eProsima RTPS User Manual

DOCUMENT INFORMATION	
Name	eProsima – RTPS User Manual
Product	Documentation
Author	Gonzalo Rodríguez Canosa
Distribution	R 0.5.1
Document Version	1.2
Date	27-Julio-2014
Status	Release
Contact	gonzalorodriguez@eprosima.com
Document Code	

eProsima
Proyectos y Sistemas de Mantenimiento SL
Ronda del poniente 2 – 1ºG
28760 Tres Cantos Madrid
Tel: + 34 91 804 34 48
info@eProsima.com – www.eProsima.com

Technical Support

- Phone: +34 91 804 34 48
- Email: Support@eProsima.com

Revision history

Date	Author	Description
16-May-2014	Gonzalo Rodriguez (gonzalorodriguez@eprosima.com)	Version 1: First version.
10-Jun-2014	Gonzalo Rodríguez	API changes added to Manual.
27-Jul-2014	Gonzalo Rodríguez	API changes and new Qos added to the manual.

Contents

Figure List

Table List

1 Introduction.

1.1 Purpose.

Real-Time Publish-Subscribe (RTPS) is the wire interoperability protocol defined for the Data-Distribution Service (DDS) standard by the Object Management Group (OMG) consortium. This protocol standard is also defined by the OMG in the specification document "The Real-Time Publish-Subscribe Wire Protocol. DDS Interoperability Wire Protocol Specification (DDS-RTPS)" that can be obtained from the OMG web-page. The main objective of this specification is that all participants in a DDS communication structure, even if they use different vendor's implementations, can interoperate.

However, most existing RTPS implementations are either included in full DDS implementations (for example, RTI-DDS, OpenDDS,...) or use past versions of the specification (ORTE). For this reason, a standalone open access RTPS implementation will likely be welcome as a useful tool for developers worldwide.

1.2 Business Context.

eProsima is a SME company focused on networking middleware with special attention to the OMG standard called Data Distribution Service for Real-time systems (DDS). eProsima develops new features and plug-ins for DDS, interoperability tools, and personalized networking middleware solutions for its customers.

Since eProsima is mainly focused on networking middleware and systematically uses OMG standards for its developed products, having its own RTPS implementation will benefit all their products and clients. The RTPS protocol is an OMG standard and every implementation must be interoperable with all the other existing implementations. Even so, having its own implementation allows eProsima to extend the protocol behavior to suit its own applications; always maintaining the interoperability.

Additionally, eProsima RTPS implementation is going to be published as a standalone opensource library in order to allow the community of DDS-RTPS users to benefit from its creation. This creates a two-way synergy between the DDS community and eProsima.

1.3 Scope.

The scope of this implementation is going to be limited by the RTPS protocol specification of the OMG. Since the main purpose of this design is to facilitate the implementation of a standalone RTPS wire protocol as close to the specification as possible, the included features will be the ones described in the OMG document. The OMG specification defines message formats, interpretation and usage scenarios for any application willing to use RTPS protocol

The most important features are:

- Enable best-effort and reliable publish-subscribe communications for real-time applications.
- Plug and play connectivity so that any new applications are automatically discovered by any other members of the network.

- Modularity and scalability to allow continuous growth with complex and simple devices in the network.

A complete reading of this document should be enough for a developer to convert this design into software. This implementation will be coded using C++ as the programming language. Any examples included in this document will be written in this language.

1.4 Intended Audience.

The intended audience for this document are the eProsima software engineers who will finally convert this implementation design to an open source library. Additionally, any software engineer willing to use, modify and/or extend the described implementation will benefit of reading this document. As explained above, this implementation will be carried out using C++, so the reader should have a basic understanding of this language. Some knowledge of DDS and RTPS will also be useful.

1.5 Reference Material.

The reader should look into the OMG's RTPS specification document that can be found in the following web-page: http://www.omg.org/spec/DDS-RTPS/2.1/.

Additionally the user should consult the eProsima RTPS – Manual document where use cases and examples are included; as well as the Public – API doxygen documentation generated along the project.

1.6 Document organization

This document can be divided into two main parts. The first part (composed of Chapters 2 and 3) contains a general view of the implementation specification. The second part describes the most important classes and methods (Chapter 5) and additional observations that need to be taken into account .

A brief summary of each chapter is presented in the following points:

- Chapter 2: This chapter contains an overview of the implementation specification; as well as an introduction to the implementation scope and limitations.
- Chapter 3: This chapter presents the implementation design. The four relevant modules are presented and explained.
- Chapter 5: This chapter includes a detailed description of the public API.
- Chapter 6: This chapter presents some examples of the most important use cases.

2 Implementation overview: scope and limitations

There are multiple DDS and RTPS bundle implementations (e.g., RTI DDS[1], OpenDDS[2], OpenSpliceDDS[3], …) and some standalone RTPS implementations (e.g., ORTE[4]). However, this last group is somewhat reduced and the implementations they provided are antiquated (e.g., ORTE implements RTPS protocol 1.2 with a C and Java API). For this reason a new up-to-date accessible RTPS protocol implementation is being designed and released by eProsima.

2.1 Implementation scope

The eProsima RTPS implementation will follow the RTPS specification document as close as possible. This entails that, where possible, the same classes, attributes and methods names will be used and implemented.

eProsima RTPS (eRTPS) is going to be released as a standalone library with an API directly accessible by DDS applications.

2.2 Implementation limitations

Ref	Issue	Action
1.	Not all QoS are supported in the implementation	Correctly describe which Qos are natively supported and which one need to be implemented outside the library.

2.3 Implementation Dependencies

List the main dependencies regarding the design effort.

Ref	Dependency	Action
1.	Boost libraries (1.53)	Boost asio will be used to manage sockets and communications; as well as other less important boost libraries (PropertyTree to read XML files, for example).

1 https://www.rti.com/products/dds/

2 http://www.opendds.org/

3 http://www.prismtech.com/opensplice

4 http://orte.sourceforge.net/

3 Implementation Specification

The eProsima implementation of the RTPS wire protocol will provide the users with the means to easily exchange RTPS messages in a DDS network. This exchange will be managed through APIs. A high-level API will be provided to control the creation and access of a Participant in the DDS network, and two user friendly APIs to publish and subscribe. These APIs together with the other modules and dependencies will be explained in the following sections. A general view of the system architecture is also presented.

3.1 System Architecture

A general view of the eProsima RTPS library together with other components of a DDS network is shown in Figure 3.1. This diagram schematically shows how the user will be able to create and manage a participant on the network, as well as send and receive message through the Publisher and Subscriber APIs. This process will be internally managed by the Writer and Reader endpoints. The User APIs, although included in the library, are not part of the RTPS implementation. These are created to facilitate the testing of the system. To achieve full DDS functionality the user should interact with the Writer and Reader through a DDS application.

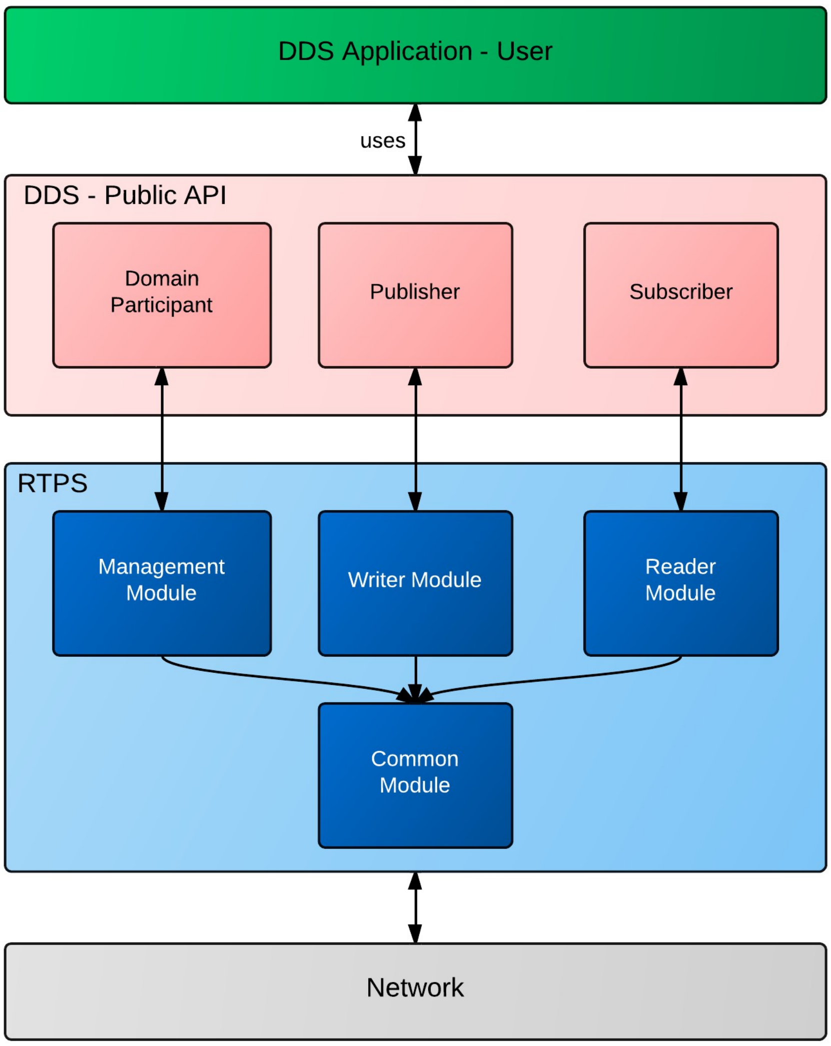

Figure 3.1: System architecture

The different modules included in Figure 3.1 and their interaction with the user and between them will be explained in the next sections.

3.1.1 Modules

There are four different modules in the designed implementation. This four modules are explained below:

- **Management Module:** This module contains the classes regarding the management of communications and events (Resource classes), as well as the Discovery Module and Liveliness Module.
- **Writer Module:** This module manages all elements related to the writer part of the communication structure, including the definition of the different types of writer and the access to the History of transmitted data.
- **Reader Module:** This module manages the subscription end of the RTPS protocol. Its a mirrored module of the Writer Module, since it manages the information reception of the data received from the publisher.
- **Common Module:** All common data structures and data types are included here. A structure type is considered common if it is needed by two or more modules. The *HistoryCache* and the RTPS *Messages* definitions are one of the most important of this common structures. More detail on these structures and others will be provided in following sections.
- **DDS – Public API Module:** This module contains the Public API available to the users to control their applications, creating Participants, Publishers and Subscribers.

3.1.2 Interaction

There are two different types on interaction defined in this architecture. On one side the interaction with the user of the protocol and on the other, the interaction between the modules of the library. These two types of interaction will now be detailed:

- **User Interaction:** The user of this library will only interact with the Public API module of this library. Using the APIs the user will be able to create a Participant and add Publisher and/or Subscriber. When this entities have been created the user will be able to send data as RTPS Messages, assign Callback methods and read or take information from a Subscriber.
- **Modules Interaction:** Both the *Writer* and the *Reader* include in their classes an instance of *HistoryCache* where they maintain information about the changes in the data-objects. Depending on the user interaction with the corresponding methods of the API, the *Writer* and the *Reader* will add, get or remove *CacheChanges* from theirs corresponding history.

3.2 Behavior Implementation

The correct behavior of the RTPS protocol will be achieved by using an event-based implementation. Multiple events will be handled by different threads allowing the application to efficiently tackle the different tasks. The thread structure, as well as the main events will be discussed in this section.

3.2.1 Thread structure

For each RTPS participant, different threads will be created that will manage different aspects of the RTPS implementation. Each application using this implementation will have at least these threads:

- **Main thread:** This thread will be the thread managed by the dds application or the user. The user will interact with the objects through the defined public API's.

- **Event thread(s):** This thread will be in charge of processing the different events triggered wither by some periodic condition or by actions performed by the user in the main thread. In this version a single event thread will be created and used per Participant.

- **Receive threads:** These threads will be in charge of listening to the different ports. Since these threads will be blocked until a RTPS message is received there would be a different thread for each different IP-port combination that the Participant is listening from. Multiple endpoints can be assigned to the same listen thread.

3.2.2 Resource structure

There are three main types of resources in the implementation, that directly correspond with three classes. All resources are managed by the Participant.

- **ResourceListen**: Each listen resource is assigned to a single IP:port combination. It receives and process messages directed to that socket and performs the necessary actions in one or more of the associated Writers or Readers. In this version each ResourceListen runs a single thread listening to a single socket. Future versions may alter this behaviour assigning multiple sockets (multiple ResourceListen objects) to a single listen thread.

- **ResourceEvent**: This resource manages the time-delayed event triggered periodically or by some message-based event. A single resource is implemented per participant, with a single thread performing all the actions. Future versions may include multiple ResourceEvents running in multiple threads to improve performance.

- **ResourceSend**: This resource manages ALL send operations in the Participant. This means that all endpoints included in a Participant send their messages through the socket defined in this resource. All messages are sent synchronously. Future versions will include multiple ResourceSend objects and the possibility to asynchronously send messages.

3.2.3 Main events

There are multiple events that are triggered wither directly by some action performed by the user, the reception of messages or even periodically. A list of the main events and the actions that need to be performed after them is included below, whereas a detailed description of all the events associated with each class of the design will be included in the detailed implementation chapter.

- **User-triggered events:** These events are triggered directly after the user performs some action, either directly to the RTPS Writer or its associated HistoryCache. These events are usually executed synchronously, directly in the Main thread, thus not using the ResourceEvent class.

- **Message-triggered events:** These events are triggered by the reception of an RTPS message. For example, the reception of an ACKNACK message would trigger a change in the status of some CacheChanges in the HistoryCache and, maybe, the re-send of some packets to a specific Reader.
- **Periodic events:** Some events must be periodically triggered according to DDS rules. For example, heartbeat packages must be sent each heartbeatPeriod to all matching Readers.

4 Object Specification

Object oriented design is the process of planning a system of interacting objects for the purpose of solving a software problem. It is one possible approach to software design. An object contains encapsulated data and procedures grouped together to represent an entity. The “object interface”, how the object can be interacted with, is also defined. An object-oriented program is described by the interaction of these objects.

This programming paradigm has been selected for this implementation since this library will use entities to represent each of the models integrating the protocol. The object oriented design can be specified in many ways. One of them is by using UML (Unified Modeling Language). UML is a standardized, general-purpose modeling language in the field of software engineering developed by Grady Booch, Ivar Jacobson and James Rumbaugh at Rational Software in the 1990s. It was adopted by the OMG in 1997. The Unified Modeling Language includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems.

A complete view of all the objects included in this implementation is shown in Figure 4.1. For the sake of simplicity the attributes and methods of most of the classes have been omitted, although they will be explained in Chapter 5. Also, all common data types will be detailed and explained in Chapter 5.

After the UML diagram a table defining and explaining the most important object types is included (Table 1).

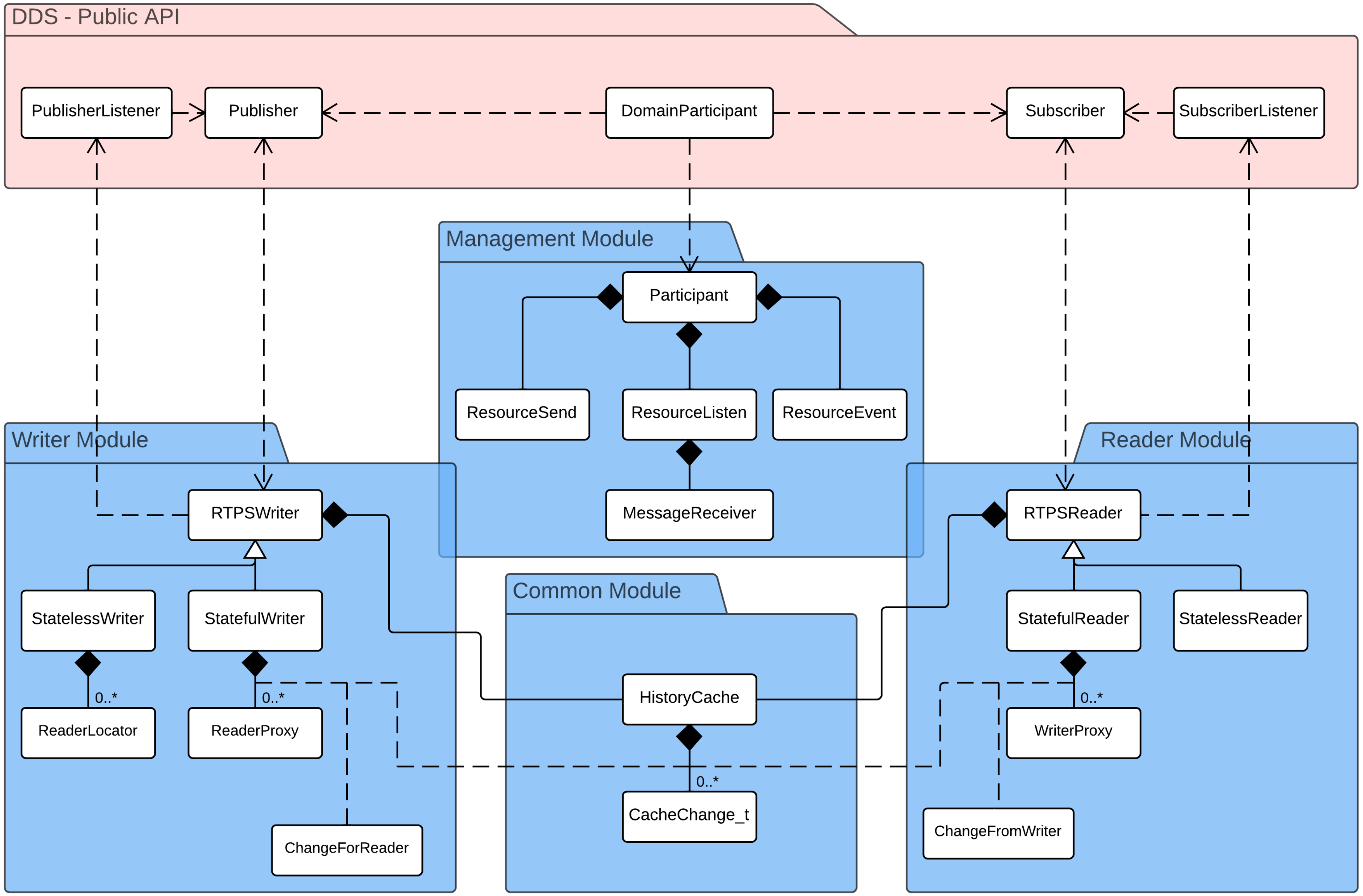

Figure 4.1: Object view of the architecture, only the most important classes are included.

The objects included in the conceptual design of this implementation are listed and explained in Table 1.

Object Name	Object Type	Comment
DDS – Public API Module		
DomainParticipant	Class	This class allows the user to control the creation and removal of publishers and subscribers. It does not belong to the RTPS implementation but is included to manage the protocol.
ParticipantAttributes	Structure	This structure contains all the necessary parameters to create a Participant.
DDSTopicDataType	Class	This class is used to provide the DomainParticipant with the methods to serialize, deserialize and get the key of a specific data type.
Publisher	Class	This class allows the user to send new information to the subscribers. Additionally, the class allows the user to add and remove Reader destinations to where the messages need to be send.
PublisherAttributes	Structure	This structure contains all the necessary parameters that defined the behavior of a Publisher. It must be provided during construction.
PublisherListener	Class	Base class the user should use to implement specific callbacks to certain actions on the Publisher side.
Subscriber	Class	This class allows the user to retrieve information from the reader history. Additionally, the interface allows the user to add and remove Writer origins from where the messages are going to be sent if no Discovery Protocol is used.
SubscriberAttributes	Structure	This structure contains all the necessary parameters that define the behavior of a Subscriber. It must be provided during construction.
SubscriberListener	Class	Base class the user should use to implement specific callbacks to certain actions in the Subscriber side.
Management Module		
Participant	Class	RTPS Class Public interface that controls the management operations of the RTPS protocol. The user should not use this class directly, but instead use the static methods of the DomainParticipant class.
ResourceEvent	Class	Class used to manage the temporal events.
ResourceListen	Class	Class used to listen to a specific socket for RTPS messages.
ResourceSend	Class	Class used to manage the send operation.
TimedEvent	Class	Base class used to specify each of the temporal events.
PDPSimple	Class	This class announces the participant to predefined Locators in the network as well as to all other Participants that have been discovered.
EDPStatic	Class	This class read and process the XML file where the static Endpoint are described as well as produce endpoint matching.

Object Name	Object Type	Comment
EDPSimple	Class	This class is implements the Simple Endpoint Discovery Protocol.
Writer Module		
RTPSWriter	Class	This class directly implements the *Publisher* API. It has its own *HistoryCache* and depending on the writer type maintains a list of *ReaderLocators* or *ReaderProxys* that contain information regarding the related *Readers*.
StatelessWriter	Class	This interface controls the *ReaderLocators* in Writer in case the user defined the Publisher as BEST_EFFORT. This implementation is optimized for scalability. Is ideally suited for best-effort communication, since it does not keep state information about remote entities.
ReaderLocator	Class	This type is used by the *StatelessWriter* to keep track of the location of its matching remote *Readers* and manage the changes in HistoryCache as they relate to that particular Reader.
StatefulWriter	Class	This interface controls the *ReaderProxys* in Writer in case the user defined the Publisher as RELIABLE. It can guarantee reliable communication and is able to apply Qos-based or content-based filtering on the Writer side. This reference implementation minimized bandwith usage.
ReaderProxy	Class	This type represents the information the *Writer* maintains on each matched *RTPSReader* and manages the changes in *HistoryCache* as they relate to that particular Reader.
ChangeForReader	Structure	This object maintains information of a *CacheChange* in the *Writer HistoryCache* as it pertains to the *Reader* represented by the *ReaderProxy*.
Reader Module		
RTPSReader	Class	This class directly implements the *Subscriber*, either directly or by some of its elements. It has its own *HistoryCache* and depending on the reader type maintains a list *WriterProxys* that contain information regarding the related writers.
StatefulReader	Class	This interface controls the publisher that are matched with this Reader maintaining a state on each matched Writer. This state is encapsulated in the WriterProxy object.
WriterProxy	Class	This type represents the information the *StatefulReader* maintains on each matched writer.
ChangeFromWriter	Structure	This structure maintains information of a *CacheChange* in the *Reader HistoryCache* as it pertains to the *RTPSWriter* represented by the *WriterProxy*.
Common Module		
HistoryCache	Class	This class implements the required behavior for the chronology of

Object Name	Object Type	Comment
		CacheChanges both the *Writer* and the *Reader* must maintain. Is composed by a collection of *CacheChanges*.
CacheChange	Structure	This structure is used to represent each change added to the *HistoryCache.*
Data	Data	This represents the data values associated with a particular change. Some changes may not have any associated data.
Message	Structure	Structure defining a RTPS Message. Is composed by a *Header* and one or multiple *Submessages.*
Header	Structure	This structure must appear at the beginning of each RTPS Message. It identifies the message as belonging to the RTPS protocol. It identifies the protocol version and the vendor that send the messages.
SubmessageHeader	Structure	This structure identifies the kind of *Submessage* and the optional elements within that *Submessage*.
MessageReceiver	Class	This object type interprets each of the received RTPS Messages. Since the meaning of each *Submessage* may depend on the previous *Submessages* a state of the whole *Message* must be maintained. This object type is reseted each time a new *Message* is received.
RTPSMessageCreator	Class	This class allows the generation of serialized CDR RTPS Messages.
External Dependencies		
Boost ASIO	API	This API allows the control of the asynchronous events.
Boost Signals2	API	This API allows to add multiple tasks to a single event.

Table 1: Objects included in the implementation design

4.1 Data Design

This section describes the data design; i.e., the structures the software defines and uses. The data structures this library is going to use can be divided in three different groups:

- Internal Data Structures: Data exchanged among implementation modules.
- Global Data Structures: Data available to major portions of the architecture.
- Temporary Data Structures: Data created for temporal use.

These three groups are described in the following sections.

4.1.1 Internal Data Structures

The Internal Data Structures are passed among the architecture modules. In case of this implementation the internal data structures are the *CacheChanges* stored in each module *HistoryCache*. Both the *RTPSWriter* and the *RTPSReader* use this type of structure to modify its own *HistoryCache* and to create RTPS *Messages*.

4.1.2 Global Data Structures

The Global Data Structures are data that need to be known by major portions of the architecture.; i.e., data shared between the modules and with the end user. For our implementation these classes are: *ReaderLocator*, *ReaderProxy* and *WriterProxy*.

Another commonly shared global structure is the eProsima FastCDR library, since both the RTPSReader and RTPSWriter will need it to serialize data and compose messages.

4.1.3 Temporary Data Structures

For this implementation the most important Temporary Data Structures are the RTPS Messages. These structures are usually created from *CacheChanges* and eliminated immediately after they are sent.

5 Public API Specification

This chapter includes a detailed implementation specification where each of the objects is thoroughly discussed and explained. For each of these objects an UML diagram and a table explaining the most important aspects of the class are included. For a detailed description of each method please consult the doxygen documentation.

5.1 Common Data Types

The types of attributes that compose each of the classes of this implementation are summarized in Table 2. A detailed explanation of their purpose can be found in Table 8.2 of the OMG RTPS Specification document (OMG.RTPS.2.1 from now on).

Attribute type	**Attribute Implementation**	**OMG.RTPS.2.1 Reference**	
		Purpose	**Mapping and Reserved values**

GUID_t	typedef struct{
	GuidPrefix_t guidPrefix;
	EntityId_t entityId;
	} GUID_t
		Table 8.2	Page 153

GuidPrefix_t	typedef octet[12] GuidPrefix_t;	Table 8.2	Page 150

EntityId_t	typedef struct {
	octet[3] entityKey;
	octet entityKind;
	} EntityId_t;
		Table 8.2	Pages 150-152

Locator_t	typedef struct{
	long kind;
	unsigned long port;
	octet[16] address;
	} Locator_t
		Table 8.2	Page 155

SequenceNumber_t	typedef struct{
	long high;
	Unisgned long low
	} SequenceNumber_t
		Table 8.2	Page154

TopicKind_t	typedef struct{ long value; } TopicKind_t	Table 8.2	Page 155

ChangeKind_t	typedef unsigned long ChangeKind_t;
	const ChangeKind_t ALIVE = 0x0001 << 0;
	const ChangeKind_t NOT_ALIVE_DISPOSED = 0x0001 << 1;
	const ChangeKind_t NOT_ALIVE_UNREGISTERED = 0x0001 << 2;
		Table 8.2	Obtained from the DDS Specification Document.

ReliabilityKind_t	typedef struct{ long value; } ReliabilityKind_t	Table 8.2	Page 156

Attribute type	Attribute Implementation	OMG.RTPS.2.1 Reference	
		Purpose	Mapping and Reserved values
InstanceHandle_t	typedef HANDLE_TYPE_NATIVE InstanceHandle_t;	Table 8.2	-
ProtocolVersion_t	typedef struct{ octet major; octet minor; } ProtocolVersion_t	Table 8.2	Page 156
VendorId_t	typedef struct{ octet[2] vendorId; } VendorId_t	Table 8.2	Page 153
Time_t	typedef struct{ long seconds; unsigned long fraction; } Time_t	Table 8.13	Page 153
Count_t	typedef struct {long value;} Count_t	Table 8.13	Page 156
KeyHash_t	typedef struct {octet[16] value;} KeyHash_t	Page 190	Page 156
StatusInfo_t	typedef struct {octet[4] value;} StatusInfo_t	Page 191	Page 156
ParameterId_t	typedef struct{ short value;} ParameterId_t	Table 8.13	Page 156
Duration_t	typedef Time_t Duration_t;	Table 8.46	-
Delay_t	typedef Time_t Delay_t;	Table 8.46	-
ProtocolId_t	enum ProtocolId_t{ PROTOCOL_RTPS = 'RTPS'};	Table 8.13	
SubmessageFlag	typedef bool SubmessageFlag;	Table 8.13	
SubmessageKind	enum SubmessageKind{ ... };	Table 8.13	Page 168

Table 2: Class Attributes Implementation

5.2 DDS – Public API

5.2.1 DomainParticipant

This singleton class can be used to create Participants, Publishers and Subscribers and to register data types. An UML diagram of this class can be observed in Figure 5.1. Only the public methods are shown in the diagram. For a detailed

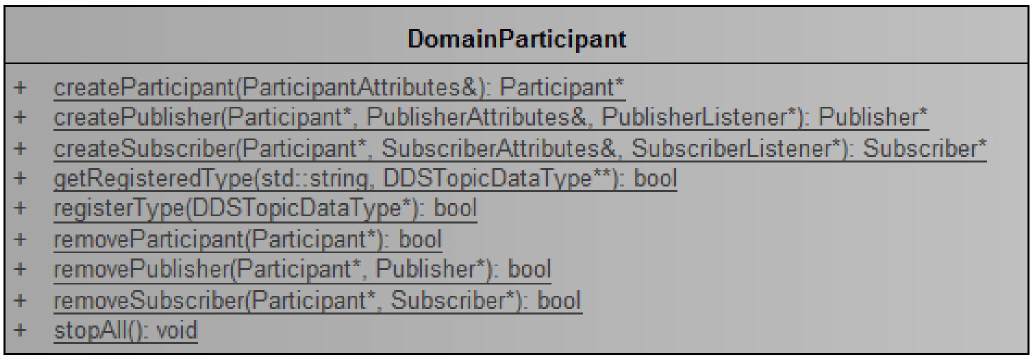
DomainParticipant
+ createParticipant(ParticipantAttributes&): Participant*
+ createPublisher(Participant*, PublisherAttributes&, PublisherListener*): Publisher*
+ createSubscriber(Participant*, SubscriberAttributes&, SubscriberListener*): Subscriber*
+ getRegisteredType(std::string, DDSTopicDataType**): bool
+ registerType(DDSTopicDataType*): bool
+ removeParticipant(Participant*): bool
+ removePublisher(Participant*, Publisher*): bool
+ removeSubscriber(Participant*, Subscriber*): bool
+ stopAll(): void

Figure 5.1: DomainParticipant UML class diagram

Aspect	Comment
Construction	This class cannot be directly constructed. The first time one of the static methods is used the singleton object is created.
Destruction	This class cannot be directly destructed. When the stopAll method is called the instance closes all Participants, Subscribers, Publishers and other created objects and removes the instance.
Attributes	This class stores all participants and all registered types, as well as the parameters used to calculate the default ports in the discovery protocols.
Interface	This class allows the creation and destruction of Participants, Publishers and Subscribers . It returns a pointer to the created element.
Thread safety	Not yet completely thread safe.
Events	-

Table 3: DomainParticipant description

5.2.2 DDSTopicDataType

Class used to serialize and deserialize the different topic data types.

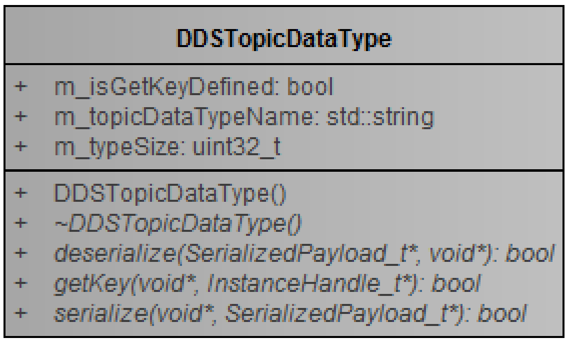
DDSTopicDataType
+ m_isGetKeyDefined: bool
+ m_topicDataTypeName: std::string
+ m_typeSize: uint32_t
+ DDSTopicDataType()
+ ~DDSTopicDataType()
+ deserialize(SerializedPayload_t*, void*): bool
+ getKey(void*, InstanceHandle_t*): bool
+ serialize(void*, SerializedPayload_t*): bool

Figure 5.2: DDSTopicDataType class diagram

Aspect	**Comment**
Construction	The user should use this as a base class to implement the methods needed by the DomainParticipant.
Destruction	-
Attributes	The topicaDataTypeName must be unique for each defined type.
Interface	The serialization and deserialization methods are must be implemented, whereas the getKey method depends on whether the topic is WITH_KEY.
Thread safety	Not safe. If two Publishers, Subscribers need to use the same object the user should define two separate instances with different names and give each endpoint one of them (by making their respective topics use different topicDataTypeNames, even if internally is the same type).
Events	-

Table 4: DDSTopicDataType description

5.2.3 Publisher

This class allows the user to control how the information is send. It is associated with a single RTPSWriter.

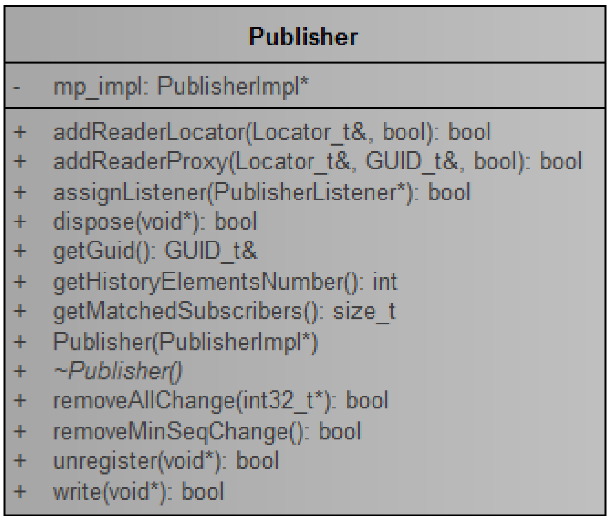
Publisher
- mp_impl: PublisherImpl*
+ addReaderLocator(Locator_t&, bool): bool
+ addReaderProxy(Locator_t&, GUID_t&, bool): bool
+ assignListener(PublisherListener*): bool
+ dispose(void*): bool
+ getGuid(): GUID_t&
+ getHistoryElementsNumber(): int
+ getMatchedSubscribers(): size_t
+ Publisher(PublisherImpl*)
+ ~Publisher()
+ removeAllChange(int32_t*): bool
+ removeMinSeqChange(): bool
+ unregister(void*): bool
+ write(void*): bool

Figure 5.3: Publisher UML class diagram

Aspect	**Comment**
Construction	The user should NOT construct this object directly but use the DomainParticipant to create it inside the designed Participant.
Destruction	The destruction should also be handled by the DomainParticipant.
Attributes	This class has an associated RTPSWriter and an associated DDSTopicDataType that are used to perform the write operations.
Interface	The main methods available to this class are associated with the write, dispose and un-register operations. The interface also allows to assign a Listener and to add Reader-Locators and ReaderProxies in case no discovery is used.
Thread safety	Not safe.
Events	Depending on the reliability type there are some events associated with this class although they will be explained in the RTPSWriter class.

Table 5: Publisher description

5.2.3.1 PublisherListener

An object of this class is assigned to each Publisher to define specific actions. The user must create a new class with PublisherListener as public base class and define the methods he wants to be called.

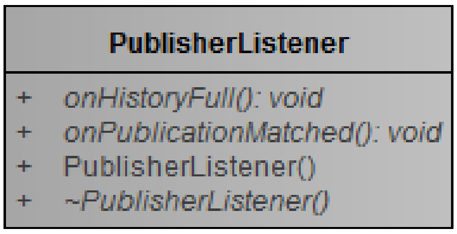

Figure 5.4: PublisherListener UML class diagram

5.2.4 Subscriber

This class allows the user to control the received information. Each instance its associated with its own RTPSReader.

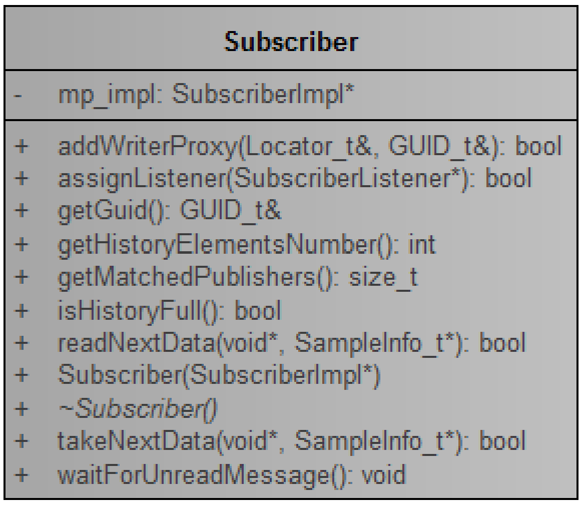
Subscriber
- mp_impl: SubscriberImpl*
+ addWriterProxy(Locator_t&, GUID_t&): bool
+ assignListener(SubscriberListener*): bool
+ getGuid(): GUID_t&
+ getHistoryElementsNumber(): int
+ getMatchedPublishers(): size_t
+ isHistoryFull(): bool
+ readNextData(void*, SampleInfo_t*): bool
+ Subscriber(SubscriberImpl*)
+ ~Subscriber()
+ takeNextData(void*, SampleInfo_t*): bool
+ waitForUnreadMessage(): void

Figure 5.5: Subscriber UML class diagram

Aspect	Comment
Construction	The user should NOT construct this object directly but use the DomainParticipant to create it inside the designed Participant.
Destruction	The destruction should also be handled by the DomainParticipant.
Attributes	This class has an associated RTPSReader and an associated DDSTopicDataType that are used to perform the read and take operations.
Interface	Three main methods are associated with this class: read, take and waitForUnread. The first two give the user information about samples received by the RTPSReader while the third one halts the execution until new messages are received.
Thread safety	Not safe.
Events	Depending on the reliability type there are some events associated with this class al-though they will be explained in the RTPSReader class.

Table 6: Subscriber description

5.2.4.1 SubscriberListener

An object of this class is assigned to each Subscriber to define specific actions. The user must create a new class with SubscriberListener as public base class and define the methods he wants to be called.

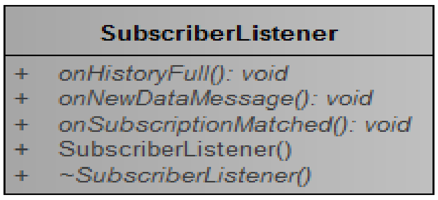

Figure 5.6: SubscriberListener UML class diagram

5.2.5 Attributes

When trying to create a new Participant, Publisher or Subscriber the corresponding Attributes class must be provided, specifying the parameters that define the element. Each of the Attributes are thoroughly explained in the doxygen documentation.

6 Use examples

This section will present real examples of how the pulic API should be used.

6.1 Type registration

To register a type a class that inherits from DDSTopicDataType must be defined, as shown in Figure 3.1. The user is responsible to correctly implement the serialize and deserialize methods (mandatory) and the getKey method.

```
typedef struct TestType{
    char name[6]; //KEY
    int32_t value;
    double price;
    TestType()
    {
        value = -1;
        price = 0;
        COPYSTR(name,"UNDEF");
    }
    void print()
    {
        cout << "Name: ";
        printf("%s",name);
        cout << " |Value: "<< value;
        cout << " |Price: "<< price;
        cout << endl;
    }
}TestType;
class TestTypeDataType:public DDSTopicDataType
{
public:
    TestTypeDataType()
{
        m_topicDataTypeName = "TestType";
        m_typeSize = sizeof(TestType);
        m_isGetKeyDefined = true;
};
    ~TestTypeDataType(){};
    bool serialize(void*data,SerializedPayload_t* payload);
    bool deserialize(SerializedPayload_t* payload,void * data);
    bool getKey(void*data,InstanceHandle_t* ihandle);
};
```

Figure 6.1: Data Type definition

An example implementation of these methods is included in Figure 6.2.

```
bool TestTypeDataType::serialize(void*data,SerializedPayload_t* payload){
    payload->length = 6+4+sizeof(double);
    payload->encapsulation = CDR_LE;
    if(payload->data !=NULL)
        free(payload->data);
    payload->data = (octet*)malloc(payload->length);
    memcpy(payload->data,data,payload->length);
    return true;
}
bool TestTypeDataType::deserialize(SerializedPayload_t* payload,void * data){
    //cout << "Deserializing length: " << payload->length << endl;
    memcpy(data,payload->data,payload->length);
    return true;
}
bool TestTypeDataType::getKey(void*data,InstanceHandle_t* handle){
    TestType* tp = (TestType*)data;
    handle->value[0]  = 0;
    handle->value[1]  = 0;
    handle->value[2]  = 0;
    handle->value[3]  = 5; //length of string in CDR BE
    handle->value[4]  = tp->name[0];
    handle->value[5]  = tp->name[1];
    handle->value[6]  = tp->name[2];
    handle->value[7]  = tp->name[3];
    handle->value[8]  = tp->name[4];
    for(uint8_t i=9;i<16;i++)
        handle->value[i]  = 0;
    return true;
}
```

Figure 6.2: Test Type method example

Finally, to actually register the type an instance of the defined class must be created and passed to the DomainParticipant through the provided interface, as shown in Figure 6.4.

```
int main()
{
...
TestTypeDataType TestTypeData;
DomainParticipant::registerType((DDSTopicDataType*)&TestTypeData);
...
};
```

Figure 6.3: Test Type registration

6.2 Participant creation

An example of the creation of a Participant can be observed in Figure 6.5.

```
ParticipantAttributes PParam;
PParam.name = "participant1";
PParam.defaultSendPort = 10042;
PParam.builtin.domainId = 50;
PParam.builtin.use_SIMPLE_ParticipantDiscoveryProtocol = true;
PParam.builtin.resendSPDPDataPeriod_sec = 30;
PParam.builtin.use_STATIC_EndpointDiscoveryProtocol = true;
PParam.builtin.m_staticEndpointXMLFilename = "StaticEndpointDefinition.xml";
Locator_t loc;
loc.kind = 1; loc.port = 10046; loc.set_IP4_address(192,168,1,16);
PParam.defaultUnicastLocatorList.push_back(loc);
Participant* p = DomainParticipant::createParticipant(PParam);
if(p!=NULL)
{
//Participant correctly created
}
```

Figure 6.4: Participant creation

There are some parameters that must be defined:

- Participant name: If not defined the creation will fail.
- DomainId: This parameter is used to calculate the discovery ports and it is important to separate different applications working in the same network.
- Discovery: If the static endpoint discovery protocol is used then the filename must be provided.
- Default locator list: These locators will be used when the user defined and endpoint with no locators.
- A maximum of 1000 Publishers and 1000 Subscribers can be created. Future versions will allow the user to determine this number per participant.

6.3 Publisher API

6.3.1 Publisher creation

In Figure 6.5 an example of the creation of a Publisher using the DomainParticipant interface is shown.

```
PublisherAttributes Wparam;
Wparam.userDefinedId = 1; //Only necessary if StaticEDP is used.
Wparam.historyMaxSize = 20;
Wparam.pushMode = true; //false only available with RELIABLE.
Wparam.topic.topicKind = WITH_KEY; //Other possible value: NO_KEY
Wparam.topic.topicDataType = "TestType";
Wparam.topic.topicName = "Test_topic";
Wparam.qos.m_reliability.kind = RELIABLE_RELIABILITY_QOS; //Other value: BEST_EFFORT
Wparam.times.heartbeatPeriod.seconds = 2;
Wparam.times.nackResponseDelay.seconds = 5;
Wparam.times.nackSupressionDuration.nanoseconds = 200*1000*1000;
Locator_t loc;
loc.kind = 1;loc.port = 10046; //If the IP is 0.0.0.0, all interfaces will be used.
Wparam.unicastLocatorList.push_back(loc); //If no locator is given the default is
used.
Publisher* pub = DomainParticipant::createPublisher(p,Wparam);
```

Figure 6.5: Publisher creation

The DomainParticipant performs a number of verifications before actually creating the Publisher:

- If the Participant was created using Static Endpoint Discovery Protocol, then the userDefinedId must be defined and greater than zero.
- The data type must have been registered before creating the Publisher.
- If the topic is defined as WITH_KEY, the registered data type must have its getKey method implemented.

6.3.2 PublisherListener

The PublisherListener is provided to a Publisher to perform certain actions when certain events occur. In Figure 6.6 an example of PublisherListener is shown.

```
class TestTypeListener: public PublisherListener
{
public:
    Participant* p;
    ParticipantAttributes Pparam;
    eprosima::dds::Publisher* pub;
    PublisherAttributes Pubparam;
    TestTypeListener()
    {
        //The participant should have been created and accessible to this method.
        p = DomainParticipant::createParticipant(Pparam);
        //PublisherAttributes must be set to the user preferences.
        pub = DomainParticipant::createPublisher(p,Pubparam,
                                                (PublisherListener*) this);

    };
    ~TestTypeListener(){};
    void onHistoryFull()
    {
        pub->removeMinSeqCache();
    }
    void onPublicationMatched(MatchingInfo info)
    {
        if(info.status == MATCHED_MATCHING)
            cout<< "New Subscriber found"<<endl;
        else if(info.status = REMOVED_MATCHING)
            cout << "Subscriber removed" <<endl;
    }
};
```

Figure 6.6: PublisherListener creation

In this example the creation of the Participant and the Publisher is perform in the constructor of the PublisherListener. This method provides the PublisherListener methods access to the Publisher pointer. However, this is not mandatory. The creation could be performed elsewhere and then the pointer to be given as a argument to the class. In this case, only one of the virtual methods defined for PublisherListener has been implemented.

It is important to avoid loops or blocking sentences in these methods since that would cause the execution threads to block.

6.3.3 Data writing and removal

The Publisher can perform three types of operations with a TestType object:

- Write: Sends an instance of the data type to all associated readers.
- Dispose: Dispose of the given object, only available in topics WITH_KEY.
- Unregister: Unregister a given object, only available in topics WITH_KEY.

It can also remove data from its own History, either one at a time (removeMinSeqCache) or all of them (removeAll(&n_removed)).

An example of this methods can be shown in Figure 6.7.

```
TestType tp;
COPYSTR(tp.name,"Obje1");
tp.value = 0;
tp.price = 1.3;
pub->write((void*)&tp);
pub->dispose((void*)&tp);
pub->unregister((void*)&tp);
pub->removeMinSeqChange();
int n_removed;
pub->removeAllChange(&n_removed);
```

Figure 6.7: Data writing and removal

6.4 Subscriber API

6.4.1 Subscriber creation

The Subscriber creation process is similar to the one presented for the Publisher creation. The verifications performed before the creation are also the ones presented in the previous sections.

```
SubscriberAttributes Rparam;
Rparam.userDefinedId = 2;
Rparam.historyMaxSize = 50;
Rparam.expectsInlineQos = true;
Rparam.topic.topicDataType = "TestType";
Rparam.topic.topicName = "Test_topic2";
Rparam.topic.topicKind = NO_KEY;
Rparam.qos.m_reliability.kind= RELIABLE_RELIABILITY_QOS;
Rparam.times.heartbeatResponseDelay.seconds = 1;
Rparam.times.heartbeatSupressionDuration.nanoseconds = 500*1000*1000;
Locator_t loc;
loc.kind = 1; loc.port = 10046;
Rparam.unicastLocatorList.push_back(loc);
Subscriber* sub = DomainParticipant::createSubscriber(p,Rparam);
```

Figure 6.8: Subscriber creation

6.4.2 SubscriberListener

The SubscriberListener also needs to be provided to each Subscriber. An example of SubscriberListener creation is shown in Figure 6.9.

```
class TestTypeListener: public SubscriberListener{
public:
    TestTypeListener(){};
    ~TestTypeListener(){};
    void onNewDataMessage() {
        cout <<"New Message"<<endl;
    }
    void onSubscriptionMatched(MatchingInfo info)
    {
        if(info.status == MATCHED_MATCHING)
            cout<< “New Publisher found”<<endl;
        else if(info.status = REMOVED_MATCHING)
            cout << “Publisher removed” <<endl;
    }
};

//somewhere else: (If the Listener is assigned at a later point (not in the creation)
it cannot be guaranteed that the method onSubscriptionMatched of the discovery will be
called.).
TestTypeListener listener;
sub->assignListener((SubscriberListener*)&listener);
```

Figure 6.9: SubscriberListener definition

In this case the SubscriberListener is instantiated and assigned to the Subscriber in a different part of the user program. As before, the user should avoid loops with no clear end condition since the execution of the receiving and/or event threads will be blocked until the user-defined operations are finished.

6.4.3 Subscriber data read and take.

The subscriber also has interfaces to read and take data from its own History. A method to block the execution until unread messages are available is also provided.

```
TestType tp;
SampleInfo_t info;
sub->waitForUnreadMessage();
sub->readNextData((void*)&tp,&info);
sub->takeNextData((void*)&tp,&info);
if(info.sampleKind == ALIVE)
     tp.print();
```

Figure 6.10: Subscriber Data reading

6.5 QoS Examples

In order to help with the implementation of additional Qos, an additional structure is provided each time a sample is read or taken from the history. This structure is of type SampleInfo_t and contains:

- sampleKind: Indicates whether the sample is ALIVE (a data structure of the type of the topic) or DISPOSED or UNREGISTERED.
- WriterGUID: GUID_t of the writer of the sample (useful to implement ownership Qos).
- OwnershipStrength: ownership strength of the writer when the sample was received.
- SourceTimestamp: timestamp indicating when the sample was send, useful to implement time-based filters.

Another important structure that helps with the implementation of additional Qos is the one provided with each call to the *onPublicationMatched* or *onSubscriptionMatched* method. This structure has the following parameters:

- remoteEndpointGUID: GUID_t of the writer or reader that has been matched.
- Status: MATCHED_MATCHING or REMOVED_MATCHING, indicating whether the method has been called when a new endpoint has been matched or when one of the already amtched endpoints has been removed.

The next subsections include different examples of additional Qos that can be easily implemented with the library API.

6.5.1 Time-Based Filter Qos

An example of this behavior implementation is shown below.

```
void TestTypeListener::onNewDataMessage()
{
    TestType tp;
    SampleInfo_t info;
    sub->readNextData((void*)&tp,&info) //Read Data
    if(info.sampleKind == ALIVE)
    {
         if(TimeConv::Time_tAbsDiff2DoubleMillisec
             (last_info.sourceTimestamp,info.sourceTimestamp)>=
           TimeConv::Time_t2MilliSecondsDouble(
            m_attributes.qos.m_timeBasedFilter.minimum_separation))
         {
               //USE THE DATA
         }
    }
}
```

6.5.2 Content-based Filter

An example of the implementation of a content-base filter in the subscriber side is presented below.

```
void TestTypeListener::onNewDataMessage()
{
    TestType tp;
    sub->readNextData((void*)&tp,&info) //Read Data
    if(info.sampleKind == ALIVE)
    {
         if(passFilter(tp))
         {
               //USE THE DATA
         }
    }
}
bool TestTypeListener::passFilter(TestType& tp)
{
    //CHECK THE DATA AGAINST THE FILTER YOU WANT
}
```

6.5.3 Ownership Qos

If the ownership kind is set to EXCLUSIVE_OWNERSHIP_QOS, only one writer can update the sample sample. In this case the writer with the greatest ownership strength. An example of this behavior is included below. In this example we assume that only one key is used. For a multiple keyed topic, a vector of owners must be implemented since each key can have a different owner.

```
//Global variables (or class members)
TestType tp;
GUID_t ownerGUID;
uint32_t ownerStrength = 0;
bool hasOwner = false;

void TestTypeListener::onNewDataMessage()
{
    sub->readNextData((void*)&tp,&info) //Read Data
    if(info.sampleKind == ALIVE)
    {
        if(!hasOwner || info.ownershipStrength > ownerStrength ||
           (info.ownershipStrength == ownerStrength &&
                info.writerGUID < ownerGUID) //Change OWNER conditions
            )
        {
             hasOwner = true;
             ownerGUID = info.writerGUID;
             ownerStrength = info.ownershipStrength;
             sampleAccepted(tp);
        }
        else if(ownerGUID == writerGUID)
             sampleAccepted(tp);
    }
}
void TestTypeListener::sampleAccepted(TestTYpe& tp)
{
    //DO SOMETHING WITH THE SAMPLE
}
void TestTypeListener::onSubscriptionMatched(MatchingInfo info)
{
    if(info.status ==MATCHED_MATCHING)
    {
        cout << "Subscriber MATCHES Pub: " << info.remoteEndpointGuid << endl;
    }
    else if(info.status == REMOVED_MATCHING)
    {
       cout << "Subscriber REMOVED Pub: " << info.remoteEndpointGuid << endl;
       if(info.remoteEndpointGUID == ownerGUID)
       {
            hasOwner = false;
       }
    }
}
```