MicroXRCE-DDS Documentation

Release 1.1.0

eProsima

Jul 16, 2019

General

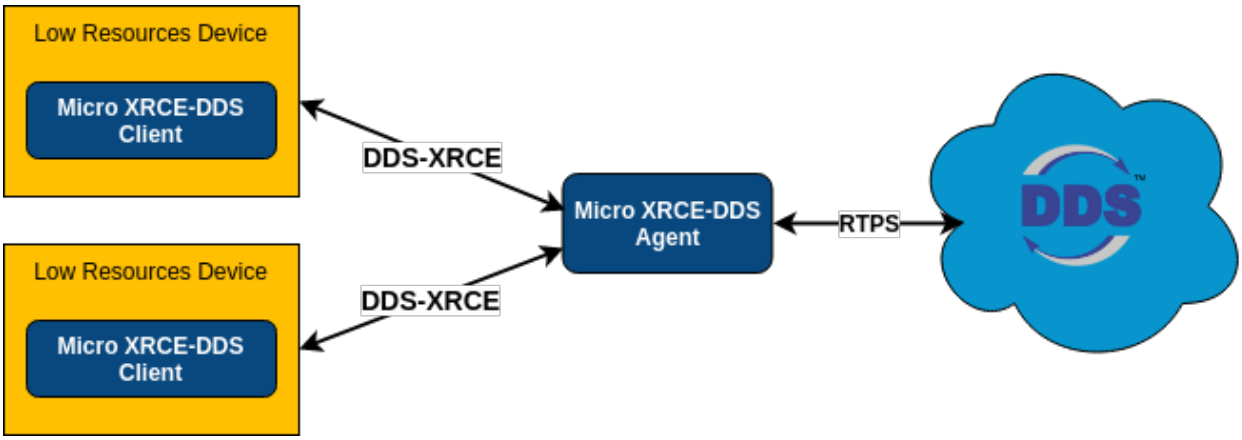
CHAPTER 1

eProsima Micro XRCE-DDS

eProsima Micro XRCE-DDS is a software solution which allows communicating eXtremely Resource Constrained Environments (XRCEs) with an existing DDS network. This implementation complies with the specification proposal, "DDS for eXtremely Resource Constrained Environments" submitted to the Object Management Group (OMG) consortium.

eProsima Micro XRCE-DDS implements a client-server protocol to enable resource-constrained devices (clients) to take part in DDS communications. *eProsima Micro XRCE-DDS Agent* (server) makes possible this communication acting on behalf of the *eProsima Micro XRCE-DDS Clients* and enables them to take part as DDS publishers and/or subscribers in the DDS Global Data Space.

eProsima Micro XRCE-DDS provides both, a plug and play *eProsima Micro XRCE-DDS Agent* and an API layer which allows you to implement your *eProsima Micro XRCE-DDS Clients*.



1.1 Main Features

High performance. Uses a static low-level serialization library (eProsima Micro CDR) that serialize in XCDR.

Low resouces. The client library is dynamic memory free and static memory free. This means that the only memory charge is due to the stack growth. It can manage a simple publisher/subscriber with less of 2KB of RAM. Besides, the client is built with a *profiles* concept, that allows adding or removing functionality to the library changing the binary size.

Multi-platform. The OS dependencies are treated as pluggable modules. The user can easily implement his platform modules to *eProsima Micro XRCE-DDS Client* library in his specific platform. By default, the project can run over *Linux*, *Windows* and *Nuttx*.

Compiler dependencies free. The client library uses pure c99 standard. No *C* compiler extensions are used.

Free and Open Source. The client library, the agent executable, the generator tool and internal dependencies as *eProsima Micro CDR* or *eProsima Fast RTPS* are all of them free and open source.

Easy to use. The project comes with examples of a publisher and a subscriber, an example of how *eProsima Micro XRCE-DDS* is deployed into a stage and an interactive demo that can be used with the ShapesDemo with the purpose of understanding the DDS-XRCE protocol and making tests. The client API is thoroughly explained, and a guided example of how to create your client application is distributed as part of the documentation.

Implementation of the DDS-XRCE standard. DDS-XRCE is a standard communication protocol of OMG consortium focused on communicating eXtremely Resource Constrained Environments with the DDS world.

Best effort and reliable communication. *eProsima Micro XRCE-DDS* supports both, *best effort* for fast and light communication and *reliable* when the communication reliability is needed.

Pluggable tansport layer. *eProsima Micro XRCE-DDS* is not built over a specific transport protocol as *Serial* or *UDP*. It is agnostic about the transport used, and give the user the possibility of implementing easily his tailored transport. By default, *UPD*, *TCP*, and *Serial* transports are provided.

Commercial support Available at *support@eprosima.com*

1.2 Installation

To install *eProsima Micro XRCE-DDS*, follow the instructions of *Installation* page.

1.3 User manual

To test *eProsima Micro XRCE-DDS* in your computer, you can follow the *Quick start* instructions. This page shows how to create a simple publisher as a XRCE client that send topics into the DDS World.

Additionally, there is an interactive example called *Shapes Demo* that allow you to create entities and to send/receive topics by instructions given by command line. This example is useful to understand how XRCE protocol works along to the DDS World.

To create your own client, you can follow the instructions of *Getting started* page. This is a tutorial that describe briefly how *eProsima Micro XRCE-DDS* API is and how it works.

To know more about *eProsima Micro XRCE-DDS*, you can read the corresponding parts to the *eProsima Micro XRCE-DDS Client* or the *eProsima Micro XRCE-DDS Agent*. If you are interested in how XRCE works, read *Entities* and *Operations* pages.

1.4 eProsima Micro XRCE-DDS Gen

To create a serialization/deserialization topic code for *eProsima Micro XRCE-DDS Client* and make easy the built of examples using thoses topics, there is a tool called *microxrceddsgen*. Information about this tool can be found in

eProsima Micro XRCE-DDS Gen page.

CHAPTER 2

Installation manual

2.1 External dependencies

2.1.1 Required dependences

eProsima Micro XRCE-DDS Client does not require external dependencies.

eProsima Micro XRCE-DDS Agent requires the following packages to work:

eProsima Fast RTPS *eProsima Fast RTPS* could be installed following the instructions: eProsima Fast RTPS installation guide.

Windows

Microsoft Visual C++ 2015 or 2017 *eProsima Micro XRCE-DDS* is supported on Windows over Microsoft Visual C++ 2015 and 2017 framework.

2.1.2 Additional Dependencies

The following dependences are not necessary to run *eProsima Micro XRCE-DDS*.

GTEST Gtest is the framework used to test the code through a complete set of tests.

Java & Gradle Java & Gradle is required to compile the code generation tool *eProsima Micro XRCE-DDS Gen*.

2.2 Installation

To compile and install the Client and the Agent modules, we use CMake.

2.2.1 Installing Agent and Client

Clone the project from GitHub:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS.git
$ cd Micro-XRCE-DDS
$ mkdir build && cd build
```

On Linux, inside of `build` folder, execute the following commands:

```
$ cmake ..
$ make
$ sudo make install
```

On Windows choose the Visual Studio version using the CMake option *-G*, for example:

```
$ cmake -G "Visual Studio 14 2015 Win64" ..
$ cmake --build . --target install
```

Now you have *eProsima Micro XRCE-DDS Agent* and *eProsima Micro XRCE-DDS Client* installed in your system.

2.2.2 Installing the Agent stand-alone

Clone the project from GitHub:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Agent.git
$ cd Micro-XRCE-DDS-Agent
$ mkdir build && cd build
```

On Linux, inside of `build` folder, execute the following commands:

```
$ cmake ..
$ make
$ sudo make install
```

On Windows first select the Visual Studio version:

```
$ cmake -G "Visual Studio 14 2015 Win64" ..
$ cmake --build .
$ cmake --build . --target install
```

Now you have the executable *eProsima Micro XRCE-DDS Agent* installed in your system. Before running it, you need to add `/usr/local/lib` to the dynamic loader-linker directories.

```
sudo ldconfig /usr/local/lib/
```

2.2.3 Installing the Client stand-alone

Clone the project from GitHub:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Client.git
$ cd Micro-XRCE-DDS-Client
$ mkdir build && cd build
```

On Linux, inside of `build` folder, execute the following commands:

```
$ cmake ..
$ make
$ sudo make install
```

On Windows first select the Visual Studio version:

```
$ cmake -G "Visual Studio 14 2015 Win64" ..
$ cmake --build .
$ cmake --build . --target install
```

If you want to install the *eProsima Micro XRCE-DDS Client* examples, you can add `-DUCLIENT_BUILD_EXAMPLES=ON` to the CMake command line options. This flag will enable the compilation of the examples when the project is compiled. There are several CMake definitions for configuring the build of the client library at compile time. You can found them in *eProsima Micro XRCE-DDS Client* page under *Configuration* section.

2.2.4 Compiling an app with the Client library

For building your Client app you need to build against the following libs:

```
gcc <your_main.c> -lmicrocdr -lmicroxrcedds_client
```

CHAPTER 3

User manual

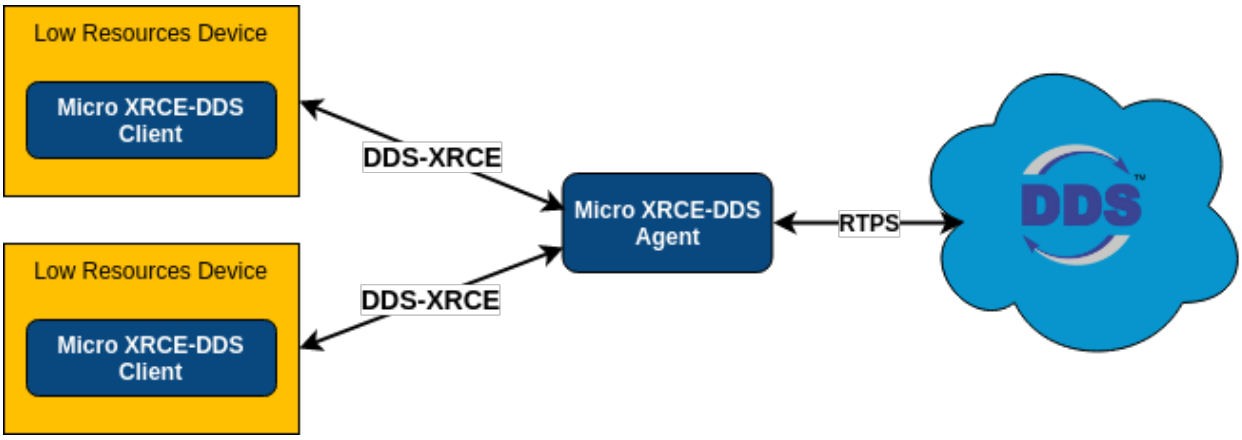
3.1 Introduction

3.1.1 DDS-XRCE protocol

eProsima Micro XRCE-DDS implements DDS-XRCE protocol specified in the *DDS for eXtremely Resource Constrained Environments* proposal submitted to the *Object Management Group (OMG)* consortium. This protocol allows resource constrained devices to communicate with a larger *DDS (Data Distribution Service)* network. This communication is done using a client-server architecture, where the server (Agent) acts as an intermediary between clients and *DDS Global Data Space*.

DDS-XRCE protocol defines a wire protocol between XRCE Agents and XRCE Clients. The messages exchanged revolve around operations and their responses. XRCE Clients ask the XRCE Agents to run operations and the XRCE Agents reply with the result of those requested operations. Making use of those operations, XRCE Clients are able to create the DDS entities hierarchy necessary to publish and/or receive data from DDS. DDS entities are created and stored on the XRCE Agent side so the XRCE Clients can reuse them at will.

eProsima Micro XRCE-DDS implements the DDS-XRCE protocol using a *eProsima Micro XRCE-DDS Agent* as server and providing a C API for developing XRCE Clients applications. *eProsima Micro XRCE-DDS Agent* uses *eProsima Fast RTPS* to reach the DDS Global Data Space.



Fast RTPS

eProsima Fast RTPS is a C++ implementation of the *RTPS (Real-Time Publish-Subscribe)* protocol, which provides publisher-subscriber communications over unreliable transports such as UDP, as defined and maintained by the *OMG* consortium. XRCE-DDS is also the wire interoperability protocol defined for the *DDS* standard, again by the *OMG*.

For deeper information please refer to the official documentation: eProsima Fast RTPS

3.1.2 Operations and entities

eProsima Micro XRCE-DDS communication between XRCE Client and XRCE Agent is based upon *Operations* and responses. Clients request operations to the Agent which will generate responses with the result of these operations. Clients may process the responses to know how an operations request has ended in the Agent.

eProsima Micro XRCE-DDS Client can request a variety of operations to the *eProsima Micro XRCE-DDS Agent*:

- Create and delete sessions.
- Create and delete entities.
- Write and read Data.

First of all, the *Agent* must know about any *Client*. This is done by sending a *Create session* operation from the *Client* to the *Agent*, which will register the *Client*. If you do not register the session, all the operations sending to the *Agent* will be refused. Once registered, the *Client* can request operations to the *Agent*. The *Client* can create and query entities using operations. The communication with *DDS* is handled by *Agent* using *Entities*.

Create entity Operation is the request used to create entities in the *Agent*. Each one of these entities corresponds with a *Fast RTPS* object. The entities you can create are:

- Participants.
- Topics.
- Publisher.
- Subscriber.
- DataWriter.
- DataReader.

For sending and receiving data from/to DDS, *Client* has access to the *DataWriter* and *DataReader* entities. These entities handle the writing/reading operations. For sending and receiving any topic the *Write Data* and *Read Data* operations must be used.

If you want to remove any entity from the *Agent*, you can use *Delete entity* operations. Also, if you want to log out a *Client* session from the *Agent*, you can use the *Delete session* operation.

3.1.3 Topic Type

The data sent by the *Client* to the *DDS Global Data Space* uses the same principles as in *Fast RTPS*. The Interface Definition Language (IDL) is used to define the type and must be known by the *Client*. Having the type defined as *IDL* we provide the *eProsima Micro XRCE-DDS Gen* tool. This tool is able to generate a compatible type that the *eProsima Micro XRCE-DDS Client* can use to send and receive. The type should match the one used on the DDS Side.

3.2 Quick start

Prosima Micro XRCE-DDS provides a C API which allows the creation of *eProsima Micro XRCE-DDS Clients* that publish and/or subscribe to topics from DDS Global Data Space. The following example shows how to create a simple *eProsima Micro XRCE-DDS Client* and *eProsima Micro XRCE-DDS Agent* for publishing and subscribing to the DDS world, using this HelloWorld.idl:

```
struct HelloWorld
{
    unsigned long index;
    string message;
};
```

First of all, we launch the *Agent*. For this example, the *Client* - *Agent* communication will be done through UDP:

```
$ cd /usr/local/bin && MicroXRCEAgent udp -p 2019 -r <references-file>
```

Along with the *Agent*, the *PublishHelloWorldClient* example provided in the source code is launched. This *Client* example will publish in the DDS World the HelloWorld topic.

```
$ examples/uxr/client/PublishHelloWorld/PublishHelloWorldClient 127.0.0.1 2019
```

The code of the *PublishHelloWorldClient* is the following:

```
// Copyright 2019 Proyectos y Sistemas de Mantenimiento SL (eProsima).
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#include "HelloWorld.h"

#include <uxr/client/client.h>
#include <ucdr/microcdr.h>

#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
#include <string.h> //strcmp
#include <stdlib.h> //atoi

#define STREAM_HISTORY  8
#define BUFFER_SIZE     UXR_CONFIG_UDP_TRANSPORT_MTU * STREAM_HISTORY

int main(int args, char** argv)
{
    // CLI
    if(3 > args || 0 == atoi(argv[2]))
    {
        printf("usage: program [-h | --help] | ip port [<topics>]\n");
        return 0;
    }

    char* ip = argv[1];
    uint16_t port = (uint16_t)atoi(argv[2]);
    uint32_t max_topics = (args == 4) ? (uint32_t)atoi(argv[3]) : UINT32_MAX;

    // Transport
    uxrUDPTransport transport;
    uxrUDPPlatform udp_platform;
    if(!uxr_init_udp_transport(&transport, &udp_platform, ip, port))
    {
        printf("Error at create transport.\n");
        return 1;
    }

    // Session
    uxrSession session;
    uxr_init_session(&session, &transport.comm, 0xAAAABBBB);
    if(!uxr_create_session(&session))
    {
        printf("Error at create session.\n");
        return 1;
    }

    // Streams
    uint8_t output_reliable_stream_buffer[BUFFER_SIZE];
    uxrStreamId reliable_out = uxr_create_output_reliable_stream(&session, output_
↪reliable_stream_buffer, BUFFER_SIZE, STREAM_HISTORY);

    uint8_t input_reliable_stream_buffer[BUFFER_SIZE];
    uxr_create_input_reliable_stream(&session, input_reliable_stream_buffer, BUFFER_
↪SIZE, STREAM_HISTORY);

    // Create entities
    uxrObjectId participant_id = uxr_object_id(0x01, UXR_PARTICIPANT_ID);
    const char* participant_xml = "<dds>"
                                      "<participant>"
                                          "<rtps>"
                                              "<name>default_xrce_participant</name>"
                                          "</rtps>"
                                      "</participant>"
                                  "</dds>";
    uint16_t participant_req = uxr_buffer_create_participant_xml(&session, reliable_
↪out, participant_id, 0, participant_xml, UXR_REPLACE);
```

(continues on next page)

(continued from previous page)

```
    uxrObjectId topic_id = uxr_object_id(0x01, UXR_TOPIC_ID);
    const char* topic_xml = "<dds>"
                                "<topic>"
                                    "<name>HelloWorldTopic</name>"
                                    "<dataType>HelloWorld</dataType>"
                                "</topic>"
                            "</dds>";
    uint16_t topic_req = uxr_buffer_create_topic_xml(&session, reliable_out, topic_id,
↪ participant_id, topic_xml, UXR_REPLACE);

    uxrObjectId publisher_id = uxr_object_id(0x01, UXR_PUBLISHER_ID);
    const char* publisher_xml = "";
    uint16_t publisher_req = uxr_buffer_create_publisher_xml(&session, reliable_out,␣
↪publisher_id, participant_id, publisher_xml, UXR_REPLACE);

    uxrObjectId datawriter_id = uxr_object_id(0x01, UXR_DATAWRITER_ID);
    const char* datawriter_xml = "<dds>"
                                    "<data_writer>"
                                        "<topic>"
                                            "<kind>NO_KEY</kind>"
                                            "<name>HelloWorldTopic</name>"
                                            "<dataType>HelloWorld</dataType>"
                                        "</topic>"
                                    "</data_writer>"
                                "</dds>";
    uint16_t datawriter_req = uxr_buffer_create_datawriter_xml(&session, reliable_out,
↪ datawriter_id, publisher_id, datawriter_xml, UXR_REPLACE);

    // Send create entities message and wait its status
    uint8_t status[4];
    uint16_t requests[4] = {participant_req, topic_req, publisher_req, datawriter_req}
↪;
    if(!uxr_run_session_until_all_status(&session, 1000, requests, status, 4))
    {
        printf("Error at create entities: participant: %i topic: %i publisher: %i␣
↪darawriter: %i\n", status[0], status[1], status[2], status[3]);
        return 1;
    }

    // Write topics
    bool connected = true;
    uint32_t count = 0;
    while(connected && count < max_topics)
    {
        HelloWorld topic = {count++, "Hello DDS world!"};

        ucdrBuffer mb;
        uint32_t topic_size = HelloWorld_size_of_topic(&topic, 0);
        uxr_prepare_output_stream(&session, reliable_out, datawriter_id, &mb, topic_
↪size);
        HelloWorld_serialize_topic(&mb, &topic);

        connected = uxr_run_session_time(&session, 1000);
        if(connected)
        {
            printf("Sent topic: %s, id: %i\n", topic.message, topic.index);
```

(continues on next page)

(continued from previous page)

```
        }
    }

    // Delete resources
    uxr_delete_session(&session);
    uxr_close_udp_transport(&transport);

    return 0;
}
```

After it, we will launch the *SubscriberHelloWorldClient*. This *Client* example will subscribe to HelloWorld topic from the DDS World.

```
$ examples/uxr/client/SubscriberHelloWorld/SubscribeHelloWorldClient 127.0.0.1 2019
```

The code of the *SubscriberHelloWorldClient* is the following:

```
// Copyright 2019 Proyectos y Sistemas de Mantenimiento SL (eProsima).
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#include "HelloWorld.h"

#include <uxr/client/client.h>
#include <string.h> //strcmp
#include <stdlib.h> //atoi
#include <stdio.h>

#define STREAM_HISTORY  8
#define BUFFER_SIZE     UXR_CONFIG_UDP_TRANSPORT_MTU * STREAM_HISTORY

void on_topic(uxrSession* session, uxrObjectId object_id, uint16_t request_id,␣
↪uxrStreamId stream_id, struct ucdrBuffer* mb, void* args)
{
    (void) session; (void) object_id; (void) request_id; (void) stream_id;

    HelloWorld topic;
    HelloWorld_deserialize_topic(mb, &topic);

    printf("Received topic: %s, id: %i\n", topic.message, topic.index);

    uint32_t* count_ptr = (uint32_t*) args;
    (*count_ptr)++;
}

int main(int args, char** argv)
```

(continues on next page)

(continued from previous page)

```
{
    // CLI
    if(3 > args || 0 == atoi(argv[2]))
    {
        printf("usage: program [-h | --help] | ip port [<topics>]\n");
        return 0;
    }

    char* ip = argv[1];
    uint16_t port = (uint16_t)atoi(argv[2]);
    uint32_t max_topics = (args == 4) ? (uint32_t)atoi(argv[3]) : UINT32_MAX;

    // State
    uint32_t count = 0;

    // Transport
    uxrUDPTransport transport;
    uxrUDPPlatform udp_platform;
    if(!uxr_init_udp_transport(&transport, &udp_platform, ip, port))
    {
        printf("Error at create transport.\n");
        return 1;
    }

    // Session
    uxrSession session;
    uxr_init_session(&session, &transport.comm, 0xCCCCDDDD);
    uxr_set_topic_callback(&session, on_topic, &count);
    if(!uxr_create_session(&session))
    {
        printf("Error at create session.\n");
        return 1;
    }

    // Streams
    uint8_t output_reliable_stream_buffer[BUFFER_SIZE];
    uxrStreamId reliable_out = uxr_create_output_reliable_stream(&session, output_
↪reliable_stream_buffer, BUFFER_SIZE, STREAM_HISTORY);

    uint8_t input_reliable_stream_buffer[BUFFER_SIZE];
    uxrStreamId reliable_in = uxr_create_input_reliable_stream(&session, input_
↪reliable_stream_buffer, BUFFER_SIZE, STREAM_HISTORY);

    // Create entities
    uxrObjectId participant_id = uxr_object_id(0x01, UXR_PARTICIPANT_ID);
    const char* participant_xml = "<dds>"
                                      "<participant>"
                                          "<rtps>"
                                              "<name>default_xrce_participant</name>"
                                          "</rtps>"
                                      "</participant>"
                                  "</dds>";
    uint16_t participant_req = uxr_buffer_create_participant_xml(&session, reliable_
↪out, participant_id, 0, participant_xml, UXR_REPLACE);

    uxrObjectId topic_id = uxr_object_id(0x01, UXR_TOPIC_ID);
    const char* topic_xml = "<dds>"
```

(continues on next page)

(continued from previous page)

```
                                    "<topic>"
                                        "<name>HelloWorldTopic</name>"
                                        "<dataType>HelloWorld</dataType>"
                                    "</topic>"
                                "</dds>";
    uint16_t topic_req = uxr_buffer_create_topic_xml(&session, reliable_out, topic_id,
↪ participant_id, topic_xml, UXR_REPLACE);

    uxrObjectId subscriber_id = uxr_object_id(0x01, UXR_SUBSCRIBER_ID);
    const char* subscriber_xml = "";
    uint16_t subscriber_req = uxr_buffer_create_subscriber_xml(&session, reliable_out,
↪ subscriber_id, participant_id, subscriber_xml, UXR_REPLACE);

    uxrObjectId datareader_id = uxr_object_id(0x01, UXR_DATAREADER_ID);
    const char* datareader_xml = "<dds>"
                                    "<data_reader>"
                                        "<topic>"
                                            "<kind>NO_KEY</kind>"
                                            "<name>HelloWorldTopic</name>"
                                            "<dataType>HelloWorld</dataType>"
                                        "</topic>"
                                    "</data_reader>"
                                "</dds>";
    uint16_t datareader_req = uxr_buffer_create_datareader_xml(&session, reliable_out,
↪ datareader_id, subscriber_id, datareader_xml, UXR_REPLACE);

    // Send create entities message and wait its status
    uint8_t status[4];
    uint16_t requests[4] = {participant_req, topic_req, subscriber_req, datareader_
↪req};
    if(!uxr_run_session_until_all_status(&session, 1000, requests, status, 4))
    {
        printf("Error at create entities: participant: %i topic: %i subscriber: %i␣
↪datareader: %i\n", status[0], status[1], status[2], status[3]);
        return 1;
    }

    // Request topics
    uxrDeliveryControl delivery_control = {0};
    delivery_control.max_samples = UXR_MAX_SAMPLES_UNLIMITED;
    uint16_t read_data_req = uxr_buffer_request_data(&session, reliable_out,␣
↪datareader_id, reliable_in, &delivery_control);

    // Read topics
    bool connected = true;
    while(connected && count < max_topics)
    {
        uint8_t read_data_status;
        connected = uxr_run_session_until_all_status(&session, UXR_TIMEOUT_INF, &read_
↪data_req, &read_data_status, 1);
    }

    // Delete resources
    uxr_delete_session(&session);
    uxr_close_udp_transport(&transport);

    return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

At this moment, the subscriber will receive the topics that are sending by the publisher.

In order to see the messages from the DDS Global Data Space point of view, you can use *eProsima Fast RTPS* HelloWorld example running a subscriber (Fast RTPS HelloWorld):

```
$ cd /usr/local/examples/C++/HelloWorldExample
$ sudo make && cd bin
$ ./HelloWorldExample subscriber
```

3.2.1 Learn More

To learn more about DDS and *eProsima Fast RTPS*: eProsima Fast RTPS

To learn how to install *eProsima Micro XRCE-DDS* read: *Installation*

To learn more about *eProsima Micro XRCE-DDS* read: *Introduction*

To learn more about *eProsima Micro XRCE-DDS Gen* read: *eProsima Micro XRCE-DDS Gen*

3.3 Getting started

This page shows how to get started with the *eProsima Micro XRCE-DDS Client* development. We will create a *Client* that can publish and subscribe a topic. This tutorial has been extract from the examples found into `examples/PublisherHelloWorld` and `examples/SubscriberHelloWorld`.

First, we need to have on the system:

- *eProsima Micro XRCE-DDS Client*.
- *eProsima Micro XRCE-DDS Agent*.
- *eProsima Micro XRCE-DDS Gen*.

3.3.1 Generate from the IDL

We will use HelloWorld as our Topic whose IDL is the following:

```
struct HelloWorld
{
    unsigned long index;
    string message;
};
```

In the *Client* we need to create an equivalent C type with its serialization/deserialization code. This is done automatically by *eProsima Micro XRCE-DDS Gen*:

```
$ microxrceddsgen HelloWorld.idl
```

3.3.2 Initialize a Session

In the source example file, we include the generated type code, in order to have access to its serialization/deserialization functions along to the writing function. Also, we will specify the max buffer for the streams and its historical associated for the reliable streams.

```
#include "HelloWorldWriter.h"

#define STREAM_HISTORY  8
#define BUFFER_SIZE     UXR_CONFIG_UDP_TRANSPORT_MTU * STREAM_HISTORY
```

Before create a Session we need to indicate the transport to use (the *Agent* must be configured for listening from UDP at port 2018).

```
uxrUDPTransport transport;
uxrUDPPlatform udp_platform;
if(!uxr_init_udp_transport(&transport, &udp_platform, "127.0.0.1", 2018))
{
    printf("Error at create transport.\n");
    return 1;
}
```

Next, we will create a session that allows us interact with the *Agent*:

```
uxrSession session;
uxr_init_session(&session, &transport.comm, 0xABCDABCD);
uxr_set_topic_callback(&session, on_topic, NULL);
if(!uxr_create_session(&session))
{
    printf("Error at create session.\n");
    return 1;
}
```

The first function `uxr_init_session` initializes the `session` structure with the transport and the *Client Key* (the session identifier for an *Agent*). The `uxr_set_topic_callback` function is for registering the function `on_topic` which will be called when the *Client* receive a topic. Once the session has been initialized, we can send the first message for login the *Client* in the *Agent* side: `uxr_create_session`. This function will try to connect with the *Agent* by `CONFIG_MAX_SESSION_CONNECTION_ATTEMPTS` attempts (configurable at `client.config`).

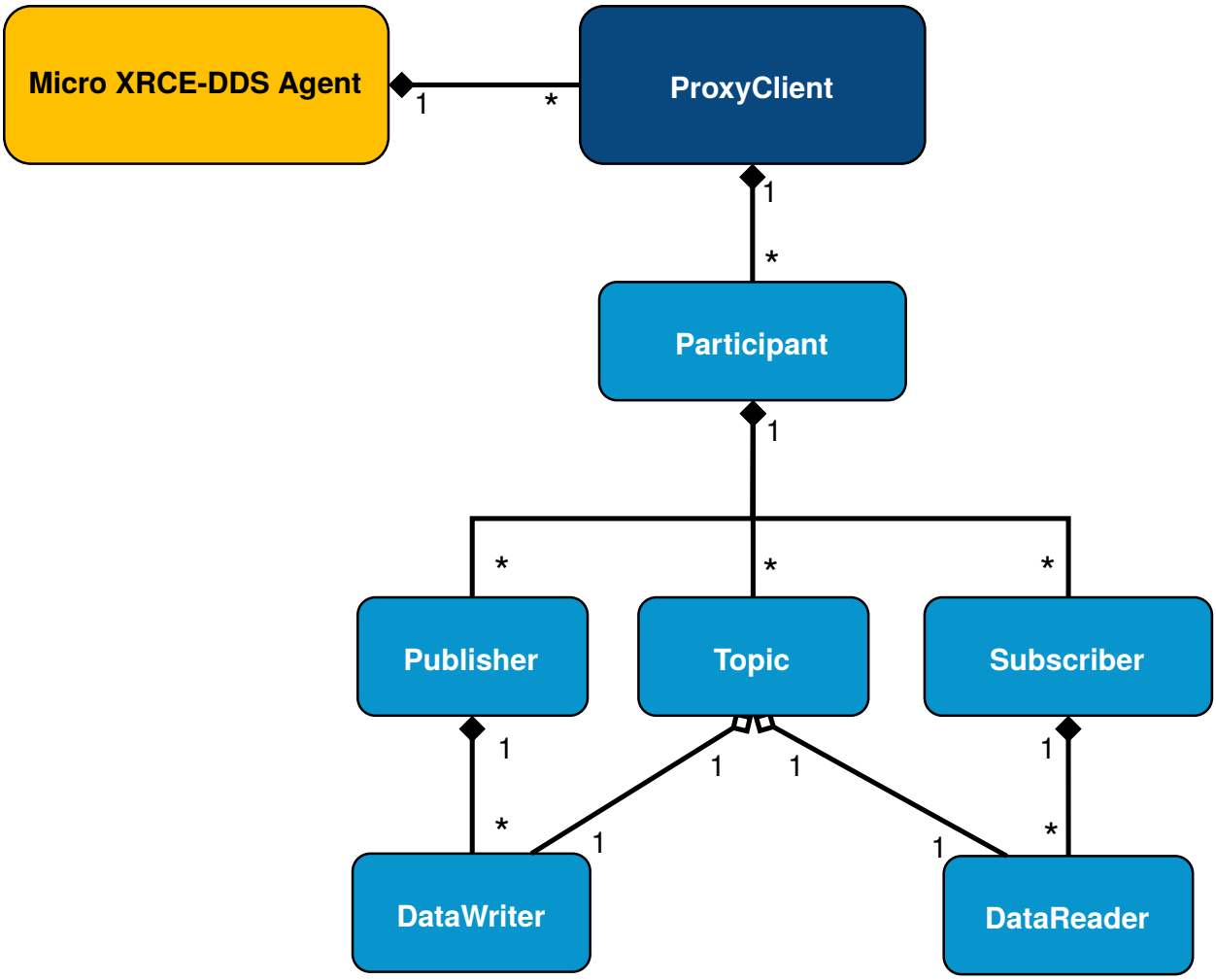
Optionally, we also could add a status callback with the function `uxr_set_status_callback`, but for the purpose of this example we do not need it.

Once we have login the session successful, we can create the streams that we will use. In this case, we will use two, both reliables, for input and output.

```
uint8_t output_reliable_stream_buffer[BUFFER_SIZE];
uxrStreamId reliable_out = uxr_create_output_reliable_stream(&session, output_
↪reliable_stream_buffer, BUFFER_SIZE, STREAM_HISTORY);

uint8_t input_reliable_stream_buffer[BUFFER_SIZE];
uxrStreamId reliable_in = uxr_create_input_reliable_stream(&session, input_reliable_
↪stream_buffer, BUFFER_SIZE, STREAM_HISTORY);
```

In order to publish and/or subscribe a topic, we need to create a hierarchy of XRCE entities in the *Agent* side. These entities will be created from the *Client*.



3.3.3 Setup a Participant

For establishing DDS communication we need to create a *Participant* entity for the *Client* in the *Agent*. We can do this calling *Create participant* operation:

```
uxrObjectId participant_id = uxr_object_id(0x01, UXR_PARTICIPANT_ID);
const char* participant_xml = "<dds>"
                                  "<participant>"
                                      "<rtps>"
                                          "<name>default_xrce_participant</name>"
                                      "</rtps>"
                                  "</participant>"
                              "</dds>";
uint16_t participant_req = uxr_buffer_create_participant_ref(&session, reliable_out,␣
↪participant_id, participant_xml, UXR_REPLACE);
```

In any *XRCE Operation* that creates an entity, an *Object ID* is necessary. It is used to represent and manage the entity in the *Client* side. In this case we will create the entity by its XML description, but also could be done by a reference of the entity in the *Agent*. Each operation, returns a *Request ID*. This identifier of the operation can be used later for associating the status with the operation. In this case, the operation has been written into the stream `reliable_out`. Later, in the `run_session` function, the data written in the stream will be sent to the *Agent*.

3.3.4 Creating topics

Once the *Participant* has been created, we can use *Create topic* operation for register a *Topic* entity within the *Participant*.

```
uxrObjectId topic_id = uxr_object_id(0x01, UXR_TOPIC_ID);
const char* topic_xml = "<dds>"
                            "<topic>"
                                "<name>HelloWorldTopic</name>"
                                "<dataType>HelloWorld</dataType>"
                            "</topic>"
                        "</dds>";
uint16_t topic_req = uxr_buffer_create_topic_xml(&session, reliable_out, topic_id,␣
↪participant_id, topic_xml, UXR_REPLACE);
```

As any other XRCE Operation used to create an entity, an Object ID must be specified to represent the entity. The `participant_id` is the participant where the Topic will be registered. In order to determine which topic will be used, an XML is sent to the *Agent* for creating and defining the Topic in the DDS Global Data Space. That definition consists of a name and a type.

3.3.5 Publishers & Subscribers

Similar to Topic registration we can create *Publishers* and *Subscribers* entities. We create a publisher or subscriber on a participant entity, so it is necessary to provide the ID of the *Participant* which will hold those *Publishers* or *Subscribers*.

```
uxrObjectId publisher_id = uxr_object_id(0x01, UXR_PUBLISHER_ID);
const char* publisher_xml = "";
uint16_t publisher_req = uxr_buffer_create_publisher_xml(&session, reliable_out,␣
↪publisher_id, participant_id, publisher_xml, UXR_REPLACE);

uxrObjectId subscriber_id = uxr_object_id(0x01, UXR_SUBSCRIBER_ID);
const char* subscriber_xml = "";
uint16_t subscriber_req = uxr_buffer_create_subscriber_xml(&session, reliable_out,␣
↪subscriber_id, participant_id, subscriber_xml, UXR_REPLACE);
```

The *Publisher* and *Subscriber* XML information is given when the *DataWriter* and *DataReader* are created.

3.3.6 DataWriters & DataReaders

Analogous to publishers and subscribers entities, we create the *DataWriters* and *DataReaders* entities. These entities are responsible to send and receive the data. *DataWriters* are referred to a publisher, and *DataReaders* are referred to a subscriber. The configuration about how these *DataReaders* and data writers works is contained in the xml.

```
uxrObjectId datawriter_id = uxr_object_id(0x01, UXR_DATAWRITER_ID);
const char* datawriter_xml = "<dds>"
                                 "<data_writer>"
                                     "<topic>"
                                         "<kind>NO_KEY</kind>"
                                         "<name>HelloWorldTopic</name>"
                                         "<dataType>HelloWorld</dataType>"
                                     "</topic>"
                                 "</data_writer>"
                             "</dds>";
```

(continues on next page)

(continued from previous page)

```
uint16_t datawriter_req = uxr_buffer_create_datawriter_xml(&session, reliable_out,␣
↪datawriter_id, publisher_id, datawriter_xml, UXR_REPLACE);

uxrObjectId datareader_id = uxr_object_id(0x01, UXR_DATAREADER_ID);
const char* datareader_xml = "<dds>"
                                 "<data_reader>"
                                     "<topic>"
                                         "<kind>NO_KEY</kind>"
                                         "<name>HelloWorldTopic</name>"
                                         "<dataType>HelloWorld</dataType>"
                                     "</topic>"
                                 "</data_reader>"
                             "</dds>";
uint16_t datareader_req = uxr_buffer_create_datareader_xml(&session, reliable_out,␣
↪datareader_id, subscriber_id, datareader_xml, UXR_REPLACE);
```

3.3.7 Agent response

In operations such as create session, create entity or request data from the *Agent*, a status is sent from the *Agent* to the *Client* indicating what happened.

For *Create session* or *Detele session* operations the status value is stored into the `session.info.last_request_status`. For the rest of the operations, the status are sent to the input reliable stream `0x80`, that is, the first input reliable stream created, with index 0.

The different status values that the *Agent* can send to the *Client* are the following (defined in `uxr/client/core/session/session_info.h`):

```
UXR_STATUS_OK
UXR_STATUS_OK_MATCHED
UXR_STATUS_ERR_DDS_ERROR
UXR_STATUS_ERR_MISMATCH
UXR_STATUS_ERR_ALREADY_EXISTS
UXR_STATUS_ERR_DENIED
UXR_STATUS_ERR_UNKNOWN_REFERENCE
UXR_STATUS_ERR_INVALID_DATA
UXR_STATUS_ERR_INCOMPATIBLE
UXR_STATUS_ERR_RESOURCES
UXR_STATUS_NONE (never send, only used when the status is known)
```

The status can be handled by the `on_status_callback` callback (configured in `uxr_set_status_callback` function) or by the `run_session_until_all_status` as we will see.

```
uint8_t status[6]; // we have 6 request to check.
uint16_t requests[6] = {participant_req, topic_req, publisher_req, subscriber_req,␣
↪datawriter_req, datareader_req};
if(!uxr_run_session_until_all_status(&session, 1000, requests, status, 6))
{
    printf("Error at create entities\n");
    return 1;
}
```

The `run_session` functions are the main functions of the *eProsima Micro XRCE-DDS Client* library. They perform serveral tasks: send the stream data to the *Agent*, listen data from the *Agent*, call callbacks, and manage the reliable connection. There are five variations of `run_session` function: - `uxr_run_session_time`

- uxr_run_session_until_timeout - uxr_run_session_until_confirmed_delivery - uxr_run_session_until_all_status - uxr_run_session_until_one_status

Here we use the uxr_run_session_until_all_status variation that will perform these actions until all status have been confirmed or the timeout has been reached. This function will return true in case all status were *OK*. After call this function, the status can be read from the status array previously declared.

3.3.8 Write Data

Once we have created a valid data writer entity, we can write data into the DDS Global Data Space using the writing operation. For creating a message with data, first we must decide which stream we want to use, and write that topic in this stream.

```
HelloWorld topic = {count++, "Hello DDS world!"};

ucdrBuffer ub;
uint32_t topic_size = HelloWorld_size_of_topic(&topic, 0);
(void) uxr_prepare_output_stream(&session, reliable_out, datawriter_id, &ub, topic_
↪size);
(void) HelloWorld_serialize_topic(&ub, &topic);

uxr_run_session_until_confirmed_delivery(&session, 1000);
```

HelloWorld_size_of_topic and HelloWorld_serialize_topic functions are automatically generated by *eProsima Micro XRCE-DDS Gen* from the IDL. The function uxr_prepare_output_stream requests a writing for a topic of topic_size size into the reliable stream represented by reliable_out, with a datawriter_id (correspond to the data writer entity used for sending the data in the *DDS World*). If the stream is available and the topic fits in it, the function will initialize the ucdrBuffer structure ub. Once the ucdrBuffer is prepared, the topic can be serialized into it. We are careless about uxr_prepare_output_stream return value because the serialization only will occur if the ucdrBuffer is valid.

After the writing function, as happened with the creation of entities, the topic has been serialized into the buffer but it has not been sent yet. To send the topic is necessary call to a run_session function. In this case, we call to uxr_run_session_until_confirmed_delivery that will wait until the message was confirmed or until the timeout has been reached.

3.3.9 Read Data

Once we have created a valid *DataReader* entity, we can read data from the DDS Global Data Space using the read operation. This operation configures how the *Agent* will send the data to the *Client*. Current implementation sends one topic to the *Client* for each read data operation of the *Client*.

```
uxrDeliveryControl delivery_control = {0};
delivery_control.max_samples = UXR_MAX_SAMPLES_UNLIMITED;

uint16_t read_data_req = uxr_buffer_request_data(&session, reliable_out, datareader_
↪id, reliable_in, &delivery_control);
```

In order to configure how the *Agent* will send the topic, we must set the input stream. In this case, we use the input reliable stream previously defined. datareader_id corresponds with the *DataDeader* entity used for receiving the data. The delivery_control parameter is optional, and allows specifying how the data will be delivered to the *Client*. For the example purpose, we set it as *unlimited*, so any number HelloWorld topic will be delivered to the *Client*.

The run_session function will call the topic callback each time a topic will be received from the *Agent*.

```
void on_topic(uxrSession* session, uxrObjectId object_id, uint16_t request_id,␣
↪uxrStreamId stream_id, struct ucdrBuffer* ub, void* args)
{
    (void) session; (void) object_id; (void) request_id; (void) stream_id; (void)␣
↪args;

    HelloWorld topic;
    HelloWorld_deserialize_topic(ub, &topic);
}
```

To know which kind of Topic has been received, we can use the `object_id` parameter or the `request_id`. The `id` of the `object_id` corresponds to the *DataReader* that has read the Topic, so it can be useful to discretize among diferent topics. The `args` argument correspond to user free data, that has been given at *uxr_set_status_callback* function.

3.3.10 Closing the Client

To close a *Client*, we must perform two steps. First, we need to tell the *Agent* that the session is no longer available. This is done sending the next message:

```
uxr_delete_session(&session);
```

After this, we can close the transport used by the session.

```
uxr_close_udp_transport(&transport);
```

3.4 Shapes Demo

ShapesDemo is an interactive example for testing how *eProsima Fast RTPS* working in the *DDS Global Data Space*. Because *eProsima Micro XRCE-DDS* aims to connect an *XRCE Client* to the *DDS World*, in this example, we will create a *Client* which will interact with the *Shapes Demo*. It can be found at *examples/uxr/client/ShapeDemoClient* inside of the installation directory. This interactive *Client* waits for user input indicating commands to execute.

The available commands are the following:

create_session Creates a Session, if exists, reuse it.

create_participant <participant id>: Creates a Participant on the current session.

create_topic <topic id> <participant id>: Registers a Topic using <participant id> participant.

create_publisher <publisher id> <participant id>: Creates a Publisher on <participant id> participant.

create_subscriber <subscriber id> <participant id>: Creates a Subscriber on <participant id> participant.

create_datawriter <datawriter id> <publisher id>: Creates a DataWriter on the publisher <publisher id>.

create_datareader <datareader id> <subscriber id>: Creates a DataReader on the subscriber <subscriber id>.

write_data <datawriter id> <stream id> [<x> <y> <size> <color>]: Writes data into a <stream id> using <data writer id> DataWriter.

request_data <datareader id> <stream id> <samples>: Reads <sample> topics from a <stream id> using <datareader id> DataReader,

cancel_data <datareader id>: Cancels any previous request data of <datareader id> DataReader.

delete <id_prefix> <type>: Removes object with <id prefix> and <type>.

stream, default_output_stream <stream_id>: Changes the default output stream for all messages except of write data. <stream_id> can be 1-127 for best effort and 128-255 for reliable. The streams must be initially configured.

exit: Closes session and exit.

tree, entity_tree <id>: Creates the necessary entities for a complete publisher and subscriber. All entities will have the same <id> as id.

h, help: Shows this message.

For example, to create a publisher *Client* that sends a square Topic in reliable mode, you need to run the following commands:

```
> create_session
> create_participant 1
> create_topic 1 1
> create_publisher 1 1
> create_datawriter 1 1
> write_data 1 128 200 200 40 BLUE
```

This *Client* will publish a topic in reliable mode that will have color BLUE, x coordinate 200, y coordinate 200 and size 40.

In case of a subscriber *Client* that receives square topics in a reliable mode, run the following:

```
> create_session
> create_participant 1
> create_topic 1 1
> create_subscriber 1 1
> create_datareader 1 1
> request_data 1 128 5
```

This *Client* will receive 5 topics in reliable mode.

To create the entities tree easily, you can run the command `entity_tree <id>`. For example, the following command creates the necessary entities for publishing and subscribing data with id *3*:

```
> entity_tree 3
create_participant 3
create_topic 3 3
create_publisher 3 3
create_subscriber 3 3
create_datawriter 3 3
create_datareader 3 3
```

To modify the output default stream, you can change it with *stream* <*id*>.

The maximum available streams corresponds with the `CONFIG_MAX_OUTPUT_BEST_EFFORT_STREAMS` and `CONFIG_MAX_OUTPUT_RELIABLE_STREAMS` properties configurable in `client.config` file.

> stream 1

Now the messages will be sent in best-effort mode.

3.5 eProsima Micro XRCE-DDS Client

In *eProsima Micro XRCE-DDS*, a *Client* can communicate with DDS Network as any other DDS actor could do. *Clients* can publish and subscribe to data Topics in the DDS Global Data Space.

eProsima Micro XRCE-DDS provides you with a C API to create *eProsima Micro XRCE-DDS Clients* application. All functions needed to set up the *Client* can be found into `client.h` header. This is the only header you need to include.

3.5.1 Profiles

The *Client* library follows a profile concept that enables to choose, add or remove some features in configuration time. This allows to customize the *Client* library size, if there are features that are not used. The profiles can be chosen in `client.config` and start with the prefix `PROFILE`. As part of these profiles, you can choose between several transport layers. Communication with the *Agent* is done through the transport you choose.

The implementation of the transport depends on the platform. The next tables show the current implementation.

Transport	Linux	Windows	NuttX
UDP	X	X	X
TCP	X	X	X
Serial	X		X

The addition of a new transport or an existant transport for a new platform can be easily implemented setting the callbacks of a `uxrCommunication` structure. See the current transport implementations as an example for new custom transport.

3.5.2 Configuration

There are several definitions for configuring and building of the *Client* library at **compile time**. These definitions allow you to create a version of the library according to your requirements. These definitions can be modified at `client.config` file. For incorporating the changes to your project, is necessary to run the `cmake` command every time the definitions change.

`PROFILE_CREATE_ENTITIES_REF=<bool>` Enables or disables the functions related to create entities by reference.

`PROFILE_CREATE_ENTITIES_XML=<bool>` Enables or disables the functions related to create entities by XML.

`PROFILE_READ_ACCESS=<bool>` Enables or disables the functions related to read topics.

`PROFILE_WRITE_ACCESS=<bool>` Enables or disables the functions related to write topics.

`PROFILE_DISCOVERY=<bool>` Enables or disables the functions of the discovery feature (currently, only for Linux).

`PROFILE_UDP_TRANSPORT=<bool>` Enables or disables the posibility to connect with the *Agent* by UDP.

`PROFILE_TCP_TRANSPORT=<bool>` Enables or disables the posibility to connect with the *Agent* by TCP.

`PROFILE_SERIAL_TRANSPORT=<bool>` Enables or disables the posibility to connect with the *Agent* by Serial.

`CONFIG_MAX_OUTPUT_BEST_EFFORT_STREAMS=<number>` Configures the maximun output best-effort streams that a session could have. The calls to `uxr_create_output_best_effort_stream` function for a session must be less or equal that this value.

`CONFIG_MAX_OUTPUT_RELIABLE_STREAMS=<number>` Configures the maximun output reliable streams that a session could have. The calls to `uxr_create_output_reliable_stream` function for a session must be less or equal that this value.

CONFIG_MAX_INPUT_BEST_EFFORT_STREAMS=<number> Configures the maximun input best-effort streams that a session could have. The calls to `uxr_create_input_best_effort_stream` function for a session must be less or equal that this value.

CONFIG_MAX_INPUT_RELIABLE_STREAMS=<number> Configures the maximun input reliable streams that a session could have. The calls to `uxr_create_input_reliable_stream` function for a session must be less or equal that this value.

CONFIG_MAX_SESSION_CONNECTION_ATTEMPTS=<number> This value indicates the number of attempts that `create_session` and `delete_session` will perform until receiving a status message. After a failed attempt, the wait time for the next attempt will be double.

CONFIG_MIN_SESSION_CONNECTION_INTERVAL=<number> This value represents how long it will take to send a new `create_session` or `delete_session` if the first sent has not answered. The wait time for each attempt will be double until reach `CONFIG_MAX_SESSION_CONNECTION_ATTEMPTS`. It is measured in milliseconds.

CONFIG_MIN_HEARTBEAT_TIME_INTERVAL=<number> Into a reliable communication, this value represents how long it will take the first heartbeat to be sent. The wait time for the next heartbeat will be double. It is measured in milliseconds.

CONFIG_BIG_ENDIANNESS=<bool> This value must be correspond to the memory endianness of the device in which the *Client* is running. *FALSE* implies that the machine is little endian and *TRUE* implies big endian.

CONFIG_UDP_TRANSPORT_MTU=<number> This value corresponds to the *Maximum Transmission Unit* able to send and receive by UDP. Internally a buffer is created with this size.

CONFIG_TCP_TRANSPORT_MTU=<number> This value corresponds to the *Maximum Transmission Unit* able to send and receive by TCP. Internally a buffer is created with this size.

CONFIG_SERIAL_TRANSPORT_MTU=<number> This value corresponds to the *Maximum Transmission Unit* able to send and receive by Serial. Internally a buffer is created proportional to this size.

3.5.3 Streams

The client communication is performed by streams. The streams can be seen as communication channels. There are two types of streams: best-effort and reliable streams and you can create several of them.

- Best-effort streams will send and receive the data leaving the reliability to the transport layer. As a result, the best effort streams consume fewer resources than a reliable stream. Also, the message size sent or received by a best-effort stream must be less or equal than the *MTU* defined in the transport used.
- Reliable streams perform the communication without lost regardless of the transport layer and allow message fragmentation in order to send and receive messages longer than the *MTU*.

  To avoid message losses, the reliable streams use additional messages to confirm the delivery, along to a history of the messages sent and received. The history is used to store messages that can not be currently processed because of several reasons such us: delivery order, incomplete fragments or messages that can not be confirmed yet. If the history is full:

  – The messages that will be written to the agent will be discarded until the history get space to store them. So, the user must wait to write in those streams (they can be considered blocked).

  – The messages received from the agent will be discarded. The library will try to recover the discarded messages requesting them to the agent (increasing the bandwidth consumption in that process).

  For that, a low history causes more messages to be discarded, increasing the data traffic because they need to be sent again. A long history will reduce the data traffic of confirmation messages in transports with a high loss rate. This internal management of the communication implies that a reliable stream is more expensive than

best-effort streams, in both, memory and bandwidth, but is possible to play with these values using the history size.

The streams are maybe the highest memory load part of the application. For that, the choice of a right configuration for the application purpose is highly recommendable, especially when the target is a limited resource device. The *Memory optimization* page explain more about how to archive this.

3.5.4 API

As a nomenclature, *eProsima Micro XRCE-DDS Client* API uses a `uxr_` prefix in all of their public API functions and `uxr` prefix in the types. In constants values an `UXR_` prefix is used. The functions belonging to the public interface of the library are only those with the tag `UXRDDLAPI` in their declarations.

Session

These functions are available even if no profile has been enabled in `client.config` file. The declaration of these function can be found in `uxr/client/core/session/session.h`.

```
void uxr_init_session(uxrSession* session, uxrCommunication* comm, uint32_t key);
```

Initializes a session structure. Once this function is called, a `create_session` call can be performed.

session Session structure where manage the session data.

key The key identifier of the *Client*. All *Clients* connected to an *Agent* must have different key.

comm Communication used for connecting to the *Agent*. All different transports have a common attribute uxrCommunication. This parameter can not be shared between active sessions.

```
void uxr_set_status_callback(uxrSession* session, uxrOnStatusFunc on_status_func,␣
↪void* args);
```

Assigns the callback for the *Agent* status messages.

session Session structure previously initialized.

on_status_func Function callback that will be called when a valid status message comes from the *Agent*.

args User pointer data. The args will be provided to `on_status_func` function.

```
void uxr_set_topic_callback(uxrSession* session, uxrOnTopicFunc on_topic_func, void*␣
↪args);
```

Assigns the callback for topics. The topics will be received only if a `request_data` function has been called.

session Session structure previously initialized.

on_status_func Function callback that will be called when a valid data message comes from the *Agent*.

args User pointer data. The args will be provided to `on_topic_func` function.

```
bool uxr_create_session(uxrSession* session);
```

Creates a new session with the *Agent*. This function logs in a session, enabling any other XRCE communication with the *Agent*.

session Session structure previously initialized.

```
bool uxr_delete_session(uxrSession* session);
```

Deletes a session previously created. All *XRCE* entities created with the session will be removed. This function logs out a session, disabling any other *XRCE* communication with the *Agent*.

session Session structure previously initialized.

```
uxrStreamId uxr_create_output_best_effort_stream(uxrSession* session, uint8_t* buffer,
↪ size_t size);
```

Creates and initializes an output best-effort stream for writing. The `uxrStreamId` returned represents the new stream and can be used to manage it. The number of available calls to this function must be less or equal than `CONFIG_MAX_OUTPUT_BEST_EFFORT_STREAMS` value of the `client.config` file.

session Session structure previously initialized.

buffer Memory block where the messages will be written.

size Buffer size.

```
uxrStreamId uxr_create_output_reliable_stream(uxrSession* session, uint8_t* buffer,␣
↪size_t size, size_t history);
```

Creates and initializes an output reliable stream for writing. The `uxrStreamId` returned represents the new stream and can be used to manage it. The number of available calls to this function must be less or equal than `CONFIG_MAX_OUTPUT_RELIABLE_STREAMS` value of the `client.config` file.

session Session structure previously initialized.

buffer Memory block where the messages will be written.

size Buffer size.

history History used for the reliable connection. The buffer size will be splited into smaller buffers using this value. The history must be a power of two.

```
uxrStreamId uxr_create_input_best_effort_stream(uxrSession* session);
```

Creates and initializes an input best-effort stream for receiving messages. The `uxrStreamId` returned represents the new stream and can be used to manage it. The number of available calls to this function must be less or equal than `CONFIG_MAX_INPUT_BEST_EFFORT_STREAMS` value of the `client.config` file.

session Session structure previously initialized.

```
uxrStreamId uxr_create_input_reliable_stream(uxrSession* session, uint8_t* buffer,␣
↪size_t size, size_t history);
```

Creates and initializes an input reliable stream for receiving messages. The returned `uxrStreamId` represents the new stream and can be used to manage it. The number of available calls to this function must be less or equal than `CONFIG_MAX_INPUT_RELIABLE_STREAMS` value of the `client.config` file.

session Session structure previously initialized.

buffer Memory block where the messages will be storaged.

size Buffer size.

history History used for the reliable connection. The buffer will be splited into smaller buffers using this value. The history must be a power of two.

```
void uxr_flash_output_streams(uxrSession* session);
```

Flashes all output streams sending the data through the transport.

session Session structure previously initialized.

```
void uxr_run_session_time(uxrSession* session, int time);
```

This function processes the internal functionality of a session. This implies:

1. Flashes all output streams sending the data through the transport.
2. If there is any reliable stream, it will perform the asociated reliable behaviour to ensure the communication.
3. Listens messages from the *Agent* and call the associated callback if exists (a topic callback or a status callback).

The `time` suffix function version will perform these actions and will listen messages for a `time` duration. Only when the time waiting for a message overcome the `time` duration, the function finishes. The function will return `true` if the sending data have been confirmed, `false` otherwise.

session Session structure previously initialized.

time Time for waiting, in milliseconds. For waiting without timeout, set the value to `UXR_TIMEOUT_INF`

```
void uxr_run_session_until_timeout(uxrSession* session, int timeout);
```

This function processes the internal functionality of a session. This implies:

1. Flashes all output streams sending the data through the transport.
2. If there is any reliable stream, it will perform the asociated reliable behaviour to ensure the communication.
3. Listens messages from the *Agent* and call the associated callback if exists (a topic callback or a status callback).

The `_until_timeout` suffix function version will perform these actions until receiving one message. Once the message has been received or the timeout has been reached, the function finishes. Only when the time waiting for a message overcome the `timeout` duration, the function finishes. The function will return `true` if has received a message, `false` if the timeout has been reached.

session Session structure previously initialized.

timeout Time for waiting a new message, in milliseconds. For waiting without timeout, set the value to UXR_TIMEOUT_INF

```
bool uxr_run_session_until_confirm_delivery(uxrSession* session, int timeout);
```

This function processes the internal functionality of a session. This implies:

1. Flashes all output streams sending the data through the transport.
2. If there is any reliable stream, it will perform the asociated reliable behaviour to ensure the communication.
3. Listenes messages from the *Agent* and call the associated callback if exists (a topic callback or a status callback).

The _until_confirm_delivery suffix function version will perform these actions during timeout or until the output reliable streams confirm that the sent messages have been received by the *Agent*. The function will return true if the sent data have been confirmed, false otherwise.

session Session structure previously initialized.

timeout Maximun waiting time for a new message, in milliseconds. For waiting without timeout, set the value to UXR_TIMEOUT_INF

```
bool uxr_run_session_until_all_status(uxrSession* session, int timeout, const uint16_
↪t* request_list, uint8_t* status_list, size_t list_size);
```

This function processes the internal functionality of a session. This implies:

1. Flashes all output streams sending the data through the transport.
2. If there is any reliable stream, it will perform the asociated reliable behaviour to ensure the communication.
3. Listenes messages from the *Agent* and call the associated callback if exists (a topic callback or a status callback).

The _until_all_status suffix function version will perform these actions during timeout duration or until all requested status had been received. The function will return true if all status have been received and all of them have the value UXR_STATUS_OK or UXR_STATUS_OK_MATCHED, false otherwise.

session Session structure previously initialized.

timeout Maximun waiting time for a new message, in milliseconds. For waiting without timeout, set the value to UXR_TIMEOUT_INF

request_list An array of request to confirm with a status.

status_list An uninitialized array with the same size as request_list where the status values will be written. The position of a status in the list corresponds to the request at the same position in request_list.

list_size The size of request_list and status_list arrays.

```
bool uxr_run_session_until_one_status(uxrSession* session, int timeout, const uint16_
↪t* request_list, uint8_t* status_list, size_t list_size);
```

This function processes the internal functionality of a session. This implies:

1. Flashes all output streams sending the data through the transport.
2. If there is any reliable stream, it will perform the asociated reliable behaviour to ensure the communication.

3. Listenes messages from the *Agent* and call the associated callback if exists (a topic callback or a status callback).

The `_until_one_status` suffix function version will perform these actions during `timeout` duration or until one requested status had been received. The function will return `true` if one status have been received and has the value `UXR_STATUS_OK` or `UXR_STATUS_OK_MATCHED`, `false` otherwise.

session Session structure previously initialized.

timeout Maximun waiting time for a new message, in milliseconds. For waiting without timeout, set the value to `UXR_TIMEOUT_INF`

request_list An array of request that can be confirmed.

status_list An uninitialized array with the same size as `request_list` where the statu value will be written. The position of the status in the list corresponds to the request at the same position in `request_list`.

list_size The size of `request_list` and `status_list` arrays.

Create entities by XML profile

These functions are enabled when `PROFILE_CREATE_ENTITIES_XML` is selected in the `client.config` file. The declaration of these functions can be found in `uxr/client/profile/session/create_entities_xml.h`.

```
uint16_t uxr_buffer_create_participant_xml(uxrSession* session, uxrStreamId stream_id,
↪ uxrObjectId object_id, uint16_t domain, const char* xml, uint8_t mode);
```

Creates a *participant* entity in the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to `uxr_flash_output_streams` or to `uxr_run_session` function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_PARTICIPANT_ID`

xml A XML representation of the new entity.

mode Determines the creation entity mode. The Creation Mode Table decribes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags.

```
uint16_t uxr_buffer_create_topic_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id, uxrObjectId participant_id, const char* xml, uint8_t mode);
```

Creates a *topic* entity in the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to `uxr_flash_output_streams` or to `uxr_run_session` function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_TOPIC_ID`

participant_id The identifier of the associated participant. The type must be `UXR_PARTICIPANT_ID`

xml A XML representation of the new entity.

mode Determines the creation entity mode. The Creation Mode Table decribes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags.

```
uint16_t uxr_buffer_create_publisher_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id, uxrObjectId participant_id, const char* xml, uint8_t mode);
```

Creates a *publisher* entity in the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to uxr_flash_output_streams or to uxr_run_session function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be UXR_PUBLISHER_ID

participant_id The identifier of the associated participant. The type must be UXR_PARTICIPANT_ID

xml A XML representation of the new entity.

mode Determines the creation entity mode. The Creation Mode Table decribes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags.

```
uint16_t uxr_buffer_create_subscriber_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id, uxrObjectId participant_id, const char* xml, uint8_t mode);
```

Creates a *subscriber* entity in the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to uxr_flash_output_streams or to uxr_run_session function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be UXR_SUBSCRIBER_ID

participant_id The identifier of the associated participant. The type must be UXR_PARTICIPANT_ID

xml A XML representation of the new entity.

mode Determines the creation entity mode. The Creation Mode Table decribes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags.

```
uint16_t uxr_buffer_create_datawriter_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id, uxrObjectId publisher_id, const char* xml, uint8_t mode);
```

Creates a *datawriter* entity in the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to uxr_flash_output_streams or to uxr_run_session function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be UXR_DATAWRITER_ID

publisher_id The identifier of the associated participant. The type must be UXR_PUBLISHER_ID

xml A XML representation of the new entity.

mode Determines the creation entity mode. The Creation Mode Table decribes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags.

```
uint16_t uxr_buffer_create_datareader_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id, uxrObjectId subscriber_id, const char* xml, uint8_t mode);
```

Creates a *datareader* entity in the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to uxr_flash_output_streams or to uxr_run_session function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be UXR_DATAREADER_ID

subscriber_id The identifier of the associated participant. The type must be UXR_SUBSCRIBER_ID

xml A XML representation of the new entity.

mode Determines the creation entity mode. The Creation Mode Table decribes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags.

Create entities by reference profile

These functions are enabled when PROFILE_CREATE_ENTITIES_REF is selected in the client.config file. The declaration of these functions can be found in uxr/client/profile/session/create_entities_ref.h.

```
uint16_t uxr_buffer_create_participant_ref(uxrSession* session, uxrStreamId stream_id,
↪ uxrObjectId object_id, const char* ref, uint8_t mode);
```

Creates a *participant* entity in the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to uxr_flash_output_streams or to uxr_run_session function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be UXR_PARTICIPANT_ID

ref A reference to the new entity.

mode Determines the creation entity mode. The Creation Mode Table decribes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags.

```
uint16_t uxr_buffer_create_topic_ref(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id, uxrObjectId participant_id, const char* ref, uint8_t mode);
```

Creates a *topic* entity in the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to uxr_flash_output_streams or to uxr_run_session function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_TOPIC_ID`

participant_id The identifier of the associated participant. The type must be `UXR_PARTICIPANT_ID`

ref A reference to the new entity.

mode Determines the creation entity mode. The Creation Mode Table decribes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags.

```
uint16_t uxr_buffer_create_datawriter_ref(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id, uxrObjectId publisher_id, const char* ref, uint8_t mode);
```

Creates a *datawriter* entity in the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to `uxr_flash_output_streams` or to `uxr_run_session` function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_DATAWRITER_ID`

publisher_id The identifier of the associated publisher. The type must be `UXR_PUBLISHER_ID`

ref A reference to the new entity.

mode Determines the creation entity mode. The Creation Mode Table decribes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags.

```
uint16_t uxr_buffer_create_datareader_ref(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id, uxrObjectId subscriber_id, const char* ref, uint8_t mode);
```

Creates a *datareader* entity in the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to `uxr_flash_output_streams` or to `uxr_run_session` function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_DATAREADER_ID`.

subscriber_id The identifier of the associated subscriber. The type must be `UXR_SUBSCRIBER_ID`.

ref A reference to the new entity.

mode Determines the creation entity mode. The Creation Mode Table decribes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags.

Create entities common profile

These functions are enabled when PROFILE_CREATE_ENTITIES_XML or PROFILE_CREATE_ENTITIES_REF are selected in the client.config file. The declaration of these functions can be found in uxr/client/profile/session/common_create_entities.h.

```
uint16_t uxr_buffer_delete_entity(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id);
```

Removes an entity. The message is only written into the stream buffer. To send the message is necessary call to uxr_flash_output_streams or to uxr_run_session function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The identifier of the object which will be deleted.

Read access profile

These functions are enabled when PROFILE_READ_ACCESS is selected in the client.config file. The declaration of these functions can be found in uxr/client/profile/session/read_access.h.

```
uint16_t uxr_buffer_request_data(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId datareader_id, uxrStreamId data_stream_id, uxrDeliveryControl* delivery_
↪control);
```

This function requests a read from a datareader of the *Agent*. The returned value is an identifier of the request. All received topic will have the same request identifier. The topics will be received at the callback topic through the run_session function. If there is no error with the request data, the topics will be received generating a status callback with the value UXR_STATUS_OK. If there is an error, a status error will be sent by the *Agent*. The message is only written into the stream buffer. To send the message is necessary call to uxr_flash_output_streams or to uxr_run_session function.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

object_id The *datareader* ID that will read the topic from the DDS World.

data_stream_id The input stream ID where the data will be received.

delivery_control Optional information about how the delivery must be. A NULL value is accepted, in this case, only one topic will be received.

Write access profile

These functions are enabled when PROFILE_WRITE_ACCESS is selected in the client.config file. The declaration of these functions can be found in uxr/client/profile/session/write_access.h.

```
bool uxr_prepare_output_stream(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId datawriter_id,
                               struct ucdrBuffer* mb_topic, uint32_t topic_size);
```

Requests a writing into a specific output stream. For this function will initialize an `ucdrBuffer` struct where a topic of `topic_size` size must be serialized. Whether the necessary gap for writting a `topic_size` bytes into the stream, the returned value is `true`, otherwise `false`. The topic will be sent in the next `run_session` function.

NOTE: All `topic_size` bytes requested will be sent to the *Agent* after a `run_session` call, no matter if the `ucdrBuffer` has been used or not.

session Session structure previously initialized.

stream_id The output stream ID where the message will be written.

datawriter_id The *datawriter* ID that will write the topic to the DDS World.

mb_topic An `ucdrBuffer` struct used to serialize the topic. This struct points to a requested gap into the stream.

topic_size The bytes that will be reserved in the stream.

Discovery profile

The discovery profile allows discovering *Agents* in the network by UDP. The reachable *Agents* will respond to the discovery call sending information about them, as their IP and port. There are two modes: multicast and unicast. The discovery phase can be performed before the *uxr_create_session* call in order to determine the *Agent* to connect with. These functions are enabled when `PROFILE_DISCOVERY` is selected in the `client.config` file. The declaration of these functions can be found in `uxr/client/profile/discovery/discovery.h`.

This feature is only available on Linux.

```
bool uxr_discovery_agents_multicast(uint32_t attempts, int period,
                                    uxrOnAgentFound on_agent_func, void* args,␣
↪uxrAgentAddress* chosen);
```

Searches into the network using multicast IP "239.255.0.2" and port 7400 (default used by the *Agent*) in order to discover *Agents*.

attempts The number of attempts to send the discovery message to the network.

period How will often be sent the discovery message to the network.

on_agent_func The callback function that will be called when an *Agent* was discovered. The callback returns a boolean value. A *true* means that the discovery routine will be finished. The current *Agent* will be selected as *chosen*. A *false* implies that the discovery routine must continue searching *Agents*.

args User arguments passed to the callback function.

chosen If the callback function was returned *true*, this value will contain the *Agent* value of the callback.

```
bool uxr_discovery_agents_unicast(uint32_t attempts, int period,
                                  uxrOnAgentFound on_agent_func, void* args,␣
↪uxrAgentAddress* chosen,
                                  const uxrAgentAddress* agent_list, size_t agent_
↪list_size);
```

Searches into the network using a list of unicast directions in order to discover *Agents*.

attempts The number of attempts to send the discovery message to the network.

period How will often be sent the discovery message to the network.

on_agent_func The callback function that will be called when an *Agent* is discovered. The callback returns a boolean value. A `true` means that the discovery routine will be finished. The current *Agent* will be selected as *chosen*. A `false` implies that the discovery routine must continue searching *Agents*.

args User arguments passed to the callback function.

chosen If the callback function was returned `true`, this value will contain the *Agent* value of the callback.

agent_list The list of addresses where discover *Agent*. By default, the *Agents* will be listened at **port 7400** the discovery messages.

agent_list_size The size of the `agent_list`.

Topic serialization

Functions to serialize and deserialize topics. These functions are generated automatically by *eProsima Micro XRCE-DDS Gen* utility over an IDL file with a topic *TOPICTYPE*. The declaration of these function can be found in the generated file `TOPICTYPE.h`.

```
bool TOPICTYPE_serialize_topic(struct ucdrBuffer* writer, const TOPICTYPE* topic);
```

Serializes a topic into an `ucdrBuffer`. The returned value indicates if the serialization was successful.

writer An `ucdrBuffer` representing the buffer for the serialization.

topic Struct to serialize.

```
bool TOPICTYPE_deserialize_topic(struct ucdrBuffer* reader, TOPICTYPE* topic);
```

Deserializes a topic from an ucdrBuffer. The returned value indicates if the serialization was successful.

reader An ucdrBuffer representing the buffer for the deserialization.

topic Struct where deserialize.

```
uint32_t TOPICTYPE_size_of_topic(const TOPICTYPE* topic, uint32_t size);
```

Counts the number of bytes that the topic will need in an *ucdrBuffer*.

topic Struct to count the size.

size Number of bytes already written into the *ucdrBuffer*. Typically, its value is *0* if the purpose is to use in `uxr_prepare_output_stream` function.

General utilities

Utility functions. The declaration of these functions can be found in `uxr/client/core/session/stream_id.h` and `uxr/client/core/session/object_id.h`.

```
uxrStreamId uxr_stream_id(uint8_t index, uxrStreamType type, uxrStreamDirection␣
↪direction);
```

Creates a stream identifier. This function does not create a new stream, only creates its identifier to be used in the *Client* API.

index Identifier of the stream, its value correspond to the creation order of the stream, different for each *type*.

type The type of the stream, it can be `UXR_BEST_EFFORT_STREAM` or `UXR_RELIABLE_STREAM`.

direction Represents the direccion of the stream, it can be `UXR_INPUT_STREAM` or `UXR_OUTPUT_STREAM`.

```
uxrStreamId uxr_stream_id_from_raw(uint8_t stream_id_raw, uxrStreamDirection␣
↪direction);
```

Creates a stream identifier. This function does not create a new stream, only creates its identifier to be used in the *Client* API.

stream_id_raw Identifier of the stream. It goes from 0 to 255. 0 is for internal library use. 1 to 127, for best effort. 128 to 255, for reliable.

direction Represents the direction of the stream, it can be `UXR_INPUT_STREAM` or `MT_OUTPUT_STREAM`.

```
uxrObjectId uxr_object_id(uint16_t id, uint8_t type);
```

Creates an identifier for reference an entity.

id Identifier of the object, different for each *type* (can be several IDs with the same ID if they have different types).

type The type of the entity. It can be: `UXR_PARTICIPANT_ID`, `UXR_TOPIC_ID`, `UXR_PUBLISHER_ID`, `UXR_SUBSCRIBER_ID`, `UXR_DATAWRITER_ID` or `UXR_DATAREADER_ID`.

Transport

These functions are platform dependent. The values `PROFILE_XXX_TRANSPORT` found into `client.config` allow to enable some of them. The declaration of these function can be found in `uxr/client/profile/transport/` folder. The common init transport functions follow the next nomenclature.

```
bool uxr_init_udp_transport(uxrUDPTransport* transport, uxrUDPPlatform* platform,␣
↪const char* ip, uint16_t port);
```

Initializes a UDP connection.

transport The uninitialized structure used for managing the transport. This structure must be accessible during the connection.

platform Structure which contains platform dependent members.

ip *Agent* IP.

port *Agent* port.

```
bool uxr_init_tcp_transport(uxrTCPTransport* transport, uxrTCPPlatform* platform,␣
↪const char* ip, uint16_t port);
```

Initializes a TCP connection. If the TCP is used, the behaviour of best-effort streams will be similar to reliable streams in UDP.

transport The uninitialized structure used for managing the transport. This structure must be accessible during the connection.

platform Structure which contains platform dependent members.

ip *Agent* IP.

port *Agent* port.

```
bool uxr_init_serial_transport(uxrSerialTransport* transport, uxrSerialPlatform*␣
↪platform, const int fd, uint8_t remote_addr, uint8_t local_addr);
```

Initializes a Serial connection using a file descriptor

transport The uninitialized structure used for managing the transport. This structure must be accessible during the connection.

platform Structure which contains platform dependent members.

fd File descriptor of the serial connection. Usually, the fd comes from the `open` OS function.

remote_addr Identifier of the *Agent* in the serial connection. By default, the *Agent* identifier in a serial is 0.

local_addr Identifier of the *Client* in the serial connection.

```
bool uxr_close_PROTOCOL_transport(PROTOCOLTransport* transport);
```

Closes a transport previously opened. *PROTOCOL* can be `udp`, `tcp` or `serial`.

transport The transport to close.

Creation Mode Table

The following table summarize the behaviour of the *Agent* under entity creation request.

Creation flags	Entity exists	Result
Don't care	NO	Entity is created.
`0`	YES	No action is taken, and `UXR_STATUS_ERR_ALREADY_EXITS` is returned.
`UXR_REPLACE`	YES	Existing entity is deleted, requested entity is created and `UXR_STATUS_OK` is returned.
`UXR_REUSE`	YES	If entity matches no action is taken and `UXR_STATUS_OK_MATCHED` is returned. If entity does not match no action is taken and `UXR_STATUS_ERR_MISMATCH` is returned.
`UXR_REUSE | UXR_REPLACE`	YES	If entity matches no action is taken and `UXR_STATUS_OK_MATCHED` is returned. If entity does not match, exiting entity is deleted, requested entity is created and `UXR_STATUS_OK` is returned.

3.6 eProsima Micro XRCE-DDS Gen

eProsima Micro XRCE-DDS Gen is a Java application used to generate source code for the *eProsima Micro XRCE-DDS* software.

This tool is able to generate from a given IDL specification file, the C struct associated with the Topic, as well as, the serialization and deserialization methods. Also, it has the possibility to generate a sample demo that works with the proposed topic. As an example of the potential of this tool, the following shows the source code generated from the ShapeDemo IDL file.

```
// ShapeType.idl

struct ShapeType {
    @key string color;
    long x;
    long y;
    long shapesize;
};
```

If we will perform the following command:

```
$ microxrceddsgen ShapeType.idl
```

it will generate the following header file and its corresponding source:

```
/*!
 * @file ShapeType.h
 * This header file contains the declaration of the described types in the IDL file.
 *
 * This file was generated by the tool gen.
 */

#ifndef _ShapeType_H_
#define _ShapeType_H_

#include <stdint.h>
#include <stdbool.h>

/*!
 * @brief This struct represents the structure ShapeType defined by the user in the␣
↪IDL file.
 * @ingroup SHAPETYPE
 */
typedef struct ShapeType
{
    char color[255];
    int32_t x;
    int32_t y;
    int32_t shapesize;

} ShapeType;

struct ucdrBuffer;

bool ShapeType_serialize_topic(struct ucdrBuffer* writer, const ShapeType* topic);
bool ShapeType_deserialize_topic(struct ucdrBuffer* reader, ShapeType* topic);
uint32_t ShapeType_size_of_topic(const ShapeType* topic, uint32_t size);

#endif // _ShapeType_H_
```

eProsima Micro XRCE-DDS Gen is also able to generate both *publisher* and *subscriber* source code examples, related with the topic specified in the IDL file, by adding the flag `-example`:

```
$ microxrceddsgen -example <file.idl>
```

In order to use these examples, the *Client* library must be compiled with the `WRITE_ACCESS_PROFILE` option for the *publisher* and the `READ_ACCESS_PROFILE` option for the *subscriber*.

3.6.1 Installation

In order to use *eProsima Micro XRCE-DDS Gen*, it is needed to follow the next steps:

1. Install its dependencies:
    1.1 Gradle.
    1.2 Java JDK.
2. Clone the code from the GitHub repository.

```
$ git clone --recursive https://github.com/eProsima/micro-XRCE-DDS-gen.git
$ cd micro-XRCE-DDS-gen
```

3. Build the code with Gradle.

```
$ gradle build
```

3.6.2 Notes

At the present time, *eProsima Micro XRCE-DDS Gen* only supports Structs composed of integer, string, array and sequence types even though it is planned to enhance the capabilities of the *eProsima Micro XRCE-DDS Gen* tool in a near future.

3.7 eProsima Micro XRCE-DDS Agent

eProsima Micro XRCE-DDS Agent acts as a server between the DDS Network and *eProsima Micro XRCE-DDS Clients* applications. *Agents* receive messages containing operations from *Clients*. Also, *Agents* keep track of the *Clients* and the entities they create. The *Agent* uses the entities to interact with the DDS Global Data Space on behalf of the *Clients*.

The communication between a *Client* and an *Agent* currently supports UDP, TCP and Serial (dependent on the platform). While it is running, the *Agent* will attend any received request from the *Clients* and answers back with the result of that request.

3.7.1 Configuration

There are several configuration parameters which can be set at **compile time** in order to configure the *eProsima Micro XRCE-DDS Agent*. These parameters can be selected as CMake flags (`-D<parameter>=<value>`) before the compilation. The following is a list of the aforementioned parameters:

`UAGENT_CONFIG_RELIABLE_STREAM_DEPTH` Specify the history of the reliable streams (default 16).

`UAGENT_CONFIG_BEST_EFFORT_STREAM_DEPTH` Specify the history of the best-effort streams (default 16).

`UAGENT_CONFIG_HEARTBEAT_PERIOD` Specify the `HEARTBEAT` message period in millisecond (default 200).

`UAGENT_CONFIG_TCP_MAX_CONNECTIONS` Specify the maximum number of connections, the *Agent* is able to manage (default 100).

`UAGENT_CONFIG_TCP_MAX_BACKLOG_CONNECTIONS` Specify the maximum number of incoming connections (pending to establish), the *Agent* is able to manage (default 100).

3.7.2 Run an Agent

To run the *Agent* you should build it as indicated in *Installation*. Once it is built successfully, you just need to launch it executing one of the following commands:

UDP server

```
$ ./MicroXRCEAgent udp [OPTIONS]

Options:
  -h,--help                                Print this help message and exit
  -p,--port UINT REQUIRED                  Select the port
  -m,--middleware TEXT in {ced,dds}=dds    Select the kind of Middleware
  -r,--refs FILE                           Load a reference file
  -v,--verbose UINT in {0,1,2,3,4,5,6}=4   Select log level from less to more verbose
  -d,--discovery                           Activate the Discovery server
  --disport UINT=7400 Needs: --discovery   Select the port for the Discovery server
  --p2p UINT                               Activate the P2P profile using the given␣
↪port
```

TCP server

```
$ ./MicroXRCEAgent tcp [OPTIONS]

Options:
  -h,--help                                Print this help message and exit
  -p,--port UINT REQUIRED                  Select the port
  -m,--middleware TEXT in {ced,dds}=dds    Select the kind of Middleware
  -r,--refs FILE                           Load a reference file
  -v,--verbose UINT in {0,1,2,3,4,5,6}=4   Select log level from less to more verbose
  -d,--discovery                           Activate the Discovery server
  --disport UINT=7400 Needs: --discovery   Select the port for the Discovery server
  --p2p UINT                               Activate the P2P profile using the given␣
↪port
```

Serial server (only Linux)

```
$ ./MicroXRCEAgent serial [OPTIONS]

Options:
  -h,--help                                Print this help message and exit
  --dev FILE REQUIRED                      Select the serial device
  -b,--baudrate TEXT=115200                Select the baudrate
  -m,--middleware TEXT in {ced,dds}=dds    Select the kind of Middleware
  -r,--refs FILE                           Load a reference file
  -v,--verbose UINT in {0,1,2,3,4,5,6}=4   Select log level from less to more verbose
  -d,--discovery                           Activate the Discovery server
  --disport UINT=7400 Needs: --discovery   Select the port for the Discovery server
  --p2p UINT                               Activate the P2P profile using the given␣
↪port
```

Pseudo-Serial server (only Linux)

```
$ ./MicroXRCEAgent pseudo-serial [OPTIONS]

Options:
  -h,--help                                Print this help message and exit
  --dev FILE REQUIRED                      Select the serial device
  -b,--baudrate TEXT=115200                Select the baudrate
  -m,--middleware TEXT in {ced,dds}=dds    Select the kind of Middleware
  -r,--refs FILE                           Load a reference file
  -v,--verbose UINT in {0,1,2,3,4,5,6}=4   Select log level from less to more verbose
  -d,--discovery                           Activate the Discovery server
  --disport UINT=7400 Needs: --discovery   Select the port for the Discovery server
```

(continues on next page)

(continued from previous page)

```
  --p2p UINT                                Activate the P2P profile using the given␣
↪port
```

- The reference file shall be composed by a set of Fast RTPS profiles following the XML syntax described in Fast RTPS. The `profile_name` attribute of each profile represents a reference to an XRCE-Entity so that it could be used by the *Clients* to create entities by reference.
- The `-b, --baudrate <baudrate>` options sets the baud rate of the communication. It can take the following values: 0, 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 240, 4800, 9600, 19200, 38400, 57600, 115200 (default), 230400, 460800, 500000, 576000, 921600, 1000000, 1152000, 1500000, 2000000, 2500000, 3000000, 3500000 or 4000000 Bd.
- The `-v, --verbose <level[0-6]>` option sets log level from less to more verbose, in level 0 the logger is off.
- `-m, --middleware <middleware-impl>`: set the middleware implementation to use. There are two: DDS (specified by the XRCE standard) and Centralized (topic are managed by the Agent similarly MQTT).
- The `--p2p <port>` option enables P2P communication. Centralized middleware is necessary for this option.

3.8 Entities

The protocol under *eProsima Micro XRCE-DDS* (XRCE), defines entities that have a direct correspondence with their analogous actors on *eProsima Fast RTPS* (DDS). The entities manage the communication between *eProsima Micro XRCE-DDS Client* and the DDS Global Data Space. Entities are stored in the *eProsima Micro XRCE-DDS Agent* and the *eProsima Micro XRCE-DDS Client* can create, use and destroy these entities.

The entities are uniquely identified by an ID called *Object ID*. The *Object ID* is the way a *Client* refers to them inside an *Agent*. In most of the *Client* request operations is necessary to specify an ID referring to one of the *Client* entities stored in the *Agent*.

3.8.1 Type of Entities

These are the entities that the *Client* can interact with.

Participants Participants can hold any number of Publishers and/or Subscribers

Publisher Publishers can hold any number of data writers.

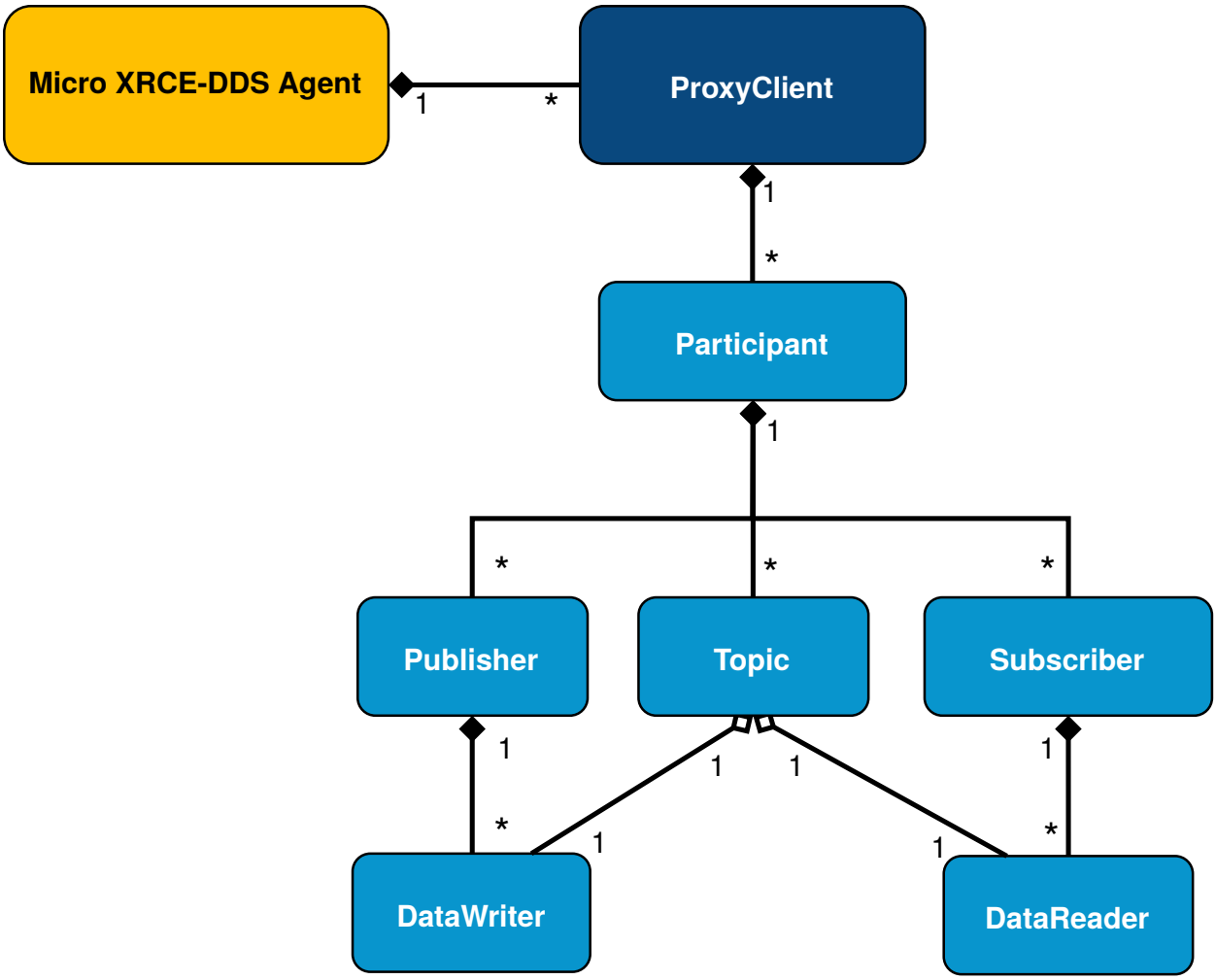
Subscriber Subscribers can hold any number of data readers.

Topic Topic data is the base of the communication. A Topic is composed of a name and a data type.

DataWriter This is the endpoint able to write Topic data.

DataReader This is the endpoint able to read Topic data.

This figure shows the entities hierarchy



3.9 Operations

Operations are the possible actions your *eProsima Micro XRCE-DDS Client* can ask to the *eProsima Micro XRCE-DDS Agent*. Operations revolve around *Entities*. *eProsima Micro XRCE-DDS Agent* will respond to all the requests with the status of the operation.

3.9.1 Types

Create session *Create session* asks the *Agent* to register a session. This is the first operation that you must request. If this operation fails or you don't request it, any of the following operations will not work. This operation will create the session corresponding to the *Client-Agent* connection.

Delete session Delete connection *Client-Agent* and remove all entities associated with this relation. After this operation, any other operation except *Create session* will fail.

Create entity A session can create all the entities it needs. There is a *Create entity* operation for each entity your session can handle. Each *Create entity* operation are related to an ID for its management.

Delete entity Analogous to create entities a session can drop the entities on the *Agent*. To drop an entity you need to request a deletion of the entity to the *Agent* using the entity ID.

Request Data This operation configures how do you want to receive data, and the *Agent* will deliver it from the DDS to your *Client*. This data will be received asynchronously, in accordance with the data delivery control setted in this operation. Reading data is done using a *DataReader* entity.

3.9.2 About XML Representation

Participats, *Topics*, *DataWriters* and *DataReaders* creation needs to be done using DDS XML configuration of the object to create. That XML configuration follows the same rules as in *eProsima Fast RTPS*.

3.10 Deployment example

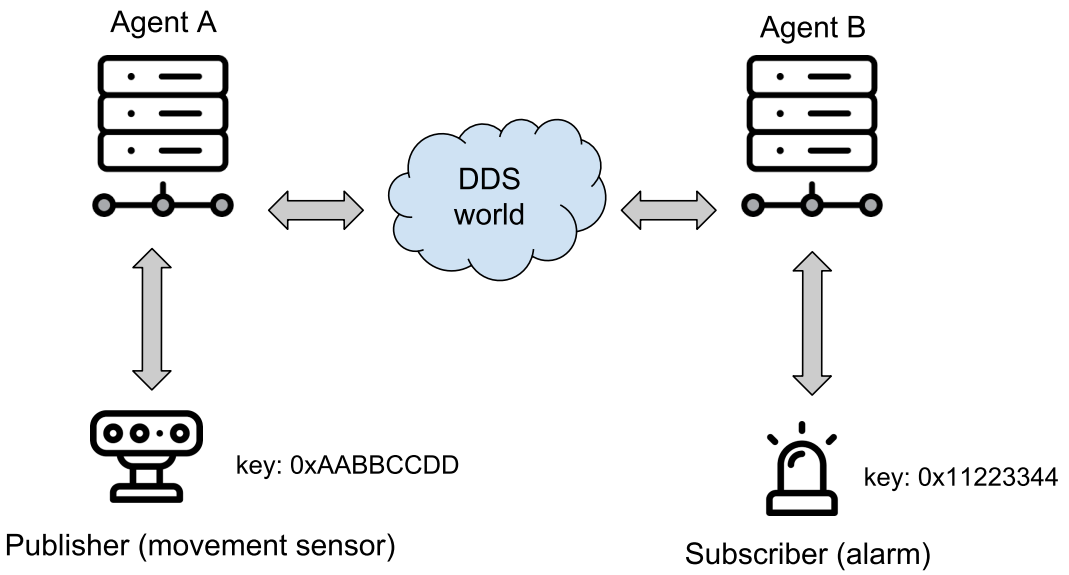
This part will show how to deploy a system using *eProsima Micro XRCE-DDS* in a real environment. An example of this can be found into `examples/Deployment` folder.

Previous tutorials are based in *all in one* examples, that is, examples that create entities, publish or subscribe and then delete the resources. One possible real purpose of this, consists in differentiate the logic of *creating entities* and the actions of *publishing and subscribing*. This can be done creating two differents *Clients*. One in charge of configure the entities in the *Agent*, and run possibly once, only for creating the entities at configuration time. And other/s that logs in the same session as the configured *Client* (sharing the entities) and only publishes or subscribes data.

This way allows to easily create *Clients* in a real scenario only with the purpose of send and receive data. Related to it, the concept of *profile* allows to build the *Client* library only with the chosen behavior (only publish or only subscribe, for example). See *eProsima Micro XRCE-DDS Client* for more information about this.

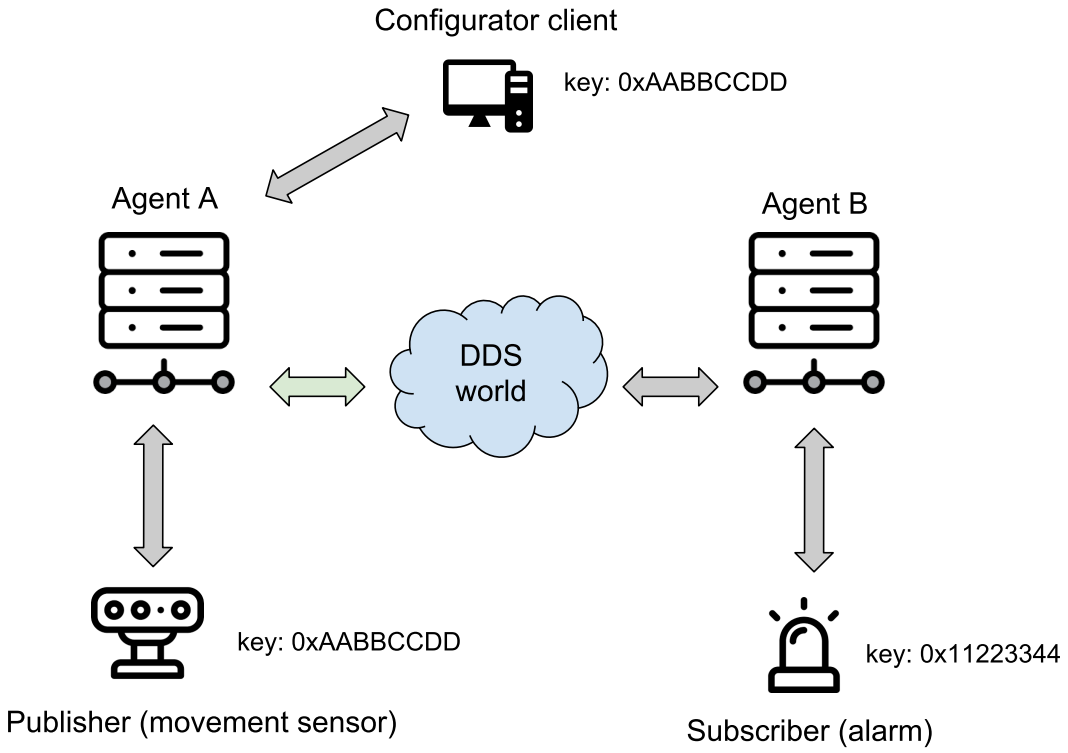
Next diagram shows an example about how to configure the environment using a *configurator client*.

3.10.1 Initial state



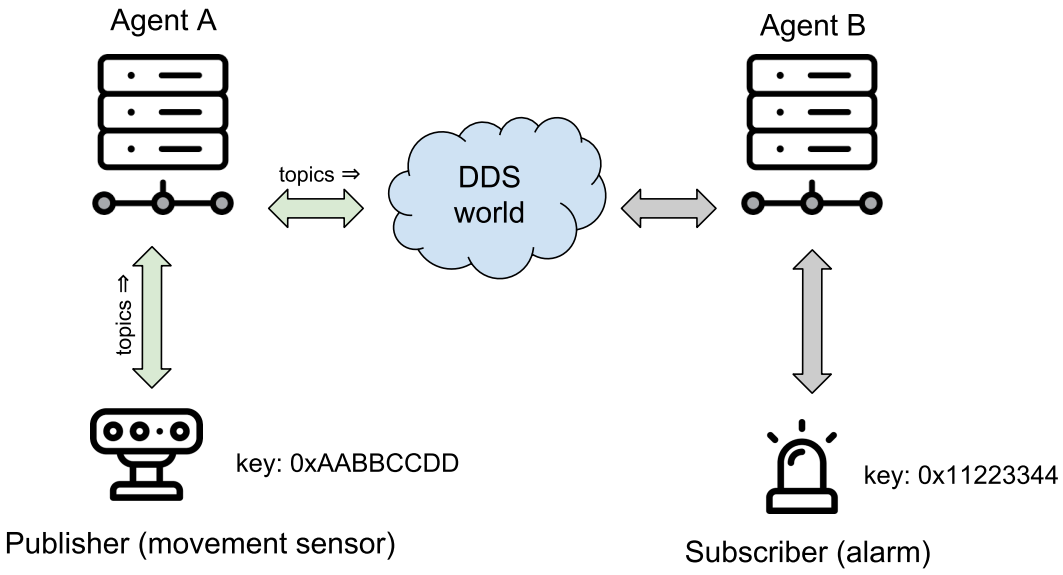
The environment contains two *Agents* (is perfectly possible to use only one *Agent* too), and two *Clients*, one for publishing and another for subscribing.

3.10.2 Configurate the publisher



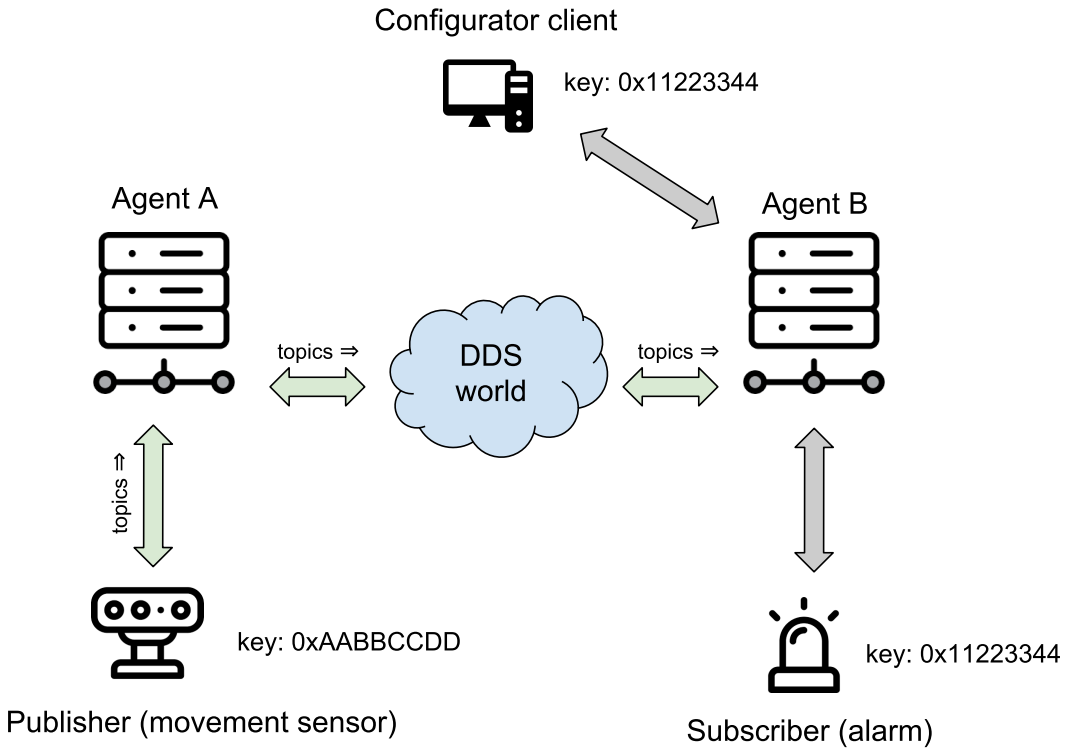
In this state a *configurator client* is connected to the *Agent A* with the *client key* that will be used by the future *publisher client* (0xAABBCCDD). Once a session is logged in, the *configurator client* creates all the necessary entities for the *publisher client*. This implies the creation of *participant*, *topic*, *publisher*, and *datawriter* entities. These entities have a representation as DDS entities, and can be reached now from the DDS world. That implies that a possible *subscriber DDS entity* could already be listening topics if it matches with a *publisher DDS entity* through *DDS* world.

3.10.3 Publish



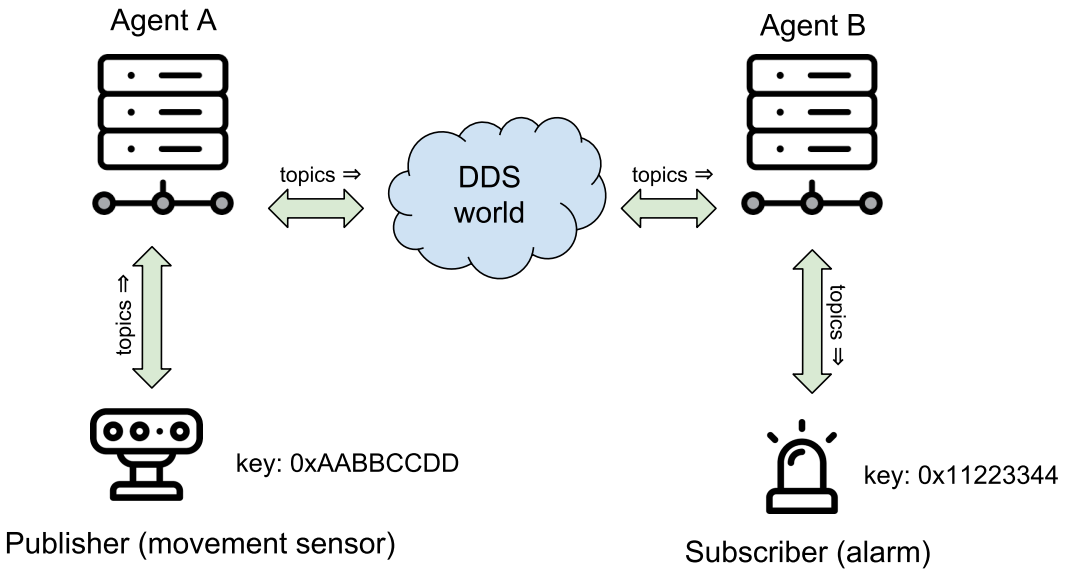
Then, the *publisher client* is connected to the *Agent A*. This *Client* logs in session with its *Client* key (0xAABBCCDD). At that moment, it can use all entities created related to this *client key*. Because all entities that it used were successfully created by the *configurator client*, the *publisher client* can immediately publish to *DDS*.

3.10.4 Configurate the subscriber



Again, the *configurator client* connects and logs in, this time to *Agent B*, now with the subscriber's key (0x11223344). In this case, the entities that the *configurator client* creates are a *participant*, a *topic*, a *subscriber*, and a *datareader*. The entities created by the *configuraton client* will be available until the session is deleted.

3.10.5 Subscriber



Once the subscriber is configured, the *subscriber client* logs in the *Agent B*. As all their entities have been created previously, so it only needs to configure the read after log in. Once the data request message has been sent, the subscriber will receive the topics from the publisher through *DDS* world.

3.11 Memory optimization

3.11.1 Executable code size

In order to manage the executable code size, the library can be compiled enabling or disabling several profiles. To add or remove a profile from the library, edit the `client.config` file. More information can be found at: *eProsima Micro XRCE-DDS Client*.

NOTE: If you are compiling your app with *gcc*, its highly recommended compile it with the linker flag: `-Wl,--gc-sections`. This will remove the code that your app does not use from the final executable.

3.11.2 Runtime size

The *Client* is dynamic memory free and static memory free. This implies that all memory charge depends only on how the stack grows during the execution. To control the stack, there are some values that can be modified and that impact to a large degree:

Stream buffers

1. Make sure you define only the streams that will be used. You can define a maximum *127* best-effort streams and *128* reliable streams, but for the majority of purposes, only one stream of either best effort or reliable can be used.

2. The history of a reliable streams is used for recovering from lost messages and speeding up the communication. For output streams, more history value will allow writing and sending more messages without waiting confirmation. If a history of an output stream is full (no messages were still confirmed by the *Agent*), no more messages can be stored into the stream. For input streams, the history is used for recovering faster when the messages are lost using less extra band-width. If your connection is highly reliable and to save memory is a priority, a reduced history can be used.
3. The *size* of the stream corresponds to the total reserved memory for the stream. This *size* must be according to the next value: `MAX_MESSAGE_SIZE * HISTORY`. The `MAX_MESSAGE_SIZE` value represents the maximum message size that can be sent without fragment the message. `MAX_MESSAGE_SIZE` must be less or equal than the transport *MTU* used.

Transport MTUs

Each transport have a different *MTU*. The *MTU* value can be defined into the `client.config` file. The *MTU* will represent the `MAX_MESSAGE_SIZE` that can be sent or received. The transport uses the *MTU* value to create an internal buffer, so this value will affect increasing transport struct size.

3.12 Transport

This section shows how the transport layer is implemented in both *eProsima Micro XRCE-DDS Agent* and *eProsima Micro XRCE-DDS Client*. Furthermore, this section describes how to add your custom transport in *eProsima Micro XRCE-DDS*.

3.12.1 Introduction

In contrast to other IoT middleware such as MQTT and CoaP, which work over a particular transport protocol, the XRCE protocol is design to support multiple transport protocol natively. This feature of XRCE is enhanced by *eProsima Micro XRCE-DDS* in two ways. On the one hand, both *Agent* and *Client* have the logic of the protocol complete separate from the transport protocol through a set of interfaces, which will be explained in the following sections. On the other hand, taking advantage of the transport interface flexibility, one custom Serial Transport has been implemented to provide support to serial communication.

3.12.2 Agent Transport Architecture

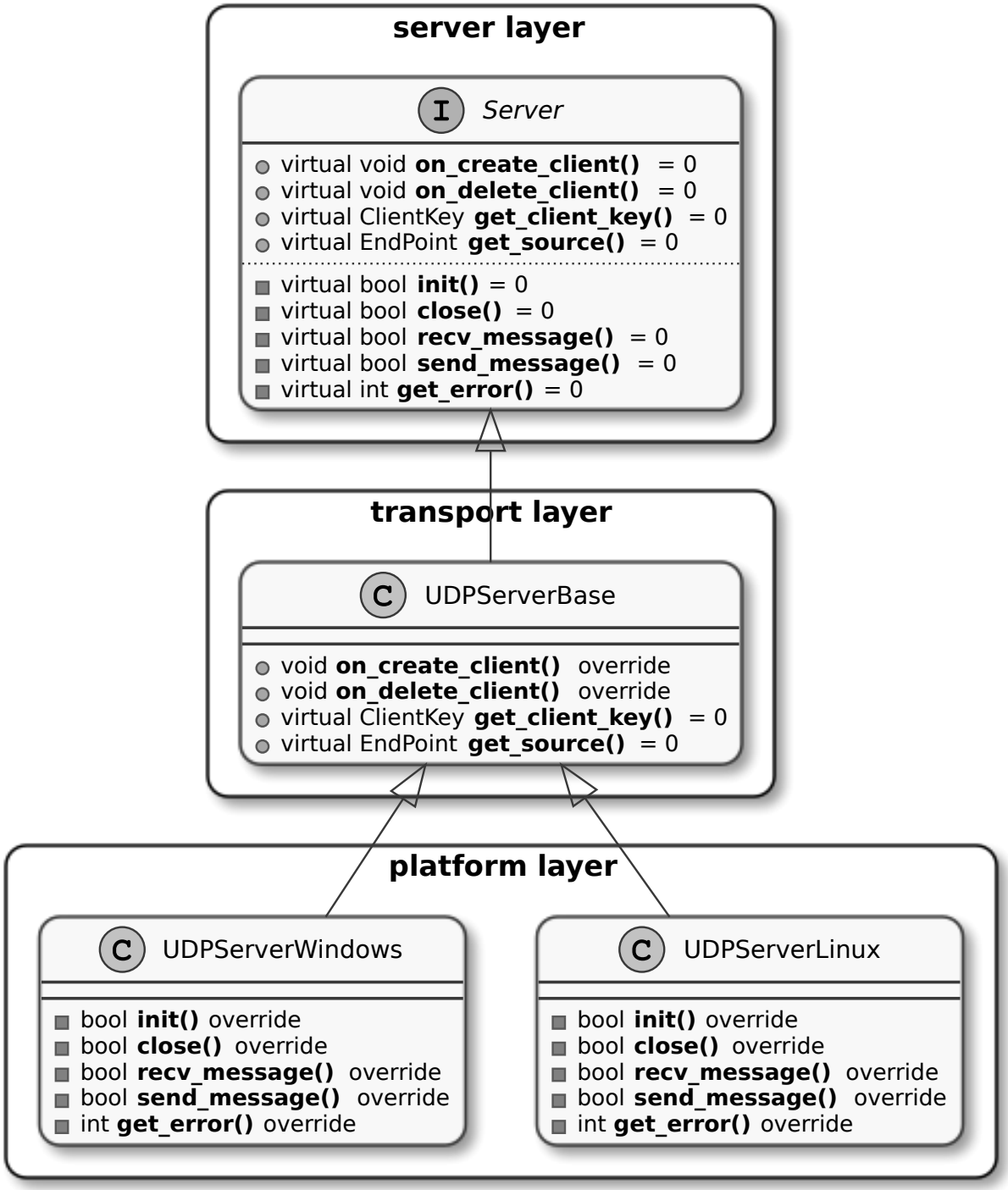
The *Agent* transport architecture is composed by 3 different layers:

- **Server Layer**: is an interface from which each transport specific server inherits. This interface implements four different threads:
  - *Sender thread*: in charge of sending the messages to the *Clients*.

  - *Receiver thread*: in charge of receiving the messages from the *Clients*.
  - *Processing thread*: in charge of processing the messages received from the *Clients*.
  - *Heartbeat thread* in charge of handling reliability with the *Clients*.
- **Transport Layer**: is a transport specific class which manages the sessions established between the *Agent* and the *Clients*. This class inherits from the *Server* interface.
- **Platform Layer**: is a platform-specific class which implements the sending and receiving functions for a given transport in a given platform. It should be noted that it is the only class which has platform dependencies.

UDP Server Example

As an example, this subsection describes how the UDP server is implemented in *eProsima Micro XRCE-DDS Agent*. The figure below shows the *Agent* transport architecture for the UDP servers.



At the top of this architecture, there is a `Server` interface (Server Layer). This `Server` interface has the following pure virtual functions:

```
/* Transport Layer */
virtual void on_create_client(EndPoint* source, const dds::xrce::ClientKey& client_
↪key) = 0;
virtual void on_delete_client(EndPoint* source) = 0;
```

(continues on next page)

(continued from previous page)

```
virtual const dds::xrce::ClientKey get_client_key(EndPoint* source) = 0;
virtual std::unique_ptr<EndPoint> get_source(const dds::xrce::ClientKey& client_key)␣
↪= 0;

/* Platform Layer */
virtual bool init() = 0;
virtual bool close() = 0;
virtual bool recv_message(InputPacket& input_packet, int timeout) = 0;
virtual bool send_message(OutputPacket output_packet) = 0;
virtual int get_error() = 0;
```

The first four virtual functions are transport specific (Transport Layer). These functions are overridden by the `UDPServerBase` class which is in charge of managing the sessions between *Clients* and the *Agent*.

On the other hand, the last five virtual functions are platform specific (Platform Layer). These functions are override by the `UDPServerLinux` and `UDPServerWindows` for Linux and Windows systems respectively.

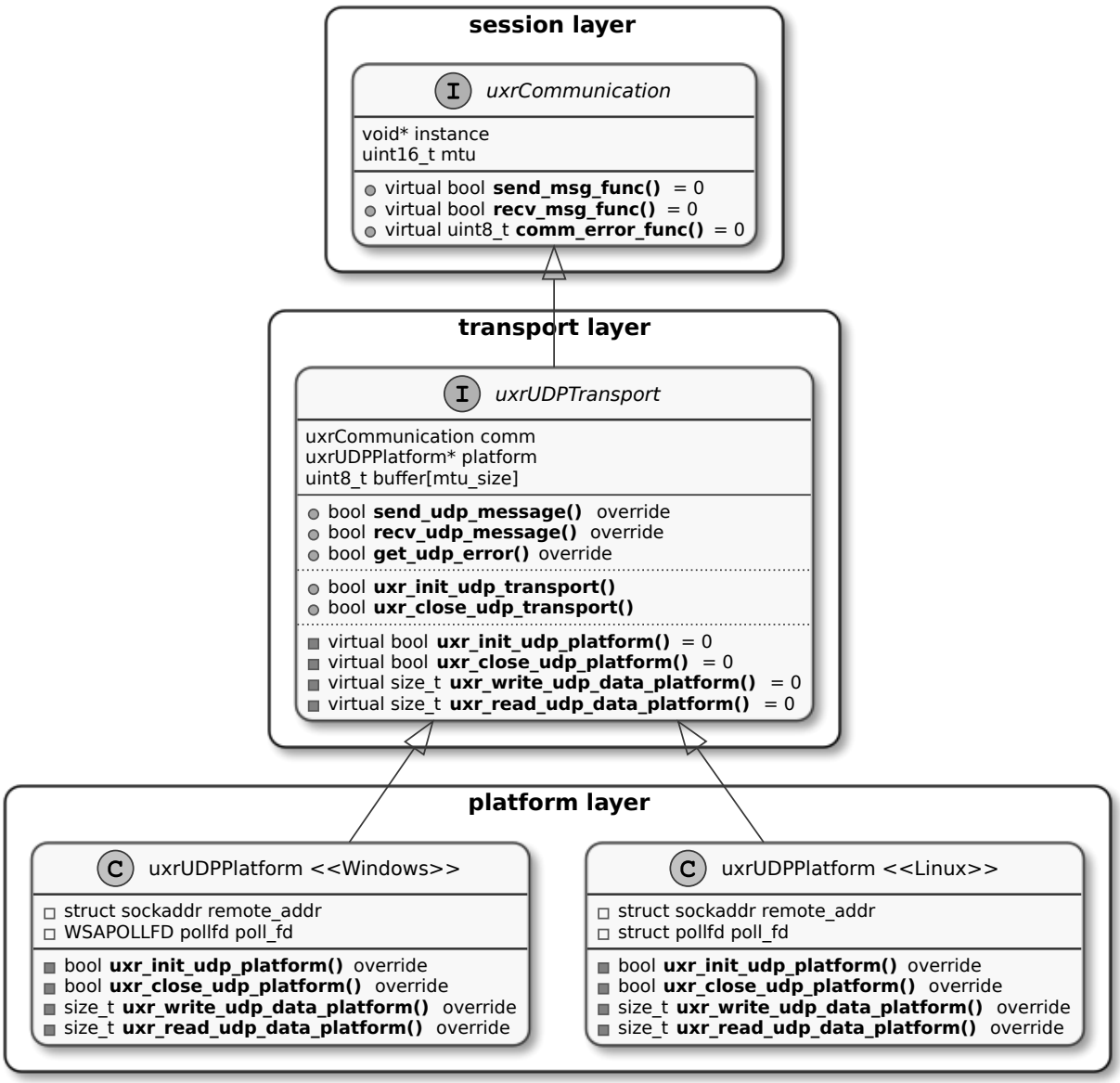
3.12.3 Client Transport Architecture

The *Client* transport architecture is analogous to the *Agent* architecture. There are also three different layers, but instead of the Server Layer, there is a Session Layer.

- **Session Layer**: implements the XRCE protocol logic and it only knows about sending and receiving messages.
- **Transport Layer**: implements the sending and receiving **message functions** for each transport protocol, calling to the Platform Layer functions. This layer only knows about sending and receiving messages through a given transport protocol.
- **Platform Layer**: implements the sending and receiving **data functions** for each platform. This layer only knows about sending and receiving raw data through a given transport in a given platform.

UDP Transport Example

As an example, this subsection describes how the UDP transport is implemented in *eProsima Micro XRCE-DDS Client*. The figure below shows the *Client* transport architecture for UDP transport.



Similar to the *Agent* architecture, there is also an interface, `uxrCommunication`, whose function pointers are used from the Session Layer. That is, each time a `run_session` is called, the Session Layer calls to `send_msg_func` and `recv_msg_func` without worrying about the transport protocol or the platform in use. This struct has the following function pointers:

```
bool send_msg_func(void* instance, const uint8_t* buf, size_t len);
bool recv_msg_func(void* instance, uint8_t** buf, size_t* len, int timeout);
uint8_t comm_error_func(void);
```

These functions are implemented by the `uxrUDPTransport`, which is in charge of two main tasks:

1. Provide an implementation for the communication interface functions. For example, in the case of the UDP protocol, these functions are the following:

```
bool send_udp_msg(void* instance, const uint8_t* buf, size_t len);
bool recv_udp_msg(void* instance, uint8_t** buf, size_t* len, int timeout);
```

(continues on next page)

(continued from previous page)

```
uint8_t get_udp_error(void);
```

2. Offer to the user the initialization and close functions related to the transport protocol. For example, in the case of the UDP protocol, these functions are the following:

```
bool uxr_init_udp_transport(uxrUDPTransport* transport, uxrUDPPlatform* platform,␣
↪const char* ip, uint8_t port);
bool uxr_close_udp_transport(uxrUDPTransport* transport);
```

For each platform, there is an implementation of these functions defined in the Transport Layer interface. For example, in the case of Linux under UDP transport protocol, the `uxrUDPPlatform` implements the following functions:

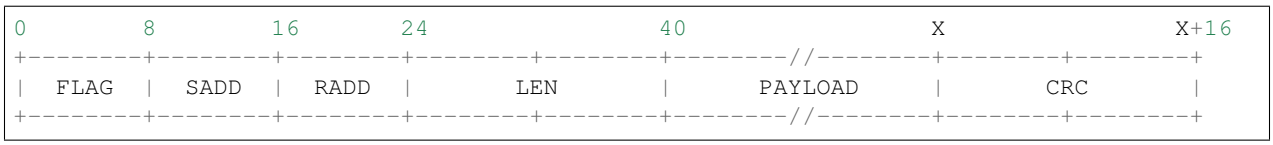
```
bool uxr_init_udp_platform(uxrUDPPlatform* platform, const char* ip, uint16_t port);
bool uxr_close_udp_platform(uxrUDPPlatform* platform);
size_t uxr_write_udp_data_platform(uxrUDPPlatform* platform, const uint8_t* buf, size_
↪t len, uint8_t* errcode);
size_t uxr_read_udp_data_platform(uxrUDPPlatform* platform, uint8_t* buf, size_t len,␣
↪int timeout, uint8_t* errcode);
```

3.12.4 Custom Serial Transport

eProsima Micro XRCE-DDS has a **Custom Serial Transport** with the following features:

- **HDLC Framing**: each serial framing begins with a `begin_frame` octet `(0x7E)`, and the rest of the frame is byte stuffing using the `space` octet `(0x7D)` following by the original octet exclusive-or with `0x20`. For example, if the frame contains the octet *0x7E* it is encoded as *0x7D, 0x5E*; and the same for the octet *0x7E* which is encoded as *0x7D, 0x5D*.
- **CRC Calculation**: serial frames end with the CRC-16 for detecting frame corruption. The CRC-16 is computed using the polynomial `x^16 + x^12 + x^5 + 1` after the frame stuffing for each octet of the frame and including the `begin_frame`, as it is describes in RFC 1662 (see sec. C.2).
- **Routing header**: the Serial Transport provides `source` and `remote` address in the framing, which could be used for implement a routing protocol.

All the aforementioned features are addressed using the following frame format:

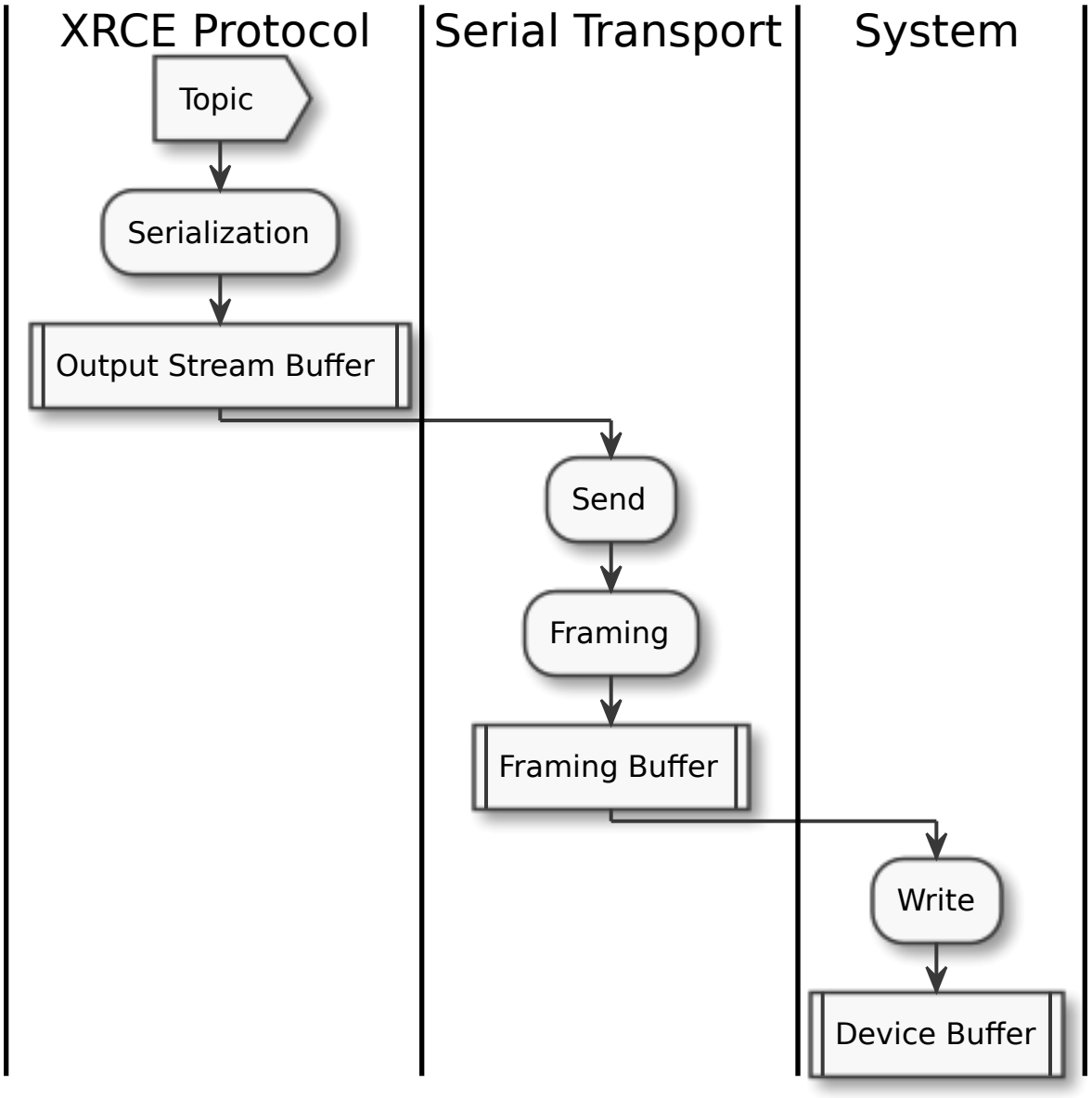


- `FLAG`: is a `begin_frame` octet for frame initialization.
- `SADD`: is the address of the device which sent the message, that is, the `source` address.
- `RADD`: is the address of the device which should receive the message, that is, the `remote` address.
- `LEN`: is the length of the **payload without framing**. It is encoded as a 2-bytes array in little endian.
- `PAYLOAD`: is the payload of the message.
- `CRC`: is the CRC of the message **after the stuffing**.

Data Sending

The figure below shows the workflow of the data sending. This workflow could be divided into the following steps:

1. A publisher application calls the *Client* library to send a given topic.
2. The *Client* library serializes the topic inside an XRCE message using *Micro CDR*. As a result, the XRCE message with the topic is stored in an **Output Stream Buffer**.
3. The *Client* library calls the Serial Transport to send the serialized message.
4. The Serial Transport frames the message, that is, adds the header, payload and CRC of the frame taking into account the stuffing. This step takes place in an auxiliary buffer called **Framing Buffer**.
5. Each time the Framing Buffer is full, the data is flushed into the **Device Buffer** calling the writing system function.



This approach has some advantages which should be pointed out:

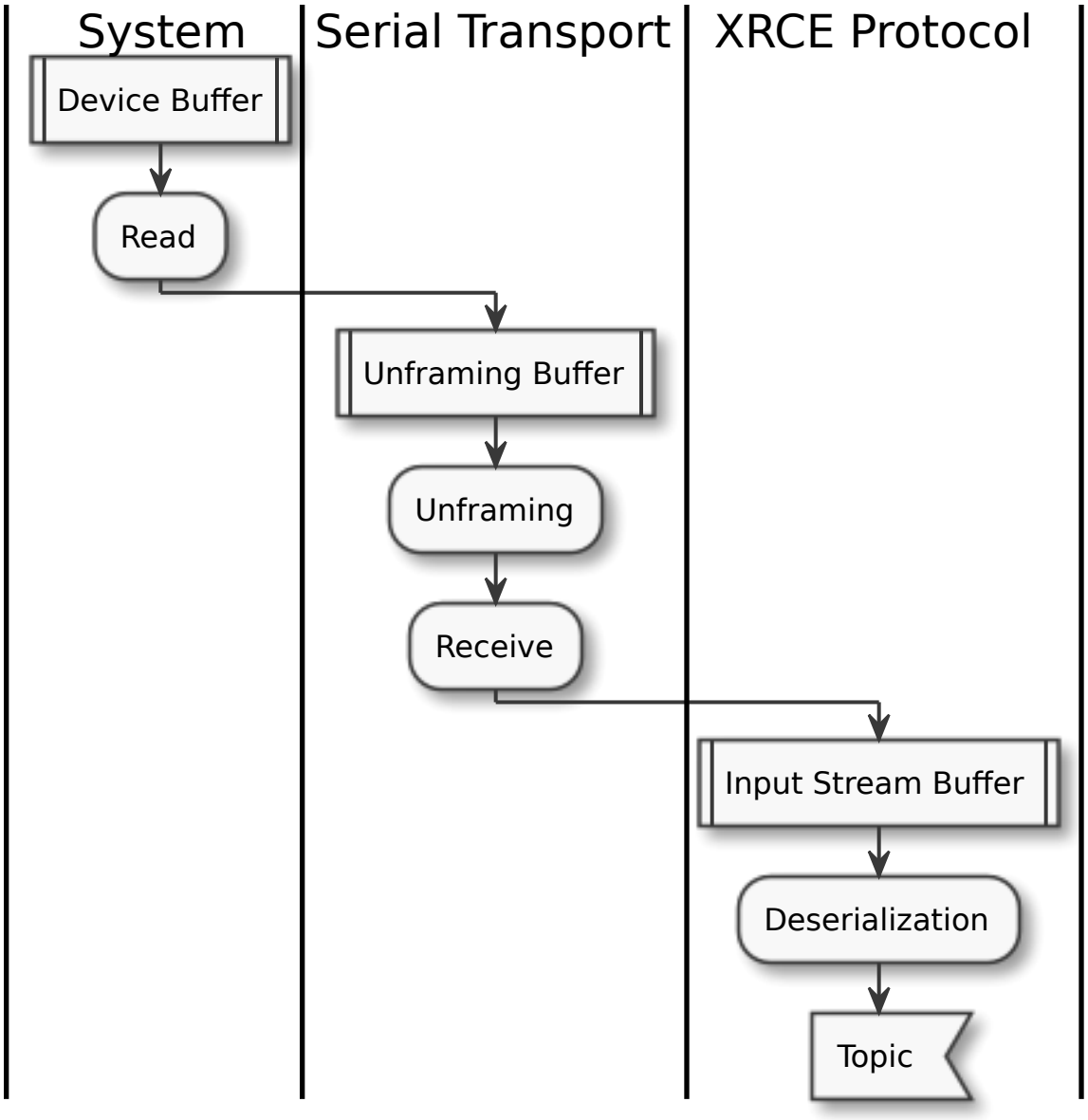
1. The HDLD framing and the CRC control provides **integrity** and **security** to the Serial Transport.
2. The framing technique allows to **reducing memory usage**. This is because the Framing Buffer size (42 bytes) bounds the Device Buffer size.

3. The framing technique also allows sending **large data** over serial. This is because the message size is not bounded by the Device Buffer size, since the message is fragmented and stuffing during the framing stage.

Data Receiving

The workflow of the data receiving is analogous to the data sending workflow:

1. A subscriber application calls the *Client* library to receive a given topic.
2. The *Client* library calls the Serial Transport to receive the serial message.
3. The Serial Transport reads data from the **Device Buffer** and unframes the raw data received from the Device Buffer in the **Unframing Buffer**.
4. Once the Unframing Buffer is full, the Serial Transport appends the fragment into the **Input Stream Buffer**. This operation is repeated until a complete message is received.
5. The *Client* library deserializes the topic from the Input Stream Buffer to the user topic struct.



It should point out that this approach has the same advantages that the sending one.

Shapes Topic Example

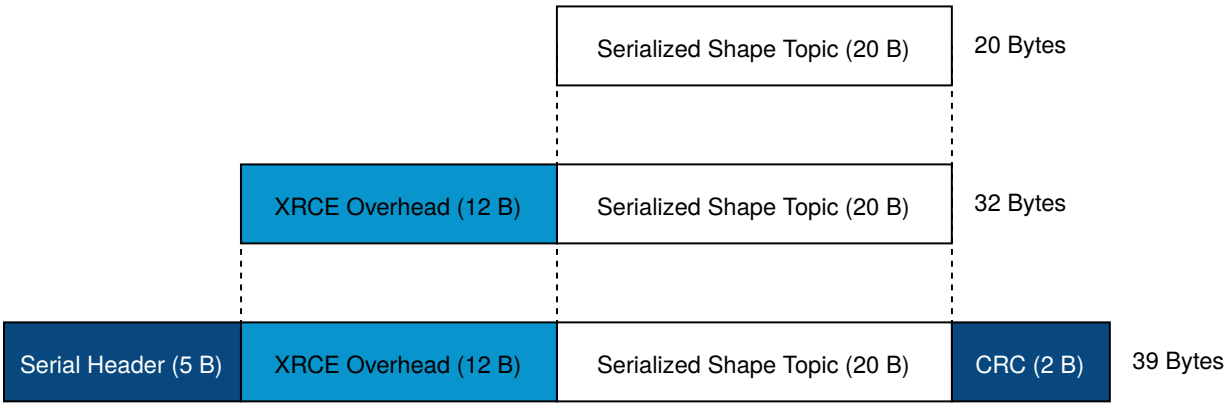
This subsection shows how a **Shapes Topic**, defined by the IDL below, is packed into the Serial Transport.

```
typedef struct ShapeType
{
    char color[128];
    int32_t x;
    int32_t y;
    int32_t shapesize;
} ShapeType;

ShapeType topic = {"red", 11, 11, 89};
```

In Serial Transport, the topic packaging could be divided into two steps:

1. The Session Layer adds the XRCE header and subheader. It adds an overhead of 12 bytes to the topic.
2. The Serial Transport adds the serial header, CRC and stuffing the payload. In the best case, it adds an overhead of 7 bytes to the topic.



The figure above shows the overhead added by Serial Transport. In the best case, it is **only 19 bytes**, but it should be noted that in this example the message stuffing has been neglected.

CHAPTER 4

Release notes

4.1 Version 1.1.0

Agent 1.1.0 | Client 1.1.1

This version include the following changes in both Agent and Client:

- **Agent 1.1.0:**
  - **Add**
    - Message fragmentation.
    - P2P communication.
    - API.
    - Time synchronization.
    - Windows discovery support.
    - New unitary tests.
    - API documentation.
    - Logger.
    - Command Line Interface.
    - Centralized middleware.
    - Remove Asio dependency.
  - **Change**
    - CMake approach.
    - Server's thread pattern.
    - Fast RTPS version upgraded to 1.8.0.
  - **Fix**

        * Serial transport.

- **Client 1.1.1:**
    - **Add**
        * Message fragmentation.
        * Time synchronization.
        * Windows discovery support.
        * New unitary tests.
        * API documentation.
        * Raspberry Pi support.
    - **Change**
        * Memory usage improvement.
        * CMake approach.
        * Discovery API.
        * Examples usage.
    - **Fix**
        * Acknack reading.
        * User data bad aligment.

4.1.1 Previous Versions

4.2 Version 1.0.3

Agent 1.0.3 | Client 1.0.2

This version includes the following changes in both Agent and Client:

- **Agent 1.0.3:**
    - Fast RTPS version upgraded to 1.7.2.
    - Baud rate support improvements.
    - Bugfixes.
- **Client 1.0.2:**
    - Uses new Fast RTPS 1.7.2 XML format.
    - Add Raspberry Pi toolchain.
    - Fix bugs.

4.3 Version 1.0.2

Agent 1.0.2 | Client 1.0.1

This version includes the following changes in the Agent:

- **Agent 1.0.2:**
  - Fast RTPS version upgraded to 1.7.0.
  - Added dockerfile.
  - Documentation fixes.

4.4 Version 1.0.1

Agent 1.0.1 | Client 1.0.1

This release includes the following changes in both Agent and Client:

- **Agent 1.0.1:**
  - Fixed Windows installation.
  - Fast CDR version upgraded.
  - Simplified CMake code.
  - Bug fixes.
- **Client 1.0.1:**
  - Fixed Windows configuration.
  - MicroCDR version upgraded.
  - Cleaned unused code.
  - Fixed documentation.
  - Bug fixes.

4.5 Version 1.0.0

This release includes the following features:

- Extended C topic code generation tool (strings, sequences and n-dimensional arrays).
- Discovery profile.
- Native write access profile (without using *eProsima Micro XRCE-DDS Gen*)
- Creation and configuration by XML.
- Creation by reference.
- Added *REUSE* flag at create entities.
- Added prefix to functions.
- Transport stack modification.
- More tests.
- Reorganized project.
- Bug fixes.
- API changes.

4.6 Version 1.0.0Beta2

This release includes the following features:

- Reliability.
- Stream concept (best-effort, reliable).
- Multiples streams of same type.
- Configurable data delivery control.
- C Topic example code generation tool.