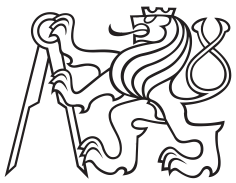
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of measurement

Implementation of actual version of DDSI-RTPS protocol for distributed control in Ethernet network

Jiri Hubacek
Open Informatics

January 2016
Supervisor: Pavel Pisa

Draft: 28. 12. 2015

Acknowledgement / Declaration

Podekovani..

Prohlasuji..

.....

Abstrakt / Abstract

Czech abstract..

Klíčová slova: RTPS, ORTE, Ethernet, Real-Time

Překlad titulu: Implementace aktuální verze protokolu DDSI-RTPS pro distribuované řízení v síti Ethernet

English abstract..

Keywords: RTPS, ORTE, Ethernet, Real-Time

Contents /

1 Introduction 1
2 Technology overview 2
2.1 DDS 2
2.2 DCPS 2
2.3 RTPS 2
2.4 ORTE 2
3 Actual RTPS protocol 3
3.1 Structure Module 3
3.1.1 Participant 4
3.1.2 Writer Endpoint 4
3.1.3 Reader Endpoint 5
3.1.4 History Cache 5
3.1.5 Cache Change 5
3.1.6 Participant Proxy 6
3.1.7 Reader Proxy 6
3.1.8 Writer Proxy 6
3.2 Messages Module 7
3.2.1 Header 8
3.2.2 Submessage Header 8
3.2.3 Interpreter Submessages 8
3.2.4 Entity Submessages 8
3.3 Behavior Module 9
3.3.1 Interoperability 9
3.3.2 Implementation 10
3.3.3 Stateless Writer 10
3.3.4 Stateless Reader 10
3.3.5 Stateful Writer 11
3.3.6 Stateful Reader 11
3.4 Discovery Module 11
3.4.1 SPDP 12
3.4.2 SEDP 12
3.5 RTPS 1.0 12
4 Required changes in ORTE 13
5 Testing of implementation 14
6 Shape for Android 15
6.1 Shape demo 15
6.2 Familiarization with ORTE 15
6.3 Classes 16
6.4 Compatibility 16
7 Security for DDS 18
7.1 Threats 18
7.2 Securing of messages 19
7.3 Plugin architecture 20
7.3.1 Authentication plugin 20
7.3.2 Access Control plugin 20
7.3.3 Cryptographic plugin 20
7.3.4 Logging plugin 21
7.3.5 Data Tagging plugin 21
7.4 Interoperability 22
7.4.1 Requirements 22
7.4.2 Considerations 22
7.5 Implementation 23
7.5.1 Builtin Endpoints 23
7.5.2 Builtin Plugins 23
8 Conclusion 24
References 25
A Symbols 27

[intro]

Chapter 1
Introduction

The Real-Time Publish-Subscribe (RTPS)[1] is the protocol of Data Distribution Service (DDS)[2] family, supporting Data-Centric Publish-Subscribe in real time and specifying communication in a decentralized network, where multiple nodes needs to send and/or receive data in real time. Specification of protocol is developed by Object Management Group[3] - international, open membership, not-for-profit technology standards consortium, since version 1.0 on February 2002 till version 2.2 on September 2014.

This thesis aims on upgrading ORTE implementation of RTPS protocol to be compatible with the latest standard version 2.2. The structure is as follows. In Chapter 1, there is an introduction to RTPS and ORTE. Chapter 3 compares implemented RTPS 1.0 with the latest RTPS 2.2, chapter 4 covers changes needed for compatibility with version 2.2 of the RTPS protocol and chapter 5 covers testing of new implementation of RTPS protocol in ORTE. In chapter 6, demo application of ORTE called *Shape* for Android is introduced. Original ORTE version based demonstration application has been developed as part of preparation for main work to gain experience with RTPS protocol and its implementation. Security for DDS is discussed in chapter 7.

[over]

Chapter 2
Technology overview

2.1 DDS

There are two main models used in Data Distribution Services. *Centralized* model, where single server for the whole network is needed and all communication goes throw it, introduces single point of failure. When the server is unreachable, the whole network is non-functional. By contrast, *decentralized* approach has no central server, no single point of failure. When one node of the network is non-functional, the rest of the network can continue in data transfers.

2.2 DCPS

In the Data-Centric Publish-Subscribe network, data are sent by *Publishers* and received by *Subscribers*. Node can be *Publisher*, *Subscriber* or both and each node can be interested in different data, timing and reliability. Data-Centric Publish-Subscribe network is responsible for delivery of right data between right nodes with right parameters.

[rfc-1]

2.3 RTPS

Real-Time Publish-Subscribe is wire protocol developed to ensure interoperability between DDS implementations. It has been designed to be fault tolerant (decentralized), scalable, tunable, with plug-and-play connectivity and ability of best-effort and reliable communication in real time applications.

2.4 ORTE

Open Real-Time Ethernet (ORTE)[4] is the implementation of RTPS 1.0. It's implemented in Application layer of UDP/IP stack, written in C, under open source license, with own API. Because there are no special requirements, it should be easy to port ORTE to many platforms, where UDP/IP stack is implemented.

[compare]

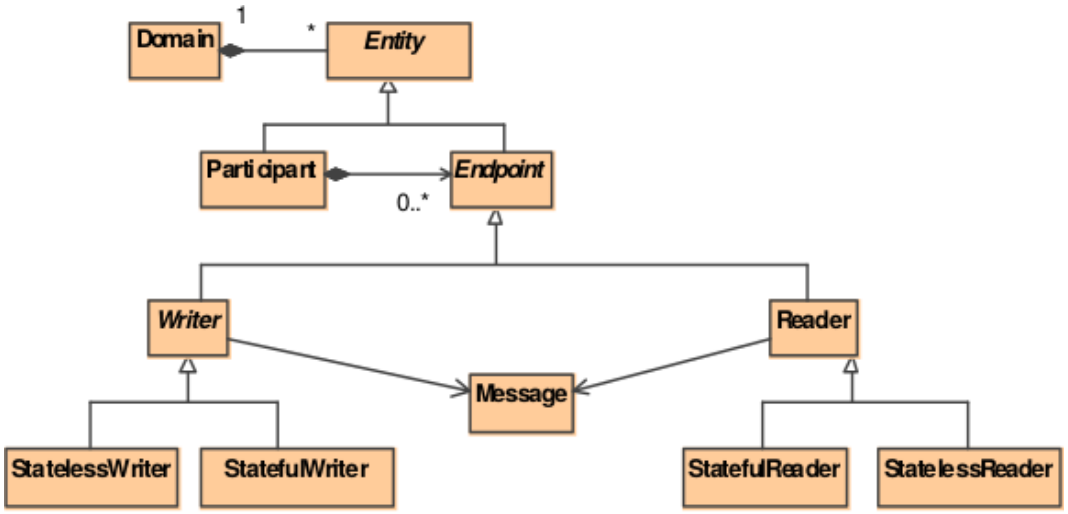
Chapter 3
Actual RTPS protocol

This chapter follows Platform Independent Model (PIM) of the RTPS protocol introduced in chapter 8 in [1]. It contains four modules - basic objects are discussed in section 3.1, messages used for communication is described in 3.2 and behavior - messages exchange between objects - is discussed in 3.3. The last module of discovering *Domain* is convered in section 3.4. In the last section 3.5, versions 2.2 and 1.0 of RTPS protocol are compared.

[struct]

3.1 Structure Module

The communication take place in the RTPS *Domain*, consisting of multiple *Entities*. Each *Entity* can be either *Participant* or *Endpoint*, where *Endpoints* can be specialized as *Writer* or *Reader*. Each *Endpoint* has it's own database of *Cache Changes* called *History Cache*. The whole structure is shown in figure 3.1.



[pic:struct] **Figure 3.1.** Structure Module diagram (chapter 7.1 in [1])

It should be mentioned that there is also proxy *Participant* - *Participant Proxy* and another two proxy *Endpoints* - *Reader Proxy* and *Writer Proxy*. These objects represents remote *Participants* and their *Writers* and *Readers* and are introduced in chapter 8.4 in [1]. The local *Participant* maintains the topology of the *Domain* and needs to store information about remote *Participants*. For this purpose, *Participant Proxy* is used. Sometimes, local *Writer* needs to store information about remote *Reader* and therefore, *Reader Proxy* is used. Also local *Readers* are sometimes in need of storing information about remote *Writers* and then *Writer Proxy* is used.

3.1.1 Participant

Domain Participant is container for *Endpoints* within the same application. It has the following attributes:

- GUID
- Protocol Version
- Vendor Id
- Default Unicast Locator List
- Default Multicast Locator List

Where the *GUID* is globally-unique RTPS-entity identifier consisting of *GUID Prefix* and *EntityId*, *Protocol Version* is version of actual implementation and vendor of implementation is represented by *Vendor Id*. *Default Unicast Locator List* and *Default Multicast Locator List* are lists of IP address and port combinations used to send User-data traffic to, when there is no such an information contained in *Writers* of the *Participant*.

Each *Endpoint* within the same *Participant* has to have the same *GUID Prefix*.

3.1.2 Writer Endpoint

Is the source of *Cache Changes* which are sent to *Readers*. It has the following attributes:

- GUID
- Topic Kind
- Reliability Level
- Unicast Locator List
- Multicast Locator List
- Push Mode
- Heartbeat Period
- Nack Response Delay
- Nack Suppression Duration
- Last Change Sequence Number
- Writer Cache

Where *Topic Kind* can be either NO_KEY or WITH_KEY. WITH_KEY is used, when the topic consists of more than one data instances identified by *key*. *Reliability Level* can be either BEST_EFFORT or RELIABLE, saying if it should be verified that *Cache Change* reached the *Reader*. *Unicast Locator List* and *Multicast Locator List* are lists of IP address and port combinations on which is the *Writer* listening. If there is no IP address and port combination in list, it's presumed that *Writer* is listening on *Default Unicast Locator List* respective *Default Multicast Locator List* of the *Participant*. *Push Mode* defines if data are sent (*Push Mode* is set to TRUE) or just Heartbeats with Sequence Numbers of available *Cache Changes* and *Reader* has to ask for *Cache Change* delivery. *Heartbeat Period*, *Nack Response Delay* and Nack Suppression Duration are time parameters used for protocol tunning, which defines announce interval of available data, how long the response to data request should be delayed respective how long can be data request for just sent data ignored. *Last Change Sequence Number* is the highest Sequence Number in *History Cache* and the *Writer Cache* is the *History Cache* of the *Writer* containing *Cache Changes* associated with the *Writer* itself.

3.1.3 Reader Endpoint

Is the destination of *Cache Changes* which are sent by *Writers*. It has the following attributes:

- GUID
- Topic Kind
- Reliability Level
- Unicast Locator List
- Multicast Locator List
- Expects Inline Qos
- Heartbeat Response Delay
- Heartbeat Suppression Duration
- Reader Cache

Where *Unicast Locator List* and *Multicast Locator List* are lists of IP address and port combinations on which is the *Reader* listening. If there is no IP address and port combination in list, it's presumed that *Reader* is listening on *Default Unicast Locator List* respective *Default Multicast Locator List* of the *Participant*. The value of *Expects Inline Qos* is set to TRUE if the *Reader* expects in-line Qos to be sent along with data. *Heartbeat Response Delay* and *Heartbeat Suppression Duration* are time parameters used for protocol tunning, which defines how long the acknowledgement of data should be delayed respective how long can be Heartbeat announces ignored after just received Heartbeat. *Reader Cache* is the *History Cache* of the *Reader* containing *Cache Changes* associted with the *Reader* itself.

3.1.4 History Cache

Is the database of *Cache Changes* serving as the API for *Writer* and *Reader Endpoints*. It has the following attributes:

- Changes

Where *Changes* are *Cache Changes* stored in the *History Cache*.

3.1.5 Cache Change

Is the change of the data object that should be propagated from the *Writer* to the matching *Readers*. It has the following attributes:

- Change Kind
- Writer GUID
- Instance Handle
- Sequence Number
- Data value

Where *Change Kind* can be ALIVE, NOT_ALIVE_DISPOSED or NOT_ALIVE_UNREGISTERED and is used to distinguish the change that was made to a data object. *Writer GUID* is the identifier of the source of the *Cache Change*, *Instance Handle* identifies the instance of the data object (in DDS the value of the *key* is used) and the *Sequence Number* is unique identifier of the *Cache Change* in the *History Cache* of the *Endpoint*. The last attribute, *Data value*, represents data associated to the *Cache Change*.

3.1.6 Participant Proxy

Represents the information about remote *Participant* in the *Domain*. It has the following attributes:

- Protocol Version
- Guid Prefix
- Vendor Id
- Expects Inline Qos
- Available Builtin Endpoints
- Metatraffic Unicast Locator List
- Metatraffic Multicast Locator List
- Default Multicast Locator List
- Default Unicast Locator List
- Manual Liveliness Count
- Lease Duration

Where *Protocol Version* specify the version of the RTPS protocol implementation used by remote *Participant* and the vendor of this implementation is represented by the *Vendor Id*. *Guid Prefix* is the common part of the GUID for the *Participant* and all of it's *Endpoints*, *Expects Inline Qos* describes whether the *Readers* of the remote *Participant* expects in-line Qos sent along with data and *Available Builtin Endpoints* parameter specify which builtin *Endpoints* used for plug-and-play interoperability are available by remote *Participant*. *Metatraffic Unicast Locator List* and *Metatraffic Multicast Locator List* are IP address and port combinations that can be used to reach the remote builtin *Endpoints* and *Default Unicast Locator List* and *Default Multicast Locator List* are IP address and port combinations that can be used to reach the remote *Endpoints* defined by user that serve for user data exchange. *Manual Liveliness Count* is used to implement MANUAL_BY_PARTICIPANT liveliness Qos and *Lease Duration* specify the time period for which the remote *Participant* should be considered alive.

3.1.7 Reader Proxy

Represents the information about remote *Reader*. It has the following attributes:

- Remote Reader GUID
- Unicast Locator List
- Muticast Locator List
- Changes for Reader
- Expects Inline Qos
- Is Active

Where *Remote Reader GUID* is unique identifier of remote *Reader*, *Unicast Locator List* and *Multicast Locator List* are lists of IP address and port combinations on which is the remote *Reader* listening, *Changes for Reader* is the list of *Cache Changes* that should be sent to the remote *Reader*, *Expects Inline Qos* attribute specify if the remote *Reader* expects in-line Qos to be sent along with data and attribute *Is Active* is set to TRUE if the remote *Reader* is responsive to the local *Writer*.

3.1.8 Writer Proxy

Represents the information about remote *Writer*. It has the following attributes:

- Remote Writer GUID

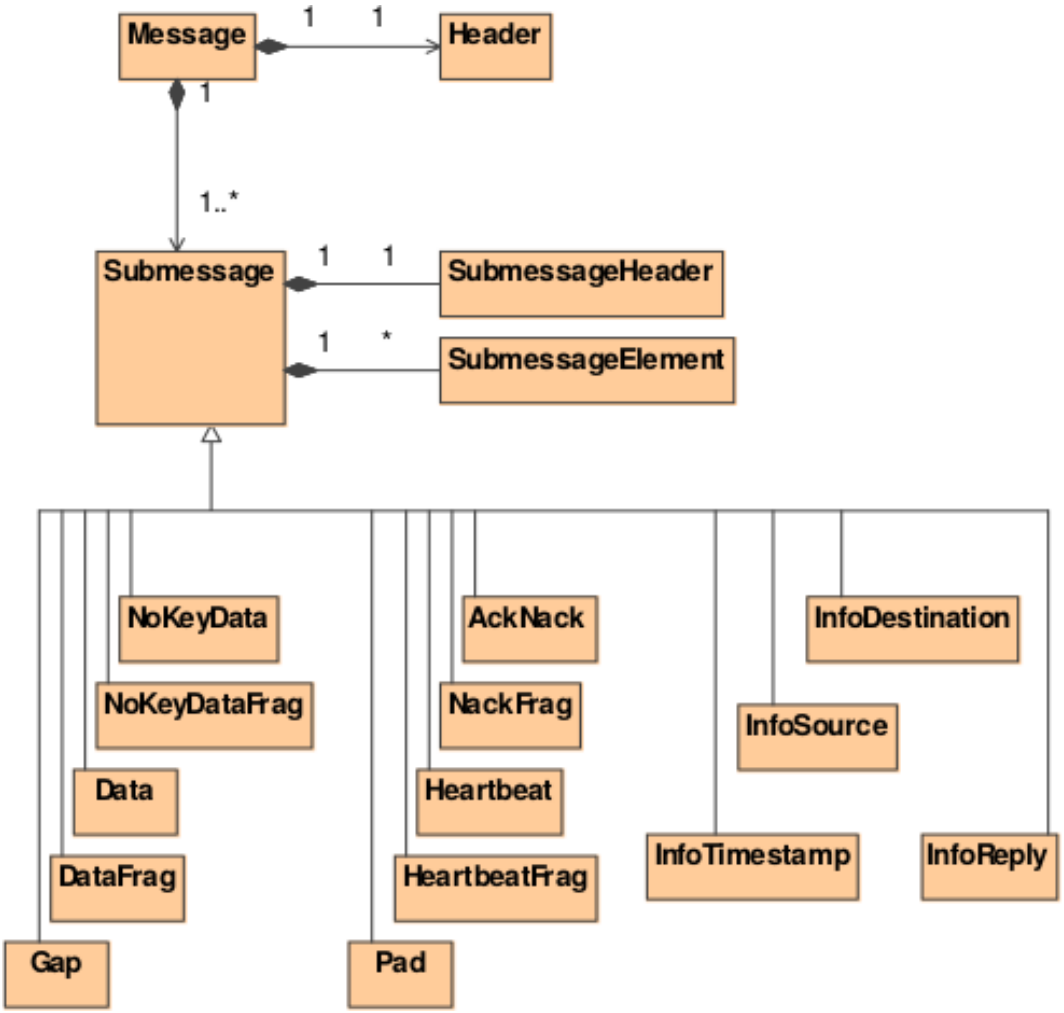
- Unicast Locator List
- Multicast Locator List
- Changes from Writer

Where *Remote Writer GUID* is unique identifier of remote *Writer*, *Unicat Locator List* and *Multicast Locator List* are lists of IP address and port combinations on which is the remote *Writer* listening and *Changes from Writer* is the list of *Cache Changes* that are received or expected from the remote *Writer*.

[message]

3.2 Messages Module

For communication between *Writers* and *Readers*, RTPS messages are used. Each message consists of *Header* and one or more *Submessages*, where each *Submessage* has it's own *Submessage Header* and *Submessage Elements* based on the kind of the *Submessage*. The structure of RTPS message is shown in figure 3.2.



[pic:msg] **Figure 3.2.** Structure of RTPS message (chapter 8.3.3 in [1])

The interpretation of *Submessage* may depend on previously received *Submessages* within the same *Message*, therefore it's needed to store the state for each *Message*. *Message Receiver* ensures this function by storing information about *Source Protocol*

Version, *Source Vendor Id*, *Source GUID Prefix*, *Destination GUID Prefix*, *Unicast Reply Locator List*, *Multicast Reply Locator List*, *Have Timestamp* and *Timestamp*. State of the *Message Receiver* is reset to default values each time the *Message* is received.

Submesssages can be divided to *Entity Submessages* which target an RTPS *Entity* affecting it's behavior and *Interpreter Submessages* changing the state of the *Message Receiver*.

3.2.1 Header

The first change in the state of *Message Receiver* is made by *Header* of the received *Message*. The *Header* identify RTPS *Message*, *Protocol Version* used, *Vendor* of the implementation and common part of GUID - *GUID Prefix* - used to interpret source *EntityId* in the *Submessage*.

3.2.2 Submessage Header

Submessage Header is included in each *Submessage*, it has the following attributes:

- Submessage Kind
- Flags
- Submessage Lenght

Where *Submessage Kind* specify the meaning of *Submessage*, *Flags* is an array of 8 bits, which identifies endianness used in the *Submessage* (LSB), the presence of optional elements and possibly changes the interpretation of the *Submessage*. The *Submessage Length* specify the length of the *Submessage*.

3.2.3 Interpreter Submessages

List of *Submessages* and their changes to the *Message Receiver* follows.

InfoDestination is sent from *Writer* to *Reader* to modify the *Destination GUID Prefix* value of *Message Receiver* used to interpret *Reader EntityId* in the *Submessage*.

InfoReply is sent from *Reader* to *Writer* to explicitly define where to send a reply to the *Submessage* that follow.

InfoSource modifies the source *Protocol Version*, *Vendor Id* and *GUID Prefix* of the *Submessages* that follow.

InfoTimestamp is used to send the *Timestamp* that apply to the *Submessages* that follow.

3.2.4 Entity Submessages

List of *Submessages* and their interpretation follows.

AckNack is sent by *Reader* to inform the *Writer* about which sequence numbers are received and which are missing.

Data is sent from *Writer* to *Reader* and is used to inform about changes to a data object. Changes may be in value or in lifecycle of the object.

DataFrag is used when the **Data** *Submessage* exceeds the MTU of lower communication layers, otherwise it has the same function as **Data** *Submessage*.

Gap sent by *Writer* indicates to the *Reader* which sequence numbers are no longer relevant.

Heartbeat announces to *Readers* sequence numbers of *Cache Changes* that are available in *Writer*.

HeartbeatFrag is used when fragmentation occures and until all fragments are available. Once all fragments are available, **Heartbeat** *Submessage* is used.

NackFrag is sent by *Reader* to inform the *Writer* about which fragments are missing.
Pad is padding message used for desired alignment. It has no other meaning.

[behavior]
3.3 Behavior Module

The purpose of the RTPS protocol can be simplified to propagating *Cache Change* from *Writer* to *Reader*. The manner of propagation in relation to properties of communication is discussed in this section. There are two main properties of communication:

- Topic Kind
- Reliability Level

Where *Topic Kind* can be either NO_KEY or WITH_KEY and *Reliability Level* can be either BEST_EFFORT or RELIABLE. Pairing of *Writers* with *Readers* must acknowledge the following restrictions.

Writer and *Reader* can be paired up and the communication may begin when the *Topic Kind* matches, because both *Endpoints* relate to the same DDS Topic which is either NO_KEY or WITH_KEY.

The reliability depends on the *Reader*. If the *Reader* has the *Reliability Level* set to RELIABLE, the corresponding *Writer* has to also have the *Reliability Level* set to RELIABLE, while the *Reader* has the *Reliability Level* set to BEST_EFFORT, the *Reliability Level* of the corresponding *Writer* doesn't matter.

The behavior required for interoperability between implementations is summarized in section 3.3.1. In chapter 8.4 in [1], there are two reference implementations of *Endpoints* that are summarized in section 3.3.2. In the remaining sections, these implementations are discussed in detail.

[inte]
3.3.1 Interoperability

In order to be compliant with protocol specification and interoperable with other implementations, the implementation of the RTPS protocol must satisfy the following requirements.

- General Requirements:
  - Only RTPS messages are used for communication.
  - Message Receiver must be implemented.
  - Timing characteristics must be tunable.
  - SPDP and SEDP must be implemented.
  - Writer Liveliness Protocol must be implemented.
- Required Writer Behavior:
  - Data must be sent in-order.
  - If requested, in-line Qos must be included.
- Additional requirements for Reliable Writer:
  - Periodic HEARTBEAT messages must be sent.
  - Response to ACKNACK message must be eventually sent.
- Required Reader Behavior:
  - No specific behavior needed as best-effort reader is completely passive.
- Additional requirements for Reliable Reader:

- Response to HEARTBEAT with final flag not set must be eventually sent.
- Response to HEARTBEAT indicating missing sample must be eventually sent.
- Once acknowledged, always acknowledged.
- ACKNACK can be only sent in response to HEARTBEAT.

Writer Liveliness Protocol is required by DDS to exchange information about the Liveliness of *Writers* by *Participants*. Builtin *Endpoints* are used for sending samples at rate faster than the smallest lease duration among *Writers* sharing the same liveliness Qos.

[impl]

3.3.2 Implementation

In chapter 8.4 in [1], two reference implementations of RTPS protocol are introduced.

Stateless implementation maintains no state about remote *Endpoints*, which suits for best-effort communication over multicast. While this implementation scales well in large systems and memory usage is reduced, additional bandwidth may be required.

Stateful implementation, on the other hand maintains full state on remote *Endpoints*. More memory is needed and scalability is limited, but bandwidth usage decreases. Also, strict reliable communication and Qos based filtering on the *Writer's* side may be applied.

It's important to mention that these are *reference implementations*. It means that there is no need for two kinds of *Endpoints* - *Stateless* and *Stateful*, because the only behavior needed for interoperability is introduced in 3.3.1. *Reference implementations* just helps to understand and implement this behavior. Both *reference implementations*, *Stateless* as well as *Stateful* may be BEST_EFFORT, RELIABLE, NO_KEY or WITH_KEY. Excaptions are discussed and explained below.

3.3.3 Stateless Writer

To be compliant with 3.3.2, *Stateless Writer* must send data in order and include in-line Qos if needed. BEST_EFFORT *Stateless Writer* has no other requirements and data are sent each *Resend Data Period* or on demand by user application.

If *Stateless Writer* is RELIABLE, periodic HEARTBEATs with available data are sent. Sending of data then depend on *Push Mode*. If *Push Mode* is TRUE, data are sent each *Resend Data Period* or on demand by user application, if it's FALSE, data are stored in *History Cache* of *Writer* and sent only in response to ACKNACK.

Readers uses ACKNACK messages to request interesting data from *Reliable Writer*. DATA submessage is sent if data are still available in *Writer's History Cache* or GAP submessage indicating that data are no longer relevant.

Even the bandwidth can be reduced if the information of acknowledged Sequence Numbers would be stored for remote *Readers*, it's not needed for interoperability.

3.3.4 Stateless Reader

Receive and process data is the meaning of BEST_EFFORT *Stateless Reader*. However, to ensure RELIABLE communication, the problem arise, because at least Sequence Numbers of announced, requested but not received *Cache Changes* are needed, so some information about remote *Writer* has to be stored and therefore *Reader* can't be *Stateless*.

As mentioned above, this is the only exception. *Stateless Reader* can't be RELIABLE.

3.3.5 Stateful Writer

For each remote *Reader*, *Stateful Writer* stores information in *Reader Proxy* structure. When *Cache Change* is added to the *History Cache* of *Writer*, filtering can occures to determine if *Cache Change* is relevant for *Reader* and consequently stored in *Changes for Reader* of *Reader Proxy*.

BEST_EFFORT traffic is sent on demand by user application to each *Reader Proxy* whenever there are any unsent changes in *Changes for Reader*.

For RELIABLE *Stateful Writer*, periodic HEARTBEATs must be sent to each *Reader Proxy*. Sending of data then depends on *Push Mode* - when the value is TRUE, *Cache Change* pass the filter and is added to *Changes for Reader*, the *Cache Change* is marked as unsent and will be sent as soon as the needed resources would be available. When tha value of *Push Mode* is FALSE, only HEARTBEATs are sent and *Reader* has to ask for data by sending ACKNACK for interested data. In response to ACKNACK, *Reliable Writer* then sends DATA submessage if the data are still available or GAP submessage when the data are no longer relevant.

It should be mentioned that in general, *Reader's Entity Id* of submessages is set to ENTITYID_UNKNOWN, stating that each *Reader* of remote *Participant* should receive the data. The only situation when *Writer* knows exactly the destination and therefore may set *Reader's Entity Id* properly is sending DATA submessage in response to ACKNACK by *Stateful Writer*.

3.3.6 Stateful Reader

For BEST_EFFORT traffic, *Stateful Reader* stores information about expected Sequence Number for each remote *Writer*. This information is stored in *Writer Proxy* structure. Storing Sequence Number ensures that there are no duplicated nor out-of-order data changes.

If the communication is RELIABLE and DATA submessage or GAP is received, expected Sequence Number is set correspondingly. When HEARTBEAT is received, databases of missing and lost changes are updated for *Writer Proxy* and the rest of behavior depends on *Final* and *Liveliness Flags*. When both, *Final* and *Liveliness Flags* are set, nothing happens. When *Liveliness Flag* is not set, then ACKNACK with missing changes may be sent and when *Final Flag* is not set, ACKNACK must be sent.

[discovery]

3.4 Discovery Module

The communication as described in 3.3 between *Endpoints* described in ?? throw the *Messages* described in 3.2 assumes that both ends of communication are known. *Discovery Module* introduces process of probing *Domain* due to discovering *Participants* called *Simple Participant Discovery Protocol* and theirs *Endpoints* known as *Simple Endpoint Discovery Protocol*.

SPDP and SEDP are only discovery protocols described in RTPS specification and have to be implemented in order to enable interoperability between implementations. However vendor specific discovery protocols can be implemented in addition to overcome some drawbacks of *Simple Discovery Protocols*.

The difference between *Builtin* and *User-defined Endpoints* needs to be clear. *Builtin Endpoints* are predefined by RTPS specification and once a *Participant* is discovered, it can be assumed that *Builtin Endpoints* are present, while *User-defined Endpoints* are defined by user application and it's purpose and can't be known in advance. Therefore the purpose of *Discovery Module* can be roughly simplified to discovering *User-defined*

Endpoints of remote *Participants* with help of *Builtin Endpoints*. *Builtin Endpoints* are used by *Simple Discovery Protocols*.

3.4.1 SPDP

Best-effort Writer with predefined Entity Id[1]) and *Best-effort Reader* with predefined Entity Id[2]) are used to exchange *SPDPdiscoveredParticipantData* containing *Participant Proxy* information about remote *Participant*. This traffic is sent to predefined IP address and port discussed in PSM in [1]. Remote *Participants* and their attributes are discovered by SPDP.

3.4.2 SEDP

Reliable Writer with predefined Entity Id[3]) and *Reliable Reader* with predefined Entity Id[4]) are used to exchange *DiscoveredWriterData* containing information about *Writers* of remote *Participant*.

Reliable Writer with predefined Entity Id[5]) and *Reliable Reader* with predefined Entity Id[6]) are used to exchange *DiscoveredReaderData* containing information about *Readers* of remote *Participant*.

The last pair of *Reliable Endpoints*[7]) can be used to exchange *DiscoveredTopicData*, but these *Endpoints* aren't mandatory and the interoperability is not affected by them.

[rtps10]

3.5 RTPS 1.0

[1]) ENTITYID_SPDP_BUILTIN_PARTICIPANT_WRITER
[2]) ENTITYID_SPDP_BUILTIN_PARTICIPANT_READER
[3]) ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER
[4]) ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER
[5]) ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER
[6]) ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER
[7]) ENTITYID_SEDP_BUILTIN_TOPIC_WRITER and ENTITYID_SEDP_BUILTIN_TOPIC_READER

[upgrade]

Chapter 4
Required changes in ORTE

[test]

Chapter 5
Testing of implementation

[shape]

Chapter 6
Shape for Android

6.1 Shape demo

With ORTE implementation of RTPS 1.0 protocol, demo application called Shape is delivered. Shape demo demonstrates the functionality of ORTE - when the color (Blue, Green, Red, Black, Yellow) is choosen, the *Publisher* is created as random shape (Circle, Square, Triangle) moving on the screen. Then, under the topic of color name, object's shape, color and coordinates are published to the network. It's possible to receive and interpret object's data (to see colored shapes moving on the screen) by adding the *Subscribers* of specific topics (colors).

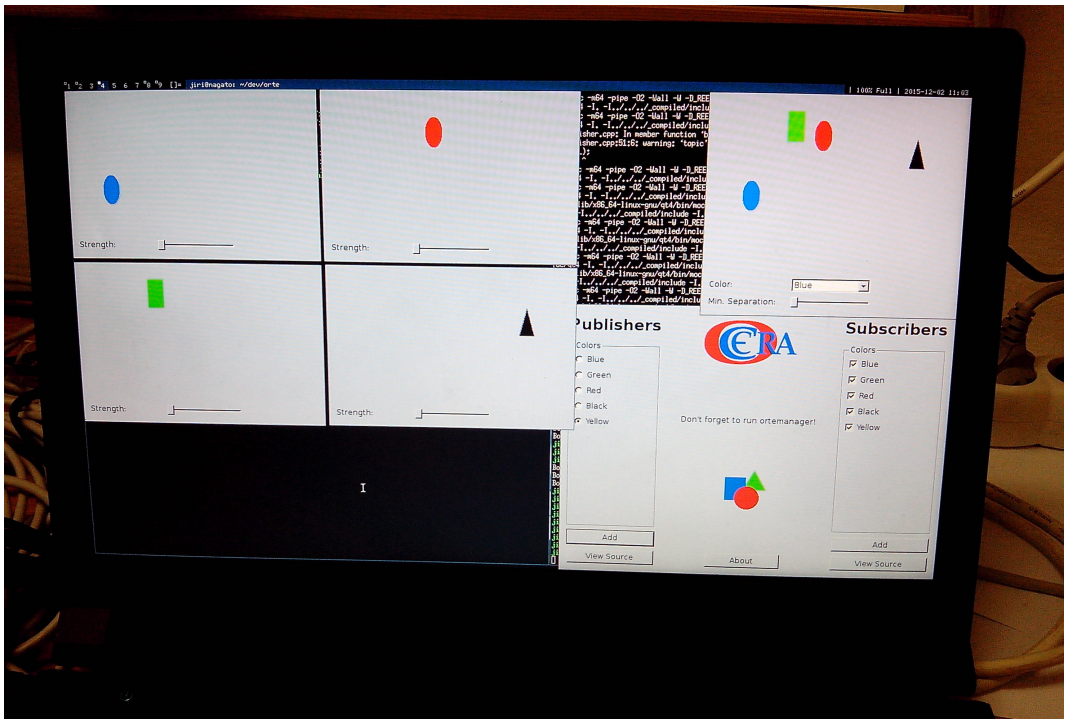


Figure 6.1. Shape demo - *Publishers* and *Subscribers*

6.2 Familiarization with ORTE

The familiarization with ORTE was done by creating demo application for Android compatible with Shape. Because the port of ORTE to Android has been already done in [5] and is available as library, the main task was application design and compatibility ensurance. The application was designed to be as simple as possible. *Publishers* view allows to create new *Publisher* of specific color and random shape, *Subscribers* view allows to set up *Subscribers* of specified colors. Finally, Settings and Help views are present.

6.3 Classes

As in Shape demo, in Shape for Android the *BoxType* class is presented, allowing to create, send and receive objects. *BoxType* consists of *color* (integer), *shape* (integer) and *rectangle* (*BoxRect*), where *BoxRect* is class for storing coordinates - *top_left_x* (short), *top_left_y* (short), *bottom_right_x* (short), *bottom_right_y* (short). The *BoxType* is extension of *MessageData* class delivered with ORTE library for Android. It allows to send and receive objects.

PublisherShape class stores *BoxType* information about *Publisher*, it's properties needed for ORTE, methods for communication with object and prepares data to send. In *Publisher* view, *Publisher* objects are created, stored in *ArrayList* and drawed on screen. Data objects are sent in *Publisher* activity each time objects are redrawn.

SubscriberElement class receives *BoxType* object from ORTE and stores it's data and methods needed for presentation. In *Subscriber* view, all received objects are stored in *ArrayList* and periodically redrawn.

Settings view allows to set up scaling, needed because of various dimensions of screens. It also contains a list of managers - in RTPS 1.0 special application called manager is used for communication of available *Publishers*/*Subscribers* between nodes. In RTPS 2.2 Simple Participant Discovery Protocol (SPDP) and Simple Endpoint Discovery Protocol (SEDP) are used.

Help view contains information about ORTE, Shape and application usage.

6.4 Compatibility

BoxType in Shape and Shape for Android is a little bit different. The reason is just familiarization with ORTE implementation and RTPS protocol, where misunderstooding was not fully avoided. Suggestions for improvements follows.

The first property of *BoxType* is *color*. In Shape demo, *color* is typed as CORBA_octet (macro for uint8_t, 1 byte) and in Shape for Android, *color* is of integer type (4 bytes). The reason why this approach does not break the compatibility is following: each data-type serialized by CORBA is aligned to 4 byte boundary. In this case, object color is first byte and the rest until the boundary is filled by zero bytes. This data representation corresponds to Little Endian in which the message is encoded by default (endianness is operating system dependent), so when Shape for Android deserialize data, Little Endian encoded integer is obtained. It also works in opposite direction - value of the *color* is serialized as integer, encoded as Little Endian and on the side of Shape demo, CORBA_octet is deserialized and 3 zero bytes skipped because of boundary alignment. The problem could arise when *color* would be sent as integer with Big Endian encoding and received as CORBA_octet, because the value of the first byte would be then zero. Also, the problem wouldn't persist in the opposite direction, because endianness is always part of the RTPS message so even node with Big Endian default encoding would receive Little Endian encoded message correctly.

The second property of *BoxType* is *shape*. The type in Shape demo is CORBA_long (macro for int32_t, 4 bytes) and integer (4 bytes) in Shape for Android. Therefore there is no problem with *shape* property.

The last property of *BoxType* is *BoxRect* consisting of coordinates of object. Each value of *BoxRect* is CORBA_short type (2 bytes) in Shape demo and short type (2 bytes) in Shape for Android. Because *BoxRect* is presented as CORBA autonomous data-type, the whole data-type (8 bytes) is aligned to 4 bytes boundary.

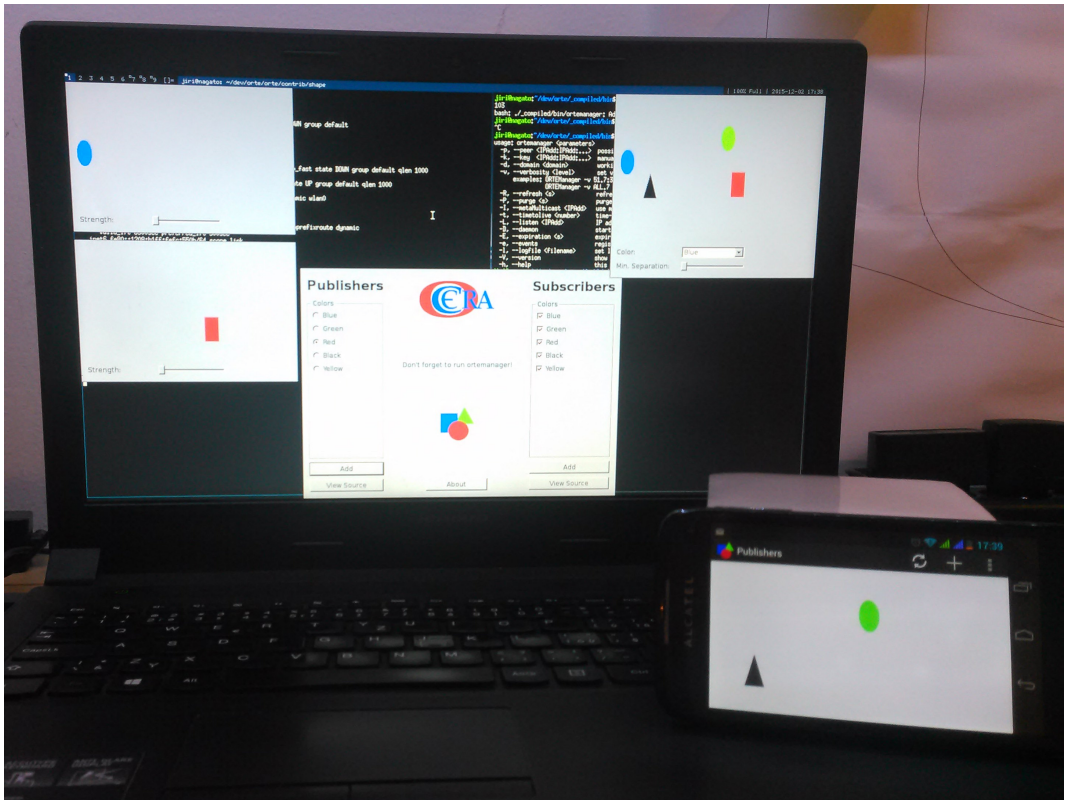


Figure 6.2. Shape for Android - Publishers view

The suggestion for the future improvement of Shape demo and Shape for Android is the revision of *BoxType* data-type.

[sec]

Chapter 7
Security for DDS

In the modern world, security is often considered. Technologies for securing communication differs by TCP/IP layers [6] - security at Media access layer consists of preventing deterioration of physical media, environmental noise and access to media. At Network layer, IPsec (IP Security Architecture) protocol is used while Transport layer uses TLS (Transport Layer Security) protocol. In this chapter, Application layer security for DDS standard [7] and possibilities of implementation in RTPS protocol are considered.

7.1 Threats

From point of view of DDS standard, communication takes place in the domain consisting of participants with various number of publishers and subscribers. In this context, Application layer security threats are following:

- Unauthorized[1]) subscription
- Unauthorized publication
- Tampering and replay
- Unauthorized access to data

Unauthorized subscription is a situation when malicious participant receives data for which it is not allowed to. In network infrastructure where access to media is shared, communication runs over multicast or participants sits on one node, it's practically unavoidable to restrict access to data. The solution is making data unreadable for malicious participant - in other words, applying encryption on publisher's side and sharing keys with authenticated subscribers only.

When malicious participant attempts to send data which it is not allowed to, it's called *Unauthorized publication*. For subscriber it's important to receive data only from valid publishers to avoid influence of malicious participant on data. The solution is to include authentication information to data sent by valid publishers so subscribers would be able to recognize data by authenticated publishers from data sent by malicious participant. Two ways how to accomplish authentication of publishers in data are Hash-based message authentication code (HMAC) and digital signature. HMAC creates authentication code using secret key shared between publisher and subscriber. Digital signature is based on private/public key pair - authentication code is created as message digest encrypted by private key of publisher. Each subscriber has access to public key of publisher and can use it to decrypt the authentication code to message digest and compare it with message digest calculated by itself. The point is that these two message digests equals if and only if the authentication code is encrypted by publisher's private key and decrypted by publisher's public key. Digital signature is called *asymmetric*

[1]) Difference between authentication and authorization has to be clear. Authentication is verification of (in this context) participant - that the participant is really the one it claims to be. On the other hand, authorization is process of allowing access to data for already authenticated participant.

cryptography (private/public key pair) and is much slower then *symmetric cryptography* (shared key), therefore the use of HMAC is preferred because of performance reasons.

Valid publisher would send data to subcsriber and malicious participant (in this case, malicious participant will be allowed to subscribe but not to publish). However if the same key is shared between publisher, subscriber and malicious paritcipant, there is no way how to prevent malicious participant to use this shared key for mimicking publisher and sending data to subscriber. This threat is called *Tampering and Replay* and can be solved by sharing different keys between publishers and subscribers. When the communication is taken over multicast, multiple HMACs are needed to be included in data, but this solution is still more powerful than using digital signatures.

In the DDS network, some participants acts as relay participants forwarding data. These participants need to be trusted as valid publishers and subscribers, but it's not always desirable to let them understand data they work with. The solution for *Unauthorized Access to Data* is having different keys for HMAC and data encryption and to share keys for decrypting of data only with desired endpoints.

7.2 Securing of messages

Securing of messages is application dependent - sometimes it's sufficient to encrypt only user-data, in other applications, submessage's metadata as sequence numbers or writer/reader identifiers are needed to be secured too and in the most secure applications, the whole metatraffic submessages are considered confidential. In order to support of different application scenarios, mechanism called *Message Transformation* is introduced. It transforms one RTPS message into another RTPS message so that the original RTPS message or it's submessages may be encrypted into the new one and protected by HMAC.

Because of *Message Transformation*, new submessages and submessage elements are introduced and the questions about interoperability between secured and non-secured implementations of RTPS protocol arises. In implementations of RTPS protocol, unknown submessages should be skipped so the regular user-traffic should not be affected, but there is Discovery also. SPDP is used by DomainParticipants to discover each other, informations as IP address, port, vendor and version are exchanged to bootstrap the communication. Therefore it makes no sense to protect SPDP communication, better to use it for exchange of informations needed to bootstrap the secured system. For both - secured and non-secured implementations of RTPS protocol, *DCPSParticipants* Topic is used in SPDP and there is no new secured Topic for SPDP.

SEDP protocol is used for discovering *publishers* and *subscribers* of each DomainParitcipant. The *DCPSPublications* and *DCPSSubscriptions* Topics are used for communication with non-secured endpoints. However for DomainParticipants supporting DDS Security, *DCPSPublicationsSecure* and *DCPSSubscriptionsSecure* Topics and associated DataWriters (*SEDPbuiltinPublicationsSecureWriter*, *SEDPbuiltinSubscriptionsSecureWriter*) and DataReaders (*SEDPbuiltinPublicationsSecureReader*, *SEDPbuiltinSubscriptionsSecureReader*) are introduced. These Topics should be used for communication that is considered sensitive.

In RTPS protocol, Writer Liveliness Protocol is specified and because data exchange by this protocol could be considered sensitive, DDS Security specifies alternate protected way to exchange liveliness information. *BuiltinParticipantMessageWriter* and *BuiltinParticipantMessageReader* are used to communicate liveliness information with non-secured endpoints. *ParticipantMessageSecure* Topic is introduced with as-

sociated *BuiltinParticipantMessageSecureWriter* and *BuiltinParticipantMessageSecureReader*, used to communication liveliness information with endpoints considered sensitive.

Also, there are two completely new builtin Topics:

- ParticipantStatelessMessage
- ParticipantVolatileMessageSecure

ParticipantStatelessMessage Topic is used to perform mutual authentication between DomainParticipants. While the mechanism for participant-to-participant communication already exists, it suffers from weakness of reliable protocol - sequence number prediction. HeartBeat messages containing *first available sequence number* can be abused by malicious participant to prevent other participants to communicate. Therefore new Topic *ParticipantStatelessMessage* with associated *BuiltinParticipantStatelessMessageWriter* (Best-Effort StatelessWriter) and *BuiltinParticipantStatelessMessageReader* (Best-Effort StatelessReader) is introduced.

For key exchange between DomainParticipants, reliable and secure communication is needed. On top of that, DURABILITY Qos needs to be VOLATILE to address only DomainParticipants that are currently in the system. *ParticipantStatelessMessage* is not suitable because it's not reliable nor secured. *ParticipantMessageSecure* Topic is not suitable because it's QoS has DURABILITY kind TRANSIENT_LOCAL rather than VOLATILE (which is required). So new Topic *ParticipantVolatileMessageSecure* with associated *BuiltinParticipantVolatileMessageSecureWriter* and *BuitinParticipantVolatileMessageSecureReader* is introduced.

7.3 Plugin architecture

There are five SPIs:

- Authentication
- Access-Control
- Cryptographic
- Logging
- Data Tagging

Interactions of plugins are shown in figure 7.1.

7.3.1 Authentication plugin

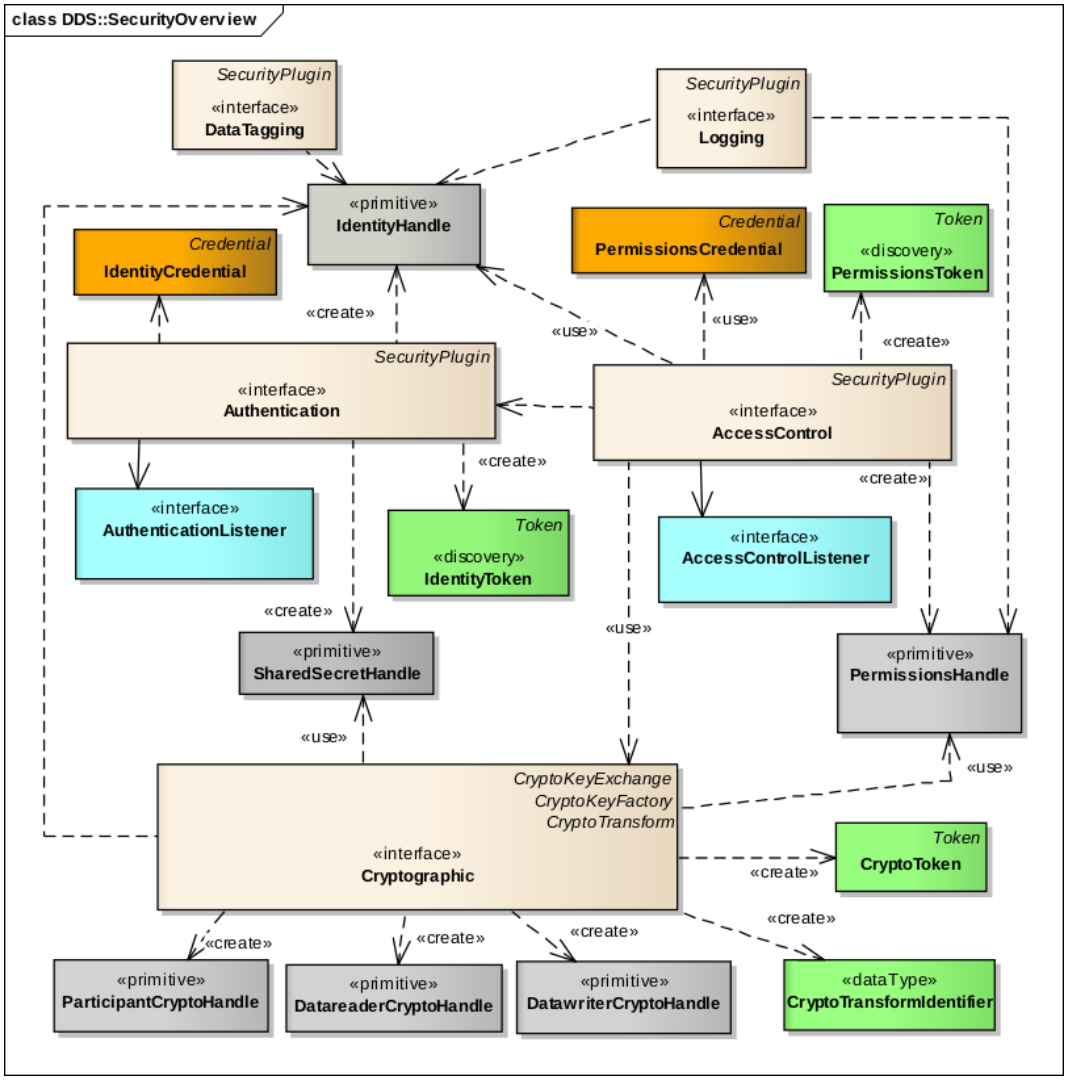
Authentication is process of verifying that (in this case) DomainParticipant is really the one it claims to be. DomainParticipant is authenticated when joining a DDS Domain, mutual authentication is supported and shared secret is established between DomainPariticpants.

7.3.2 Access Control plugin

Ensures authorization - allows or deny protectected operations of DomainPariticipant as join domain, create Topic, publish to Topic or subscribe Topic.

7.3.3 Cryptographic plugin

Encryption, decryption, digests, MAC, HMAC, key generating and exchange, signing and verifying of signatures is ensured by *Cryptographic plugin*. The plugin API has to be general enough to allow specific requirements for cryptographic libraries, encryption and digest algorithms, message authentication and signing users of DDS may need to deploy.



[pic:sec-pa] **Figure 7.1.** Plugin Architecture Model

7.3.4 Logging plugin

This plugin logs security events of DomainParticipant. Two options of collecting log data are logging all events to a local file and distributing log events securely over DDS.

7.3.5 Data Tagging plugin

Classification of data is performed by *Data Tagging plugin*. It can be used for access control based on tag, message prioritization or even don't have to be used by middleware (RTPS implementation), but by application or service. There are four kinds of tagging:

- Data Writer - used in specification, data received from DataWriter has it's tag.
- Data Instance - each instance of the data has a tag.
- Individual sample - each sample of data instance is tagged individually.
- Per field - the most complex method of tagging.

7.4 Interoperability

Out-of-the-box interoperability of DDS Security implementations is ensured analogously to RTPS implementations - while mandatory *builtin endpoints* ensures that each DomainParticipant is able to discover other DomainParticipants, in DDS Security implementations, each DomainParticipant is able to secure data by at least mandatory *builtin plugins*.

7.4.1 Requirements

This is resume of requirements for builtin plugins by out-of-the-box interoperability as presented in chapter 9.2 of [7]. Following are essential functional requirements for builtin plugins:

- Authentication of DomainParticipants joining a domain
- Access control of applications subscribing to data
- Message integrity and authentication
- Encryption of a data by different keys

Following are functions that should be required by builtin plugins:

- Sending digitally signed data
- Sending data securely over multicast
- Data tagging
- Integrating with open standard security plugins

Following are functions considered useful:

- Access control to certain samples
- Access control to certain attributes within sample
- Permissions for QoS usage by DDS Entities

Non-functional requirements are:

- Performance and Scalability
- Robustness and Availability
- Fit to DDS Data-Centric Information Model
- Reuse of existing security infrastructure and technologies
- Ease of use

7.4.2 Considerations

Usually DDS is deployed in systems where high performance for large number of DomainParticipants is needed, therefore actions performed by plugins shouldn't notably degrade system performance. In practice it means that asymmetric cryptography should be used only for discovery, authentication, session and shared-secret establishment, symmetric cryptography shoud be used for data, use of ciphers, HMACs or digital signatures should be selectable per Topic, there should be possibility of providing integrity via HMAC without data encryption and there should be support for encrypted data over multicast.

DDS used to be deployad in system where *robustness and availability* is considered critical. It's required from system to continue operating even if partial fail occures, so centralized services reprezenting single point of failure have to be avoided, DomainParticipant components have to be self-contained to be able to operate securely, multi-party

key agreement protocols should be avoided because of simplicity of discruption and tokens and keys should be compartmentalized as much as possible to avoid situations where multiple applications using same key are compromised if just one of them is compromised.

7.5 Implementation

The implementation of DDS Security into ORTE consists of changes in Modules (presented in chapter 8 of [1]). Introduction of *SecureSubMsg* Submessage and *SecuredPayload* Submessage Element assumes modification of Message Module, Discovery Module is affected by *Builtin Secure Endpoints* - if configured to, discovery of *publishers* and *subscribers* is secured and Behavior Module needs to be modified to include *Builtin Plugins* which ensures security.

7.5.1 Builtin Endpoints

This is the list of *Builtin Endpoints* presented in chapter 7.4.5 of [7]. In order to ensure out-of-the-box compatibility, following *Builtin Secure Endpoints* needs to be implemented:

- SEDPbuiltinPublicationsSecureWriter
- SEDPbuiltinPublicationsSecureReader
- SEDPbuiltinSubscriptionsSecureWriter
- SEDPbuiltinSubscriptionsSecureReader
- BuiltinParticipantMessageSecureWriter
- BuiltinParticipantMessageSecureReader
- BuiltinParticipantVolatileMessageSecureWriter
- BuiltinParticipantVolatileMessageSecureReader

7.5.2 Builtin Plugins

This is resume of *Builtin Plugins* presented in chapter 9.1 of [7]. In order to ensure out-of-the-box compatibility, following *Builtin Plugins* needs to be implemented:

- DDS:Auth:PKI-RSA/DSA-DH (Authentication plugin)
  - Uses PKI with pre-configured shared Certificate Authority
  - RSA or DSA and Diffie-Hellman for authentication and key exchange
- DDS:Access:PKI-Signed-XML-Permissions (Access control plugin)
  - Permissions document signed by shared Certificate Authority
- DDS:Crypto:AES-CTR-HMAC-RSA/DSA-DH (Cryptographic plugin)
  - AES128 for encryption (counter mode)
  - SHA1 and SHA256 for digest
  - HMAC-SHA1 and HMAC-256 for HMAC
- DDS:Tagging:DDS_Discovery (Data Tagging plugin)
  - Send Tags via Endpoint Discovery
- DDS:Logging:DDS_LogTopic (Logging plugin)
  - Logs security events to a dedicated DDS Log Topic

[concl]

Chapter 8
Conclusion

References

[OMG:DDSI-RTPS22] [1] Object Management Group (OMG). *The Real-Time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification* . 2014 .
http://www.omg.org/spec/DDSI-RTPS/2.2/ .

[OMG:DDS] [2] Object Management Group (OMG). *Data Distribution Service for Real-Time Systems* . 2007 .
http://www.omg.org/spec/DDS/ .

[www:OMG] [3] *Object Management Group* .
http://www.omg.org/index.htm .

[FEE:ORTE] [4] *ORTE - Open Real-Time Ethernet* .
http://orte.sourceforge.net/ .

[FEE:vajnar-bc] [5] Martin Vajnar. *ORTE communication middleware for Android OS* . 2014 .

[www:tcp-ip] [6] *Core Protocols in the Internet Protocol Suite* .
http://tools.ietf.org/id/draft-baker-ietf-core-04.html .

[OMG:DDS-SECURITY] [7] Object Management Group (OMG). *DDS Security*. 2014 .
http://www.omg.org/spec/DDS-SECURITY/ .

Appendix A
Symbols

AES	Advanced Encryption Standard
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
CTR	Counter
DCPS	Data-Centric Publish-Subscribe
DDS	Data Distribution Service
DH	Diffie-Hellman
DSA	Digital Signature Algorithm
HMAC	Hash-based Message Authentication Code
IP	Internet Protocol
IPsec	IP Security
LSB	Least Significant Bit
MAC	Message Authentication Code
MSB	Most Significant Bit
MTU	Maximum Transmission Unit
OMG	Object Management Group
ORTE	Open Real-Time Ethernet
PIM	Platform Independent Model
PKI	Public Key Infrastructure
PSM	Platform Specific Model
RSA	Rivest Shamir Adleman
RTPS	Real-Time Publish-Subscribe
SEDP	Simple Endpoint Discovery Protocol
SHA	Secure Hash Algorithm
SPDP	Simple Participant Discovery Protocol
SPI	Service Plugin Interface
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
XML	EXtensible Markup Language

Requests for correction

[rfc-1] obrazek