.4.2.13, ”dispose” and Section 2.1.2.4.2.14, ”dispose_w_timestamp”) on the instance prior to calling the *unregister* operation.
- The setting ‘***autodispose_unregistered_instances*** = FALSE’ will not cause this automatic disposition upon unregistering. The application can still call one of the *dispose* operations prior to unregistering the instance and accomplish the same effect. Refer to Section 2.1.3.23.3, ”Semantic difference between unregister_instance and dispose” for a description of the consequences of disposing and unregistering instances.

Note that the deletion of a ***DataWriter*** automatically unregisters all data-instances it manages (Section 2.1.2.4.1.6, ”delete_datawriter”). Therefore the setting of the ***autodispose_unregistered_instances*** flag will determine whether instances are ultimately disposed when the ***DataWriter*** is deleted either directly by means of the ***Publisher::delete_datawriter*** operation or indirectly as a consequence of calling ***delete_contained_entities*** on the ***Publisher*** or the ***DomainParticipant*** that contains the ***DataWriter***.

2.1.3.22 READER_DATA_LIFECYCLE

This policy controls the behavior of the ***DataReader*** with regards to the lifecycle of the data-instances it manages, that is, the data-instances that have been received and for which the ***DataReader*** maintains some internal resources.

The ***DataReader*** internally maintains the samples that have not been taken by the application, subject to the constraints imposed by other QoS policies such as HISTORY and RESOURCE_LIMITS.

The ***DataReader*** also maintains information regarding the identity, ***view_state*** and ***instance_state*** of data-instances even after all samples have been 'taken.' This is needed to properly compute the states when future samples arrive.

Under normal circumstances the ***DataReader*** can only reclaim all resources for instances for which there are no writers and for which all samples have been 'taken.' The last sample the ***DataReader*** will have taken for that instance will have an ***instance_state*** of either NOT_ALIVE_NO_WRITERS or NOT_ALIVE_DISPOSED depending on whether the last writer that had ownership of the instance disposed it or not. Refer to Figure 2-11 for a statechart describing the transitions possible for the ***instance_state***. In the absence of the READER_DATA_LIFECYCLE QoS this behavior could cause problems if the application "forgets" to 'take' those samples. The 'untaken' samples will prevent the ***DataReader*** from reclaiming the resources and they would remain in the ***DataReader*** indefinitely.

The ***autopurge_nowriter_samples_delay*** defines the maximum duration for which the ***DataReader*** will maintain information regarding an instance once its ***instance_state*** becomes NOT_ALIVE_NO_WRITERS. After this time elapses, the ***DataReader*** will purge all internal information regarding the instance, any untaken samples will also be lost.

The ***autopurge_disposed_samples_delay*** defines the maximum duration for which the ***DataReader*** will maintain samples for an instance once its ***instance_state*** becomes NOT_ALIVE_DISPOSED. After this time elapses, the ***DataReader*** will purge all samples for the instance.

2.1.3.23 Relationship between registration, LIVELINESS, and OWNERSHIP

The need for registering/unregistering instances stems from two use cases:

- Ownership resolution on redundant systems.
- Detection of loss in topological connectivity.

These two use cases also illustrate the semantic differences between the ***unregister_instance*** and ***dispose*** operations on a ***DataWriter***.

2.1.3.23.1 Ownership resolution on redundant systems

It is expected that users may use DDS to set up redundant systems where multiple ***DataWriter*** entities are "capable" of writing the same instance. In this situation the ***DataWriter*** entities are configured such that:

- Either both are writing the instance "constantly,"
- or else they use some mechanism to classify each other as "primary" and "secondary," such that the primary is the only one writing, and the secondary monitors the primary and only writes when it detects that the primary "writer" is no longer writing.

Both cases above use the OWNERSHIP policy kind EXCLUSIVE and arbitrate themselves by means of the OWNERSHIP_STRENGTH. Regardless of the scheme, the desired behavior from the ***DataReader*** point of view is that reader normally receives data from the primary unless the "primary" writer stops writing in which case the reader starts to receive data from the secondary ***DataWriter***.

This approach requires some mechanism to detect that a ***DataWriter*** (the primary) is no longer "writing" the data as it should. There are several reasons why this may be happening and all must be detected but not necessarily distinguished:

1. [crash] - The writing process is no longer running (e.g., the whole application has crashed).
2. [connectivity loss] - Connectivity to the writing application has been lost (e.g., network got disconnected).
3. [application fault] - The application logic that was writing the data is faulty and has stopped calling the "write" operation on the ***DataWriter***.

Arbitrating from a ***DataWriter*** to one of a higher strength is simple and the decision can be taken autonomously by the ***DataReader***. Switching ownership from a higher strength ***DataWriter*** to one of a lower strength ***DataWriter*** requires that the ***DataReader*** can make a determination that the stronger ***DataWriter*** is "no longer writing the instance."

2.1.3.23.1.1 Case where the data is periodically updated

This determination is reasonably simple when the data is being written periodically at some rate. The ***DataWriter*** simply states its offered DEADLINE (maximum interval between updates) and the ***DataReader*** automatically monitors that the ***DataWriter*** indeed updates the instance at least once per deadline ***period***. If the deadline is missed, the ***DataReader*** considers the ***DataWriter*** "not alive" and automatically gives ownership to the next highest-strength ***DataWriter*** that is alive.

2.1.3.23.1.2 Case where data is not periodically updated

The case where the ***DataWriter*** is not writing data periodically is also a very important use-case. Since the instance is not being updated at any fixed period, the "deadline" mechanism cannot be used to determine ownership. The liveliness solves this situation. Ownership is maintained while the ***DataWriter*** is "alive" and for the ***DataWriter*** to be alive it must fulfill its "LIVELINESS" QoS contract. The different means to renew liveliness (automatic, manual) combined by the implied renewal each time data is written handle the three conditions above [crash], [connectivity loss], and [application fault]. Note that to handle [application fault] LIVELINESS must be MANUAL_BY_TOPIC. The ***DataWriter*** can retain ownership by periodically writing data or else calling ***assert_liveliness*** if it has no data to write. Alternatively if only protection against [crash] or [connectivity loss] is desired, it is sufficient that some task on the writer process periodically writes data or calls ***assert_liveliness*** on the ***DomainParticipant***.

However, this scenario requires that the ***DataReader*** knows what instances are being "written" by the ***DataWriter***. That is the only way that the ***DataReader*** deduces the ownership of specific instances from the fact that the ***DataWriter*** is still "alive." Hence the need for the writer to "register" and "unregister" instances. Note that while

"registration" can be done lazily the first time the ***DataWriter*** writes the instance, "unregistration" in general cannot. Similar reasoning will lead to the fact that unregistration will also require a message to be sent to the readers.

2.1.3.23.2 Detection of loss in topological connectivity

There are applications that are designed in such a way that their correct operation requires some minimal topological connectivity, that is, the writer needs to have a minimum number of readers or alternatively the reader must have a minimum number of writers.

A common scenario is that the application does not start doing its logic until it knows that some specific writers have the minimum configured readers (e.g., the alarm monitor is up).

A more common scenario is that the application logic will wait until some writers appear that can provide some needed source of information (e.g, the raw sensor data that must be processed).

Furthermore once the application is running it is a requirement that this minimal connectivity (from the source of the data) is monitored and the application informed if it is ever lost. For the case where data is being written periodically, the DEADLINE QoS and the ***on_deadline_missed*** listener provides the notification. The case where data is not periodically updated requires the use of the LIVELINESS in combination with register/unregister instance to detect whether the "connectivity" has been lost, and the notification is provided by means of the "NO_WRITERS" view state.

In terms of the required mechanisms the scenario is very similar to the case of maintaining ownership. In both cases the reader needs to know whether a writer is still "managing the current value of an instance" even though it is not continually writing it and this knowledge requires the writer to keep its liveliness plus some means to know which instances the writer is currently "managing" (i.e., the registered instances).

2.1.3.23.3 Semantic difference between unregister_instance and dispose

The ***DataWriter*** operation ***dispose*** is semantically different from ***unregister_instance***. The ***dispose*** operation indicates that the data-instance no longer exists (e.g., a track that has disappeared, a simulation entity that has been destroyed, a record entry that has been deleted, etc.) whereas the ***unregister_instance*** operation indicates that the writer is no longer taking responsibility for updating the value of the instance.

Deleting a ***DataWriter*** is equivalent to unregistering all the instances it was writing, but is not the same as "disposing" all the instances.

For a ***Topic*** with EXCLUSIVE OWNERSHIP if the current owner of an instance disposes it, the readers accessing the instance will see the ***instance_state*** as being "DISPOSED" and not see the values being written by the weaker writer (even after the stronger one has disposed the instance). This is because the ***DataWriter*** that owns the instance is saying that the instance no longer exists (e.g., the master of the database is saying that a record has been deleted) and thus the readers should see it as such.

For a ***Topic*** with EXCLUSIVE OWNERSHIP if the current owner of an instance unregisters it, then it will relinquish ownership of the instance and thus the readers may see the value updated by another writer (which will then become the owner). This is because the owner said that it no longer will be providing values for the instance and thus another writer can take ownership and provide those values.

2.1.4 Listeners, Conditions, and Wait-sets

Listeners and conditions (in conjunction with wait-sets) are two alternative mechanisms that allow the application to be made aware of changes in the DCPS communication status.

2.1.4.1 Communication Status

The communication statuses whose changes can be communicated to the application depend on the ***Entity***. The following table shows for each entity the statuses that are relevant.

Entity	Status Name	Meaning
Topic	INCONSISTENT_TOPIC	Another topic exists with the same name but different characteristics.
Subscriber	DATA_ON_READERS	New information is available.

Entity	Status Name	Meaning
DataReader	SAMPLE_REJECTED	A (received) sample has been rejected.
	LIVELINESS_CHANGED	The liveliness of one or more ***DataWriter*** that were writing instances read through the ***DataReader*** has changed. Some ***DataWriter*** have become “active” or “inactive.”
	REQUESTED_DEADLINE_MISSED	The deadline that the ***DataReader*** was expecting through its ***QosPolicy*** DEADLINE was not respected for a specific instance.
	REQUESTED_INCOMPATIBLE_QOS	A ***QosPolicy*** value was incompatible with what is offered.
	DATA_AVAILABLE	New information is available.
	SAMPLE_LOST	A sample has been lost (never received).
	SUBSCRIPTION_MATCHED	The *DataReader* has found a *DataWriter* that matches the *Topic* and has compatible QoS, or has ceased to be matched with a ***DataWriter*** that was previously considered to be matched.
DataWriter	LIVELINESS_LOST	The liveliness that the ***DataWriter*** has committed through its ***QosPolicy*** LIVELINESS was not respected; thus ***DataReader*** entities will consider the ***DataWriter*** as no longer “active.”
	OFFERED_DEADLINE_MISSED	The deadline that the ***DataWriter*** has committed through its ***QosPolicy*** DEADLINE was not respected for a specific instance.
	OFFERED_INCOMPATIBLE_QOS	A ***QosPolicy*** value was incompatible with what was requested.
	PUBLICATION_MATCHED	The *DataWriter* has found *DataReader* that matches the *Topic* and has compatible QoS, or has ceased to be matched with a *DataReader* that was previously considered to be matched.

Those statuses may be classified in:

- Read communication statuses: i.e., those that are related to arrival of data, namely DATA_ON_READERS and DATA_AVAILABLE.
- Plain communication statuses: i.e., all the others.

Read communication statuses are treated slightly differently than the others for they don’t change independently. In other words, at least two changes will appear at the same time (DATA_ON_READERS + DATA_AVAILABLE) and even several of the last kind may be part of the set. This ‘grouping’ has to be communicated to the application. How this is done is discussed in each of the two following sections.

For each plain communication status, there is a corresponding structure to hold the status value. These values contain the information related to the change of status, as well as information related to the statuses themselves (e.g., contains cumulative counts). They are used with the two different mechanisms explained in the following sections.

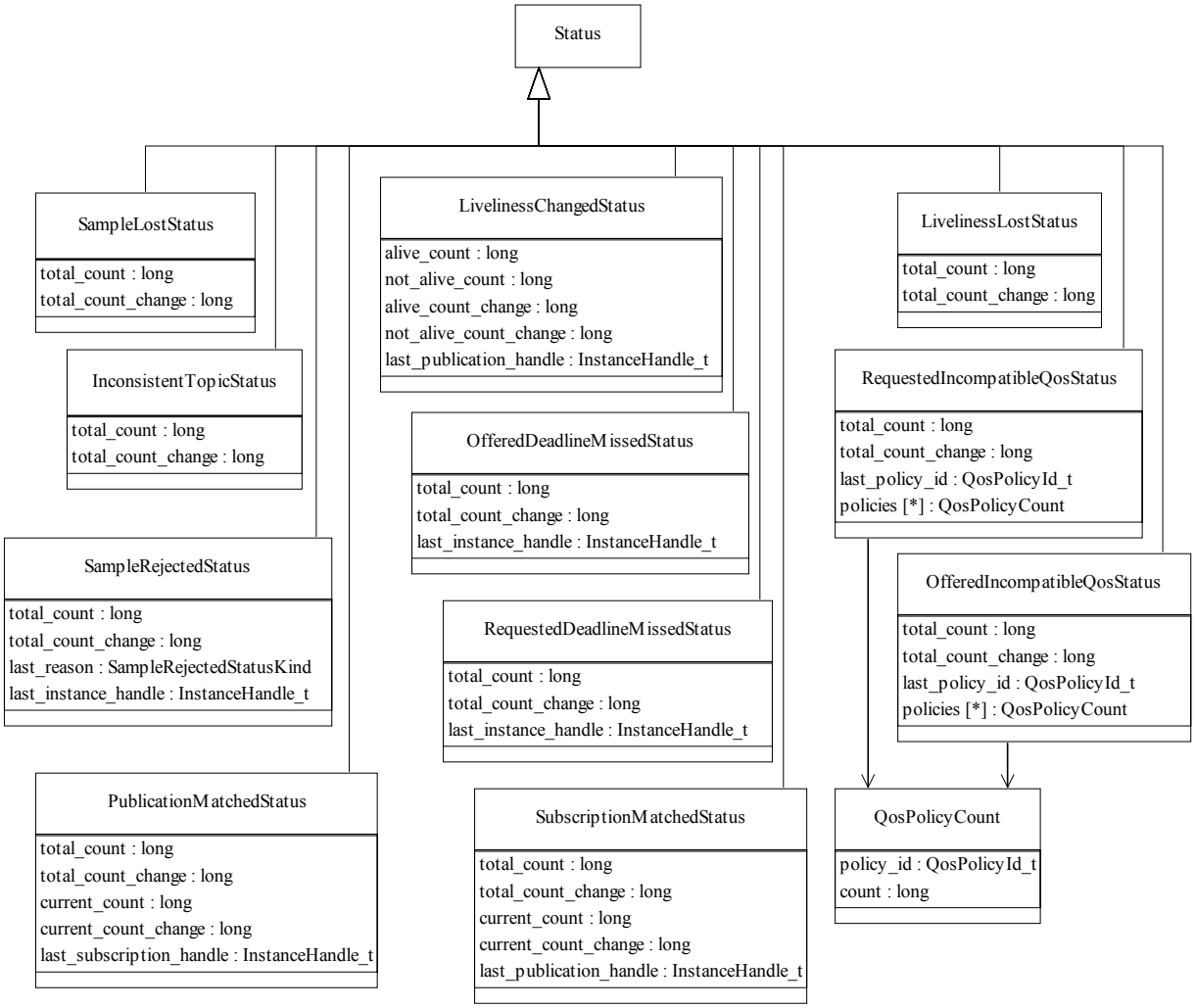

Figure 2-13 Status Values

The interpretation of the attributes for each status value is provided in the following table.

SampleLostStatus	Attribute meaning
total_count	Total cumulative count of all samples lost across of instances of data published under the ***Topic***.
total_count_change	The incremental number of samples lost since the last time the listener was called or the status was read.
SampleRejectedStatus	Attribute meaning
total_count	Total cumulative count of samples rejected by the DataReader.
total_count_change	The incremental number of samples rejected since the last time the listener was called or the status was read.
last_reason	Reason for rejecting the last sample rejected. If no samples have been rejected, the reason is the special value NOT_REJECTED.
last_instance_handle	Handle to the instance being updated by the last sample that was rejected.
InconsistentTopicStatus	Attribute meaning.
total_count	Total cumulative count of the Topics discovered whose name matches the Topic to which this status is attached and whose type is inconsistent with the Topic.
total_count_change	The incremental number of inconsistent topics discovered since the last time the listener was called or the status was read.
LivelinessChangedStatus	Attribute meaning.
alive_count	The total number of currently active DataWriters that write the Topic read by the DataReader. This count increases when a newly matched DataWriter asserts its liveliness for the first time or when a DataWriter previously considered to be not alive reasserts its liveliness. The count decreases when a DataWriter considered alive fails to assert its liveliness and becomes not alive, whether because it was deleted normally or for some other reason.
not_alive_count	The total count of currently DataWriters that write the Topic read by the DataReader that are no longer asserting their liveliness. This count increases when a DataWriter considered alive fails to assert its liveliness and becomes not alive for some reason other than the normal deletion of that DataWriter. It decreases when a previously not alive DataWriter either reasserts its liveliness or is deleted normally.
alive_count_change	The change in the alive_count since the last time the listener was called or the status was read.
not_alive_count_change	The change in the not_alive_count since the last time the listener was called or the status was read.
last_publication_handle	Handle to the last DataWriter whose change in liveliness caused this status to change.
RequestedDeadlineMissedStatus	Attribute meaning.

total_count	Total cumulative number of missed deadlines detected for any instance read by the DataReader. Missed deadlines accumulate; that is, each deadline period the *total_count* will be incremented by one for each instance for which data was not received.
total_count_change	The incremental number of deadlines detected since the last time the listener was called or the status was read.
last_instance_handle	Handle to the last instance in the DataReader for which a deadline was detected.
RequestedIncompatibleQosStatus	Attribute meaning.
total_count	Total cumulative number of times the concerned DataReader discovered a DataWriter for the same Topic with an offered QoS that was incompatible with that requested by the DataReader.
total_count_change	The change in total_count since the last time the listener was called or the status was read.
last_policy_id	The QosPolicyId_t of one of the policies that was found to be incompatible the last time an incompatibility was detected.
policies	A list containing for each policy the total number of times that the concerned DataReader discovered a DataWriter for the same Topic with an offered QoS that is incompatible with that requested by the DataReader.
LivelinessLostStatus	Attribute meaning.
total_count	Total cumulative number of times that a previously-alive DataWriter became not alive due to a failure to actively signal its liveliness within its offered liveliness period. This count does not change when an already not alive DataWriter simply remains not alive for another liveliness period.
total_count_change	The change in total_count since the last time the listener was called or the status was read.
OfferedDeadlineMissedStatus	Attribute meaning.
total_count	Total cumulative number of offered deadline periods elapsed during which a DataWriter failed to provide data. Missed deadlines accumulate; that is, each deadline period the total_count will be incremented by one.
total_count_change	The change in total_count since the last time the listener was called or the status was read.
last_instance_handle	Handle to the last instance in the DataWriter for which an offered deadline was missed.
OfferedIncompatibleQosStatus	Attribute meaning.
total_count	Total cumulative number of times the concerned DataWriter discovered a DataReader for the same Topic with a requested QoS that is incompatible with that offered by the DataWriter.
total_count_change	The change in total_count since the last time the listener was called or the status was read.

last_policy_id	The PolicyId_t of one of the policies that was found to be incompatible the last time an incompatibility was detected.
policies	A list containing for each policy the total number of times that the concerned DataWriter discovered a DataReader for the same Topic with a requested QoS that is incompatible with that offered by the DataWriter.
PublicationMatchedStatus	Attribute meaning.
total_count	Total cumulative count the concerned DataWriter discovered a "match" with a DataReader. That is, it found a DataReader for the same Topic with a requested QoS that is compatible with that offered by the DataWriter.
total_count_change	The change in total_count since the last time the listener was called or the status was read.
last_subscription_handle	Handle to the last DataReader that matched the DataWriter causing the status to change.
current_count	The number of DataReaders currently matched to the concerned DataWriter.
current_count_change	The change in current_count since the last time the listener was called or the status was read.
SubscriptionMatchedStatus	Attribute meaning.
total_count	Total cumulative count the concerned DataReader discovered a "match" with a DataWriter. That is, it found a DataWriter for the same Topic with a requested QoS that is compatible with that offered by the DataReader.
total_count_change	The change in total_count since the last time the listener was called or the status was read.
last_publication_handle	Handle to the last DataWriter that matched the DataReader causing the status to change.
current_count	The number of DataWriters currently matched to the concerned DataReader.
current_count_change	The change in current_count since the last time the listener was called or the status was read.

2.1.4.2 Changes in Status

Associated with each one of an ***Entity***'s communication status is a logical ***StatusChangedFlag***. This flag indicates whether that particular communication status has changed since the last time the status was 'read' by the application. The way the status changes is slightly different for the Plain Communication Status and the Read Communication status.

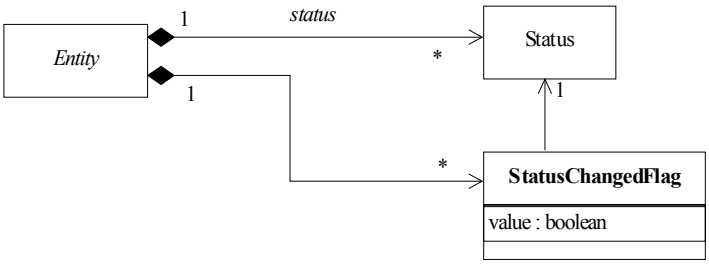

Figure 2-14 StatusChangedFlag indicates if status has changed

Note that Figure 2-14 is only conceptual; it simply represents that the ***Entity*** knows which specific statuses have changed. It does not imply any particular implementation of the ***StatusChangedFlag*** in terms of boolean values.

2.1.4.2.1 Changes in Plain Communication Status

For the plain communication status, the ***StatusChangedFlag*** flag is initially set to FALSE. It becomes TRUE whenever the plain communication status changes and it is reset to FALSE each time the application accesses the plain communication status via the proper ***get_<plain communication status>*** operation on the ***Entity***.

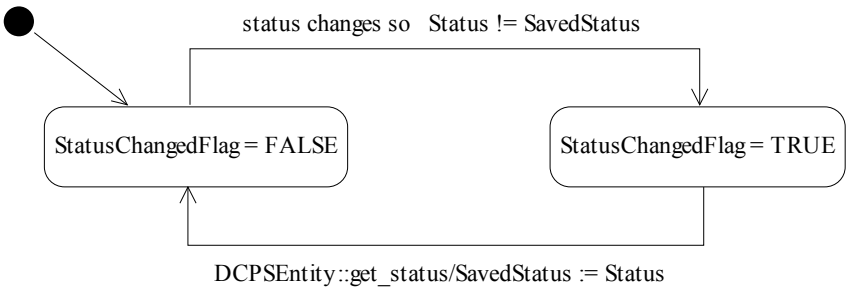

Figure 2-15 Changes in StatusChangedFlag for plain communication status

The communication status is also reset to FALSE whenever the associated listener operation is called as the listener implicitly accesses the status which is passed as a parameter to the operation. The fact that the status is reset prior to calling the listener means that if the application calls the ***get_<plain communication status>*** from inside the listener it will see the status already reset.

An exception to this rule is when the associated listener is the 'nil' listener. As described in Section 2.1.4.3.1, “Listener Access to Plain Communication Status,” on page 2-138 the 'nil' listener is treaded as a NOOP and the act of calling the 'nil' listener does not reset the communication status.

For example, the value of the ***StatusChangedFlag*** associated with the REQUESTED_DEADLINE_MISSED status will become TRUE each time new deadline occurs (which increases the ***total_count*** field within ***RequestedDeadlineMissedStatus***). The value changes to FALSE when the application accesses the status via the corresponding ***get_requested_deadline_missed_status*** method on the proper ***Entity***.

2.1.4.2.2 Changes in Read Communication Statuses

For the read communication status, the ***StatusChangedFlag*** flag is initially set to FALSE.

The ***StatusChangedFlag*** becomes TRUE when either a data-sample arrives or else the ***ViewState***, ***SampleState***, or ***InstanceState*** of any existing sample changes for any reason other than a call to ***DataReader***::**read**, ***DataReader***::**take** or their variants. Specifically any of the following events will cause the ***StatusChangedFlag*** to become TRUE:

- The arrival of new data.
- A change in the ***InstanceState*** of a contained instance. This can be caused by either:
  - The arrival of the notification that an instance has been disposed by:
    - the ***DataWriter*** that owns it if ***OWNERSHIP QoS kind***=EXLUSIVE
    - or by any ***DataWriter*** if ***OWNERSHIP QoS kind***=SHARED.
  - The loss of liveliness of the ***DataWriter*** of an instance for which there is no other ***DataWriter***.
  - The arrival of the notification that an instance has been unregistered by the only ***DataWriter*** that is known to be writing the instance.

Depending on the kind of ***StatusChangedFlag***, the flag transitions to FALSE again as follows***:***

- The DATA_AVAILABLE ***StatusChangedFlag*** becomes FALSE when either the corresponding listener operation (***on_data_available***) is called or the ***read*** or ***take*** operation (or their variants) is called on the associated ***DataReader***.
- The DATA_ON_READERS ***StatusChangedFlag*** becomes FALSE when any of the following events occurs:
  - The corresponding listener operation (***on_data_on_readers***) is called.
  - The ***on_data_available*** listener operation is called on any ***DataReader*** belonging to the ***Subscriber***.
  - The ***read*** or ***take*** operation (or their variants) is called on any ***DataReader*** belonging to the ***Subscriber.***

.

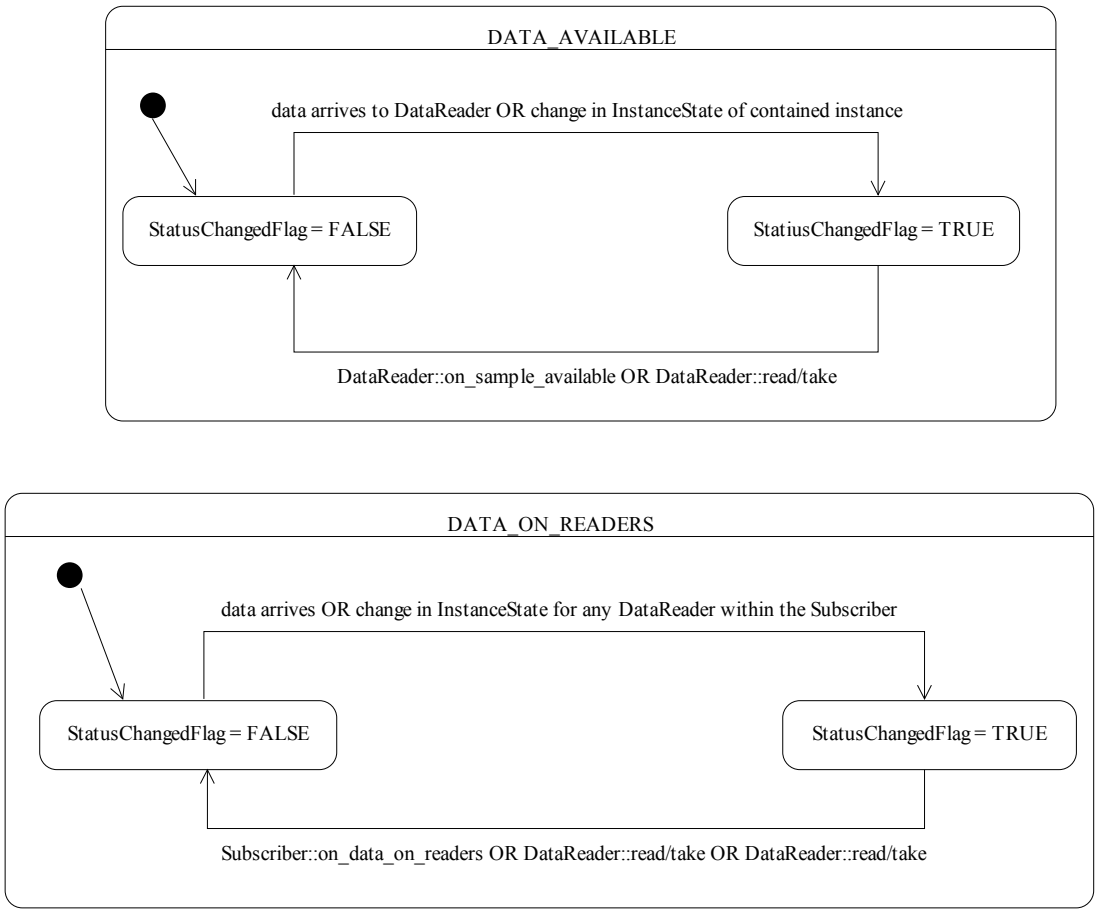

Figure 2-16 Changes in StatusChangedFlag for read communication status

2.1.4.3 Access through Listeners

Listeners provide a mechanism for the middleware to asynchronously alert the application of the occurrence of relevant status changes.

All ***Entity*** support a listener, which type of which is specialized to the specific type of the related ***Entity*** (e.g., ***DataReaderListener*** for the ***DataReader***). Listeners are interfaces that the application must implement. Each dedicated listener presents a list of operations that correspond to the relevant communication status changes (i.e., that the application may react to).

All listeners are listed on Figure 2-17, associated with the DCPS constructs that participate in this mechanism (note that only the related operations are displayed).

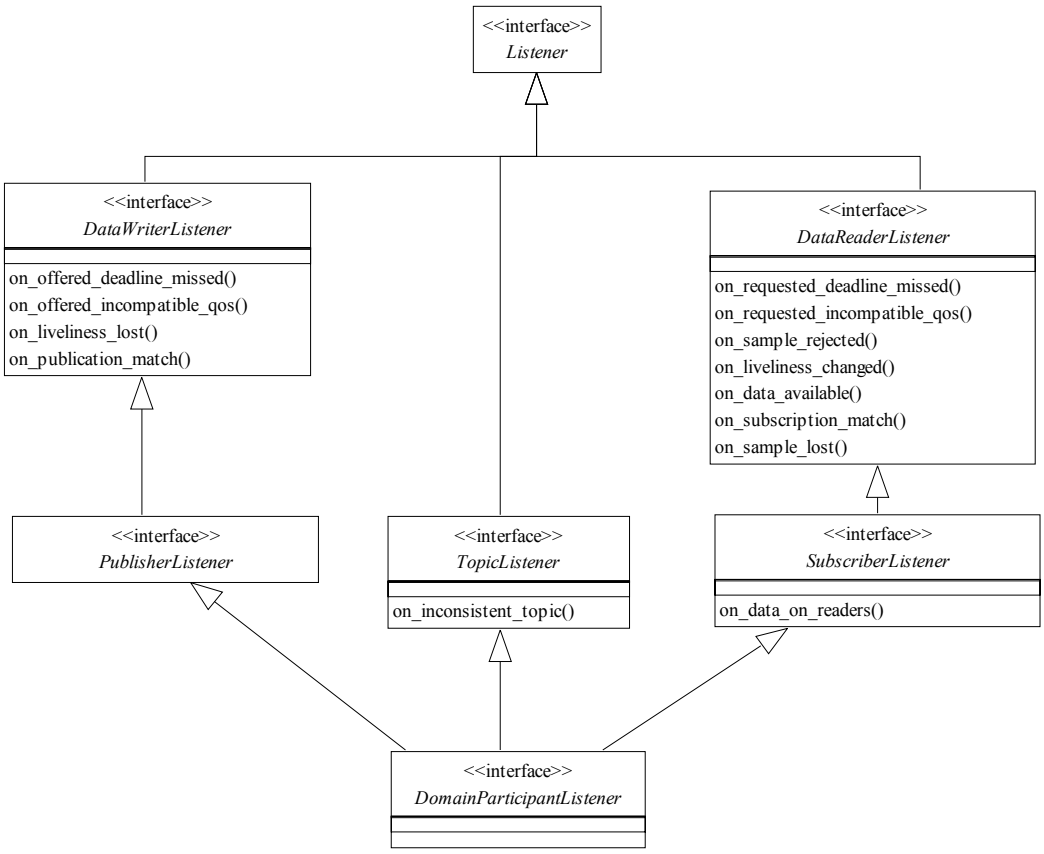

Figure 2-17 Supported DCPS Listeners

Listeners are stateless. It is thus possible to share the same ***DataReaderListener*** instance among all the ***DataReader*** objects (assuming that they will react similarly on similar status changes). Consequently, the provided parameter contains a reference to the actual concerned ***Entity***.

2.1.4.3.1 Listener Access to Plain Communication Status

The general mapping between the plain communication statuses as explained in Section 2.1.4.1, “Communication Status,” on page 2-129 and the listeners’ operations is as follows:

- For each communication status, there is a corresponding operation whose name is ***on_<communication_status>***, which takes a parameter of type ***<communication_status>*** as listed in Section 2.1.4.1, “Communication Status,” on page 2-129.
- ***on_<communication_status>*** is available on the relevant ***Entity*** as well as those that embed it, as expressed in the following figure:

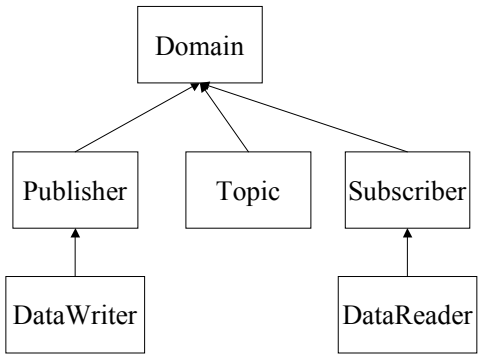

Figure 2-18

- When the application attaches a listener on an entity, it must set a mask that indicates to the middleware which operations are enabled within this listener (see operation ***Entity::set_listener***)
- When a plain communication status changes[29], the middleware triggers the most 'specific' relevant listener operation that is enabled[30]. In case the most specific relevant listener operation corresponds to an application-installed 'nil' listener the operation will be considered handled by a NO-OP operation.

This behavior allows the application to set a default behavior (e.g., in the listener associated with the ***DomainParticipant***) and setting dedicated behaviors only where needed.

2.1.4.3.2 Listener access to Read Communication Status

The two statuses related to data arrival are treated slightly differently. Since they constitute the real purpose of the Data Distribution Service, there is not really a need to provide a default mechanism as for the plain communication statuses, and more importantly, several of them may need to be treated as a whole as explained in Section 2.1.4.1, "Communication Status," on page 2-129.

The rule is as follows. Each time the read communication status changes[31]:

---

29. To be more precise, when the corresponding ***StatusChangedFlag*** described in "Changes in Plain Communication Status" becomes TRUE.

30. For example, in case of ON_OFFERED_DEADLINE_MISSED for a given ***DataWriter***: the ***DataWriter***'s listener operation ***on_offered_deadline_missed***, or by default (i.e., if there was no listener attached to that ***DataWriter***, or if the operation was not enabled), the ***Publisher***'s listener or else (no listener attached to the ***Publisher*** or operation not enabled) the ***DomainParticipant's*** listener.

31. To be more precise, when the corresponding ***StatusChangedFlag*** described in "Changes in Read Communication Statuses" becomes TRUE.

- first, the middleware tries to trigger the ***SubscriberListener*** operation ***on_data_on_readers*** with a parameter of the related ***Subscriber***;
- if this does not succeed (no listener or operation non-enabled), it tries to trigger ***on_data_available*** on all the related ***DataReaderListener*** objects, with as parameter the related ***DataReader***.

The rationale is that either the application is interested in relations among data arrivals and it must use the first option (and then get the corresponding ***DataReader*** objects by calling ***get_datareaders*** on the related ***Subscriber*** and then get the data by calling ***read***/***take*** on the returned ***DataReader*** objects[32]), or it wants to treat each ***DataReader*** fully independently and it may choose the second option (and then get the data by calling ***read***/***take*** on the related ***DataReader***).

Note that if ***on_data_on_readers*** is called, then the middleware will not try to call ***on_data_available,*** however, the application can force a call to the ***DataReader*** objects that have data by means of the ***notify_datareaders*** operation.

There is no implied "event queuing" in the invocation of the listeners in the sense that, if several changes of status of the same kind occur in sequence, it is not necessary that the DCPS implementation delivers one listener callback per "unit" change. For example, it may occur that the DCPS implementation discovers that the liveliness of a ***DataReader*** has changed in that several matching ***DataWriter*** entities have appeared; in that case the DCPS implementation may choose to invoke the ***on_liveliness_changed*** operation on the ***DataReaderListener*** just once, as long as the ***LivelinessChangedStatus*** provided to the listener corresponds to the most current one.

2.1.4.4 Conditions and Wait-sets

As previously mentioned, conditions (in conjunction with wait-sets) provide an alternative mechanism to allow the middleware to communicate communication status changes (including arrival of data) to the application.

Figure 2-19 : Wait-sets and Conditions shows all the DCPS constructs that are involved in that mechanism (note that only the related operations are displayed).

32.As detailed in Section 2.1.2.5, "Subscription Module," on page 2-64.

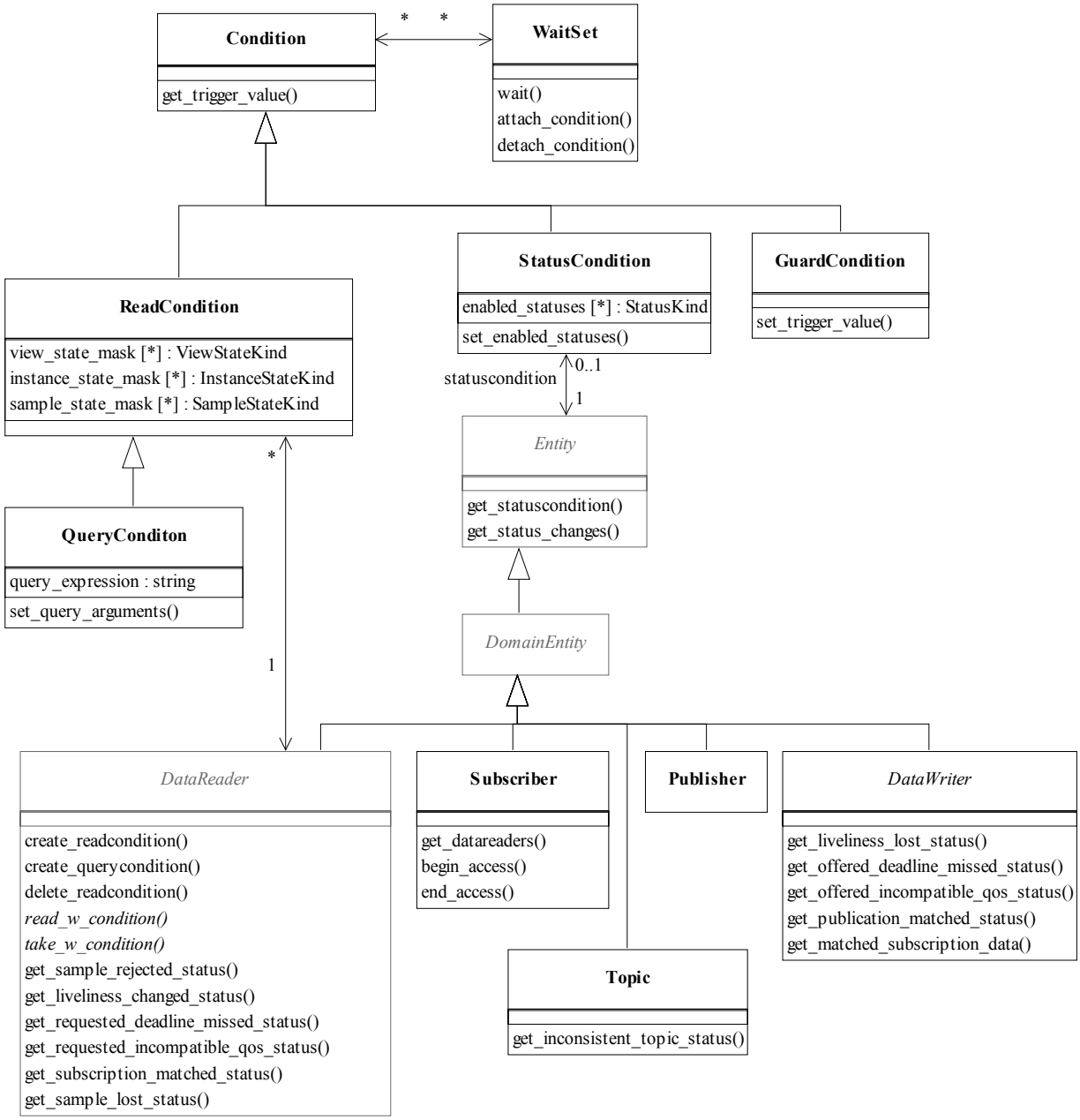

Figure 2-19 Wait-sets and Conditions

This mechanism is wait-based. Its general use pattern is as follows:

- The application indicates which relevant information it wants to get, by creating ***Condition*** objects (***StatusCondition***, ***ReadCondition*** or ***QueryCondition***[33]) and attaching them to a ***WaitSet***.

- It then waits on that ***WaitSet*** until the ***trigger_value*** of one or several ***Condition*** objects become TRUE.
- It then uses the result of the ***wait*** (i.e., the list of ***Condition*** objects with ***trigger_value***==TRUE) to actually get the information by callilng:
  - ***get_status_changes*** and then ***get_<communication_status>*** on the relevant ***Entity***. If the condition is a ***StatusCondition*** and the status changes, refer to plain communication status.
  - ***get_status_changes*** and then ***get_datareaders*** on the relevant ***Subscriber***. If the condition is a ***StatusCondition*** and the status changes, refers to DATA_ON_READERS[34].
  - ***get_status_changes*** and then ***read***/***take*** on the relevant ***DataReader***. If the condition is a ***StatusCondition*** and the status changes, refers to DATA_AVAILABLE.
  - Directly ***read_w_condition***/***take_w_condition*** on the ***DataReader*** with the ***Condition*** as a parameter if it is a ***ReadCondition*** or a ***QueryCondition***.

Usually the first step is done in an initialization phase, while the others are put in the application main loop.

As there is no extra information passed from the middleware to the application when a wait returns (only the list of triggered ***Condition*** objects), ***Condition*** objects are meant to embed all that is needed to react properly when enabled. In particular, ***Entity***-related conditions[35] are related to exactly one ***Entity*** and cannot be shared.

The blocking behavior of the ***WaitSet*** is illustrated in Figure 2-19. The result of a ***wait*** operation depends on the state of the ***WaitSet***, which in turn depends on whether at least one attached ***Condition*** has a ***trigger_value*** of TRUE. If the ***wait*** operation is called on ***WaitSet*** with state BLOCKED, it will block the calling thread. If ***wait*** is called on a ***WaitSet*** with state UNBLOCKED, it will return immediately. In addition, when the ***WaitSet*** transitions from BLOCKED to UNBLOCKED it wakes up any threads that had called ***wait*** on it.

33. See Section 2.1.2.1, “Infrastructure Module,” on page 2-11 for general definition of conditions; see Section 2.1.2.5, “Subscription Module,” on page 2-64 for ***ReadCondition*** and ***QueryCondition***.

34. And then read/take on the returned ***DataReader*** objects.

35. For instance, ***StatusCondition***, ***ReadCondition*** and ***QueryCondition***. See Section 2.1.2.1, “Infrastructure Module,” on page 2-11 on the use of basic Condition.

Similar to the invocation of listeners, there is no implied “event queuing” in the awakening of a WaitSet in the sense that, if several Conditions attached to the WaitSet have their trigger_value transition to TRUE in sequence the DCPS implementation needs to only unblock the WaitSet once.

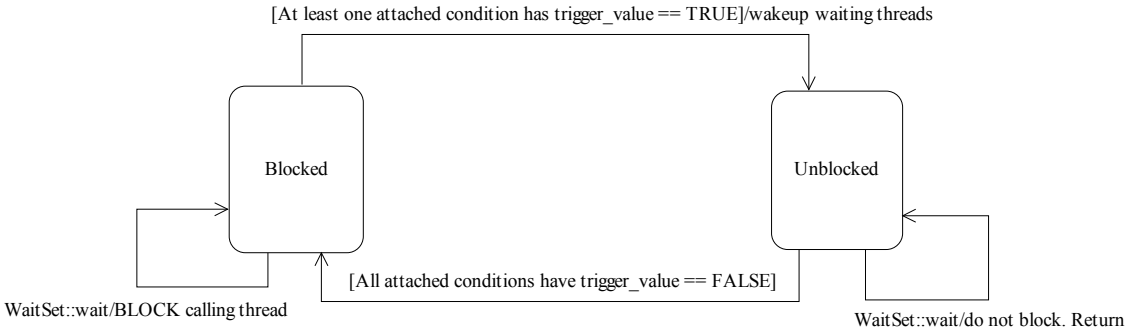

Figure 2-20 Blocking behavior of WaitSet

A key aspect of the ***Condition***/***WaitSet*** mechanism is the setting of the ***trigger_value*** of each ***Condition***.

2.1.4.4.1 Trigger State of the StatusCondition

The ***trigger_value*** of a ***StatusCondition*** is the Boolean OR of the ***ChangedStatusFlag*** of all the communication statuses to which it is sensitive. That is, ***trigger_value***==FALSE only if all the values of the ***ChangedStatusFlags*** are FALSE.

The sensitivity of the ***StatusCondition*** to a particular communication status is controlled by the list of ***enabled_statuses*** set on the condition by means of the ***set_enabled_statuses*** operation.

2.1.4.4.2 Trigger State of the ReadCondition

Similar to the ***StatusCondition***, a ***ReadCondition*** also has a ***trigger_value*** that determines whether the attached ***WaitSet*** is BLOCKED or UNBLOCKED. However, unlike the ***StatusCondition***, the ***trigger_value*** of the ***ReadCondition*** is tied to the presence of at least a sample managed by the Service with ***SampleState, ViewState,*** and ***InstanceState*** matching those of the ***ReadCondition***. Furthermore, for the ***QueryCondition*** to have a ***trigger_value***==TRUE, the data associated with the sample must be such that the ***query_expression*** evaluates to TRUE.

The fact that the trigger_value of a ***ReadCondition*** is dependent on the presence of samples on the associated ***DataReader*** implies that a single take operation can potentially change the ***trigger_value*** of several ***ReadCondition*** or ***QueryCondition*** conditions. For example, if all samples are taken, any ***ReadCondition*** and ***QueryCondition*** conditions associated with the ***DataReader*** that had their ***trigger_value***==TRUE before will see the ***trigger_value*** change to FALSE. Note that this does not guarantee that ***WaitSet*** objects that were separately attached to those conditions will not be woken up. Once we have ***trigger_value***==TRUE on a condition it may wake

up the attached ***WaitSet***, the condition transitioning to ***trigger_value***==FALSE does not necessarily 'unwakeup' the ***WaitSet*** as 'unwakening' may not be possible in general. The consequence is that an application blocked on a ***WaitSet*** may return from the wait with a list of conditions some of which are no longer "active." This is unavoidable if multiple threads are concurrently waiting on separate ***WaitSet*** objects and taking data associated with the same ***DataReader*** entity.

To elaborate further, consider the following example: A ***ReadCondition*** that has a ***sample_state_mask*** = {NOT_READ} will have ***trigger_value*** of TRUE whenever a new sample arrives and will transition to FALSE as soon as all the newly-arrived samples are either read (so their status changes to READ) or taken (so they are no longer managed by the Service). However if the same ***ReadCondition*** had a ***sample_state_mask*** = {READ, NOT_READ}, then the ***trigger_value*** would only become FALSE once all the newly-arrived samples are taken (it is not sufficient to read them as that would only change the ***SampleState*** to READ which overlaps the mask on the ***ReadCondition***.

2.1.4.4.3 Trigger State of the GuardCondition

The ***trigger_value*** of a ***GuardCondition*** is completely controlled by the application via operation ***set_trigger_value***.

2.1.4.5 Combination

Those two mechanisms may be combined in the application (e.g., using wait-sets and conditions to access the data and listeners to be warned asynchronously of erroneous communication statuses).

It is likely that the application will choose one or the other mechanism for each particular communication status (not both). However, if both mechanisms are enabled, then the listener mechanism is used first and then the ***WaitSet*** objects are signalled.

2.1.5 Built-in Topics

As part of its operation, the middleware must discover and possibly keep track of the presence of remote entities such as a new participant in the domain. This information may also be important to the application, which may want to react to this discovery, or else access it on demand.

To make this information accessible to the application, the DCPS specification introduces a set of built-in topics and corresponding ***DataReader*** objects that can then be used by the application. The information is then accessed as if it was normal application data. This approach avoids introducing a new API to access this information and allows the application to become aware of any changes in those values by means of any of the mechanisms presented in Section 2.1.4, "Listeners, Conditions, and Wait-sets," on page 2-129.

The built-in data-readers all belong to a built-in ***Subscriber***. This subscriber can be retrieved by using the method ***get_builtin_subscriber*** provided by the ***DomainParticipant***. The built-in ***DataReader*** objects can be retrieved by using the operation ***lookup_datareader***, with the Subscriber and the topic name as parameters.

The QoS of the built-in ***Subscriber*** and ***DataReader*** objects is given by the following table:

USER_DATA	<unspecified>
TOPIC_DATA	<unspecified>
GROUP_DATA	<unspecified>
DURABILITY	TRANSIENT_LOCAL
DURABILITY_SERVICE	Does not apply as DURABILITY is TRANSIENT_LOCAL
PRESENTATION	access_scope = TOPIC coherent_access = FALSE ordered_access = FALSE
DEADLINE	Period = infinite
LATENCY_BUDGET	duration = <unspecified>
OWNERSHIP	SHARED
LIVELINESS	kind = AUTOMATIC lease_duration = <unspecified>
TIME_BASED_FILTER	minimum_separation = 0
PARTITION	<unspecified>
RELIABILITY	kind = RELIABLE max_blocking_time = 100 milliseconds
DESTINATION_ORDER	BY_RECEPTION_TIMESTAMP
HISTORY	kind = KEEP_LAST depth = 1
RESOURCE_LIMITS	All LENGTH_UNLIMITED.
READER_DATA_LIFECYCLE	autopurge_nowriter_samples_delay = infinite autopurge_disposed_samples_delay = infinite
ENTITY_FACTORY	autoenable_created_entities = TRUE

Built-in entities have default listener settings as well. The built-in ***Subscriber*** and all of its built-in ***Topics*** have nil listeners with all statuses appearing in their listener masks. The built-in ***DataReaders*** have nil listeners with no statuses in their masks.

The information that is accessible about the remote entities by means of the built-in topics includes all the QoS policies that apply to the corresponding remote Entity. This QoS policies appear as normal ‘data’ fields inside the data read by means of the built-in Topic. Additional information is provided to identify the Entity and facilitate the application logic.

A built-in ***DataReader*** obtained from a given ***Participant*** will not provide data pertaining to ***Entities*** created from that same ***Participant*** under the assumption that such entities are already known to the application that created them.

The table below lists the built-in topics, their names, and the additional information--beyond the QoS policies that apply to the remote entity--that appears in the data associated with the built-in topic.

Topic name	Field Name	Type	Meaning
DCPSParticipant (entry created when a ***DomainParticipant*** object is created).	key	BuiltinTopicKey_t	DCPS key to distinguish entries.
	user_data	UserDataQosPolicy	Policy of the corresponding *DomainParticipant.*
DCPSTopic (entry created when a ***Topic*** object is created).	key	BuiltinTopicKey_t	DCPS key to distinguish entries.
	name	string	Name of the *Topic.*
	type_name	string	Name of the type attached to the *Topic.*
	durability	DurabilityQosPolicy	Policy of the corresponding *Topic.*
	durability_service	DurabilityServiceQos Policy	Policy of the corresponding *Topic.*
	deadline	DeadlineQosPolicy	Policy of the corresponding *Topic.*
	latency_budget	LatencyBudgetQos Policy	Policy of the corresponding *Topic.*
	liveliness	LivelinessQosPolicy	Policy of the corresponding *Topic.*
	reliability	ReliabilityQosPolicy	Policy of the corresponding *Topic.*
	transport_priority	TransportPriorityQos Policy	Policy of the corresponding *Topic.*
	lifespan	LifespanQosPolicy	Policy of the corresponding *Topic.*
	destination_order	DestinationOrderQos Policy	Policy of the corresponding *Topic.*
	history	HistoryQosPolicy	Policy of the corresponding *Topic.*
	resource_limits	ResourceLimitsQos Policy	Policy of the corresponding *Topic.*
	ownership	OwnershipQosPolicy	Policy of the corresponding *Topic.*
	topic_data	TopicDataQosPolicy	Policy of the corresponding *Topic.*

Topic name	Field Name	Type	Meaning
DCPSPublication (entry created when a ***DataWriter*** is created in association with its ***Publisher***).	key	BuiltinTopicKey_t	DCPS key to distinguish entries.
	participant_key	BuiltinTopicKey_t	DCPS key of the participant to which the *DataWriter* belongs.
	topic_name	string	Name of the related *Topic*.
	type_name	string	Name of the type attached to the related *Topic*.
	durability	DurabilityQosPolicy	Policy of the corresponding *DataWriter*.
	durability_service	DurabilityServiceQosPolicy	Policy of the corresponding *DataWriter*.
	deadline	DeadlineQosPolicy	Policy of the corresponding *DataWriter*.
	latency_budget	LatencyBudgetQosPolicy	Policy of the corresponding *DataWriter*.
	liveliness	LivelinessQosPolicy	Policy of the corresponding *DataWriter*.
	reliability	ReliabilityQosPolicy	Policy of the corresponding *DataWriter*.
	lifespan	LifespanQosPolicy	Policy of the corresponding *DataWriter*.
	user_data	UserDataQosPolicy	Policy of the corresponding *DataWriter*.
	ownership	OwnershipQosPolicy	Policy of the corresponding *DataWriter*.
	ownership_strength	OwnershipStrengthQosPolicy	Policy of the corresponding *DataWriter*.
	destination_order	DestinationOrderQosPolicy	Policy of the corresponding *DataWriter*.
	presentation	PresentationQosPolicy	Policy of the *Publisher* to which the *DataWriter* belongs.
	partition	PartitionQosPolicy	Policy of the *Publisher* to which the *DataWriter* belongs.
	topic_data	TopicDataQosPolicy	Policy of the related *Topic*.
	group_data	GroupDataQosPolicy	Policy of the *Publisher* to which the *DataWriter* belongs.

Topic name	Field Name	Type	Meaning
DCPSSubscription (entry created when a ***DataReader*** is created in association with its ***Subscriber***)	key	BuiltinTopicKey_t	DCPS key to distinguish entries.
	participant_key	BuiltinTopicKey_t	DCPS key of the participant to which the *DataReader* belongs.
	topic_name	string	Name of the related *Topic*.
	type_name	string	Name of the type attached to the related *Topic*.
	durability	DurabilityQosPolicy	Policy of the corresponding *DataReader*.
	deadline	DeadlineQosPolicy	Policy of the corresponding *DataReader*.
	latency_budget	LatencyBudgetQos Policy	Policy of the corresponding *DataReader*.
	liveliness	LivelinessQosPolicy	Policy of the corresponding *DataReader*.
	reliability	ReliabilityQosPolicy	Policy of the corresponding *DataReader*.
	ownership	OwnershipQosPolicy	Policy of the corresponding *DataReader*.
	destination_order	DestinationOrderQos Policy	Policy of the corresponding *DataReader*.
	user_data	UserDataQosPolicy	Policy of the corresponding *DataReader*.
	time_based_filter	TimeBasedFilterQos Policy	Policy of the corresponding *DataReader*.
	presentation	PresentationQosPolicy	Policy of the *Subscriber* to which the *DataReader* belongs.
	partition	PartitionQosPolicy	Policy of the *Subscriber* to which the *DataReader* belongs.
	topic_data	TopicDataQosPolicy	Policy of the related *Topic*.
	group_data	GroupDataQosPolicy	Policy of the *Subscriber* to which the *DataReader* belongs.

2.1.6 Interaction Model

Two interaction models are shown here to illustrate the behavior of the DCPS. The first one concerns publication, the second one subscription.

It should be noted that these models are not intended to explain how the Service is implemented. In particular, what happens on the right side of the picture (e.g., which components actually send the notifications) should be understood as how it *may* work rather than how it *actually does* work (as written inside quotes on the diagrams).

2.1.6.1 Publication View

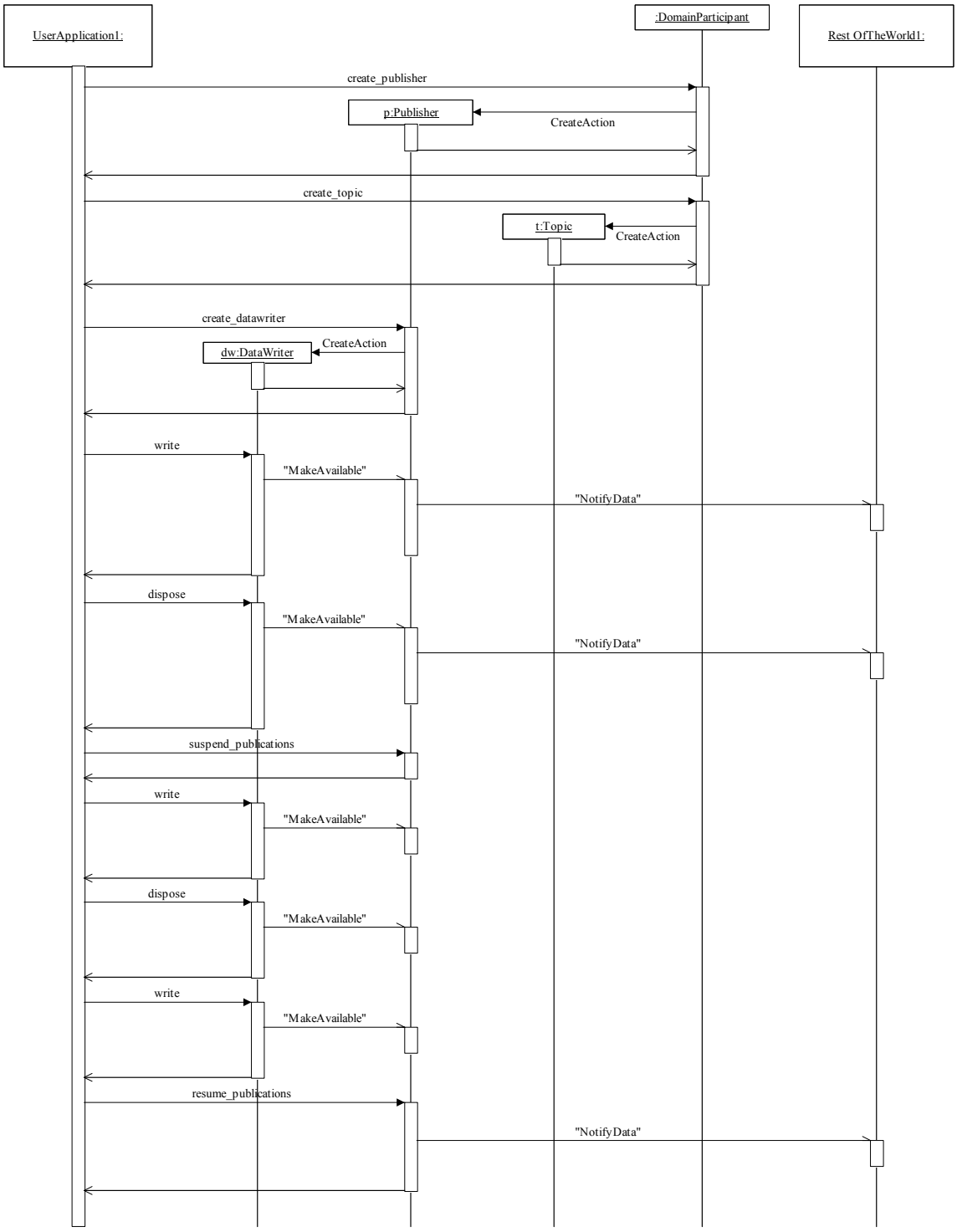

Figure 2-21 DCPS Interaction Model (Publication)

The first part of Figure 2-21 shows the ***Publisher***'s creation. The second part shows that topics must be created before they are referred to by a ***DataWriter***. It should be noted that the topic creation can be requested by a different actor than the one that will use it afterward (in that case, it has to be searched by ***TopicFactory::get_topic***).

The third part of Figure 2-21 shows the ***DataWriter***'s creation. Then, a ***write*** and a ***dispose*** operation are issued on the ***DataWriter,*** which immediately informs the ***Publisher***. Since the application has not invoked the ***suspend_publications*** operation on the ***Publisher***, the corresponding notifications are propagated according to the current ***Publisher***'s policy regarding sending.[36]

The last part of Figure 2-21 shows the same kind of interactions embedded into a pair of ***suspend_publications***/***resume_publications***. It is important to take into account that the corresponding notifications are now delayed until the last ***resume_publications***. It should also be noted that even if the diagram shows only one ***DataWriter***, several of them could be bracketed within a suspend/resume pair.

2.1.6.2 Subscription View

On the subscription side, two diagrams are given. The first one (see Figure 2-22) shows how it works when listeners are used, while the second (see Figure 2-23) shows the use of conditions and wait-sets.

2.1.6.2.1 Notification via Listeners

The first part of Figure 2-22 shows the ***Subscriber***'s and the ***DataReader***'s creation by means of the ***DomainParticipant***.

The second part shows the use of a ***SubscriberListener***: It must first be created and attached to the ***Subscriber*** (***set_listener***). Then when data arrives, it is made available to each related ***DataReader***. Then the ***SubscriberListener*** is triggered (***on_data_on_readers)***. The application must then get the list of affected ***DataReader*** objects (***get_datareaders)***; then it can ***read***/***take*** the data directly on these objects.

Alternatively, the third part of the diagram shows the use of ***DataReaderListener*** objects that are first created and attached to the readers. When data is made available on a ***DataReader***, the related listener is triggered and data can be read (***read***/***take***). It should be noted that, in this case, no link between readers can be retrieved.

Note – When the two kinds of listeners are set, the ***SubscriberListener*** supersedes the ***DataReaderListener*** ones.

36.Usually, this means that the notifications are sent immediately.

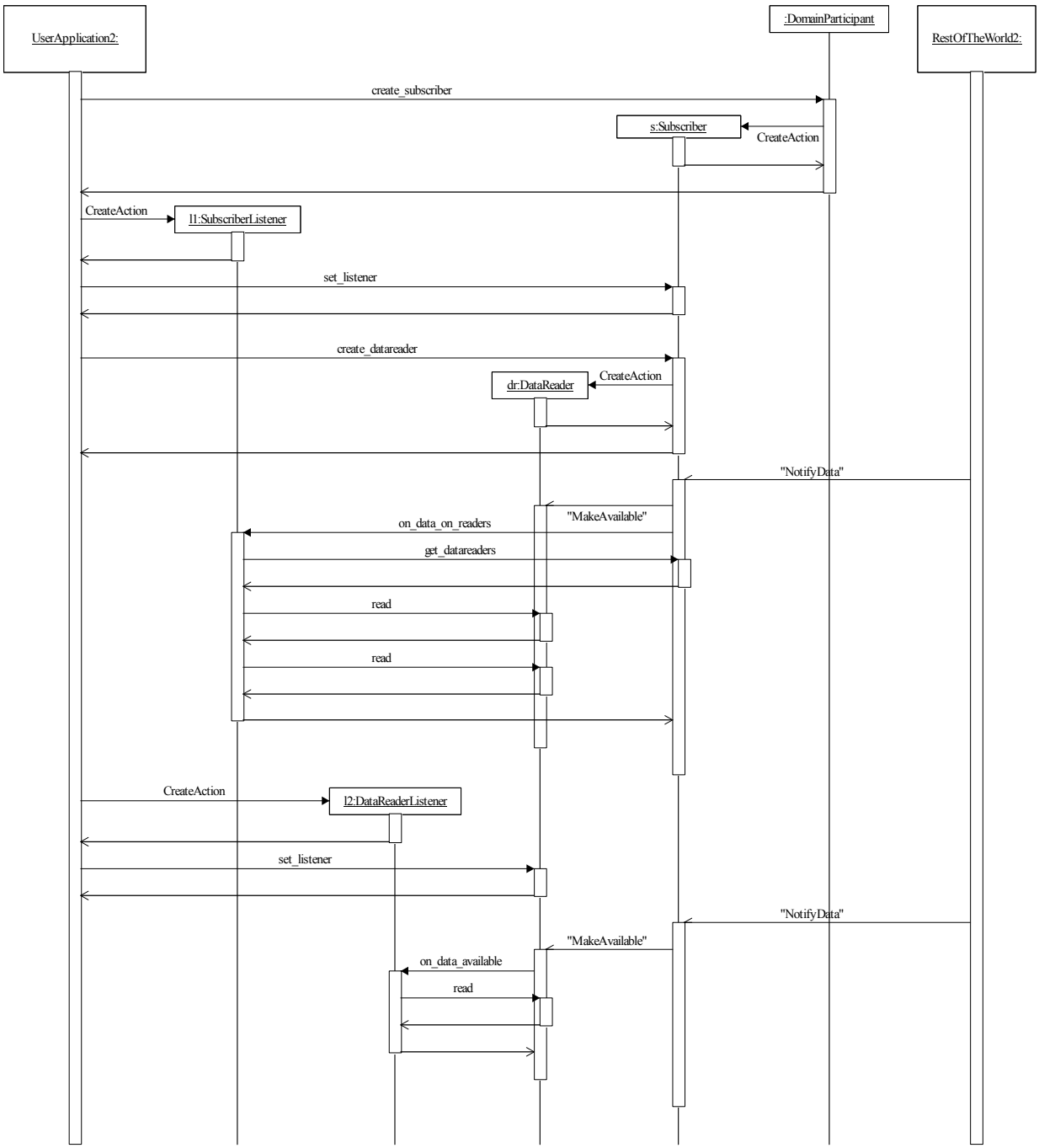

Figure 2-22 DCPS Interaction Model (Subscription with Listeners)

2.1.6.2.2 Notifications via Conditions and Wait-Sets

The first part of Figure 2-22 shows the ***Subscriber***'s and the ***DataReader***'s creation by means of the ***DomainParticipant***.

The second part shows the creation of a ***WaitSet*** and a ***ReadCondition***, the attachment of the latter to the former, and the call to the ***WaitSet::wait*** operation. Note that it is likely that several conditions (***ReadCondition***, but also ***StatusCondition***) will be created and attached to the same ***WaitSet***.

The third part of the diagram shows the information flow when data is received. Note that the ***wait*** returns the list of all the enabled conditions, in an arrival cycle: in case several ***DataReader*** objects receive available data, several conditions will be set enabled at the same time and the application will perform several ***read*** accordingly.

Note – With conditions and wait-sets, read operations are executed naturally in the user context.

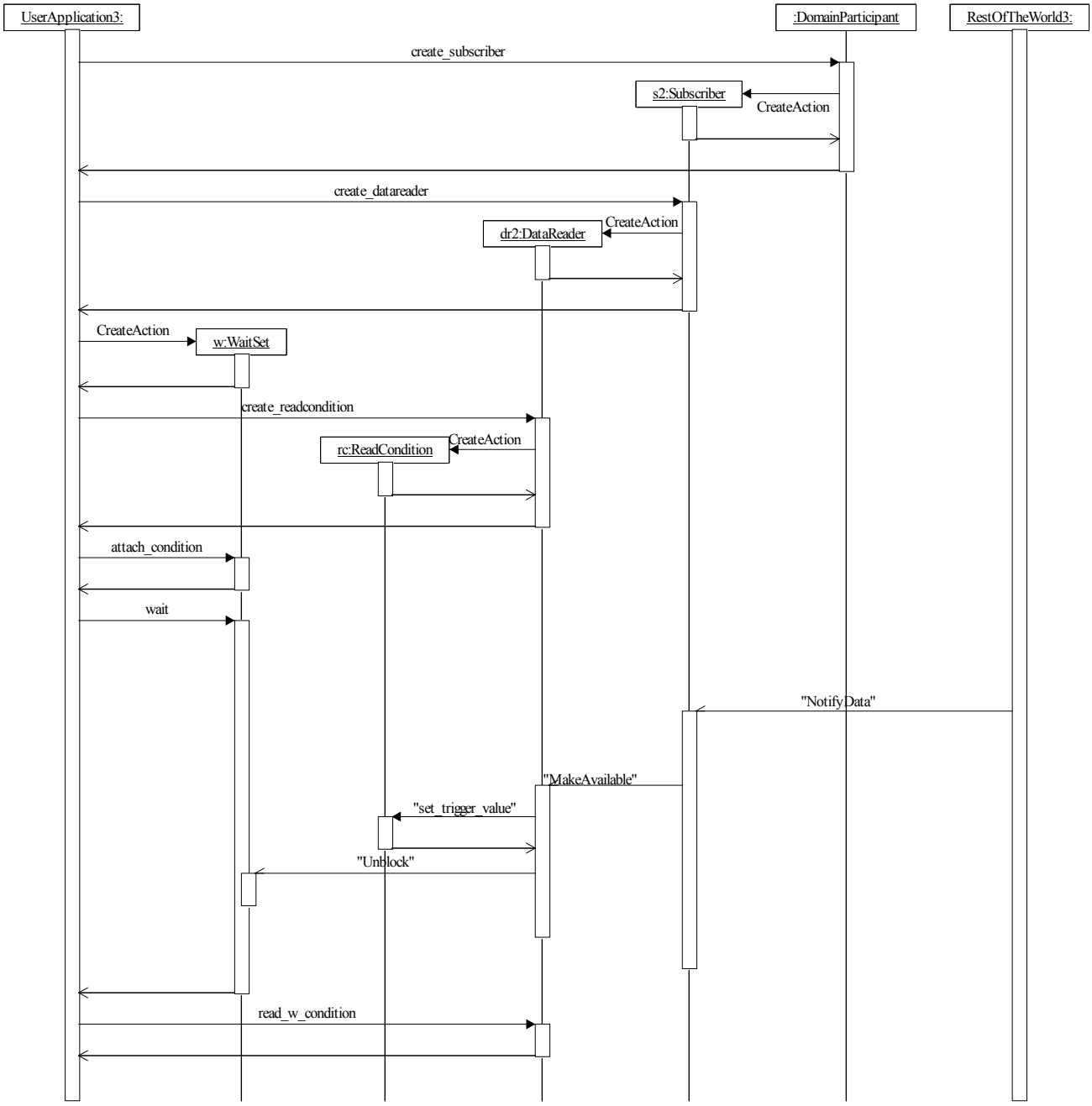

Figure 2-23 DCPS Interaction Model (Subscriptions with Conditions)

2.2 OMG IDL Platform Specific Model (PSM)

2.2.1 Introduction

The OMG IDL PSM is provided by means of the IDL that defines the interface an application can use to interact with the Service.

2.2.2 PIM to PSM Mapping Rules

A key concern in the development of the interface is performance. This is a consequence of the application space targeted by the Data Distribution Service (DDS).

'Out' parameters in the PIM are conventionally mapped to 'inout' parameters in the PSM in order to minimize the memory allocation performed by the Service and allow for more efficient implementations. The intended meaning is that the caller of such an operation should provide an object to serve as a "container" and that the operation will then "fill in" the state of that object appropriately.

The PIM to PSM mapping maps the UML interfaces and classes interfaces into IDL interfaces. Plain data types are mapped into structures.

IDL interfaces do not support overloading. The cases where a base class or interface has an abstract operation that must be redefined by a specialized class or interface has been mapped into a base IDL interface where the abstract operation appears inside comments. This serves simply as a reminder of the fact that all specializations must implement the operation.

Enumerations have been mapped into either IDL 'enum' or hand-picked IDL 'long' values that correspond to increasing powers of 2 (that is 0x01, 0x02, 0x04, etc.). The latter choice has been made when it was required to refer to multiple values as part of a function parameter. This allowed the use of a 'long' as a mask to indicate a set of enumerated values. This selection affected only the PIM 'status kind' values, namely: ***StatusKind***, ***SampleStateKind***, ***ViewStateKind***, and ***InstanceStateKind***. The name of the mask type is formed by replacing the word 'Kind' with the word 'Mask' as in ***StatusMask***, ***SampleStateMask***, etc.

Collection parameters have been mapped into IDL sequences. The only exception applies to the case where the collection elements are hand-picked IDL 'long.' In this case the collection is mapped into an IDL 'long' interpreted as a mask.

Each ***QosPolicy*** has been mapped as an IDL struct. The collection of policies suitable for each Entity has been modeled as another IDL struct that contains attributes corresponding to the policies that apply to this ***Entity***. This approach has several advantages. First, it provides compile-time checking of the suitability of setting a specific ***QosPolicy*** on a particular ***Entity***. A second advantage is that it does not require the use of the type "any" which increases code size and is not natural to use in "C." Other approaches were less attractive. IDL interfaces are not suitable because a collection of ***QosPolicy*** appears as an argument to several operations and needs to be passed "by value." IDL 'valuetype' was considered but rejected because it is not universally supported and also forces each attribute to be accessed via an operation.

Error-return values have been mapped to plain return codes of the corresponding functions. The reason is that DCPS targets “C” as one of the key deployment languages and return codes are more natural to use in “C.”

The DataSample class that associates the ***SampleInfo*** and ***Data*** collections returned from the data-accessing operations (***read*** and ***take***) have not been explicitly mapped into IDL. The collections themselves have been mapped into sequences. The correspondence between each ***Data*** and ***SampleInfo*** is represented by the use of the same index to access the corresponding elements on each of the collections. It is anticipated that additional data-accessing API’s may be provided on each target language to make this operation as natural and efficient is it can be. The reason is that accessing data is the main purpose of the Data-Distribution service, and, the IDL mapping provides a programming-language neutral representation that cannot take advantage of the strengths of each particular language.

The classes that do not have factory operations, namely ***WaitSet*** and ***GuardCondition*** are mapped to IDL interfaces. The intent is that they will be implemented as native classes on each of the implementation languages and they will be constructed using the “new” operator natural for that language. Furthermore, the implementation language mapping should offer at least a constructor that takes no arguments such that applications can be portable across different vendor implementations of this mapping.

The language implementations of the DomainParticipantFactory interface should have the static operation **get_instance** described in Section 2.1.2.2.2, “DomainParticipantFactory Class,” on page 2-34. This operation does not appear in the IDL interface ***DomainParticipantFactory*** as static operations cannot be expressed in IDL.

The two types used to represent time: ***Duration_t*** and ***Time_t*** are been mapped into structures that contain fields for the second and the nanosecond parts. These types are further constrained to always use a ‘normalized’ representation for the time, that is, the *nanosec* field must verify 0 <= *nanosec* < 1000000000.

The IDL PSM introduces a number of types that are intended to be defined in a native way. As these are opaque types, the actual definition of the type does not affect portability and is implementation dependent. For completeness the names of the types appear as typedefs in the IDL and a #define with the suffix “_TYPE_NATIVE” is used as a place-holder for the actual type. The type used in the IDL by this means is not normative and an implementation is allowed to use any other type, including non-scalar (i.e., structured types).

2.2.3 DCPS PSM : IDL

```
#define DOMAINID_TYPE_NATIVE    long
#define HANDLE_TYPE_NATIVE      long
#define HANDLE_NIL_NATIVE       0
#define BUILTIN_TOPIC_KEY_TYPE_NATIVE     long

#define TheParticipantFactory
#define PARTICIPANT_QOS_DEFAULT
#define TOPIC_QOS_DEFAULT
#define PUBLISHER_QOS_DEFAULT
```

```
#define SUBSCRIBER_QOS_DEFAULT
#define DATAWRITER_QOS_DEFAULT
#define DATAREADER_QOS_DEFAULT
#define DATAWRITER_QOS_USE_TOPIC_QOS
#define DATAREADER_QOS_USE_TOPIC_QOS

module DDS {
    typedef DOMAINID_TYPE_NATIVE  DomainId_t;
    typedef HANDLE_TYPE_NATIVE    InstanceHandle_t;

    struct BuiltinTopicKey_t {
        BUILTIN_TOPIC_KEY_TYPE_NATIVE value[3];
    };

    typedef sequence<InstanceHandle_t> InstanceHandleSeq;

    typedef long ReturnCode_t;
    typedef long QosPolicyId_t;
    typedef sequence<string> StringSeq;

    struct Duration_t {
        long sec;
        unsigned long nanosec;
    };

    struct Time_t {
        long sec;
        unsigned long nanosec;
    };

    // -----------------------------------------------------------------------
    // Pre-defined values
    // -----------------------------------------------------------------------
    const InstanceHandle_t HANDLE_NIL = HANDLE_NIL_NATIVE;

    const long LENGTH_UNLIMITED = -1;

    const long            DURATION_INFINITE_SEC   = 0x7fffffff;
    const unsigned long   DURATION_INFINITE_NSEC  = 0x7fffffff;

    const long            DURATION_ZERO_SEC       = 0;
    const unsigned long   DURATION_ZERO_NSEC      = 0;

    const long            TIME_INVALID_SEC        = -1;
    const unsigned long   TIME_INVALID_NSEC       = 0xffffffff;

    // -----------------------------------------------------------------------
    // Return codes
    // -----------------------------------------------------------------------
    const ReturnCode_t RETCODE_OK                     = 0;
    const ReturnCode_t RETCODE_ERROR                  = 1;
    const ReturnCode_t RETCODE_UNSUPPORTED            = 2;
    const ReturnCode_t RETCODE_BAD_PARAMETER          = 3;
    const ReturnCode_t RETCODE_PRECONDITION_NOT_MET   = 4;
    const ReturnCode_t RETCODE_OUT_OF_RESOURCES       = 5;
```

```
const ReturnCode_t RETCODE_NOT_ENABLED           = 6;
const ReturnCode_t RETCODE_IMMUTABLE_POLICY      = 7;
const ReturnCode_t RETCODE_INCONSISTENT_POLICY   = 8;
const ReturnCode_t RETCODE_ALREADY_DELETED       = 9;
const ReturnCode_t RETCODE_TIMEOUT               = 10;
const ReturnCode_t RETCODE_NO_DATA               = 11;
const ReturnCode_t RETCODE_ILLEGAL_OPERATION     = 12;

// ----------------------------------------------------------------------
// Status to support listeners and conditions
// ----------------------------------------------------------------------

typedef unsigned long StatusKind;
typedef unsigned long StatusMask;    // bit-mask StatusKind

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

struct InconsistentTopicStatus {
    long total_count;
    long total_count_change;
};

struct SampleLostStatus {
    long total_count;
    long total_count_change;
};

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

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

struct LivelinessLostStatus {
    long                total_count;
```

```
    long                    total_count_change;
};

struct LivelinessChangedStatus {
    long                    alive_count;
    long                    not_alive_count;
    long                    alive_count_change;
    long                    not_alive_count_change;
    InstanceHandle_t        last_publication_handle;
};

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

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

struct QosPolicyCount {
    QosPolicyId_t           policy_id;
    long                    count;
};

typedef sequence<QosPolicyCount> QosPolicyCountSeq;

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

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


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


struct SubscriptionMatchedStatus {
```

```
    long                total_count;
    long                total_count_change;
    long                current_count;
    long                current_count_change;
    InstanceHandle_t    last_publication_handle;
};

// -----------------------------------------------------------------------
// Listeners
// -----------------------------------------------------------------------

interface Listener;
interface Entity;
interface TopicDescription;
interface Topic;
interface ContentFilteredTopic;
interface MultiTopic;
interface DataWriter;
interface DataReader;
interface Subscriber;
interface Publisher;

typedef sequence<DataReader> DataReaderSeq;

interface Listener {};

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

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

interface PublisherListener : DataWriterListener {
};

interface DataReaderListener : Listener {
    void on_requested_deadline_missed(
        in DataReader the_reader,
        in RequestedDeadlineMissedStatus status);
    void on_requested_incompatible_qos(
        in DataReader the_reader,
```

```
        in RequestedIncompatibleQosStatus status);
    void on_sample_rejected(
        in DataReader the_reader,
        in SampleRejectedStatus status);
    void on_liveliness_changed(
        in DataReader the_reader,
        in LivelinessChangedStatus status);
    void on_data_available(
        in DataReader the_reader);
    void on_subscription_matched(
        in DataReader the_reader,
        in SubscriptionMatchedStatus status);
    void on_sample_lost(
        in DataReader the_reader,
        in SampleLostStatus status);
};

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


interface DomainParticipantListener : TopicListener,
                                      PublisherListener,
                                      SubscriberListener {
};


// -----------------------------------------------------------------------
// Conditions
// -----------------------------------------------------------------------

interface Condition {
    boolean get_trigger_value();
};

typedef sequence<Condition> ConditionSeq;

interface WaitSet {
    ReturnCode_t wait(
        inout ConditionSeq active_conditions,
        in Duration_t timeout);
    ReturnCode_t attach_condition(
        in Condition cond);
    ReturnCode_t detach_condition(
        in Condition cond);
    ReturnCode_t get_conditions(
        inout ConditionSeq attached_conditions);
};

interface GuardCondition : Condition {
    ReturnCode_t set_trigger_value(
        in boolean value);
};
```

```
interface StatusCondition : Condition {
    StatusMask get_enabled_statuses();
    ReturnCode_t set_enabled_statuses(
        in StatusMask mask);
    Entity get_entity();
};

// Sample states to support reads
typedef unsigned long SampleStateKind;
const SampleStateKind READ_SAMPLE_STATE                     = 0x0001 << 0;
const SampleStateKind NOT_READ_SAMPLE_STATE                 = 0x0001 << 1;

// This is a bit-mask SampleStateKind
typedef unsigned long SampleStateMask;
const SampleStateMask ANY_SAMPLE_STATE                      = 0xffff;

// View states to support reads
typedef unsigned long ViewStateKind;
const ViewStateKind NEW_VIEW_STATE                          = 0x0001 << 0;
const ViewStateKind NOT_NEW_VIEW_STATE                      = 0x0001 << 1;

// This is a bit-mask ViewStateKind
typedef unsigned long ViewStateMask;
const ViewStateMask ANY_VIEW_STATE                          = 0xffff;

// Instance states to support reads
typedef unsigned long InstanceStateKind;
const InstanceStateKind ALIVE_INSTANCE_STATE                = 0x0001 << 0;
const InstanceStateKind NOT_ALIVE_DISPOSED_INSTANCE_STATE   = 0x0001 << 1;
const InstanceStateKind NOT_ALIVE_NO_WRITERS_INSTANCE_STATE = 0x0001 << 2;

// This is a bit-mask InstanceStateKind
typedef unsigned long InstanceStateMask;
const InstanceStateMask ANY_INSTANCE_STATE                  = 0xffff;
const InstanceStateMask NOT_ALIVE_INSTANCE_STATE            = 0x006;


interface ReadCondition : Condition {
    SampleStateMask     get_sample_state_mask();
    ViewStateMask       get_view_state_mask();
    InstanceStateMask   get_instance_state_mask();
    DataReader          get_datareader();
};

interface QueryCondition : ReadCondition {
    string         get_query_expression();
    ReturnCode_t   get_query_parameters(
        inout StringSeq query_parameters);
    ReturnCode_t   set_query_parameters(
        in StringSeq query_parameters);
};

// -----------------------------------------------------------------------
// Qos
```

```
// -----------------------------------------------------------------------
const string USERDATA_QOS_POLICY_NAME              = "UserData";
const string DURABILITY_QOS_POLICY_NAME            = "Durability";
const string PRESENTATION_QOS_POLICY_NAME          = "Presentation";
const string DEADLINE_QOS_POLICY_NAME              = "Deadline";
const string LATENCYBUDGET_QOS_POLICY_NAME         = "LatencyBudget";
const string OWNERSHIP_QOS_POLICY_NAME             = "Ownership";
const string OWNERSHIPSTRENGTH_QOS_POLICY_NAME     = "OwnershipStrength";
const string LIVELINESS_QOS_POLICY_NAME            = "Liveliness";
const string TIMEBASEDFILTER_QOS_POLICY_NAME       = "TimeBasedFilter";
const string PARTITION_QOS_POLICY_NAME             = "Partition";
const string RELIABILITY_QOS_POLICY_NAME           = "Reliability";
const string DESTINATIONORDER_QOS_POLICY_NAME      = "DestinationOrder";
const string HISTORY_QOS_POLICY_NAME               = "History";
const string RESOURCELIMITS_QOS_POLICY_NAME        = "ResourceLimits";
const string ENTITYFACTORY_QOS_POLICY_NAME         = "EntityFactory";
const string WRITERDATALIFECYCLE_QOS_POLICY_NAME   = "WriterDataLifecycle";
const string READERDATALIFECYCLE_QOS_POLICY_NAME   = "ReaderDataLifecycle";
const string TOPICDATA_QOS_POLICY_NAME             = "TopicData";
const string GROUPDATA_QOS_POLICY_NAME             = "TransportPriority";
const string LIFESPAN_QOS_POLICY_NAME              = "Lifespan";
const string DURABILITYSERVICE_POLICY_NAME         = "DurabilityService";

const QosPolicyId_t INVALID_QOS_POLICY_ID             = 0;
const QosPolicyId_t USERDATA_QOS_POLICY_ID            = 1;
const QosPolicyId_t DURABILITY_QOS_POLICY_ID          = 2;
const QosPolicyId_t PRESENTATION_QOS_POLICY_ID        = 3;
const QosPolicyId_t DEADLINE_QOS_POLICY_ID            = 4;
const QosPolicyId_t LATENCYBUDGET_QOS_POLICY_ID       = 5;
const QosPolicyId_t OWNERSHIP_QOS_POLICY_ID           = 6;
const QosPolicyId_t OWNERSHIPSTRENGTH_QOS_POLICY_ID   = 7;
const QosPolicyId_t LIVELINESS_QOS_POLICY_ID          = 8;
const QosPolicyId_t TIMEBASEDFILTER_QOS_POLICY_ID     = 9;
const QosPolicyId_t PARTITION_QOS_POLICY_ID           = 10;
const QosPolicyId_t RELIABILITY_QOS_POLICY_ID         = 11;
const QosPolicyId_t DESTINATIONORDER_QOS_POLICY_ID    = 12;
const QosPolicyId_t HISTORY_QOS_POLICY_ID             = 13;
const QosPolicyId_t RESOURCELIMITS_QOS_POLICY_ID      = 14;
const QosPolicyId_t ENTITYFACTORY_QOS_POLICY_ID       = 15;
const QosPolicyId_t WRITERDATALIFECYCLE_QOS_POLICY_ID = 16;
const QosPolicyId_t READERDATALIFECYCLE_QOS_POLICY_ID = 17;
const QosPolicyId_t TOPICDATA_QOS_POLICY_ID           = 18;
const QosPolicyId_t GROUPDATA_QOS_POLICY_ID           = 19;
const QosPolicyId_t TRANSPORTPRIORITY_QOS_POLICY_ID   = 20;
const QosPolicyId_t LIFESPAN_QOS_POLICY_ID            = 21;
const QosPolicyId_t DURABILITYSERVICE_QOS_POLICY_ID   = 22;

struct UserDataQosPolicy {
    sequence<octet> value;
};

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

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

struct TransportPriorityQosPolicy {
    long value;
};

struct LifespanQosPolicy {
    Duration_t duration;
};

enum DurabilityQosPolicyKind {
    VOLATILE_DURABILITY_QOS,
    TRANSIENT_LOCAL_DURABILITY_QOS,
    TRANSIENT_DURABILITY_QOS,
    PERSISTENT_DURABILITY_QOS
};
struct DurabilityQosPolicy {
    DurabilityQosPolicyKind kind;
};

enum PresentationQosPolicyAccessScopeKind {
    INSTANCE_PRESENTATION_QOS,
    TOPIC_PRESENTATION_QOS,
    GROUP_PRESENTATION_QOS
};
struct PresentationQosPolicy {
    PresentationQosPolicyAccessScopeKind access_scope;
    boolean coherent_access;
    boolean ordered_access;
};

struct DeadlineQosPolicy {
    Duration_t period;
};

struct LatencyBudgetQosPolicy {
    Duration_t duration;
};

enum OwnershipQosPolicyKind {
    SHARED_OWNERSHIP_QOS,
    EXCLUSIVE_OWNERSHIP_QOS
};
struct OwnershipQosPolicy {
    OwnershipQosPolicyKind kind;
};

struct OwnershipStrengthQosPolicy {
    long value;
};

enum LivelinessQosPolicyKind {
    AUTOMATIC_LIVELINESS_QOS,
```

```
        MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
        MANUAL_BY_TOPIC_LIVELINESS_QOS
    };

    struct LivelinessQosPolicy {
        LivelinessQosPolicyKind kind;
        Duration_t lease_duration;
    };

    struct TimeBasedFilterQosPolicy {
        Duration_t minimum_separation;
    };

    struct PartitionQosPolicy {
        StringSeq name;
    };

    enum ReliabilityQosPolicyKind {
        BEST_EFFORT_RELIABILITY_QOS,
        RELIABLE_RELIABILITY_QOS
    };

    struct ReliabilityQosPolicy {
        ReliabilityQosPolicyKind kind;
        Duration_t max_blocking_time;
    };

    enum DestinationOrderQosPolicyKind {
        BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS,
        BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
    };
    struct DestinationOrderQosPolicy {
        DestinationOrderQosPolicyKind kind;
    };

    enum HistoryQosPolicyKind {
        KEEP_LAST_HISTORY_QOS,
        KEEP_ALL_HISTORY_QOS
    };
    struct HistoryQosPolicy {
        HistoryQosPolicyKind kind;
        long depth;
    };

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

    struct EntityFactoryQosPolicy {
        boolean autoenable_created_entities;
    };

    struct WriterDataLifecycleQosPolicy {
```

```
    boolean autodispose_unregistered_instances;
};

struct ReaderDataLifecycleQosPolicy {
    Duration_t autopurge_nowriter_samples_delay;
    Duration_t autopurge_disposed_samples_delay;
};

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

struct DomainParticipantFactoryQos {
    EntityFactoryQosPolicy entity_factory;
};

struct DomainParticipantQos {
    UserDataQosPolicy  user_data;
    EntityFactoryQosPolicy entity_factory;
};

struct TopicQos {
    TopicDataQosPolicy                  topic_data;
    DurabilityQosPolicy                 durability;
    DurabilityServiceQosPolicy          durability_service;
    DeadlineQosPolicy                   deadline;
    LatencyBudgetQosPolicy              latency_budget;
    LivelinessQosPolicy                 liveliness;
    ReliabilityQosPolicy                reliability;
    DestinationOrderQosPolicy           destination_order;
    HistoryQosPolicy                    history;
    ResourceLimitsQosPolicy             resource_limits;
    TransportPriorityQosPolicy          transport_priority;
    LifespanQosPolicy                   lifespan;

    OwnershipQosPolicy                  ownership;
};

struct DataWriterQos {
    DurabilityQosPolicy                 durability;
    DurabilityServiceQosPolicy          durability_service;
    DeadlineQosPolicy                   deadline;
    LatencyBudgetQosPolicy              latency_budget;
    LivelinessQosPolicy                 liveliness;
    ReliabilityQosPolicy                reliability;
    DestinationOrderQosPolicy           destination_order;
    HistoryQosPolicy                    history;
    ResourceLimitsQosPolicy             resource_limits;
    TransportPriorityQosPolicy          transport_priority;
    LifespanQosPolicy                   lifespan;
```

```
    UserDataQosPolicy                   user_data;
    OwnershipQosPolicy                  ownership;
    OwnershipStrengthQosPolicy          ownership_strength;
    WriterDataLifecycleQosPolicy        writer_data_lifecycle;
};

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

struct DataReaderQos {
    DurabilityQosPolicy                 durability;
    DeadlineQosPolicy                   deadline;
    LatencyBudgetQosPolicy              latency_budget;
    LivelinessQosPolicy                 liveliness;
    ReliabilityQosPolicy                reliability;
    DestinationOrderQosPolicy           destination_order;
    HistoryQosPolicy                    history;
    ResourceLimitsQosPolicy             resource_limits;

    UserDataQosPolicy                   user_data;
    OwnershipQosPolicy                  ownership;
    TimeBasedFilterQosPolicy            time_based_filter;
    ReaderDataLifecycleQosPolicy        reader_data_lifecycle;
};

struct SubscriberQos {
    PresentationQosPolicy               presentation;
    PartitionQosPolicy                  partition;
    GroupDataQosPolicy                  group_data;
    EntityFactoryQosPolicy              entity_factory;
};

// -----------------------------------------------------------------------

struct ParticipantBuiltinTopicData {
    BuiltinTopicKey_t                   key;
    UserDataQosPolicy                   user_data;
};

struct TopicBuiltinTopicData {
    BuiltinTopicKey_t                   key;
    string                              name;
    string                              type_name;
    DurabilityQosPolicy                 durability;
    DurabilityServiceQosPolicy          durability_service;
    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;
};

struct PublicationBuiltinTopicData {
    BuiltinTopicKey_t                   key;
    BuiltinTopicKey_t                   participant_key;
    string                              topic_name;
    string                              type_name;

    DurabilityQosPolicy                 durability;
    DurabilityServiceQosPolicy          durability_service;
    DeadlineQosPolicy                   deadline;
    LatencyBudgetQosPolicy              latency_budget;
    LivelinessQosPolicy                 liveliness;
    ReliabilityQosPolicy                reliability;
    LifespanQosPolicy                   lifespan;
    UserDataQosPolicy                   user_data;
    OwnershipQosPolicy                  ownership;
    OwnershipStrengthQosPolicy          ownership_strength;
    DestinationOrderQosPolicy           destination_order;

    PresentationQosPolicy               presentation;
    PartitionQosPolicy                  partition;
    TopicDataQosPolicy                  topic_data;
    GroupDataQosPolicy                  group_data;
};

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;
    OwnershipQosPolicy                  ownership;
    DestinationOrderQosPolicy           destination_order;
    UserDataQosPolicy                   user_data;
    TimeBasedFilterQosPolicy            time_based_filter;

    PresentationQosPolicy               presentation;
    PartitionQosPolicy                  partition;
    TopicDataQosPolicy                  topic_data;
    GroupDataQosPolicy                  group_data;
};

// ---------------------------------------------------------------------------
```

```
interface Entity {
//  ReturnCode_t set_qos(
//      in EntityQos qos);
//  ReturnCode_t get_qos(
//      inout EntityQos qos);
//  ReturnCode_t set_listener(
//      in Listener l,
//      in StatusMask mask);
//  Listener get_listener();

    ReturnCode_t enable();

    StatusCondition get_statuscondition();

    StatusMask get_status_changes();

    InstanceHandle_t get_instance_handle();
};

// -----------------------------------------------------------------------
interface DomainParticipant : Entity {
    // Factory interfaces
    Publisher create_publisher(
        in PublisherQos qos,
        in PublisherListener a_listener,
        in StatusMask mask);
    ReturnCode_t delete_publisher(
        in Publisher p);

    Subscriber create_subscriber(
        in SubscriberQos qos,
        in SubscriberListener a_listener,
        in StatusMask mask);
    ReturnCode_t delete_subscriber(
        in Subscriber s);
    Subscriber get_builtin_subscriber();

    Topic create_topic(
        in string topic_name,
        in string type_name,
        in TopicQos qos,
        in TopicListener a_listener,
        in StatusMask mask);

    ReturnCode_t delete_topic(
        in Topic a_topic);

    Topic find_topic(
        in string topic_name,
        in Duration_t timeout);
    TopicDescription lookup_topicdescription(
        in string name);

    ContentFilteredTopic create_contentfilteredtopic(
        in string name,
```

```
    in Topic related_topic,
    in string filter_expression,
    in StringSeq expression_parameters);

ReturnCode_t delete_contentfilteredtopic(
    in ContentFilteredTopic a_contentfilteredtopic);

MultiTopic create_multitopic(
    in string name,
    in string type_name,
    in string subscription_expression,
    in StringSeq expression_parameters);

ReturnCode_t delete_multitopic(
    in MultiTopic a_multitopic);

ReturnCode_t delete_contained_entities();

ReturnCode_t set_qos(
    in DomainParticipantQos qos);
ReturnCode_t get_qos(
    inout DomainParticipantQos qos);

ReturnCode_t set_listener(
    in DomainParticipantListener a_listener,
    in StatusMask mask);
DomainParticipantListener get_listener();

ReturnCode_t ignore_participant(
    in InstanceHandle_t handle);
ReturnCode_t ignore_topic(
    in InstanceHandle_t handle);
ReturnCode_t ignore_publication(
    in InstanceHandle_t handle);
ReturnCode_t ignore_subscription(
    in InstanceHandle_t handle);

DomainId_t get_domain_id();
ReturnCode_t assert_liveliness();

ReturnCode_t set_default_publisher_qos(
    in PublisherQos qos);
ReturnCode_t get_default_publisher_qos(
    inout PublisherQos qos);

ReturnCode_t set_default_subscriber_qos(
    in SubscriberQos qos);
ReturnCode_t get_default_subscriber_qos(
    inout SubscriberQos qos);

ReturnCode_t set_default_topic_qos(
    in TopicQos qos);
ReturnCode_t get_default_topic_qos(
    inout TopicQos qos);
```

```
    ReturnCode_t get_discovered_participants(
        inout InstanceHandleSeq participant_handles);
    ReturnCode_t get_discovered_participant_data(
        inout ParticipantBuiltinTopicData participant_data,
        in InstanceHandle_t participant_handle);

    ReturnCode_t get_discovered_topics(
        inout InstanceHandleSeq topic_handles);
    ReturnCode_t get_discovered_topic_data(
        inout TopicBuiltinTopicData topic_data,
        in InstanceHandle_t topic_handle);

    boolean contains_entity(
        in InstanceHandle_t a_handle);

    ReturnCode_t get_current_time(
        inout Time_t current_time);
};

interface DomainParticipantFactory {
    DomainParticipant create_participant(
        in DomainId_t domain_id,
        in DomainParticipantQos qos,
        in DomainParticipantListener a_listener,
        in StatusMask mask);
    ReturnCode_t delete_participant(
        in DomainParticipant a_participant);

    DomainParticipant lookup_participant(
        in DomainId_t domain_id);

    ReturnCode_t set_default_participant_qos(
        in DomainParticipantQos qos);
    ReturnCode_t get_default_participant_qos(
        inout DomainParticipantQos qos);

    ReturnCode_t set_qos(
        in DomainParticipantFactoryQos qos);
    ReturnCode_t get_qos(
        inout DomainParticipantFactoryQos qos);
};

interface TypeSupport {
//     ReturnCode_t register_type(
//         in DomainParticipant domain,
//         in string type_name);
//     string get_type_name();
};

// -----------------------------------------------------------------------
interface TopicDescription {
    string get_type_name();
    string get_name();

    DomainParticipant get_participant();
```

```
};

interface Topic : Entity, TopicDescription {
    ReturnCode_t set_qos(
        in TopicQos qos);
    ReturnCode_t get_qos(
        inout TopicQos qos);
    ReturnCode_t set_listener(
        in TopicListener a_listener,
        in StatusMask mask);
    TopicListener get_listener();
    // Access the status
    ReturnCode_t get_inconsistent_topic_status(
        inout InconsistentTopicStatus a_status);
};

interface ContentFilteredTopic : TopicDescription {
    string get_filter_expression();
    ReturnCode_t get_expression_parameters(
        inout StringSeq);
    ReturnCode_t set_expression_parameters(
        in StringSeq expression_parameters);
    Topic get_related_topic();
};

interface MultiTopic : TopicDescription {
    string get_subscription_expression();
    ReturnCode_t get_expression_parameters(
        inout StringSeq);
    ReturnCode_t set_expression_parameters(
        in StringSeq expression_parameters);
};

// ---------------------------------------------------------------------------
interface Publisher : Entity {
    DataWriter create_datawriter(
        in Topic a_topic,
        in DataWriterQos qos,
        in DataWriterListener a_listener,
        in StatusMask mask);
    ReturnCode_t delete_datawriter(
        in DataWriter a_datawriter);
    DataWriter lookup_datawriter(
        in string topic_name);

    ReturnCode_t delete_contained_entities();

    ReturnCode_t set_qos(
        in PublisherQos qos);
    ReturnCode_t get_qos(
        inout PublisherQos qos);

    ReturnCode_t set_listener(
        in PublisherListener a_listener,
        in StatusMask mask);
```

```
        PublisherListener get_listener();

        ReturnCode_t suspend_publications();
        ReturnCode_t resume_publications();

        ReturnCode_t begin_coherent_changes();
        ReturnCode_t end_coherent_changes();

        ReturnCode_t wait_for_acknowledgments(
            in Duration_t max_wait);

        DomainParticipant get_participant();

        ReturnCode_t set_default_datawriter_qos(
            in DataWriterQos qos);
        ReturnCode_t get_default_datawriter_qos(
            inout DataWriterQos qos);

        ReturnCode_t copy_from_topic_qos(
            inout DataWriterQos a_datawriter_qos,
            in TopicQos a_topic_qos);
    };

    interface DataWriter : Entity {
    //  InstanceHandle_t register_instance(
    //      in Data instance_data);
    //  InstanceHandle_t register_instance_w_timestamp(
    //      in Data instance_data,
    //      in Time_t source_timestamp);
    //  ReturnCode_t unregister_instance(
    //      in Data instance_data,
    //      in InstanceHandle_t  handle);
    //  ReturnCode_t unregister_instance_w_timestamp(
    //      in Data instance_data,
    //      in InstanceHandle_t  handle,
    //      in Time_t source_timestamp);
    //  ReturnCode_t write(
    //      in Data instance_data,
    //      in InstanceHandle_t handle);
    //  ReturnCode_t write_w_timestamp(
    //      in Data instance_data,
    //      in InstanceHandle_t handle,
    //      in Time_t source_timestamp);
    //  ReturnCode_t dispose(
    //      in Data instance_data,
    //      in InstanceHandle_t instance_handle);
    //  ReturnCode_t dispose_w_timestamp(
    //      in Data instance_data,
    //      in InstanceHandle_t instance_handle,
    //      in Time_t source_timestamp);
    //  ReturnCode_t get_key_value(
    //      inout Data key_holder,
    //      in InstanceHandle_t handle);
    //  InstanceHandle_t lookup_instance(
    //      in Data instance_data);
```

```
        ReturnCode_t set_qos(
            in DataWriterQos qos);
        ReturnCode_t get_qos(
            inout DataWriterQos qos);

        ReturnCode_t set_listener(
            in DataWriterListener a_listener,
            in StatusMask mask);
        DataWriterListener get_listener();

        Topic get_topic();
        Publisher get_publisher();

        ReturnCode_t wait_for_acknowledgments(
            in Duration_t max_wait);

        // Access the status
        ReturnCode_t get_liveliness_lost_status(
            inout LivelinessLostStatus);
        ReturnCode_t get_offered_deadline_missed_status(
            inout OfferedDeadlineMissedStatus status);
        ReturnCode_t get_offered_incompatible_qos_status(
            inout OfferedIncompatibleQosStatus status);
        ReturnCode_t get_publication_matched_status(
            inout PublicationMatchedStatus status);

        ReturnCode_t assert_liveliness();

        ReturnCode_t get_matched_subscriptions(
            inout InstanceHandleSeq subscription_handles);
        ReturnCode_t get_matched_subscription_data(
            inout SubscriptionBuiltinTopicData subscription_data,
            in InstanceHandle_t subscription_handle);
    };

    // -----------------------------------------------------------------------
    interface Subscriber : Entity {
        DataReader create_datareader(
            in TopicDescription a_topic,
            in DataReaderQos qos,
            in DataReaderListener a_listener,
            in StatusMask mask);
        ReturnCode_t delete_datareader(
            in DataReader a_datareader);
        ReturnCode_t delete_contained_entities();
        DataReader lookup_datareader(
            in string topic_name);
        ReturnCode_t get_datareaders(
            inout DataReaderSeq readers,
            in SampleStateMask sample_states,
            in ViewStateMask view_states,
            in InstanceStateMask instance_states);
        ReturnCode_t notify_datareaders();
```

```
        ReturnCode_t set_qos(
            in SubscriberQos qos);
        ReturnCode_t get_qos(
            inout SubscriberQos qos);

        ReturnCode_t set_listener(
            in SubscriberListener a_listener,
            in StatusMask mask);
        SubscriberListener get_listener();

        ReturnCode_t begin_access();
        ReturnCode_t end_access();

        DomainParticipant get_participant();

        ReturnCode_t set_default_datareader_qos(
            in DataReaderQos qos);
        ReturnCode_t get_default_datareader_qos(
            inout DataReaderQos qos);

        ReturnCode_t copy_from_topic_qos(
            inout DataReaderQos a_datareader_qos,
            in TopicQos a_topic_qos);
    };

    interface DataReader : Entity {
    //  ReturnCode_t read(
    //      inout DataSeq data_values,
    //      inout SampleInfoSeq sample_infos,
    //      in long max_samples,
    //      in SampleStateMask sample_states,
    //      in ViewStateMask view_states,
    //      in InstanceStateMask instance_states);

    //  ReturnCode_t take(
    //      inout DataSeq data_values,
    //      inout SampleInfoSeq sample_infos,
    //      in long max_samples,
    //      in SampleStateMask sample_states,
    //      in ViewStateMask view_states,
    //      in InstanceStateMask instance_states);

    //  ReturnCode_t read_w_condition(
    //      inout DataSeq data_values,
    //      inout SampleInfoSeq sample_infos,
    //      in long max_samples,
    //      in ReadCondition a_condition);

    //  ReturnCode_t take_w_condition(
    //      inout DataSeq data_values,
    //      inout SampleInfoSeq sample_infos,
    //      in long max_samples,
    //      in ReadCondition a_condition);

    //  ReturnCode_t read_next_sample(
```

```
//      inout Data data_values,
//      inout SampleInfo sample_info);

//  ReturnCode_t take_next_sample(
//      inout Data data_values,
//      inout SampleInfo sample_info);

//  ReturnCode_t read_instance(
//      inout DataSeq data_values,
//      inout SampleInfoSeq sample_infos,
//      in long max_samples,
//      in InstanceHandle_t a_handle,
//      in SampleStateMask sample_states,
//      in ViewStateMask view_states,
//      in InstanceStateMask instance_states);

//  ReturnCode_t take_instance(
//      inout DataSeq data_values,
//      inout SampleInfoSeq sample_infos,
//      in long max_samples,
//      in InstanceHandle_t a_handle,
//      in SampleStateMask sample_states,
//      in ViewStateMask view_states,
//      in InstanceStateMask instance_states);

//  ReturnCode_t read_next_instance(
//      inout DataSeq data_values,
//      inout SampleInfoSeq sample_infos,
//      in long max_samples,
//      in InstanceHandle_t previous_handle,
//      in SampleStateMask sample_states,
//      in ViewStateMask view_states,
//      in InstanceStateMask instance_states);

//  ReturnCode_t take_next_instance(
//      inout DataSeq data_values,
//      inout SampleInfoSeq sample_infos,
//      in long max_samples,
//      in InstanceHandle_t previous_handle,
//      in SampleStateMask sample_states,
//      in ViewStateMask view_states,
//      in InstanceStateMask instance_states);

//  ReturnCode_t read_next_instance_w_condition(
//      inout DataSeq data_values,
//      inout SampleInfoSeq sample_infos,
//      in long max_samples,
//      in InstanceHandle_t previous_handle,
//      in ReadCondition a_condition);

//  ReturnCode_t take_next_instance_w_condition(
//      inout DataSeq data_values,
//      inout SampleInfoSeq sample_infos,
//      in long max_samples,
//      in InstanceHandle_t previous_handle,
```

```
//      in ReadCondition a_condition);

//  ReturnCode_t return_loan(
//      inout DataSeq data_values,
//      inout SampleInfoSeq sample_infos);

//  ReturnCode_t get_key_value(
//      inout Data key_holder,
//      in InstanceHandle_t handle);

//  InstanceHandle_t lookup_instance(
//      in Data instance_data);

    ReadCondition create_readcondition(
        in SampleStateMask sample_states,
        in ViewStateMask view_states,
        in InstanceStateMask instance_states);

    QueryCondition create_querycondition(
        in SampleStateMask sample_states,
        in ViewStateMask view_states,
        in InstanceStateMask instance_states,
        in string query_expression,
        in StringSeq query_parameters);

    ReturnCode_t delete_readcondition(
        in ReadCondition a_condition);

    ReturnCode_t delete_contained_entities();

    ReturnCode_t set_qos(
        in DataReaderQos qos);
    ReturnCode_t get_qos(
        inout DataReaderQos qos);

    ReturnCode_t set_listener(
        in DataReaderListener a_listener,
        in StatusMask mask);
    DataReaderListener get_listener();

    TopicDescription get_topicdescription();
    Subscriber get_subscriber();

    ReturnCode_t get_sample_rejected_status(
        inout SampleRejectedStatus status);
    ReturnCode_t get_liveliness_changed_status(
        inout LivelinessChangedStatus status);
    ReturnCode_t get_requested_deadline_missed_status(
        inout RequestedDeadlineMissedStatus status);
    ReturnCode_t get_requested_incompatible_qos_status(
        inout RequestedIncompatibleQosStatus status);
    ReturnCode_t get_subscription_matched_status(
        inout SubscriptionMatchedStatus status);
    ReturnCode_t get_sample_lost_status(
        inout SampleLostStatus status);
```

```
        ReturnCode_t wait_for_historical_data(
            in Duration_t max_wait);

        ReturnCode_t get_matched_publications(
            inout InstanceHandleSeq publication_handles);
        ReturnCode_t get_matched_publication_data(
            inout PublicationBuiltinTopicData publication_data,
            in InstanceHandle_t publication_handle);
    };


    struct SampleInfo {
        SampleStateKind      sample_state;
        ViewStateKind        view_state;
        InstanceStateKind    instance_state;
        Time_t               source_timestamp;
        InstanceHandle_t     instance_handle;
        InstanceHandle_t     publication_handle;
        long                 disposed_generation_count;
        long                 no_writers_generation_count;
        long                 sample_rank;
        long                 generation_rank;
        long                 absolute_generation_rank;
        boolean              valid_data;
    };

    typedef sequence<SampleInfo> SampleInfoSeq;
};

// Implied IDL for type "Foo"
// Example user defined structure
struct Foo {
    long dummy;
};

typedef sequence<Foo> FooSeq;

#include "dds_dcps.idl"

interface FooTypeSupport : DDS::TypeSupport {
    DDS::ReturnCode_t register_type(
        in DDS::DomainParticipant participant,
        in string type_name);
    string get_type_name();
};

interface FooDataWriter : DDS::DataWriter {

    DDS::InstanceHandle_t register_instance(
        in Foo instance_data);
    DDS::InstanceHandle_t register_instance_w_timestamp(
        in Foo instance_data,
        in DDS::Time_t source_timestamp);
```

```
        DDS::ReturnCode_t unregister_instance(
            in Foo instance_data,
            in DDS::InstanceHandle_t  handle);
        DDS::ReturnCode_t unregister_instance_w_timestamp(
            in Foo instance_data,
            in DDS::InstanceHandle_t  handle,
            in DDS::Time_t source_timestamp);

        DDS::ReturnCode_t write(
            in Foo instance_data,
            in DDS::InstanceHandle_t handle);

        DDS::ReturnCode_t write_w_timestamp(
            in Foo instance_data,
            in DDS::InstanceHandle_t handle,
            in DDS::Time_t source_timestamp);

        DDS::ReturnCode_t dispose(
            in Foo instance_data,
            in DDS::InstanceHandle_t instance_handle);

        DDS::ReturnCode_t dispose_w_timestamp(
            in Foo instance_data,
            in DDS::InstanceHandle_t instance_handle,
            in DDS::Time_t source_timestamp);

        DDS::ReturnCode_t get_key_value(
            inout Foo key_holder,
            in DDS::InstanceHandle_t handle);

        DDS::InstanceHandle_t lookup_instance(
            in Foo key_holder);
    };

    interface FooDataReader : DDS::DataReader {
        DDS::ReturnCode_t read(
            inout FooSeq data_values,
            inout DDS::SampleInfoSeq sample_infos,
            in long max_samples,
            in  DDS::SampleStateMask sample_states,
            in  DDS::ViewStateMask view_states,
            in  DDS::InstanceStateMask instance_states);

        DDS::ReturnCode_t take(
            inout FooSeq data_values,
            inout DDS::SampleInfoSeq sample_infos,
            in long max_samples,
            in  DDS::SampleStateMask sample_states,
            in  DDS::ViewStateMask view_states,
            in  DDS::InstanceStateMask instance_states);

        DDS::ReturnCode_t read_w_condition(
            inout FooSeq data_values,
            inout DDS::SampleInfoSeq sample_infos,
```

```
    in long max_samples,
    in  DDS::ReadCondition a_condition);

DDS::ReturnCode_t take_w_condition(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq sample_infos,
    in long max_samples,
    in  DDS::ReadCondition a_condition);

DDS::ReturnCode_t read_next_sample(
    inout Foo data_value,
    inout DDS::SampleInfo sample_info);

DDS::ReturnCode_t take_next_sample(
    inout Foo data_value,
    inout DDS::SampleInfo sample_info);


DDS::ReturnCode_t read_instance(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq sample_infos,
    in long max_samples,
    in DDS::InstanceHandle_t a_handle,
    in DDS::SampleStateMask sample_states,
    in DDS::ViewStateMask view_states,
    in DDS::InstanceStateMask instance_states);

DDS::ReturnCode_t take_instance(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq sample_infos,
    in long max_samples,
    in DDS::InstanceHandle_t a_handle,
    in DDS::SampleStateMask sample_states,
    in DDS::ViewStateMask view_states,
    in DDS::InstanceStateMask instance_states);

DDS::ReturnCode_t read_next_instance(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq sample_infos,
    in long max_samples,
    in DDS::InstanceHandle_t previous_handle,
    in DDS::SampleStateMask sample_states,
    in DDS::ViewStateMask view_states,
    in DDS::InstanceStateMask instance_states);

DDS::ReturnCode_t take_next_instance(
    inout FooSeq data_values,
    inout DDS::SampleInfoSeq sample_infos,
    in long max_samples,
    in DDS::InstanceHandle_t previous_handle,
    in DDS::SampleStateMask sample_states,
    in DDS::ViewStateMask view_states,
    in DDS::InstanceStateMask instance_states);
```

Compliance Points A

This specification includes the following compliance profiles.

- *Minimum profile*: This profile contains just the mandatory features of ~~the~~ DCPS~~layer~~. None of the optional features are included.
- *Content-subscription profile*: This profile adds the optional classes: ***ContentFilteredTopic***, ***QueryCondition***, ***MultiTopic.*** This profile enables subscriptions by content. See Section 2.1.2.3, "Topic-Definition Module," on page 2-38.
- *Persistence profile*: This profile adds the optional Qos policy DURABILITY_SERVICE as well as the optional settings 'TRANSIENT' and 'PERSISTENT' of the DURABILITY QoS policy ***kind***. This profile enables saving data into either TRANSIENT memory, or permanent storage so that it can survive the lifecycle of the ***DataWriter*** and system outings. See Section 2.1.3.4, "DURABILITY," on page 2-114.
- *Ownership profile*: This profile adds two things First the optional setting 'EXCLUSIVE' of the OWNERSHIP ***kind***. Second support for the optional OWNERSHIP_STRENGTH policy. Third the ability to set a ***depth*** > 1 for the HISTORY QoS policy.
- *Group access*~~*Object model*~~ *profile*: This profile includes ~~the DLRL and also includes~~ support for the PRESENTATION ***access_scope*** setting of 'GROUP' (Section 2.1.3.6, "PRESENTATION," on page 2-115).

Syntax for ~~DCPS~~ Queries and Filters

B

A subset of SQL syntax is used in several parts of the specification:

- The ***filter_expression*** in the ***ContentFilteredTopic*** (see "ContentFilteredTopicClass" on page 2-29).
- The ***topic_expression*** in the ***MultiTopic*** (see "MultiTopic Class [optional]" on page 2-29).
- The ***query_expression*** in the ***QueryReadCondition*** (see "QueryCondition Class" on page 2-60).

Those expressions may use a subset of SQL, extended with the possibility to use program variables in the SQL expression. The allowed SQL expressions are defined with the BNF-grammar below.

The following notational conventions are made:

- *NonTerminals* are typeset in italics.
- `'Terminals'` are quoted and typeset in a fixed width font.
- TOKENS are typeset in small caps.
- The notation *(element // ',')* represents a non-empty comma-separated list of *elements*.

SQL grammar in BNF

```
Expression    ::= FilterExpression
              |   TopicExpression
              |   QueryExpression
              .
FilterExpression ::= Condition
TopicExpression  ::= SelectFrom {Where } ';'
QueryExpression  ::= {Condition}{'ORDER BY' (FIELDNAME // ',') }
                     .
```

```
SelectFrom        ::= 'SELECT' Aggregation 'FROM' Selection
                  .
Aggregation       ::= '*'
                  |   (SubjectFieldSpec // ',')
                  .
SubjectFieldSpec  ::= FIELDNAME
                  |   FIELDNAME 'AS' FIELDNAME
                  |   FIELDNAME FIELDNAME
                  .
Selection     ::= TOPICNAME
              |   TOPICTNAME NaturalJoin JoinItem
              .
JoinItem      ::= TOPICNAME
              |   TOPICNAME NaturalJoin JoinItem
              |   '(' TOPICNAME NaturalJoin JoinItem ')'
              .
NaturalJoin       ::= 'INNER NATURAL JOIN'
              |    'NATURAL JOIN'
              |   'NATURAL INNER JOIN'
              .
Where         ::= 'WHERE' Condition
              .
Condition     ::= Predicate
              |   Condition 'AND' Condition
              |   Condition 'OR' Condition
              |   'NOT' Condition
              |   '(' Condition ')'
              .
Predicate     ::= ComparisonPredicate
              |   BetweenPredicate
              .
ComparisonPredicate::= FIELDNAME RelOp Parameter
                  |   Parameter RelOp FIELDNAME
                  |   FIELDNAME RelOp FIELDNAME
                  .
BetweenPredicate     ::=  FIELDNAME 'BETWEEN' Range
                  |   FIELDNAME 'NOT BETWEEN' Range
                  .
RelOp         ::= '=' | '>' | '>=' | '<' | '<=' | '<>' | like
              .
Range         ::= Parameter 'AND' Parameter
              .
Parameter     ::= INTEGERVALUE
              |   CHARVALUE
              |   FLOATVALUE
              |   STRING
              |   ENUMERATEDVALUE
              |   PARAMETER
              .
```

Note – INNER NATURAL JOIN, NATURAL JOIN, and NATURAL INNER JOIN are all aliases, in the sense that they have the same semantics. They are all supported because they all are part of the SQL standard.

Token expression

The syntax and meaning of the tokens used in the SQL grammar is described as follows:

- FIELDNAME - A fieldname is a reference to a field in the data-structure. The dot `'.'` is used to navigate through nested structures. The number of dots that may be used in a FIELD-NAME is unlimited. The FIELDNAME can refer to fields at any depth in the data structure. The names of the field are those specified in the IDL definition of the corresponding structure, which may or may not match the field-names that appear on the language-specific (e.g., C/C++, Java) mapping of the structure.
- TOPICNAME - A topic name is an identifier for a topic, and is defined as any series of characters `'a'`, ..., `'z'`, `'A'`, ..., `'Z'`, `'0'`, ..., `'9'`, `'-'` but may not start with a digit.
- INTEGERVALUE - Any series of digits, optionally preceded by a plus or minus sign, representing a decimal integer value within the range of the system. A hexadecimal number is preceded by `0x` and must be a valid hexadecimal expression.
- CHARVALUE - A single character enclosed between single quotes.
- FLOATVALUE - Any series of digits, optionally preceded by a plus or minus sign and optionally including a floating point (`'.'`). A power-of-ten expression may be postfixed, which has the syntax e*n*, where *n* is a number, optionally preceded by a plus or minus sign.
- STRING - Any series of characters encapsulated in single quotes, except a new-line character or a right quote. A string starts with a left or right quote, but ends with a right quote.
- ENUMERATEDVALUE - An enumerated value is a reference to a value declared within an enumeration. Enumerated values consist of the name of the enumeration label enclosed in single quotes. The name used for the enumeration label must correspond to the label names specified in the IDL definition of the enumeration.
- PARAMETER - A parameter is of the form %*n*, where *n* represents a natural number (zero included) smaller than 100. It refers to the *n* + 1 th argument in the given context.

Examples

Assuming Topic “Location” has as an associated type a structure with fields “flight_name, x, y, z”, and Topic “FlightPlan” has as fields “flight_id, source, destination”. The following are examples of using these expressions.

Example of a ***topic_expression***:

- “SELECT flight_name, x, y, z AS height FROM ‘Location’ NATURAL JOIN ‘FlightPlan’ WHERE height < 1000 AND x <23”

Example of a ***query_expression*** or a ***filter_expression***:

- “height < 1000 AND x <23”

Index

Index