D3.13

FIROS2
Software new release Y1

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D3.13
Deliverable name	FIROS2
Date	December 2018
Dissemination level	public
Workpackage and task	3.4
Author	Javier Moreno (eProsima)
Contributors	Borja Outerelo Gamarra (eProsima)
Keywords	micro-ROS, ROS2, FIWARE, intercommunication, NGSIv2 protocol, communication bridges, context broker
Abstract	This document provides links to the released software and documentation for deliverable D3.13 *FIROS2_new_Release* of the Task 3.4 *FIWARE Interoperability*.

Contents

1 Sumary **2**

2 Acronyms and keywords **2**

3 Overview to Results **2**

4 Links to Software Repositories **2**

5 Annex 1: Webpage on FIROS2 **3**

5.1 Interoperability 3

5.1.1 Mechanisms for the deserialisation of incoming data in the transformation library 3

5.1.2 Integration proposals 6

5.2 Demonstration 12

5.2.1 Linux demonstration 13

6 Annex 2: README.md from FIROS2 Package **15**

6.0.1 **Table Of Contents** 15

6.0.2 Installation of FIROS2 15

6.0.3 FIROS2 configuration 16

6.0.4 Transformation, mapping and communication 18

6.0.5 Types and interfaces 18

6.1 Dynamic Types 19

1 Sumary

FIROS2 is an application that allows intercommunication between *ROS2* and *NGSIv2* protocol. Since *FIROS2* is powered by *eProsima Integration Service*, it makes possible the creation of bidirectional communication bridges with customized routing, to map between input and output attributes, and perform data modification between *ROS2* and *NGSIv2* from *FIWARE-Orion contextBroker*.

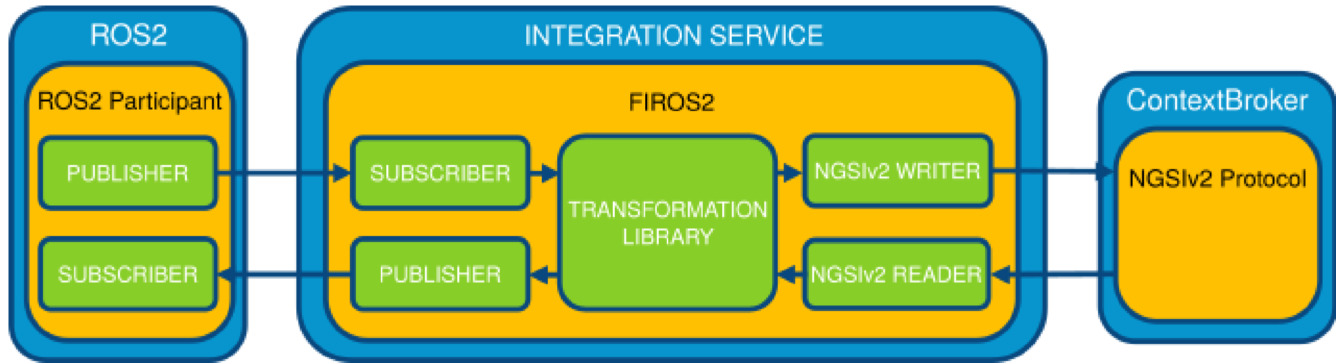

Figure 1: FIROS architecture

2 Acronyms and keywords

Term	Definition
ROS2	Robot Operating System 2
FIROS2	FIWARE - ROS2 bridge
IS	Integration service

3 Overview to Results

This document provides links to the released software and documentation for deliverable D3.13 *FIROS2_new_Release* of the Task 3.4 *FIWARE Interoperability*.

As an entry-point to all software and documentation, we created a dedicated webpage on the micro-ROS website: https://microros.github.io/FIROS2/

The annex includes a copy of this website and copies of the major documentation files of this software release.

4 Links to Software Repositories

The FIROS2 package is provided as a ROS2 package available at:

- Git repository: https://github.com/eProsima/FIROS2
  Package name: firos2
  Branch: feature/TCP_DynTypes
  Commits: a3c8e1a

FIROS2 package documentation available at:

- Git repository: https://github.com/eProsima/FIROS2
  File: ./README.md
  Branch: feature/TCP_DynTypes
  Commits: a3c8e1a

FIROS2 demo available at:

- Git repository: https://github.com/microROS/micro-ROS-demos
  Package name: int32_firos2
  Package path: ./Cpp/int32_FIROS2
  Branch: feature/FIROS2
  Commits: cb2bb33

5 Annex 1: Webpage on FIROS2

content of https://microros.github.io/FIROS2/ from 27th December 2018

5.1 Interoperability

This subsection will explain all the design alternatives for the interoperability of FIROS2 with micro-ROS.

5.1.1 Mechanisms for the deserialisation of incoming data in the transformation library

FIROS2 requires transformation libraries to convert ROS2 messages into FIWARE NGSIv2 messages and the other way around. For each message, one transformation library is required by the integration service (FIROS2).

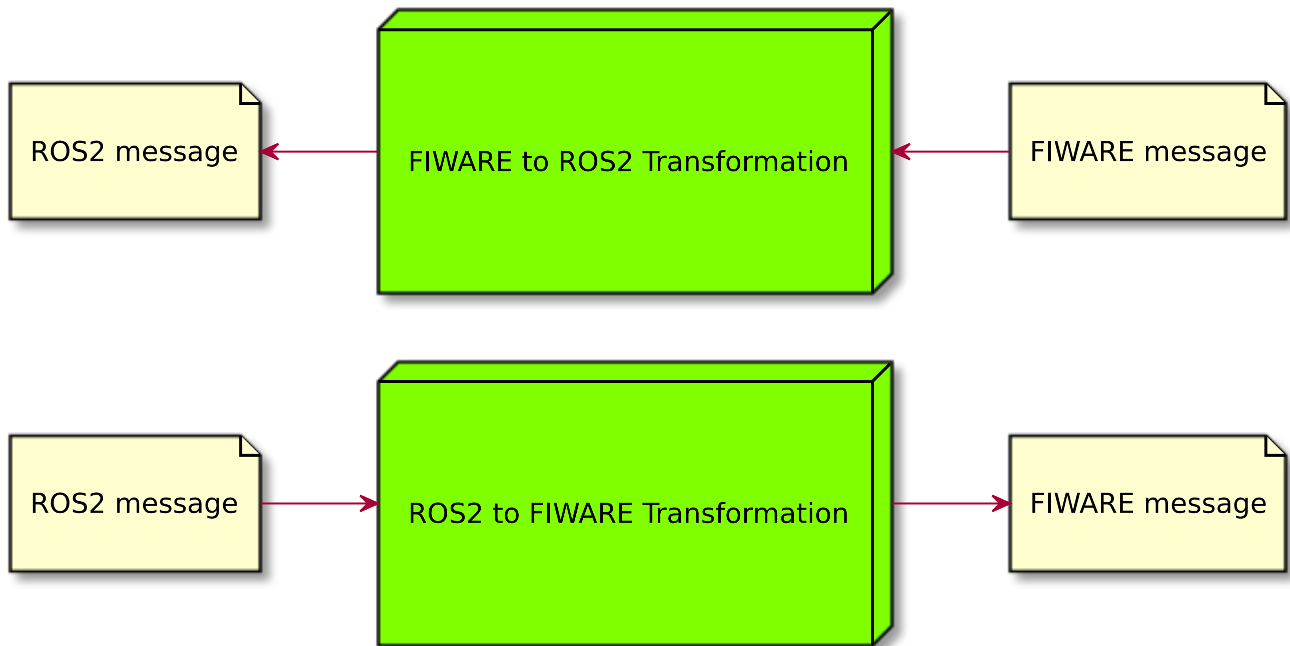

Figure 2: FIROS2 transformations

In the implementation of these transformation libraries, the user needs to be able to serialisation/deserialisation ROS2 messages. Also, an NGSIv2 serialisation/deserialisation mechanism will be used.

The FIROS2 package provides a standard NGSIv2 serialisation/deserialisation mechanisms, but ROS2 serialisation/deserialisation is not offered due to its dependencies with the message type.

For solving this issue, various methods to get it are proposed:

5.1.1.1 Use serialisation/deserialisation method provided by the middleware layer

This is currently the method used in micro-ROS - FIROS 2 integration.

In this case, the transformation library will use user selected middleware interface to serialise/deserialise the bridged ROS2 messages. This method requires to get the message typesupport for the bridged message type. This method is straightforward to implement as it does not require additional source code development. Also, the abstraction from the middleware implementation makes it more compatible with others ROS2 workspaces.

This is a portion of code used in the transformation library implementation.

```
1  extern "C" void USER_LIB_EXPORT transform(SerializedPayload_t *serialized_input,
       SerializedPayload_t *serialized_output){
2
3  // Get type support
4  const rosidl_message_type_support_t * type_support =
       rosidl_typesupport_cpp::get_message_type_support_handle<MESAGE_TYPE>();
5
```

```
6     // Convert to ROS2 serialized message
7     rmw_serialized_message_t serialized_message;
8     serialized_message.buffer = (char*)serialized_input->data;
9     serialized_message.buffer_length = serialized_input->length;
10    serialized_message.buffer_capacity = serialized_input->max_size;
11    serialized_message.allocator = rcutils_get_default_allocator();
12
13    // Deserialise
14    MESAGE_TYPE data;
15    if (rmw_deserialize(&serialized_message, type_support, (void*)&data) != RMW_RET_OK){
16        return;
17    }
18
19    // Transformation and NGSIv2 serialisation code here
20
21    }
```

Note the call to ROS 2 interface **rosidl_typesupport_cpp::get_message_type_support_handle**

5.1.1.2 Use serialisation/deserialisation method for an specific type support

In this case, the transformation library will use one specific type support to serialise/deserialise the bridged ROS2 messages. In micro-ROS case, the implementation to be used will be rosidl_typesupport_microxrcedds. This method is trivial to develop as it does not require additional source code on the micro-ROS side.

In the case of micro-ROS, the transformation library should use the serialisation/deserialisation API exposed by its typesupport, rosidl_typesupport_microxrcedds. This mechanism requires the user to have access to the typesupport API, which sometimes is not always possible.

5.1.1.3 Used serialisation/deserialisation method generated from IDL file

In this case, transformation library will use generated code to serialise/deserialise the bridged ROS2 messages. The generated code may be made using an IDL parser tool. In the micro-ROS case, Micro XRCE-DDS provides with Micro XRCE-DDS code generator, which accepts an IDL file as input and generates type code. This IDL files should correspond with those messages types the transformation is wanted. This is the integration service native method. Integration services uses this method, but it makes the development of the library slower as it needs to be generated per each message to be bridged.

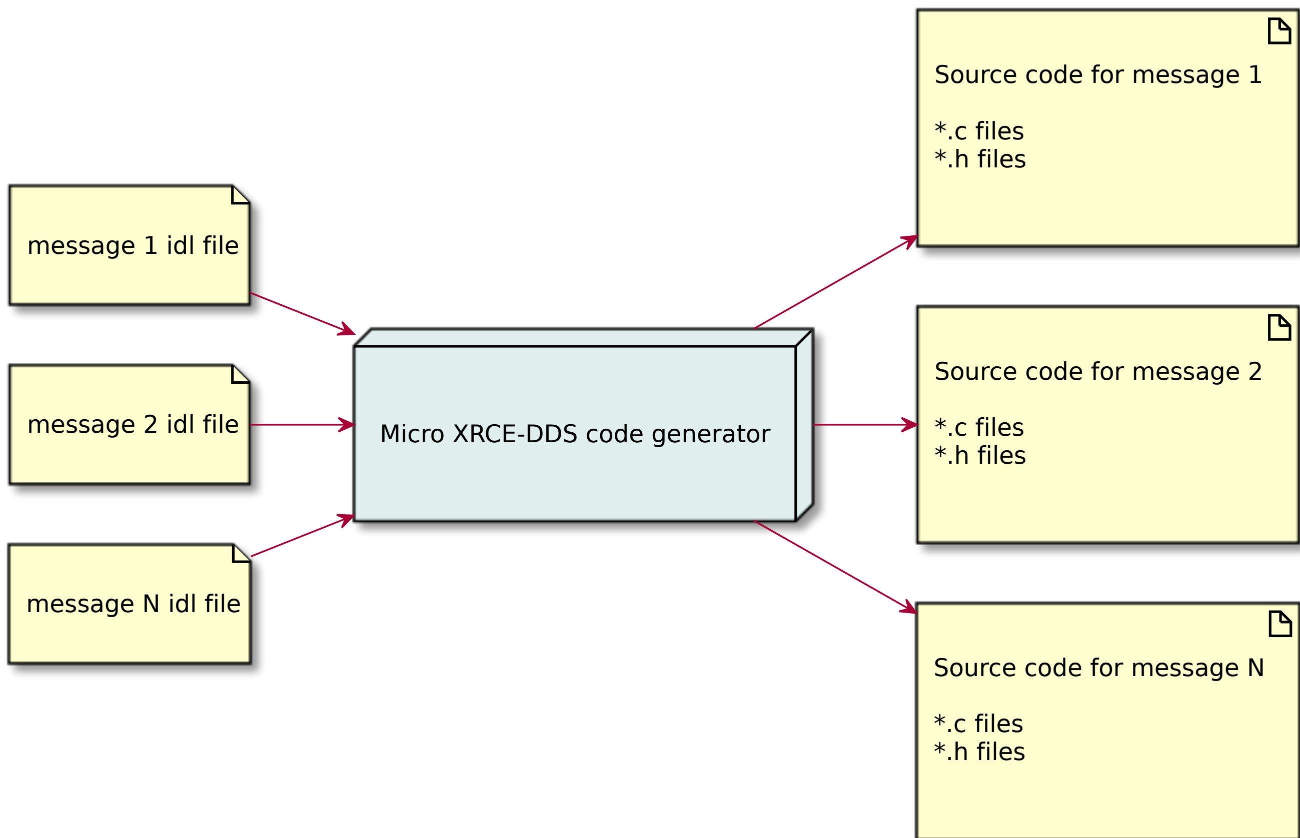

Figure 3: Micro XRCE-DDS Code generator

In the case of ROS2/micro-ROS workspaces, there are tools which generate those IDL files. The rosidl_gen package is the package micro-ROS/ROS2 could use to create IDL from ROS2 interfaces.

5.1.2 Integration proposals

Aside from transformation library implementations possibilities, micro-ROS could be integrated into different levels with FIROS2. This section presents all the integrations possibilities.

5.1.2.1 Direct integration

In this case, micro-ROS Agent will act as a bridge between DDS-XRCE and NGSIv2.

Selected bridged topics and their corresponding transformations must be configured on the micro-ROS Agent node.

This proposal requires micro-ROS Agent changes, but it is the direct native integration of micro-ROS and FIROS, without relying on DDS global data space.

Architecture

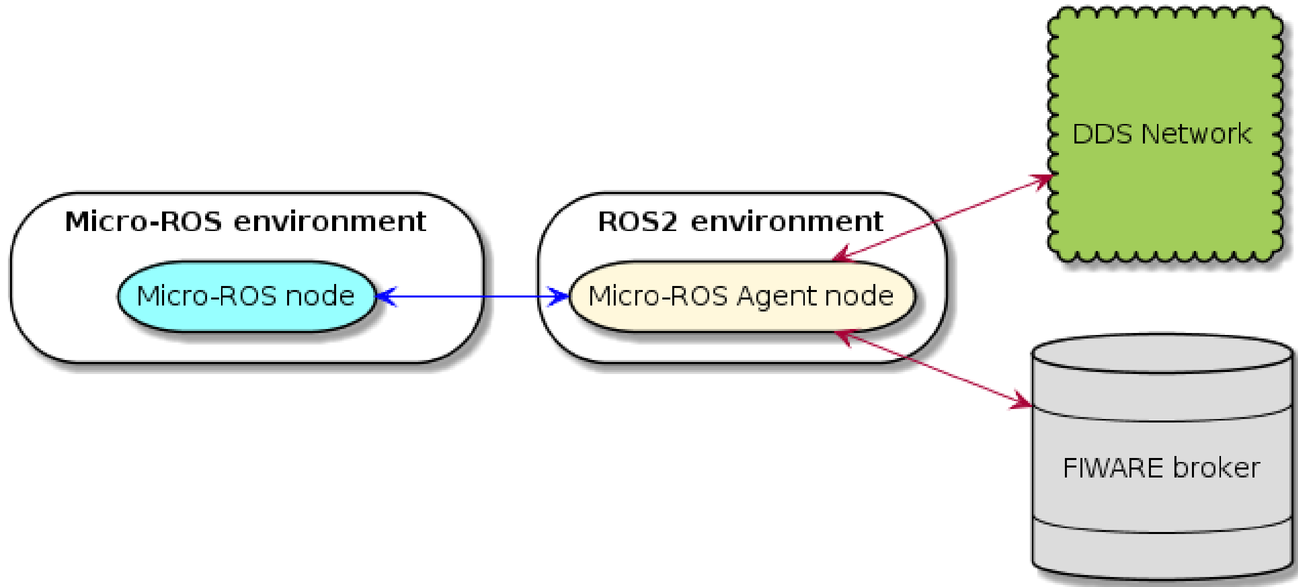

Figure 4: micro-ROS - FIROS2 direct interoperation

Use case

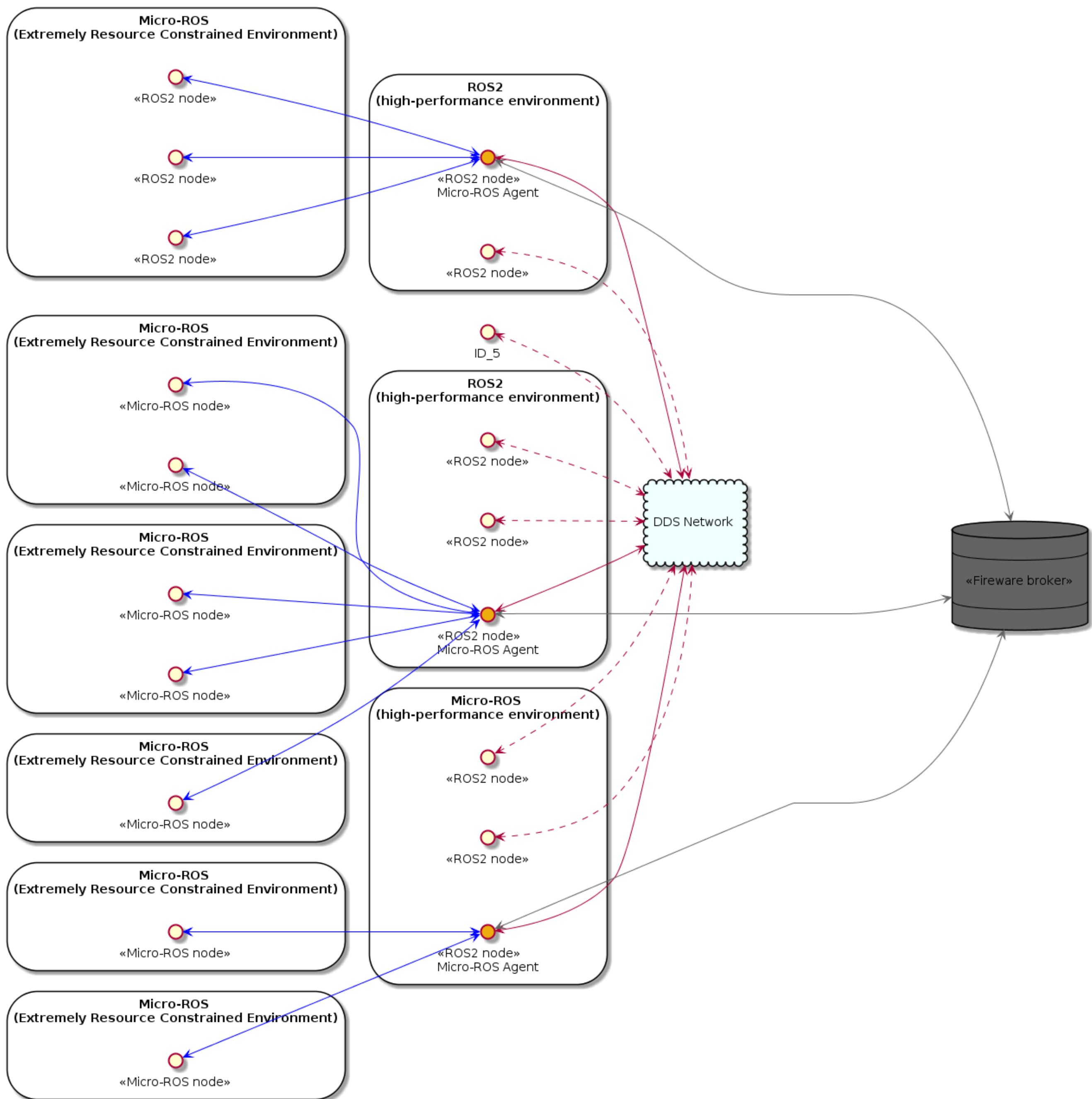

Figure 5: Direct interoperation use case example

5.1.2.2 Indirect integration with a single FIROS2 node

In this case, micro-ROS nodes will publish the configured topics on DDS, and a FIROS2 node will subscribe to those topics and convert them into NGSIv2 protocol. Selected bridged topics must be configured on that single FIROS2 node. Each ROS 2 topic type should have a corresponding transformation library configured on that FIROS2 node.

This approach is a specialisation of the following one, where all the configuration and transformation libraries are centralised in a single node.

This proposal requires transformation library development, but the integration will be the same as a regular ROS2 node, so no micro-ROS specific development should be expected.

Architecture

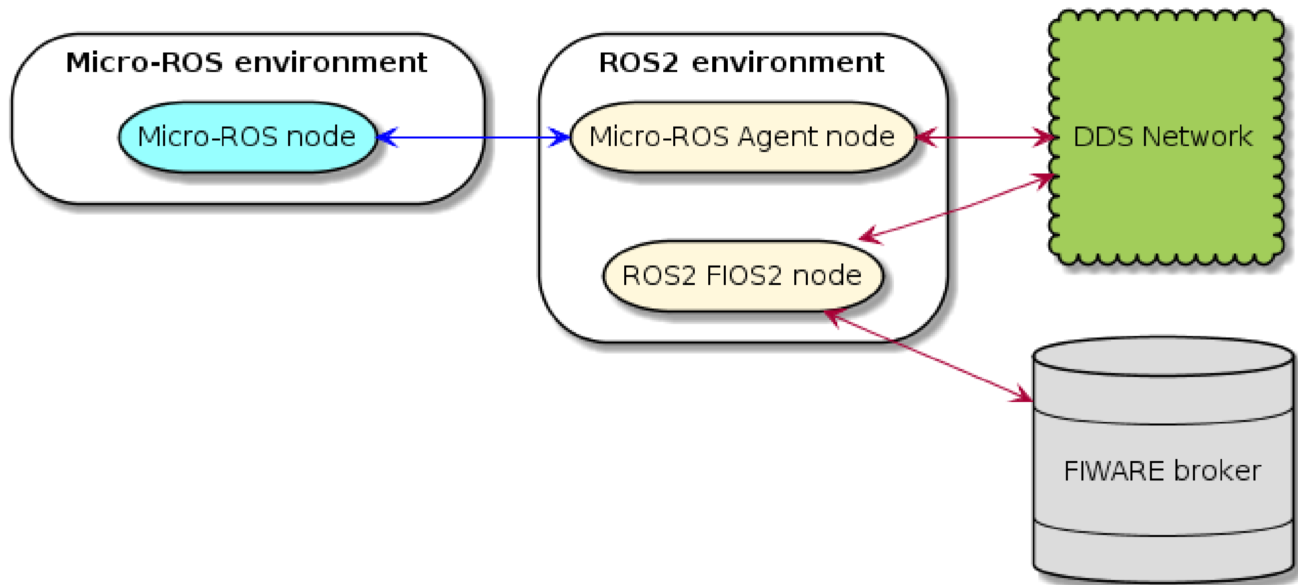

Figure 6: micro-ROS - FIROS 2 single node interoperation

Use Case

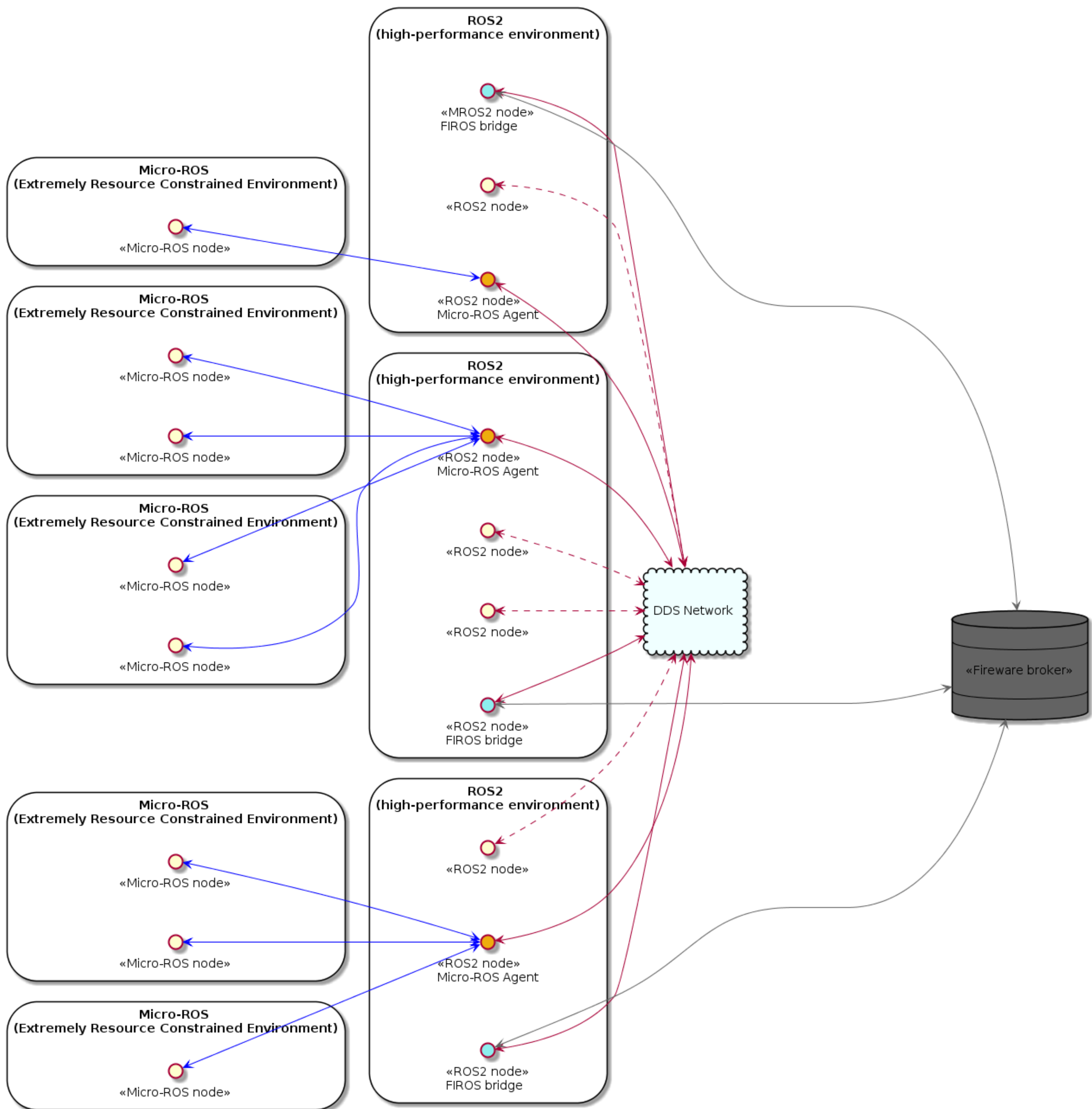

Figure 7: Single node use case example

5.1.2.3 Indirect integration with multiple FIROS2 nodes

In this case, micro-ROS nodes will publish the configured topic on DDS and multiple FIROS2 nodes, one for each set topic, will subscribe to those topics and convert them into NGSIv2 protocol.

This approach would require more nodes on the network and individual configurations.

This approach is the one followed by micro-ROS, and it is limited due to current FIROS 2 implementation.

This proposal requires transformation library development, but the integration will be the same as a regular ROS2 node, so no micro-ROS specific development should be expected.

Architecture

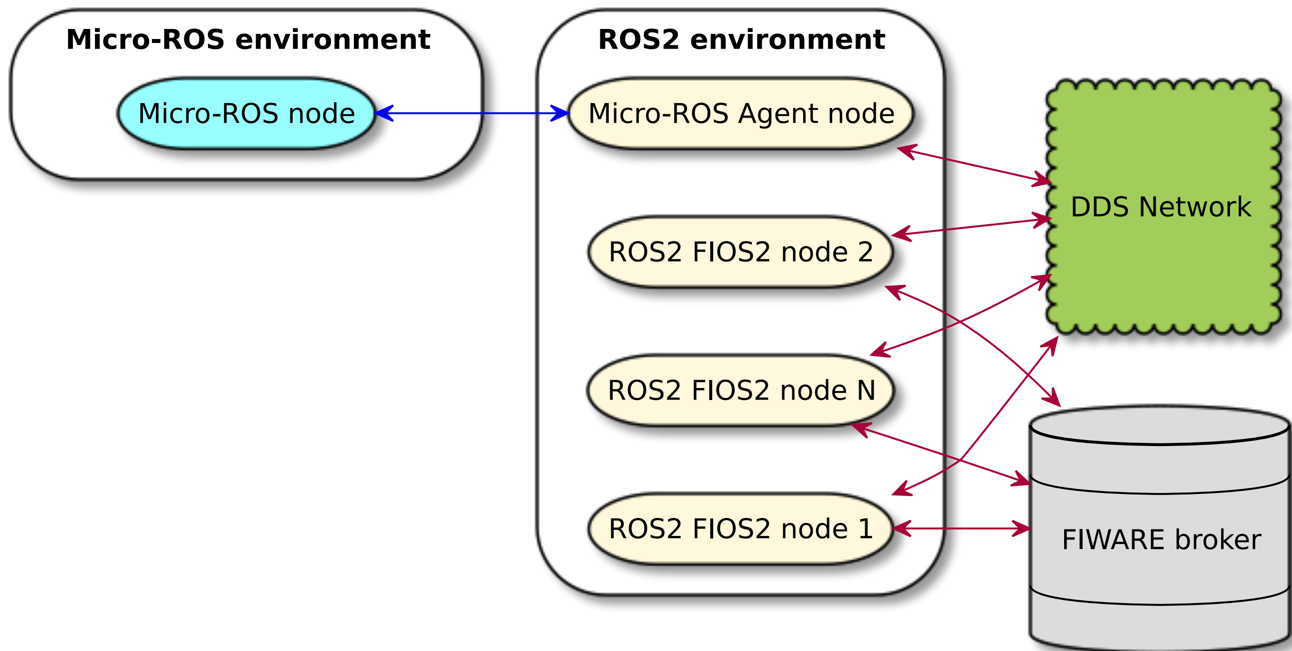

Figure 8: micro-ROS - FIROS 2 multiple nodes interoperation

Use Case

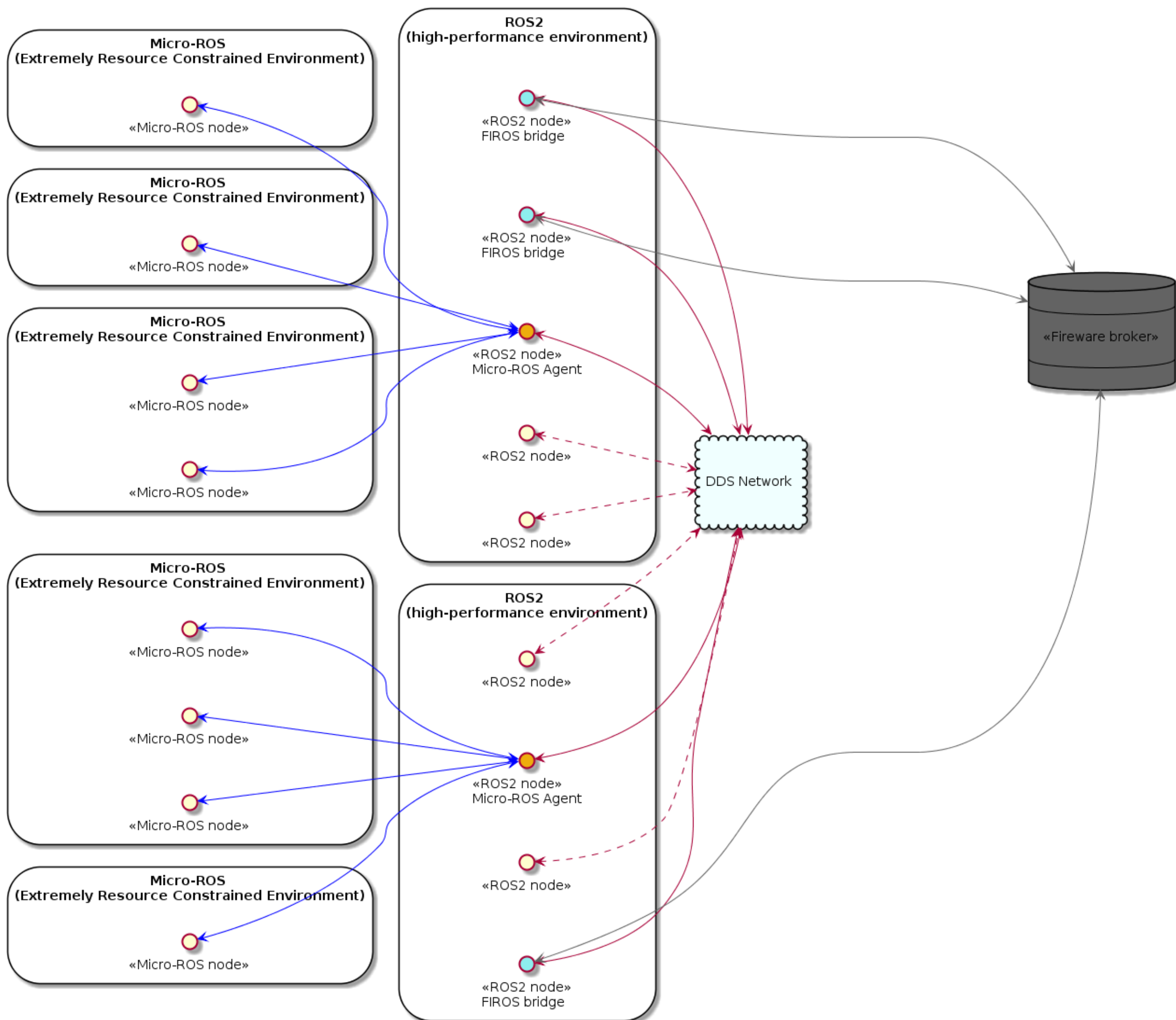

Figure 9: Multiple node use case

5.2 Demonstration

This section explains how to demonstrate the interoperability of FIROS2 with Micro-ROS. The purpose is to demonstrate the interoperability, although the final design is not closed.

To run the demonstration a step by step guide is presented in this document.

> **Note:** The only requirement to run the demonstration is to have docker CE and docker compose installed.

5.2.1 Linux demonstration

1. **Run Micro-ROS Agent node**

Download the pre-compiled agent and run it

```
docker pull microros/agent_linux
docker run  -it --rm  --privileged --net=host microros/agent_linux
```

Once inside the docker, raise the agent.

```
uros_agent udp 8888
```

> **Note:** After this step a micro-ROS Agent will be running at the localhost address and port 8888.

1. **Run a FIWARE Orion broker**

To test the communication it is necessary to have a FIWARE Orion server listening. The server will be run locally using a docker compose. The steps have been extracted from the docker hub official FIWARE repository.

For this, execute the following commands in a terminal and leave it open.

```
mkdir orion
cd orion
echo "mongo:
    image: mongo:3.4
    command: --nojournal
orion:
    image: fiware/orion
    links:
        - mongo
    ports:
        - \"1026:1026\"
    command: -dbhost mongo" > docker-compose.yml
sudo docker-compose up
```

> **Note:** After this execution a FIWARE Orion server will be running at the localhost address and port 1026.

1. **Build FIROS2 in a ROS2 workspace and run it**

To compile FIROS2, micro-ROS Agent side set of packages will be used.

For this, execute the following instructions.

```
1 docker pull microros/linux
2 docker run  -it --rm  --privileged --net=host microros/linux
```

Once in the Docker, all the necessary repositories should be downloaded and a FIROS2 node built and configured as a gateway of an int32.

```
1 mkdir -p ws/src
2 cd ws
3 wget https://raw.githubusercontent.com/microROS/micro-ROS-doc/feature/RepoListUpdate/\
4 Installation/repos/agent_minimum.repos
5 vcs import src < agent_minimum.repos
6 git clone -b feature/FIROS2 https://github.com/microROS/micro-ROS-demos.git src/uros/Demos
7 git clone --recursive -b feature/TCP_DynTypes https://github.com/eProsima/FIROS2.git src/uros/FIROS2
8 colcon build
9 . ./install/local_setup.bash
10 install/firos2/bin/firos2 install/int32_firos2/lib/int32_firos2/config.xml
```

1. **Run Micro-ROS client publisher**

Download a pre-compiled client and execute it.

```
1 docker pull microros/client_linux
2 docker run  -it --rm  --privileged --net=host microros/client_linux
```

Once in the docker run the micro-ROS Client.

```
1 int32_publisher_c
```

1. **Check that data has been uploaded into FIWARE Orion server**

In a Linux terminal execute the below sub-shell script

```
1 (
2     UPDATE_TIME="0.5"
3
4     curl  -v \
5         --include \
6         --header 'Content-Type: application/json' \
```

```
7         --request POST \
8         --data-binary '{  "id": "Helloworld",
9                           "type": "Helloworld",
10                          "$ATTRIBUTE": {
11                              "value": "0",
12                              "type": "Number"
13                          }
14                      }' \
15        'localhost:1026/v2/entities'
16
17    (
18        while (( 1 ))
19        do
20        curl -v "localhost:1026/v2/entities/Helloworld/attrs/count/value?type=Helloworld"
21        echo ""
22        sleep $UPDATE_TIME
23        done
24    )
25 )
```

For further information please refer to the official FIROS2 documentation: FIROS2 documentation

6 Annex 2: README.md from FIROS2 Package

Content of https://github.com/eProsima/FIROS2/blob/bf344cfeb44a2d6065fa8e41634767348cc30679/README.md from 27th December 2018.

eProsima FIROS2 is an application that allows intercommunication between *ROS2* and *NGSIv2* protocol. Since *FIROS2* is powered by *eProsima Integration Service*, it makes possible the creation of bidirectional communication bridges with customized routing, to map between input and output attributes, and perform data modification between *ROS2* and *NGSIv2* from *FIWARE-Orion contextBroker*.

6.0.1 Table Of Contents

Installation

Configuration

Transformation, mapping and communication

Types and interfaces

Dynamic Types

6.0.2 Installation of FIROS2

Before using *FIROS2*, it has to be installed along with the rest of *ROS2* packages of your system. If you have followed the *ROS2* installation manual, you only need to clone this repository on your *ROS2* workspace.

For cloning this project and update its submodules at the same time, don't forget to add the *–recursive* option.

In Linux, these are the steps:

```
1   $ cd ~/ros2_ws/src/ros2/
2   $ git clone --recursive https://github.com/eProsima/firos2
3   $ cd ~/ros2_ws
```

In the case of Windows:

```
1   > cd C:\dev\ros2\src\ros2
2   > git clone --recursive https://github.com/eProsima/firos2
```

On windows you must compile *cURL Library* as a thirdpary submodule:

```
1   > cd C:\dev\ros2\src\ros2\firos2\thirdparty\curl
2   > buildconf.bat
3   > cd winbuild
4   > nmake /f Makefile.vc mode=dll VC=14
5   > cd C:\dev\ros2
```

Once done, *FIROS2* is compiled like any other *ROS2* package.

For example:

```
1   $ colcon build --cmake-args -DTHIRDPARTY=ON --packages-select firos
```

There are several examples to show the behavior under the examples folder.

6.0.3 FIROS2 configuration

FIROS2 offers different parameters that the user can configure. For setting-up a *bridge*, the user has to define a configuration file with the information about input and output protocols. There is a generic example on **config.xml**

In this template is it possible to set different bridges between topics and entities. *FIROS2's bridges* subscribe to a topic and update data of the related entity and subscribe to entities and publish data to the related topic. The parameters that have to be defined are (only shown a *bridge* section of the *config.xml* file):

```
1 <!-- Declares a custom bridge named 'bridge_ngsiv2' -->
2 <bridge name="bridge_ngsiv2">
3     <!-- Path to the NGSIv2 library -->
4     <library>/path/to/ngsiv2bridge.so</library>
5
```

```
6   <reader name="sub_ngsiv2">
7       <property>
8           <name>id</name>
9           <value>entity_idPattern</value>
10      </property>
11      <property>
12          <name>host</name>
13          <value>context_broker_host</value>
14      </property>
15      <property>
16          <name>port</name>
17          <value>context_broker_port</value>
18      </property>
19      <property> <!-- optional -->
20          <name>type</name>
21          <value>entity_type</value>
22      </property>
23      <property> <!-- optional, comma separated values -->
24          <name>attrs</name>
25          <value>attr1[,attr2...]</value>
26      </property>
27      <property> <!-- optional -->
28          <name>expression</name>
29          <value>condition_expression</value>
30      </property>
31      <property> <!-- optional, comma separated values -->
32          <name>notif</name>
33          <value>notif_attr1[,notif_attr2...]</value>
34      </property>
35      <property>
36          <name>listener_host</name>
37          <value>our_listener_host</value>
38      </property>
39      <property>
40          <name>listener_port</name>
41          <value>our_listener_port</value>
42      </property>
43      <property>
44          <!-- optional. In KB, if not specified, 2KB -->
45          <name>listener_buffer_size</name>
46          <value>our_listener_buffer_size</value>
47      </property>
48      <property> <!-- optional -->
49          <name>expiration</name>
50          <value>subscription_expiration_time</value>
51      </property>
52      <property> <!-- optional -->
```

```
53            <name>throttling</name>
54            <value>subscription_throttling</value>
55        </property>
56        <property> <!-- optional -->
57            <name>description</name>
58            <value>subscription_description</value>
59        </property>
60    </reader>
61
62    <writer name="pub_ngsiv2">
63        <property>
64            <name>host</name>
65            <value>context_broker_host</value>
66        </property>
67        <property>
68            <name>port</name>
69            <value>context_broker_port</value>
70        </property>
71    </writer>
72 </bridge>
```

6.0.4 Transformation, mapping and communication

As said before, when a *bridge* is connecting two nodes with different protocols, the user has to provide a library with a function to transform and to map the attributes from one protocol to another. To make this step easier, there is an empty code template in **templatelib.cpp**.

This function will be compiled apart and loaded in *FIROS2* at runtime.

In this way, the user can map attributes from the input to the output message and at the same time apply changes over the data. The serialization and deserialization functions are generated with provided tools, so the *IDL* files used are the only thing that the user has to put in the bridge.

FIROS2 provides a built-in *NGSIv2 bridge library* named *libisbridgengsiv2lib.so* that implements an *ISBridgeNGSIv2* with *NGSIv2Publisher* and *NGSIv2Subscriber* that allows communicating RTPS and NGSIv2, implementing the interfaces *ISBridge*, *ISWriter*, and *ISReader* respectively.

You can, of course, implement and use your *bridge libraries* to define other behaviors.

You can learn more about *Bridge Libraries* and *Transformation Libraries* in the documentation of *eProsima Integration Service*.

6.0.5 Types and interfaces

For interaction with *NGSIv2* entities exists an *IDL* file (and generated files). This *IDL* is named **JsonNGSIv2.idl** and contains a structure composed of two strings, *entityId*, and *data*.

On receiving messages from *NGSIv2* protocol, it only fills the *data* field. In this case, it ignores the *entityId* field, and it's better to keep it empty. *Integration Service* will fill *data* with the complete JSON string sent to our listener by the *contextBroker* server (this is, the subscription result).

For sending messages to the *contextBroker* to update entities from changes received from *RTPS* subscriber, *JsonNGSIv2" must fill* entityId* with the *entityId* of the entity modified and *data* with a composed JSON containing the attributed to being updated.

The user must implement the interaction with the *NGSIv2* entities in the *transformation library*.

In the example **TIS_NGSIv2** the *transformation library* shows examples of both transformations using the described behavior.

You can create your *IDL* files to define the required behavior and management of the data. To get a deeper comprehension of the relation between *ROS2* and *Fast RTPS* IDL definitions, you can see this article.

6.1 Dynamic Types

An example of integration with **Dynamic Types** can be found in the *Dyn_TIS_NGSIv2* example. *RobotExample* executable uses static types (to show compatibility), but *ROS2* publisher/subscriber uses *DynamicTypes*, as well as *NGSIv2* that uses a dynamic version of the *NGSIv2 JSON* library. All *Dynamic Types* related files are under *DynNGSIv2* folder, that generates an additional dynamic library to be used with *dynamic types transformation libraries*.