Integration Service Documentation

Release 1.0.0

eProsima

Jul 28, 2020

INSTALLATION MANUAL

1 Main Features **3**

1.1 System Handles . . . 3

1.2 YAML configuration files . . . 4

1.3 Additional features . . . 5

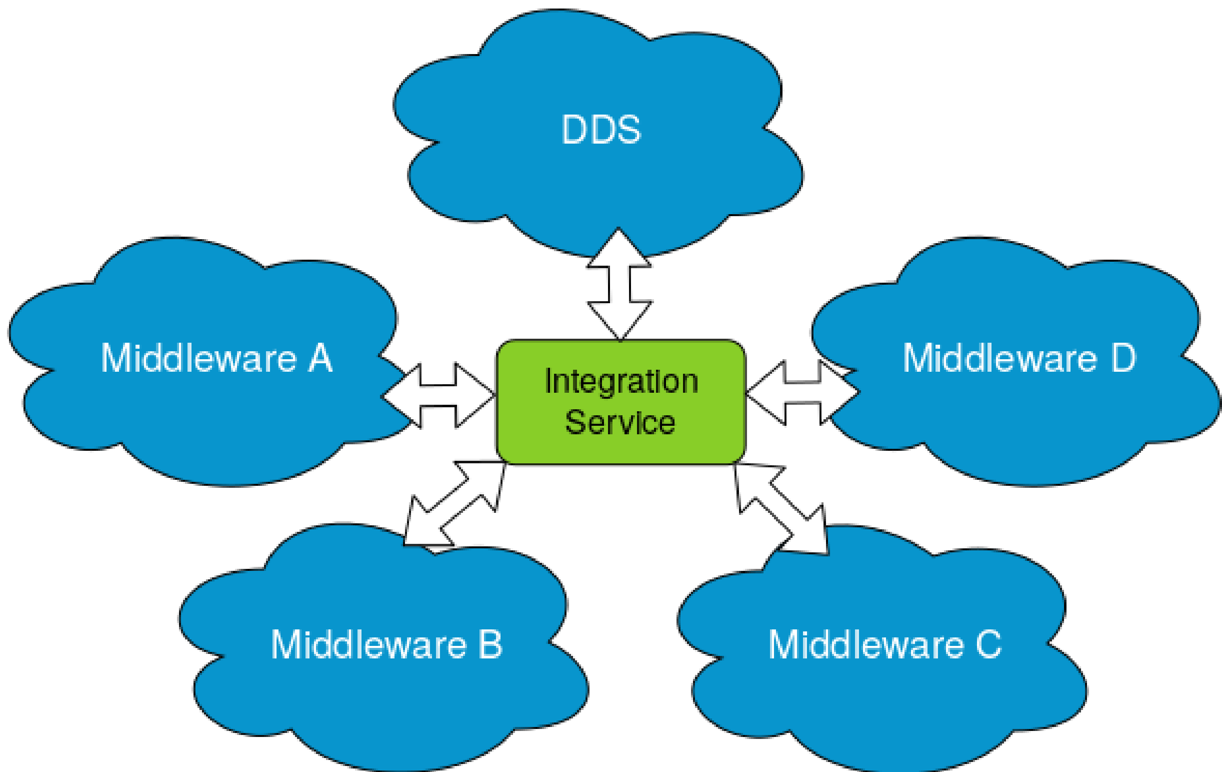
2 Structure of the Documentation **7**

2.1 Installation Manual . . . 7

2.2 User Manual . . . 10

eProsima Integration-Service is a tool based on SOSS that allows intercommunicating any *DDS*-based system with any other protocol, including other *DDS* systems, integrating them into a larger, more complex system.

eProsima Integration-Service can be configured with a YAML text file, through which the user can provide a mapping between the topics and services on the *DDS*-based middleware and those on the system(s) to which the user wants to bridge it.



CHAPTER

ONE

MAIN FEATURES

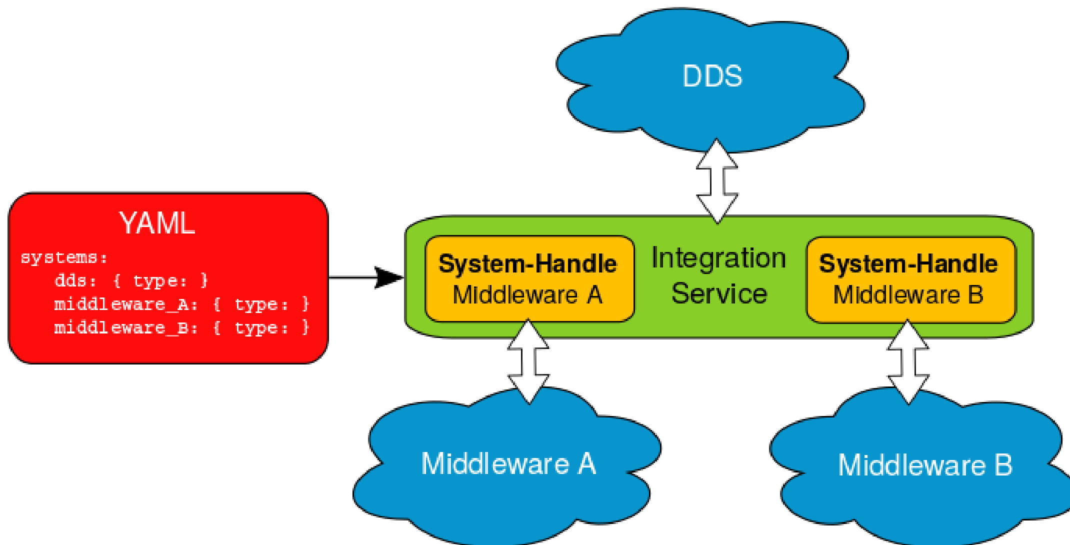
eProsima Integration-Service provides a plugin-based platform that is easily and intuitively configurable. This section explains these key features.

1.1 System Handles

An *eProsima Integration-Service* instance can connect *N* middlewares through dedicated plugins that speak the same language as the core. This common language is eProsima xtypes; a fast and lightweight OMG DDS-XTYPES standard C++11 header-only implementation. The plugins, or **System-Handles**, are discovered by *eProsima Integration-Service* at runtime after they have been installed.

Built-in **System-Handles** are provided for connecting *Orion ContextBroker*, *ROS*, *ROS2*, and *WebSocket* to the *DDS* world. New **System-Handles** for additional protocols can be easily created, automatically allowing communication of the new protocol with *DDS* and with the middlewares that are already supported (detailed information on how to create a **System-Handle** can be found here). Thanks to this, downstream users can extend *eProsima Integration-Service* to communicate *DDS*-based systems with any middleware.

The plugin-based framework is especially advantageous when it comes to integrating a new *DDS* component into a complex system where the rest of sub-systems use incompatible protocols, or viceversa. Indeed, once all protocols of interest are communicated with *eProsima Integration-Service*, each via a dedicated **System-Handle**, the integration is mediated by the core and relies on centralization rather than on the creation of dedicated bridges for each pair of components. For a system made of *N* components, this means that the number of new software parts to add grows as N rather than N^2.



1.2 YAML configuration files

eProsima Integration-Service is configured by means of a YAML file that specifies a set of compulsory fields, plus some optional ones. The most common fields required to configure a **System-Handle** are:

- `types`: specifies the IDL types used by *eProsima Integration-Service* to transmit messages.
- `systems`: specifies the middlewares involved in the communication.
  - `types-from`: allows the middleware to inherit the type from another system.
- `routes`: specifies which bridges *eProsima Integration-Service* needs to create.
- `topics/services`: specify the topics exchanged over the above bridges in either publisher/subscriber or client/server type communications.

This configuration approach is profitable when *DDS* is integrated into complex systems, since a single YAML file is needed no matter how many protocols are being communicated.

Below you can find a minimal example of the information that the YAML configuration file should contain. In this example, a single topic is translated from *ROS2* to *DDS*:

```yaml
systems:
    ros2: { type: ros2 }
    dds: { types-from: ros2 }
topics:
    chatter: { type: std_msgs/String, route: {from: ros2, to: dds} }
```

The versatility of *eProsima Integration-Service* is that it offers the possibility to operate different translations by only changing the configuration file. For example, by changing the specified middlewares, one can obtain an instance which translates between *WebSocket+JSON* (as produced and consumed by a standard Web browser) and *DDS*:

```yaml
types:
    idls:
        ->
            module std_msgs
```

(continues on next page)

(continued from previous page)

```
            {
                struct String
                {
                    string data;
                };
            };
systems:
    web: { type: websocket_client, types-from: robot, host: localhost, port: 12345 }
    robot: { type: dds }
routes:
    web2robot: {from: web, to: robot}
topics:
    chatter: { type: "std_msgs/String", route: web2robot }
```

For more information on how to configure *eProsima Integration-Service* via YAML files, please refer to the following link.

1.3 Additional features

Free and Open Source.

The *eProsima Integration-Service* core, and all **System-Handles** available to date are free and open source. Consult the *Related Links* section of the documentation to be redirected to the relevant repositories.

Easily configurable.

As detailed above, an *eProsima Integration-Service* instance is easily configurable by means of a YAML file. For more information on how to do so, please consult the link: YAML Configuration.

Easy to extend to new platforms.

New platforms can easily enter the *eProsima Integration-Service* world by generating the plugin, or **System-Handle** needed by the core to integrate them. For more information on **System-Handles**, please consult the link: System-Handle Creation.

Easy to use.

Installing and running *eProsima Integration-Service* is intuitive and straightforward. Please refer to the *Getting Started* section to be guided through the installation process.

Commercial support.

Available at support@eprosima.com

CHAPTER

TWO

STRUCTURE OF THE DOCUMENTATION

This documentation is organized into the following sections.

2.1 Installation Manual

This section is meant to provide the user with an easy-to-use installation guide and is organized as follows:

2.1.1 External Dependencies

eProsima Fast DDS

eProsima Fast DDS can be installed following the instructions at: Fast DDS.

Additional Dependencies

Each **System-Handle** may have additional dependencies. Please refer to the concrete *SOSS* documentation at the following link.

CMake

CMake 3.5 is required to build the project files.

C++

eProsima Integration-Service uses standard C++14.

colcon

If installed using colcon, colcon becomes a dependency.

Related Links

- eProsima Fast-RTPS
- SOSS documentation
- Colcon Manual
- **System-Handle** repositories

System-Handle	Repository
SOSS-ROS2, **SOSS-WEBSOCKET**, **SOSS-MOCK**, **SOSS-ECHO**	https://github.com/eProsima/soss_v2/tree/feature/xtypes-dds
SOSS-DDS	https://github.com/eProsima/SOSS-DDS/tree/feature/xtypes-ddss
SOSS-ROS1	https://github.com/eProsima/soss-ros1/tree/feature/xtypes-support
SOSS-FIWARE	https://github.com/eProsima/SOSS-FIWARE/tree/feature/xtypes-support

2.1.2 Getting Started

Table of Contents

- *Installation*
- *Deployment*
- *Getting Help*

This section is meant to provide the user with an easy-to-use installation guide, and an explication of how to launch an *eProsima Integration-Service* instance.

Installation

The *eProsima Integration-Service* repository consists of many cmake packages which can be configured and built manually, but we recommend to use colcon, as it makes the job much smoother.

Create a colcon workspace and clone the SOSS and the SOSS-DDS repositories:

```
mkdir ~/is-workspace
cd ~/is-workspace
git clone ssh://git@github.com/eProsima/soss_v2 src/soss --recursive -b feature/
↪xtypes-dds
git clone ssh://git@github.com/eProsima/SOSS-DDS src/soss-dds --recursive -b feature/
↪xtypes-dds
```

Note: The `--recursive` flag in the first `git clone` command is mandatory to download some required third-parties. The `--recursive` flag in the second command installs eProsima Fast DDS, which is a requirement of *eProsima Integration Service*. In this way, both *eProsima Integration Service* and *eProsima Fast DDS* will be installed in the same `~/is-workspace/install` directory. The use of this second `--recursive` flag can be omitted in case you have *eProsima Fast DDS* already installed in your system. Note that if your installation is local, you'll need to source it before running any *eProsima Fast DDS* application and *eProsima Integration-Service* instance.

Once *eProsima Integration-Service* is in the `src` directory of your colcon workspace, you can build the packages by running:

```
colcon build
```

If any package is missing dependencies **causing the compilation to fail**, you can add the flag `--packages-up-to soss-dds-test` to make sure that you at least build `soss-dds-test`:

```
colcon build --packages-up-to soss-dds-test
```

Note: `colcon build` will build the package `soss-core` and all the built-in **System-Handles**. If you don't want to build the built-in **System-Handles** you can execute `colcon build --packages-up-to soss-core`. If you only want a to build a sub-set of built-in **System-Handles** you can use the same directive with the name of the packages, for example:

```
colcon build --packages-up-to soss-ros2 soss-fiware
```

The built-in **System-Handles** packages are:

- `soss-ros2`: ROS2 **System-Handle**.
- `soss-websocket`: WebSocket **System-Handle**.
- `soss-mock`: Mock **System-Handle** for testing purposes.
- `soss-echo`: Echo **System-Handle** for example purposes.

Additional **System-Handles** can be found in their own repositories:

- `soss-fiware`: Fiware Orion ContextBroker System-Handle.
- `soss-ros1`: ROS System-Handle.
- `soss-dds`: DDS System-Handle.

Most of the **System-Handle** packages include a `-test` package for testing purposes.

Once that's finished building, and before launching an *eProsima Integration-Service* instance with the `soss` command, you can source the new colcon overlay:

```
source install/setup.bash
```

Deployment

You can now run an *eProsima Integration-Service* instance it in order to bring an arbitrary number of middlewares into the *DDS* world.

The workflow is dependent on the specific middlewares involved in the desired communication, given that each is integrated into *eProsima Integration-Service* via a dedicated **System-Handle**.

First of all, you will have to clone the repositories of the **System-Handles** that your use-case requires into your `is-workspace`. To know which are the **System-Handles** supported to date, refer to the *Related Links* section of this documentation.

Once all the necessary packages have been cloned, you need to build them. To do so, run:

```
colcon build
```

with the possible addition of flags depending on the specific use-case. Once that's finished building, you can source the new colcon overlay:

```
source install/setup.bash
```

The workspace is now prepared for running an *eProsima Integration-Service* instance. From the fully overlaid shell, you will have to execute the `soss` command, followed by the name of the YAML configuration file that describes how messages should be passed among *DDS* and the middlewares involved:

```
soss <config.yaml>
```

Once *eProsima Integration-Service* is initiated, the user will be able to communicate the desired protocols.

Note: The sourcing of the local colcon overlay is required every time the colcon workspace is opened in a new shell environment. As an alternative, you can copy the source command with the full path of your local installation to your `.bashrc` file as:

```
source ~/is-workspace/install/setup.bash
```

Getting Help

If you need support you can reach us by mail at support@eProsima.com or by phone at +34 91 804 34 48.

2.2 User Manual

In this section we discuss the most representative use-cases demonstrating *eProsima Integration-Service*'s functionalities. For each use-case, a related example is presented and the user is guided step-by-step through the installation protocol and environment preparation necessary to have the examples set up and working. It is organized as follows:

2.2.1 Use-cases and Examples

Typical Use-cases

Typical scenarios in which *eProsima Integration-Service* is relevant are:

- Communication of different *DDS*-based systems that use incompatible configurations.
- Communication of a *DDS* system with systems with incompatible protocols.
- Integrating *DDS* systems into arbitrarily complex systems using different protocols.
- Communication between *DDS* systems located in different geographical regions through the Internet.

In this user manual we discuss representative use-cases demonstrating these *eProsima Integration-Service*'s functionalities. For each use-case, a related example is presented and the user is guided step-by-step through the installation protocol and environment preparation necessary to have the examples set up and working.

Namely, we will go through the following:

Use-cases	Examples
DDS bridge	*Example: ROS2 communication*
Add compatibility to an unsupported protocol	*Example: ROS1 communication*
Integrate a large system	*Example: Integrate Orion Context-Broker and ROS1 into DDS*
WAN communication	*Example: WAN TCP tunneling*

Important remarks

A compulsory prerequisite for running the examples of the following sections is to have *eProsima Integration-Service* correctly installed as explained in the introductory section *Getting Started*. Please make sure to follow all the steps described in the document before proceeding.

Whenever you run the `colcon build` command in the examples provided, if any package is missing dependencies **causing the compilation to fail**, you can add the flag `--packages-up-to soss-dds-test` as follows:

```
colcon build --packages-up-to soss-dds-test
```

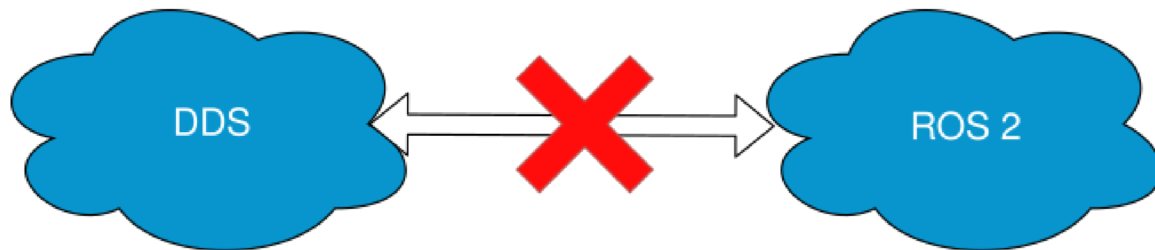
Also notice that, for being able to execute *eProsima Integration-Service* with the `soss` command at the end of each example, the shell must be fully overlaid with the sourcing of all colcon-built packages required by the specific use-case:

- The *eProsima Integration-Service* installation, along with that of any possible **System-Handle** that might be required by the specific example (e. g., **SOSS-FIWARE** and **SOSS-ROS1**).
- The *ROS2* or *ROS1* installation, when needed.

As an alternative, you can install permanently the overlays relevant to your use-case system-wide, by adding the opportune `source` commands to your `.bashrc` file.

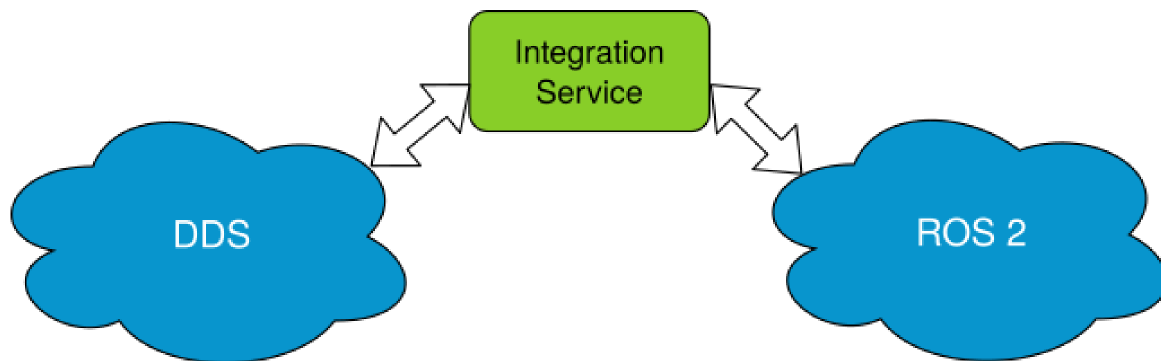
2.2.2 DDS bridge

A typical scenario faced when bridging different *DDS*-based systems is that these systems use incompatible configurations. This happens for example in the communication between *DDS* and *ROS2*.



A user with knowledge of both systems may be aware that *ROS2* uses *DDS* as a middleware but hides some of *DDS*' configuration details, thus making a direct communication between the two difficult, if not impossible. By using *eProsima Integration-Service*, this communication can be eased and achieved with minimal effort from the user's side.

This section is intended to illustrate the *DDS-ROS2* communication as an example of this type of *eProsima Integration-Service*-mediated bridge, by putting into communication a *ROS2 talker-listener* example with a *Fast-RTPS* HelloWorld example.



Example: ROS2 communication

Requirements

To prepare the deployment and setup the environment, you need to have *eProsima Integration-Service* correctly installed in your system. To do so, please follow the steps delined in *Getting Started* and read carefully the *Important remarks* section.

To get this example working, the following requirements must be met:

- Having *ROS2* (Crystal or superior) installed, with the *talker-listener* example working.
- Having the HelloWorldExample compiled. To do so, go to the `~/is-workspace/src/soss-dds/examples/common/HelloWorldExample` folder and type:

```
mkdir build
cd build
cmake ..
make
```

- Having the ros2_std_msgs compiled. To do so, go to the `~/is-workspace/src/soss-dds/examples/common/ros2_std_msgs` folder and type:

```
mkdir build
cd build
cmake ..
make
```

- Having the **SOSS-ROS2 System-Handle** installed. Unless configured otherwise, this package is built automatically when *eProsima Integration-Service* is installed.

ROS2 talker to DDS subscriber

To enable communication from ROS2 to DDS, open three terminals:

- In the first terminal, execute a *ROS2* `talker`

```
source /opt/ros/$ROS2_DISTRO/setup.bash
ros2 run demo_nodes_cpp talker
```

- In the second terminal, execute the *Fast-DDS* HelloWorld `subscriber`

```
cd ~/is-workspace
source install/setup.bash
./src/soss-dds/examples/common/HelloWorldExample/build/HelloWorldExample␣
↪subscriber
```

At this point, the two applications cannot communicate due to the incompatibility of their **topic** and **type** in their *DDS* configuration. This is where *eProsima Integration-Service* comes into play to make the communication possible.

- In the third terminal, go to the `is-workspace` folder where you have *eProsima Integration-Service* installed, and execute it using the `soss` command followed by the dds_ros2_string.yaml configuration file located in the `src/soss-dds/examples/udp/` folder:

```
cd ~/is-workspace
source /opt/ros/$ROS2_DISTRO/setup.bash
```

(continues on next page)

(continued from previous page)

```
source install/setup.bash
soss src/soss-dds/examples/udp/ros2_dds_string.yaml
```

Once the last command is executed, the two *DDS* applications will start communicating.

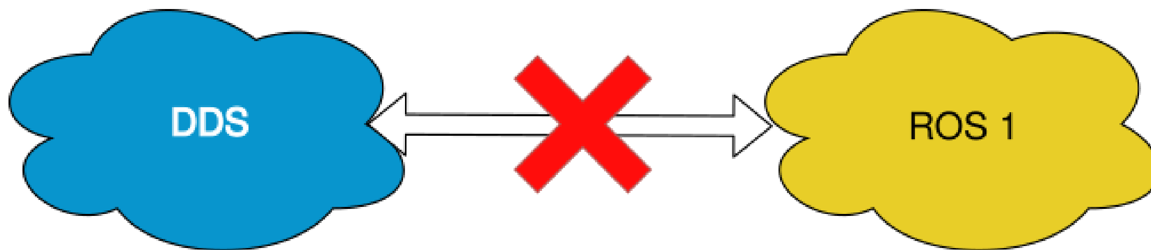
DDS publisher to ROS2 listener

To test the same communication the other way around, launch the *ROS2* `listener`, the HelloWorld `publisher` and execute *eProsima Integration-Service* using the `soss` command followed by the ros2_dds_string.yaml configuration file located in the `src/soss-dds/examples/udp/` folder:

```
cd ~/is-workspace
source /opt/ros/$ROS2_DISTRO/setup.bash
source install/setup.bash
soss src/soss-dds/examples/udp/dds_ros2_string.yaml
```

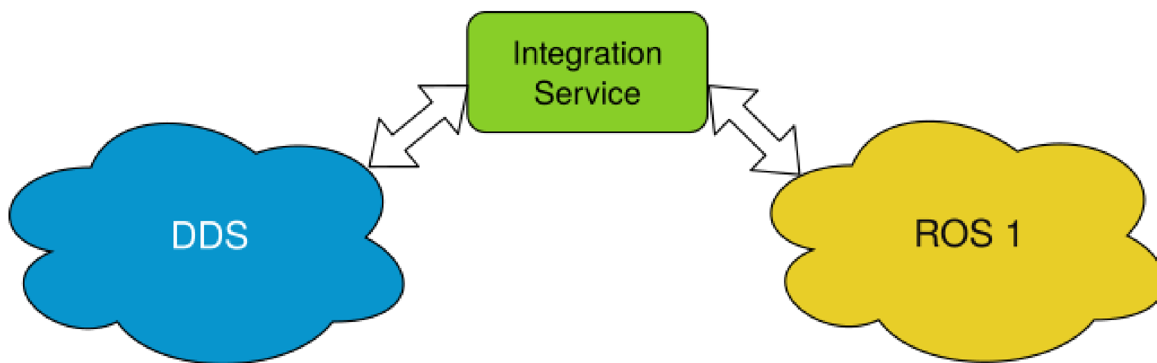
2.2.3 Add compatibility to an unsupported protocol

Another typical scenario encountered when communicating different systems is that they use different protocols, for example, *DDS* and *ROS1*.



In such a case, in the absence of the *eProsima Integration-Service* tool the user would need to create a custom *DDS* to *ROS1* bridge that won't be reusable to communicate either of the two with other protocols.

By using *eProsima Integration-Service* instead, this communication can be achieved with minimum user's effort. In this specific case, a *ROS1* **System-Handle** already exists, so the communication with *DDS* is essentially direct. However, the communication is straightforward enough even if a dedicated **System-Handle** doesn't exist yet, as the user can create his own **System-Handle**, thus becoming able to communicate with *DDS* and any other protocol already supported by *eProsima Integration-Service*. For more information regarding how to generate a **System-Handle** from scratch, please consult the System-Handle Creation section of the *SOSS* documentation.



In the example below we show how *eProsima Integration-Service* bridges a *DDS* application with a *ROS1* application, by communicating a HelloWorld *DDS* application with a *talker-listener* example.

Example: ROS1 communication

Requirements

To prepare the deployment and setup the environment, you need to have *eProsima Integration-Service* correctly installed in your system. To do so, please follow the steps delined in *Getting Started* and read carefully the *Important remarks* section.

To get this example working, the following requirements must be met:

- Having *ROS1* Melodic installed, with the *pub-echo* example working.
- Having the HelloWorldExample compiled. To do so, go to the `~/is-workspace/src/soss-dds/examples/common/HelloWorldExample` folder and type:

```
mkdir build
cd build
cmake ..
make
```

- Having the ros1_std_msgs compiled. To do so, go to the `~/is-workspace/src/soss-dds/examples/common/ros1_std_msgs` folder and type:

```
mkdir build
cd build
cmake ..
make
```

- Having the **SOSS-ROS1 System-Handle** installed, that you can download from the dedicated SOSS-ROS1 repository. Clone it into the workspace where you have *eProsima Integration-Service* installed:

```
cd ~/is-workspace
git clone ssh://git@github.com/eProsima/soss-ros1 src/soss-ros1 -b feature/xtypes-
↪support
```

  And then build the packages by running:

```
colcon build
```

ROS1 pub to DDS subscriber

Open three terminals:

- In the first terminal, launch the *ROS1* `pub` application with the commands:

```
source /opt/ros/melodic/setup.bash
rostopic pub -1 /chatter std_msgs/String -- "message"
```

  where "message" is some custom message chosen by the user.

- In the second terminal, execute the HelloWorld example as a `subscriber`:

```
cd ~/is-workspace
source install/setup.bash
./src/soss-dds/examples/common/HelloWorldExample/build/HelloWorldExample␣
↪subscriber
```

- In the third terminal, go to the is-workspace folder where you have *eProsima Integration-Service* and the **SOSS-ROS1 System-Handle** installed, and execute the former using the soss command followed by the ros1_dds.yaml configuration file located in the src/soss-dds/examples/ros1 folder:

```
cd ~/is-workspace
source /opt/ros/melodic/setup.bash
source install/setup.bash
soss src/soss-dds/examples/ros1/ros1_dds.yaml
```

Once *eProsima Integration-Service* is launched, you should see that the *ROS1* pub and the HelloWorld subscriber will start communicating.

DDS publisher to ROS1 echo

If you want to test it the other way around, launch the *ROS1* echo with the command:

```
source /opt/ros/melodic/setup.bash
rostopic echo /chatter
```

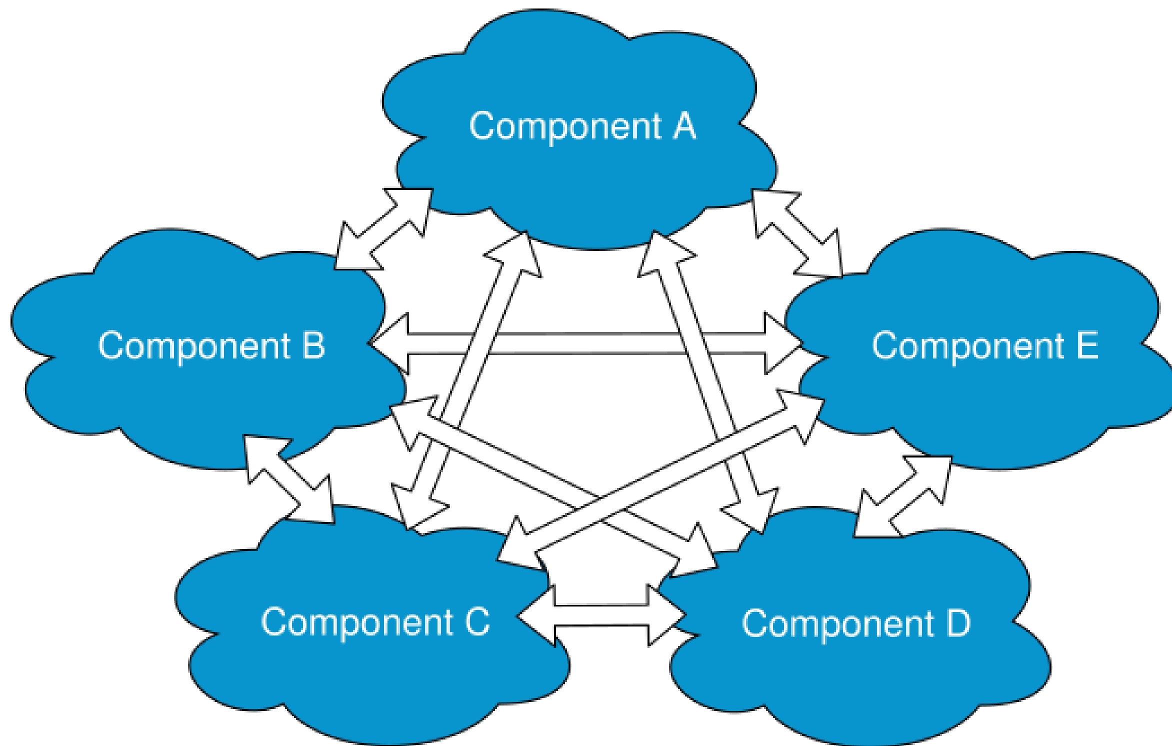
Then launch the HelloWorld as a publisher, and *eProsima Integration-Service* using the soss command followed by the dds_ros1.yaml configuration file located in the src/soss-dds/examples/ros1 folder instead:

```
cd ~/is-workspace
source /opt/ros/melodic/setup.bash
source install/setup.bash
soss src/soss-dds/examples/ros1/dds_ros1.yaml
```

2.2.4 Integrate a large system

Most systems evolve with time, undergoing the addition of new functionalities or new components. When these new components are based on software or hardware that don't use a protocol compatible with the rest of the system, an additional component must be created, usually known as *bridge*.

If the system contains several subsystems, and each component uses a different protocol, a *bridge* must be created for each existing components pair that need to be communicated, making the integration of the new component unhandy.



eProsima Integration-Service and its core ease this process. Specifically, *eProsima Integration-Service* allows to integrate any *DDS* system into an already existing system or viceversa, by providing an out-of-the-box bridge that straightforwardly puts into communication the *DDS* and the non-*DDS* protocols. Also, thanks, to its **System-Handle**-based structure, the core of *eProsima Integration-Service* allows to centralise all the possible bridges between the rest of subsystems.

Once all protocols are communicated with *eProsima Integration-Service*, the inter-components communication can be easily implemented by means of an individual YAML configuration file.

As explained in the *introductory section*, *eProsima Integration-Service* already provides the **System-Handle** for some common protocols.

This section is intended to illustrate an example of how *eProsima Integration-Service* integrates a *DDS* application into a complex system comprising *Orion Context-Broker* and *ROS1*.

Example: Integrate Orion Context-Broker and ROS1 into DDS

Requirements

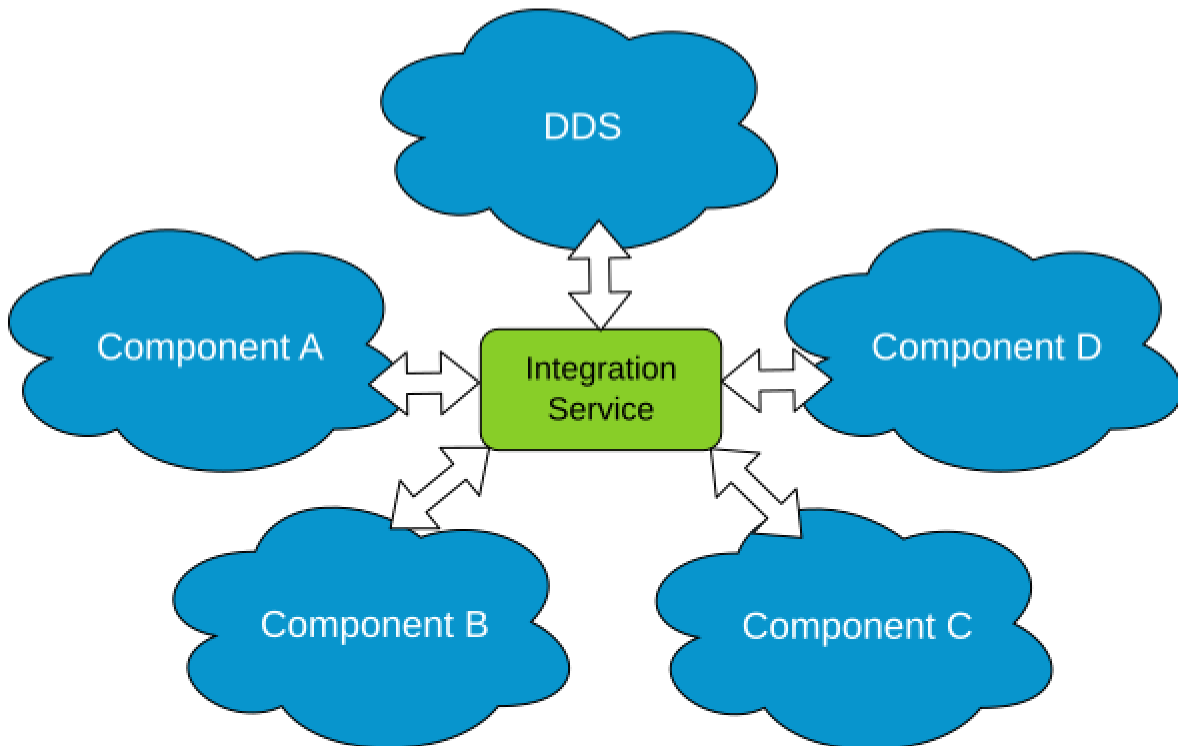
To prepare the deployment and setup the environment, you need to have *eProsima Integration-Service* correctly installed in your system. To do so, please follow the steps delined in *Getting Started* and read carefully the *Important remarks* section.

To get this example working, the following requirements must be met:

- Having *ROS1* Melodic installed, with the *pub-echo* example working.
- Having the HelloWorldExample compiled. To do so, go to the `~/is-workspace/src/soss-dds/examples/common/HelloWorldExample` folder and type:

```
mkdir build
cd build
```

(continues on next page)



(continued from previous page)

```
cmake ..
make
```

- Having the ros1_std_msgs compiled. To do so, go to the `~/is-workspace/src/soss-dds/examples/common/ros1_std_msgs` folder and type:

```
mkdir build
cd build
cmake ..
make
```

- Having the **SOSS-ROS1 System-Handle** installed, that you can download from the dedicated SOSS-ROS1 repository. Clone it into the workspace where you have *eProsima Integration-Service* installed:

```
cd ~/is-workspace
git clone ssh://git@github.com/eProsima/soss-ros1 src/soss-ros1 -b feature/xtypes-
↪support
```

  And then build the packages by running:

```
colcon build
```

- An accessible Orion Context Broker service.
- Asio and cURLpp installed.
- Having the **SOSS-FIWARE System-Handle** installed, that you can download from the dedicated SOSS-FIWARE repository. Clone it into the workspace where you have *eProsima Integration-Service* installed:

```
cd ~/is-workspace
git clone ssh://git@github.com/eProsima/SOSS-FIWARE src/soss-fiware -b feature/
↪xtypes-support
```

And then build the packages by running:

```
colcon build
```

Before proceeding, note that the dds_fiware_ros1.yaml and the fiware_dds_ros1.yaml configuration files located in the `~/is-workspace/src/soss-dds/examples/fiware` folder must be edited to match the IP address and port used by the *contextBroker* configuration in the testing environment.

DDS publisher to Context Broker and ROS1 echo

To publish from the *DDS* world to a system composed by a *contextBroker* service and a *ROS1* `echo` application, follow the steps detailed below.

Open four terminals (replace `<url>` with the location of the *contextBroker*, following the format `<ip>:<port>`):

- In the first terminal, execute the Helloworld `publisher`:

```
cd ~/is-workspace
source install/setup.bash
./src/soss-dds/examples/common/HelloWorldExample/build/HelloWorldExample publisher
```

- In the second terminal, create/check the value of the `data-binary` field in the *contextBroker*:

  1. When testing for the first time, the Helloworld entity must be created in the *contextBroker*:

```
curl --include \
    --request POST \
    --header "Content-Type: application/json" \
    --data-binary "{ \"type\": \"HelloWorld\", \"id\": \"HelloWorldTopic\", \
↪"data\": { \"value\": \"\" } }" \
    '<url>/v2/entities'
```

  2. Check if the value of the attribute already exists by typing:

```
curl "http://<url>/v2/entities/HelloWorldTopic"
```

  3. Update the value:

```
curl --include \
    --request PUT \
    --header "Content-Type: application/json" \
    --data-binary "{ \"value\": \"<Updated message>\" }" \
    '<url>/v2/entities/HelloWorldTopic/attrs/data'
```

- In the third terminal, execute the *ROS1* `echo` application with the command:

```
source /opt/ros/melodic/setup.bash
rostopic echo /chatter
```

- In the fourth terminal, go to the `is-workspace` folder where you have *eProsima Integration-Service* and the **SOSS-ROS1 System-Handle** and **SOSS-FIWARE System-Handle** installed, and execute the former using the `soss` command followed by the `dds_fiware_ros1.yaml` YAML example file previously edited:

```
cd ~/is-workspace
source /opt/ros/melodic/setup.bash
source install/setup.bash
soss src/soss-dds/examples/fiware/dds_fiware_ros1.yaml
```

- Check again the value of the `data` in the *contextBroker*:

```
curl "http://<url>/v2/entities/HelloWorldTopic"
```

Now, the value must contain information (normally, `HelloWorld`).

The *ROS1* `echo` will show the updated values as well.

Context Broker to DDS subscriber and ROS1 echo

If you want to test the communication the other way around, launch Helloworld as a `subscriber` and force an update in the *contextBroker* `data` as detailed above (step 3 of the second terminal commands) while *eProsima Integration-Service* is executing with the `fiware_dds_ros1.yaml` YAML example file previously edited. Keep the *ROS1* application in `listener` mode in order to avoid having two publishers at the same time. Indeed, while possible, having *ROS1* publishing may hinder probing the example behavior correctly.

2.2.5 WAN communication

One of the most critical and powerful use-cases is that of two systems located in different geographical regions which need to communicate through the Internet, using a *WAN* connection.

Using a pair of *eProsima Integration-Service* instances, or **System-Handles**, one for each system, this scenario can be addressed with a **secure TCP tunnel** thanks to the **SSL TCP** capabilities of *Fast-RTPS*.

In this case, we can see *eProsima Integration-Service* as a gateway to translate each system to *DDS* over *SSL-TCP*. A proper configuration of the destination router and firewalls will allow the communication.

The example below illustrates how to configure *eProsima Integration-Service* to achieve WAN communication.

Example: WAN TCP tunneling

Requirements

To prepare the deployment and setup the environment, you need to have *eProsima Integration-Service* correctly installed in your system. To do so, please follow the steps delined in *Getting Started* and read carefully the *Important remarks* section.

To test this example properly, you need two separated subnets that are not connected but both with internet access, or a testing environment simulating this scenario (for example, two routers, with one of them acting as ISP of the second).

Note that route tables and NAT must be configured so as to ensure proper port redirection before starting the test.

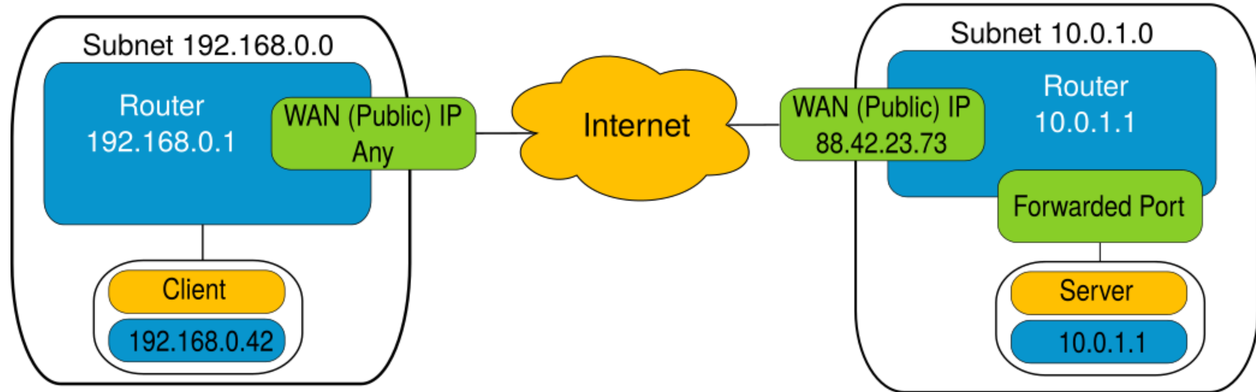


Fig. 1: The IP addresses shown only serve the purpose of illustrating the example, but the important information is the **real** public IP of the *server* machine. Also, its router must enable NAT to forward the listening port to the *server*.

Also, to get this example working, the following requirements must be met in both machines:

- Having *ROS2* (Crystal or superior) installed, with the *talker-listener* example working.
- Having the ros2_std_msgs compiled. To do so, go to the `~/is-workspace/src/soss-dds/examples/common/ros2_std_msgs` folder and type:

```
mkdir build
cd build
cmake ..
make
```

- Having the **SOSS-ROS2 System-Handle** installed. Unless configured otherwise, this package is built automatically when *eProsima Integration-Service* is installed.

Once the environment is prepared and tested (for example, using a port-scanner), modify the file `wan_config.xml` to match the IP address and port of with the WAN IP address and forwarded port of your environment.

This test will launch a *ROS2* `talker` in the *server* machine, and a *ROS2* `listener` in the *client* machine. An *eProsima Integration-Service* instance will communicate each application with the one in the other machine using the WAN-TCP communication capabilities of *Fast-DDS*.

Executing the WAN communication

Open 2 terminals in each machine:

On the *server* side:

- In the first terminal, launch the *ROS2* `talker` example:

```
source /opt/ros/$ROS2_DISTRO/setup.bash
ros2 run demo_nodes_cpp talker
```

- In the second terminal, go to the `is-workspace` folder where you have *eProsima Integration-Service* installed and execute it using the `soss` command followed by the server YAML configuration file located in the `src/soss-dds/examples/wan` folder:

```
cd ~/is-workspace
source /opt/ros/$ROS2_DISTRO/setup.bash
source install/setup.bash
soss src/soss-dds/examples/wan/wan_server_talker.yaml
```

On the *client* side:

- In the first terminal, launch the *ROS2* `listener` example:

```
source /opt/ros/$ROS2_DISTRO/setup.bash
ros2 run demo_nodes_cpp listener
```

- In the second terminal, go to the `is-workspace` folder where you have *eProsima Integration-Service* installed and sourced, and execute it using the `soss` command followed by the client YAML configuration file located in the `src/soss-dds/examples/wan` folder:

```
cd ~/is-workspace
source /opt/ros/$ROS2_DISTRO/setup.bash
source install/setup.bash
soss src/soss-dds/examples/wan/wan_client_listener.yaml
```

Once the two *eProsima Integration-Service* instances match, the *talker-listener* example will start to communicate. If the test doesn't work, review carefully your NAT configuration.