SOSS Documentation

Release 3.0.0

eProsima

Jul 10, 2020

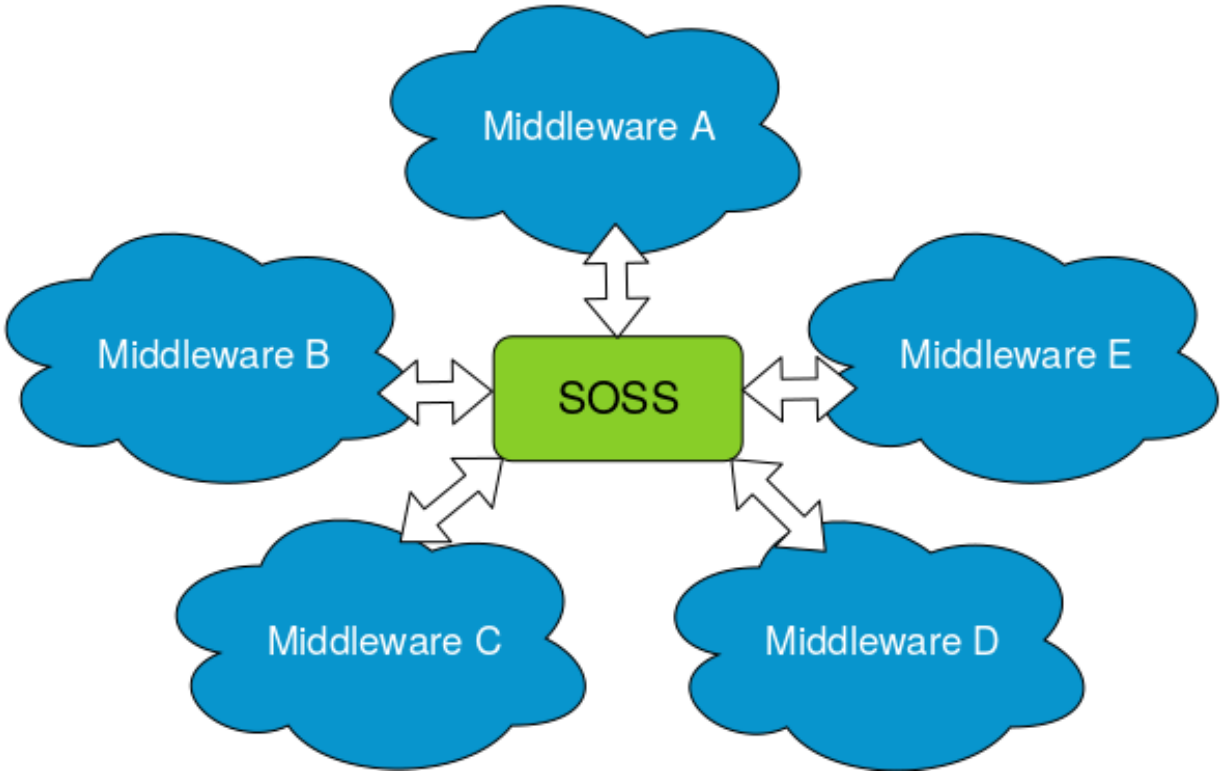
INSTALLATION MANUAL

SOSS is a *System-Of-Systems Synthesizer* that allows communication among an arbitrary number of protocols that speak different languages.

If one has a number of complex systems and wills to combine them to create a larger, even more complex system, *SOSS* can act as an intermediate message-passing tool that, by speaking a common language, centralizes and mediates the integration.

The communication between the different protocols is made possible by system-specific plugins, or **System-Handles**. These provide the necessary conversion between the target protocols and the specific language spoken by *SOSS*. Once a system is communicated with *SOSS*, it enters the *SOSS* world and can straightforwardly reach out to any other system that already exists in this world.

SOSS is configured by means of a YAML text file, through which the user can provide a mapping between the topics and services on the middlewares of the systems involved.



CHAPTER

ONE

MAIN FEATURES

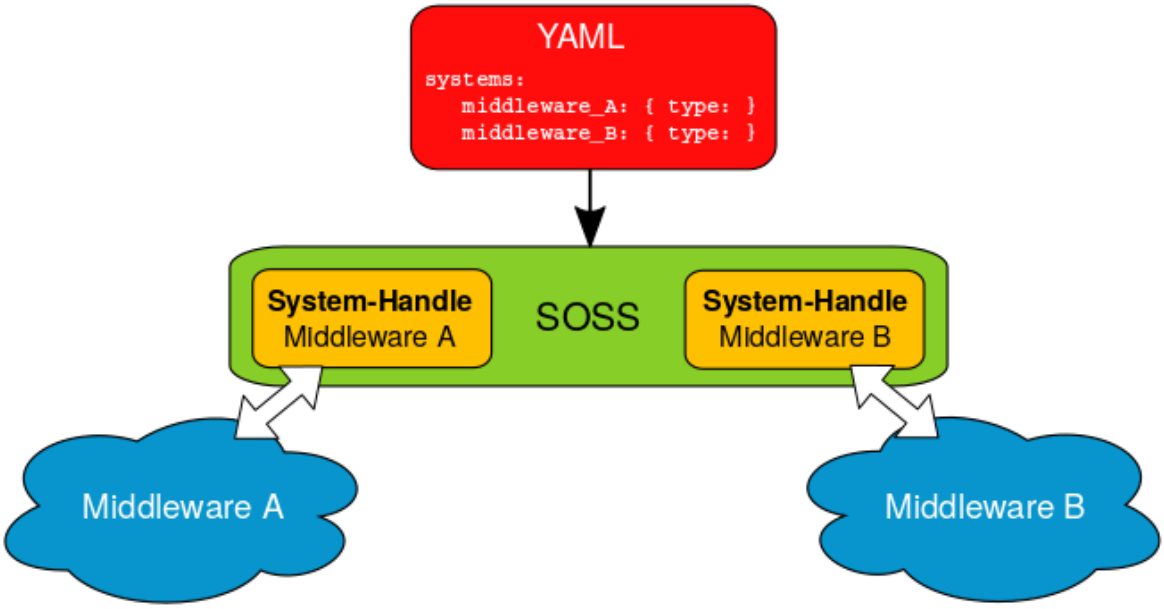
SOSS provides a plugin-based platform that is easily and intuitively configurable. This section explains these key features.

1.1 System Handles

A *SOSS* 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 *SOSS* at runtime after they have been installed.

Built-in **System-Handles** are provided for the following systems: *DDS*, *Orion ContextBroker*, *ROS*, *ROS2*, and *WebSocket*. New **System-Handles** for additional protocols can be easily created, automatically allowing communication of the new protocol with the middlewares that are already supported. Detailed information on how to create a **System-Handle** can be found in the *System-Handle Creation* section of this documentation.

The plugin-based framework is especially advantageous when it comes to integrating a new component into a complex system where the rest of sub-systems use incompatible protocols. Indeed, once all protocols of interest are communicated with *SOSS*, 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

SOSS is configured by means of a YAML file. Detailed information on how to configure a *SOSS*-mediated communication by means of one such file can be found in the *YAML Configuration* section of this documentation. Notice that 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 this configuration text file should contain. In this example, a single topic is translated from *ROS1* to *ROS2*:

```
systems:
    ros1: { type: ros1 }
    ros2: { type: ros2 }
topics:
    chatter: { type: std_msgs/String, route: {from: ros1, to: ros2} }
```

The strength of this approach is that different translations are possible by only changing the configuration file. For example, by changing the middlewares involved, we can obtain an instance which translates between *WebSocket+JSON* (as produced and consumed by a standard Web browser) and *ROS2*:

```
systems:
    web: { type: websocket_client, types-from: robot, host: localhost, port: 12345 }
    robot: { type: ros2 }
routes:
    web2robot: {from: web, to: robot}
topics:
    chatter: { type: "std_msgs/String", route: web2robot }
```

1.3 Additional features

Free and Open Source.

SOSS and all **System-Handles** available to date are free and open source.

Easily configurable.

As detailed above, a *SOSS* instance is easily configurable by means of a YAML file.

Easy to extend to new platforms.

New platforms can easily enter the *SOSS* world by generating the plugin, or **System-Handle** needed by the core to integrate them.

Easy to use.

Installing and running *SOSS* 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

Each **System-Handle** may have specific dependencies.

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

- SOSS repository
- 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

In this section, we sketch the steps necessary for installing *SOSS* and running a `soss` instance. Note that the workflow is dependent on the specific systems that are being communicated, given that each is brought into the *SOSS* world via a dedicated **System-Handle**. An example of how these generic steps are implemented in a concrete use-case can be found in the final section.

Installation

As a first step, you will need to create a colcon workspace to clone the *SOSS* repository. This repository contains the *SOSS* core library and the **System-Handles** for some of the protocols that are integrated into the *SOSS* world. It consists of many cmake packages which can be configured and built manually, but we recommend to use a colcon workspace, which makes the job much smoother. To do so, create a `soss-workspace` folder and clone *SOSS* into it:

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

Note: the `--recursive` flag is mandatory to download some required third-parties.

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

```
colcon build
```

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 the 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** 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, you can source the new colcon overlay:

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

Deployment

Now you can run a `soss` instance to put two or more middlewares into communication. Notice that the *SOSS* repository does not contain all the **System-Handles** of the protocols that are to date integrated into the *SOSS* world. For those **System-Handles** that are not built-in, you need to clone their specific repositories into the `soss-workspace` folder as well.

In the *Related Links* section you can find a table of the repositories of all the *SOSS*-supported **System-Handles**.

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 a `soss` 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 the middlewares involved:

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

Once *SOSS* is initiated, the user will be able to communicate the desired protocols.

For more information on how to configure *SOSS* via a YAML file, please refer to *YAML Configuration*. For information on how to create your own custom **System-Handle**, see *System-Handle Creation* instead.

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 ~/soss-workspace/install/setup.bash
```

The same applies for the **System-Handle** repositories.

Example: ROS1-ROS2 communication

As a demonstration of *SOSS*' capabilities and usage, we will walk you through how to set up a communication between *ROS1* and *ROS2*.

Setting up SOSS

We will assume that you have installed ROS1 Melodic and ROS2 Crystal using the ROS PPAs. To run the `soss-ros2-test` integration test, you will also need

```
sudo apt install ros-crystal-test-msgs
```

Note that the same steps are applicable to *Dashing*.

Create a colcon workspace as explained above

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

and source the *ROS2 Crystal* overlay:

```
source /opt/ros/crystal/setup.bash
```

Now, you can run:

```
colcon build
```

Note: If any packages are missing dependencies **causing the compilation to fail**, you can add the flag `--packages-up-to soss-ros2-test` to make sure that you at least build `soss-ros2-test`:

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

Once that's finished building, you can source the new colcon overlay:

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

Notice, with reference to the table shown in *Related Links*, that you now have both *SOSS* and the **SOSS-ROS2 System-Handle** installed. To get the **SOSS-ROS1 System-Handle**, you can create a new workspace, and then clone the dedicated repository into it:

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

Now source the *ROS Melodic* distribution:

```
source /opt/ros/melodic/setup.bash
```

You will likely see this message:

```
ROS_DISTRO was set to 'crystal' before. Please make sure that the environment does␣
↪not mix paths from different
distributions.
```

That's okay. The reason is that we have made a previous sourcing of *ROS2* in the same shell, but you will be able to build `soss-ros1` as long as a *ROS1* distribution was sourced more recently than a *ROS2* distribution.

Now you can use `colcon build` to build `soss-ros1`:

```
colcon build
```

And finally, you can source the new colcon overlay:

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

You may see another warning about `ROS_DISTRO`. That's okay.

Running SOSS

After following the above build instructions, **open a new shell** environment and run:

```
source /opt/ros/melodic/setup.bash
roscore
```

Then you can return to the shell environment that you were using to build. **If that shell has already been closed**, then open a new one, return to your `soss-workspace` workspace and source the overlays:

```
cd ~/soss-workspace
source /opt/ros/melodic/setup.bash
source /opt/ros/crystal/setup.bash
source install/setup.bash
```

Now from the fully-overlaid shell, you can run the `soss` instance:

```
soss src/soss-ros1/examples/hello_ros.yaml
```

In this command, the executable `soss` is given a YAML configuration file to describe how messages should be passed among whichever middlewares (in this case, *ROS1* and *ROS2*).

In another **new shell environment**, run:

```
source /opt/ros/melodic/setup.bash
rostopic echo /hello_ros1
```

In yet another **new shell environment**, run:

```
source /opt/ros/crystal/setup.bash
ros2 topic echo /hello_ros2 std_msgs/String
```

Now when you send messages to the topic `/hello_ros1` from *ROS2*, they will appear in the *ROS1* `rostopic echo` terminal. For example, open a **new shell environment** and run:

```
source /opt/ros/crystal/setup.bash
ros2 topic pub -r 1 /hello_ros1 std_msgs/String "{data: \"Hello, ros1\"}"
```

Or you can send messages from *ROS1* to *ROS2*. For example, open a **new shell environment** and run:

```
source /opt/ros/melodic/setup.bash
rostopic pub -r 1 /hello_ros2 std_msgs/String "Hello, ros2"
```

Notice that even if this demo requires 6 shell environments to run, *SOSS* itself only occupies one shell.

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

This section provides the user with an in-depth knowledge of *SOSS*' main aspects. First of all, we detail the internal structure of a **System-Handle** and guide the user through the creation a brand-new **System-Handle**. Secondly, we explain how to configure *SOSS* by means of YAML files, explaining how to fill the required fields, depending on the needs of the specific use-case.

2.2.1 System-Handle Creation

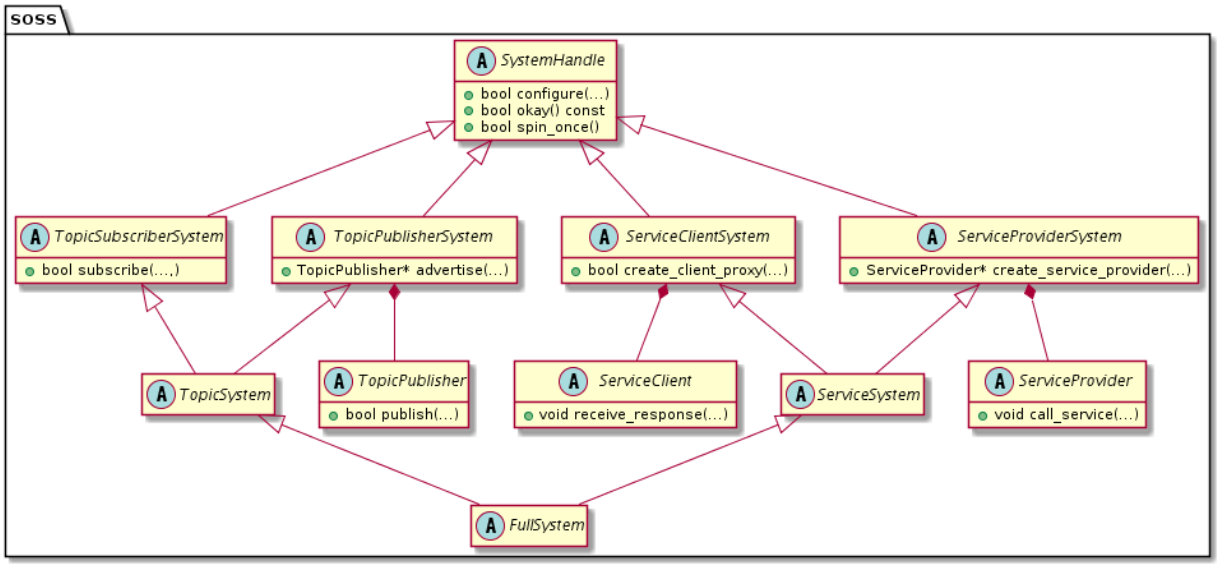
The `soss-core` library defines a set of abstract interfaces and provides some utility classes to form a plugin-based framework. A single `soss` executable instance can connect *N* middlewares where each middleware has a *SOSS*-plugin associated with it. The *SOSS*-plugin, or **System-Handle**, for a middleware is a lightweight wrapper around that middleware (e.g. a *ROS* node or a *websocket* server/client). The `soss-core` library provides cmake functions that allow these middleware **System-Handles** to be discovered by the `soss` executable at runtime after the **System-Handle** has been installed. Because of this, downstream users can extend *SOSS* to communicate with any middleware.

A single `soss` executable can route any number of topics or services to/from any number of middlewares.

SOSS provides built-in **System-Handles** for connecting to **DDS**, **Orion ContextBroker**, **ROS**, **ROS2**, and **WebSocket**. Adding a new **System-Handle** automatically allows communication with the rest of these protocols.

System-Handle hierarchy

Here you can find a diagram of a **System-Handle** class inheritance structure.

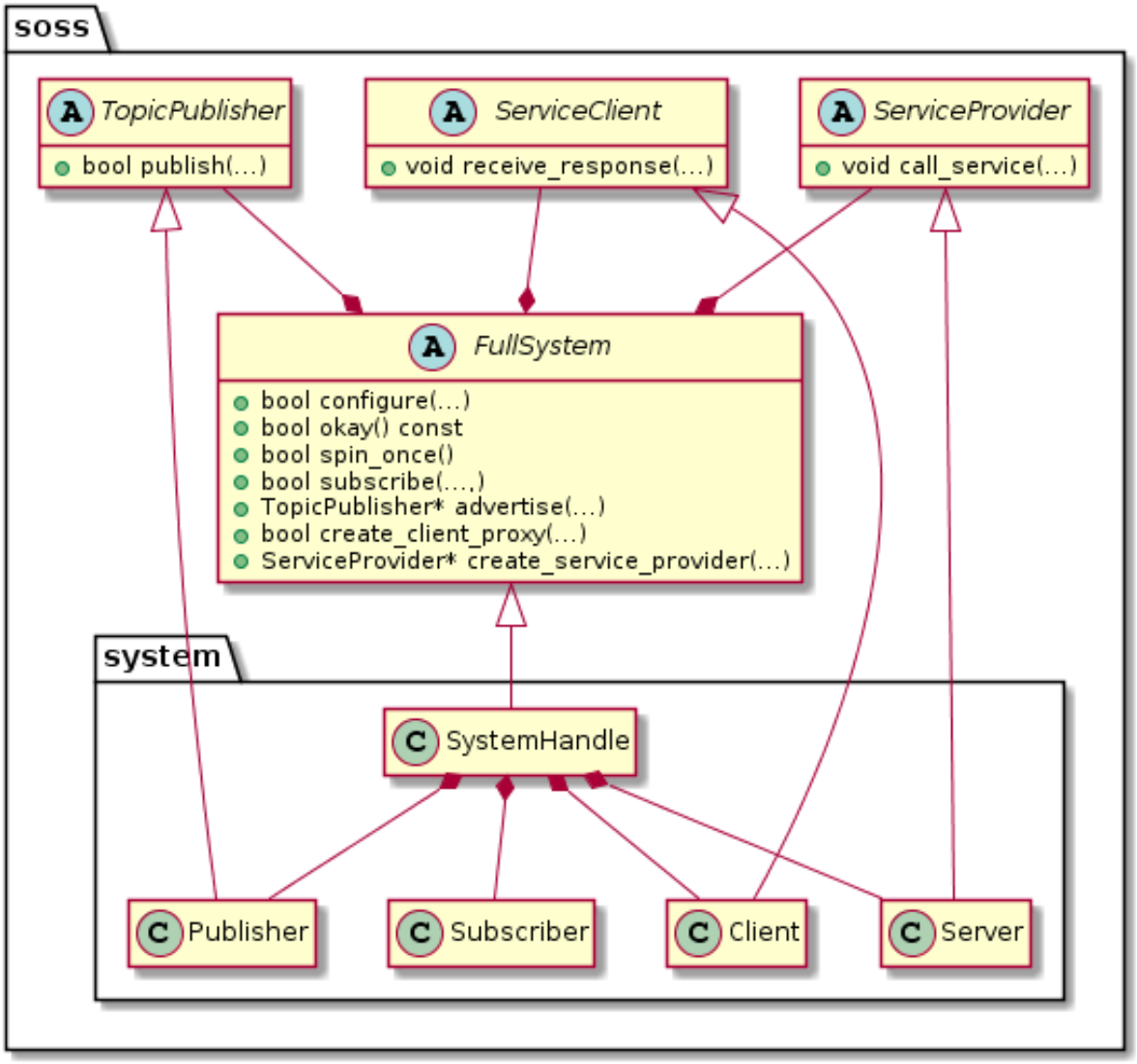


Each **System-Handle** must inherit, directly or indirectly, from the `SystemHandle` superclass. Depending on the nature of each protocol, it should implement the derived classes using multiple inheritance from `TopicSubscriberSystem`, `TopicPublisherSystem`, `ServiceClientSystem`, and/or

`ServiceProviderSystem`. To simplify this inheritance, classes `TopicSystem`, `ServiceSystem`, and `FullSystem` are available to inherit from.

System-Handle implementation

In the diagram below, the architecture of a generic "Full" **System-Handle** and its integration into `soss` is shown.



To ease the implementation, the new `system::SystemHandle` will inherit from `FullSystem`. The following sections will explain the methods to be implemented.

To implement the `TopicPublisher`, `ServiceClient`, and `ServiceProvider` interfaces, the most direct way is to create child classes, respectively `system::Publisher`, `system::Client`, and `system::Server`. An additional class `system::Subscriber` may be useful to manage the subscribers created. In the example shown in the diagram above, the `system::SystemHandle` will contain the needed instances of these classes, but any approach may be valid if the interfaces are met.

SystemHandle

All **System-Handles** must implement the `configure`, `okay`, and `spin_once` methods that belong to the superclass:

```
bool configure(
    const RequiredTypes& types,
    const YAML::Node& configuration,
    TypeRegistry& type_registry);

bool okay() const = 0;

bool spin_once();
```

The `configure` method is called to setup the **System-Handle** with the associated `configuration`, defined in the YAML file that is passed to it. The types that the SH needs to manage to implement the communication are passed to this method via the `types` argument, whereas the new types created by the **System-Handle** are expected to be filled in the `type_registry`.

The `okay` method is called by *SOSS* to check if the **System-Handle** is working. This method will verify internally if the middleware has any problem.

The `spin_once` method is called by *SOSS* to allow spinning to those middlewares that need it.

TopicSubscriberSystem

This kind of system must implement the `subscribe` method:

```
using SubscriptionCallback = std::function<void(const xtypes::DynamicData& message)>;

bool subscribe(
    const std::string& topic_name,
    const xtypes::DynamicType& message_type,
    SubscriptionCallback callback,
    const YAML::Node& configuration);
```

SOSS will call this method in order to create a new subscriber to the topic `topic_name` using `message_type` type, plus an optional `configuration`. Once the middleware system receives a message from the subscription, the message must be translated into the `message_type` and the **System-Handle** must invoke the `callback` with the translated message.

TopicPublisherSystem

This kind of system must implement the `advertise` method:

```
std::shared_ptr<TopicPublisher> advertise(
    const std::string& topic_name,
    const xtypes::DynamicType& message_type,
    const YAML::Node& configuration);
```

SOSS will call this method in order to create a new `TopicPublisher` to the topic `topic_name` using `message_type` type, and optional `configuration`.

The `TopicPublisher` is an interface that must be implemented by a `Publisher` in order to allow *SOSS* to publish messages to the target middleware. This interface defines a single method `publish`:

```
bool publish(const xtypes::DynamicData& message);
```

When *SOSS* needs to publish to the middleware system it will call the `TopicPublisher::publish` method, with a message that must be translated from the `message_type` parameter by the `advertise` method above.

ServiceClientSystem

This kind of system must implement the `create_client_proxy` method:

```
using RequestCallback =
    std::function<void(
        const xtypes::DynamicData& request,
        ServiceClient& client,
        std::shared_ptr<void> call_handle)>;

bool create_client_proxy(
    const std::string& service_name,
    const xtypes::DynamicType& service_type,
    RequestCallback callback,
    const YAML::Node& configuration);
```

SOSS will call this method in order to create a new `ServiceClient` to the service `service_name` using the `service_type` type, plus an optional `configuration`. This `ServiceClient` will be provided as an argument in the `callback` invocation when a response is received.

The `ServiceClient` is an interface that must be implemented by a `Client` in order to allow *SOSS* to relate a *request* with its *reply*. This is done by providing a `call_handle` both in the `call_service` method from `ServiceProvider` and in the `callback` from `create_client_proxy` method. When the *reply* is received by another **System-Handle**, its `ServiceProvider` will call the `receive_response` method from the `Client`:

```
void receive_response(
    std::shared_ptr<void> call_handle,
    const xtypes::DynamicData& response);
```

The `receive_response`:

- Translates the `response` from `service_type` and relate the `call_handle`, if needed, to its middleware's request;
- Replies to its middleware.

ServiceProviderSystem

This kind of system must implement the `create_service_proxy` method:

```
std::shared_ptr<ServiceProvider> create_service_proxy(
    const std::string& service_name,
    const xtypes::DynamicType& service_type,
    const YAML::Node& configuration);
```

SOSS will call this method in order to create a new `ServiceProvider` to the service `service_name` using the `service_type` type, plus an optional `configuration`.

The `ServiceProvider` is and interface that must be implemented by a `Server` in order to allow *SOSS* to *request* (or call) a service from the target middleware.

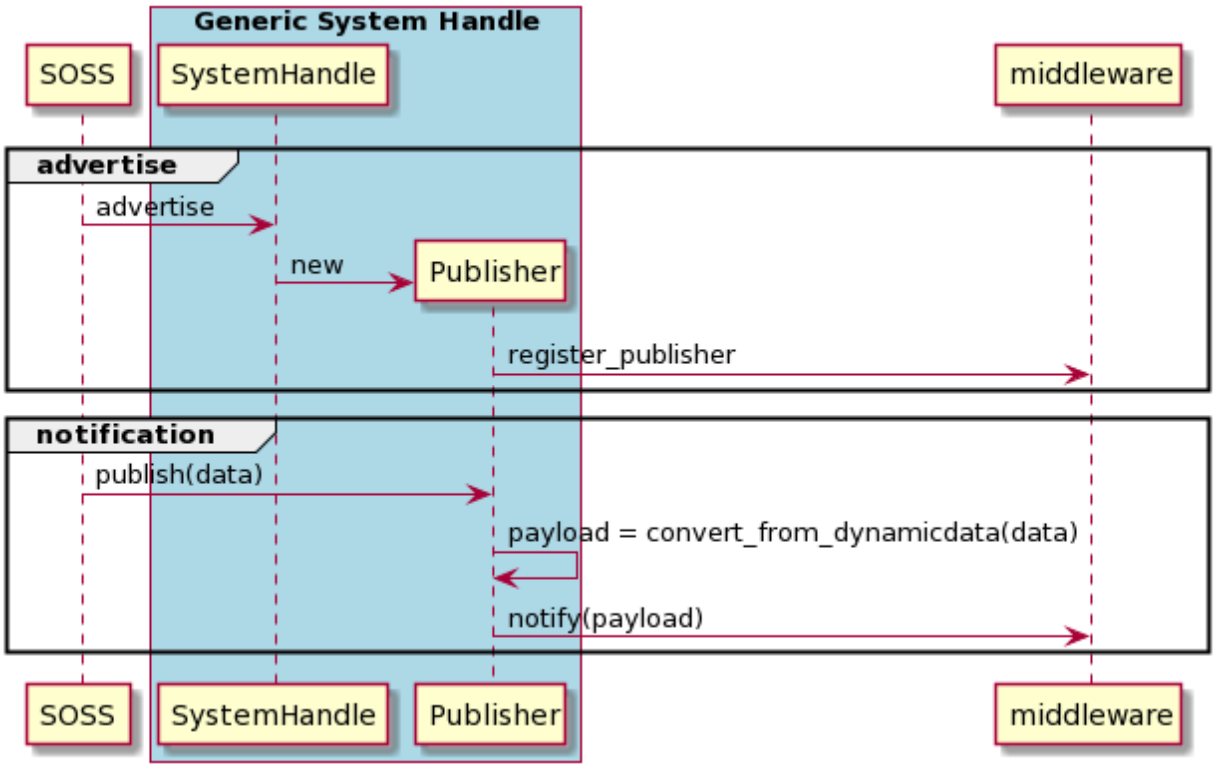
```
void call_service(
    const xtypes::DynamicData& request,
    ServiceClient& client,
    std::shared_ptr<void> call_handle);
```

This `call_service` method will translate the `request` from `service_type` and will call its middleware service, which stores the related `call_handle` and `client`. Once it receives the response from its middleware, it must translate back the response and retrieve the `call_handle` and `client` related. Then, it will invoke the `receive_response` method from the client using the `call_handle` as argument.

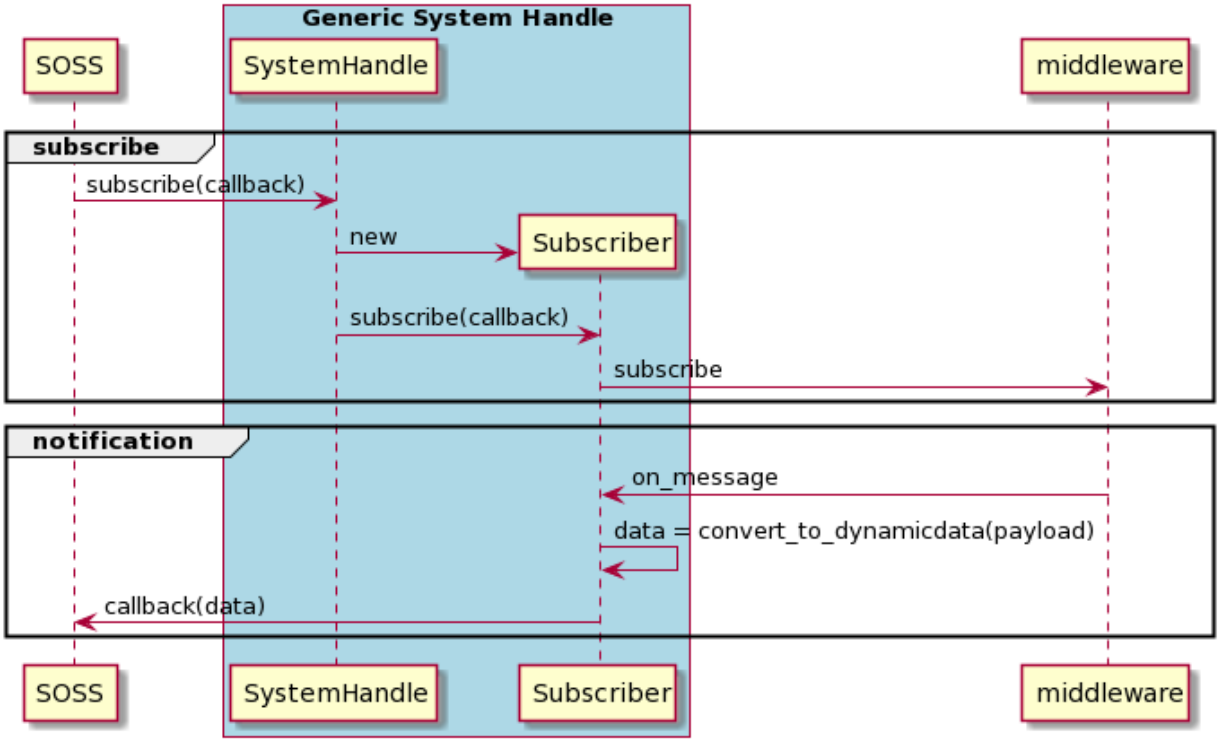
Sequence diagrams

The following diagrams illustrate the previous sections using a *generic* **System-Handle**.

TopicPublisher flow

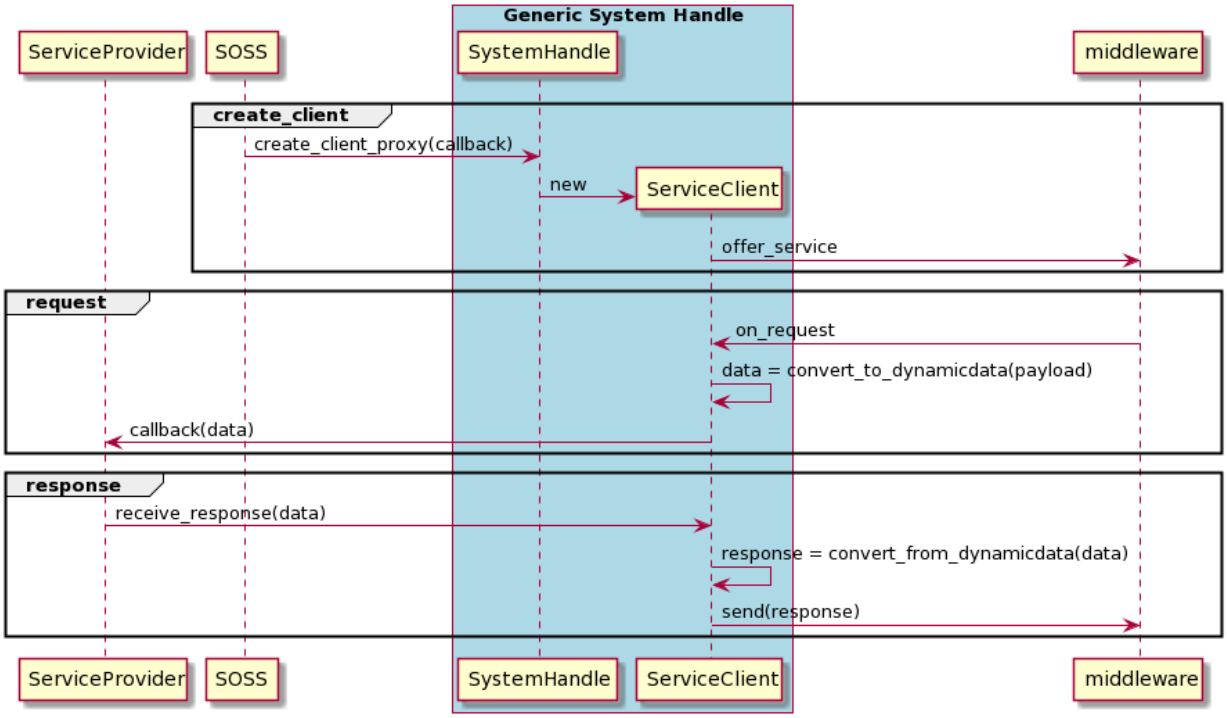


TopicSubscriber flow



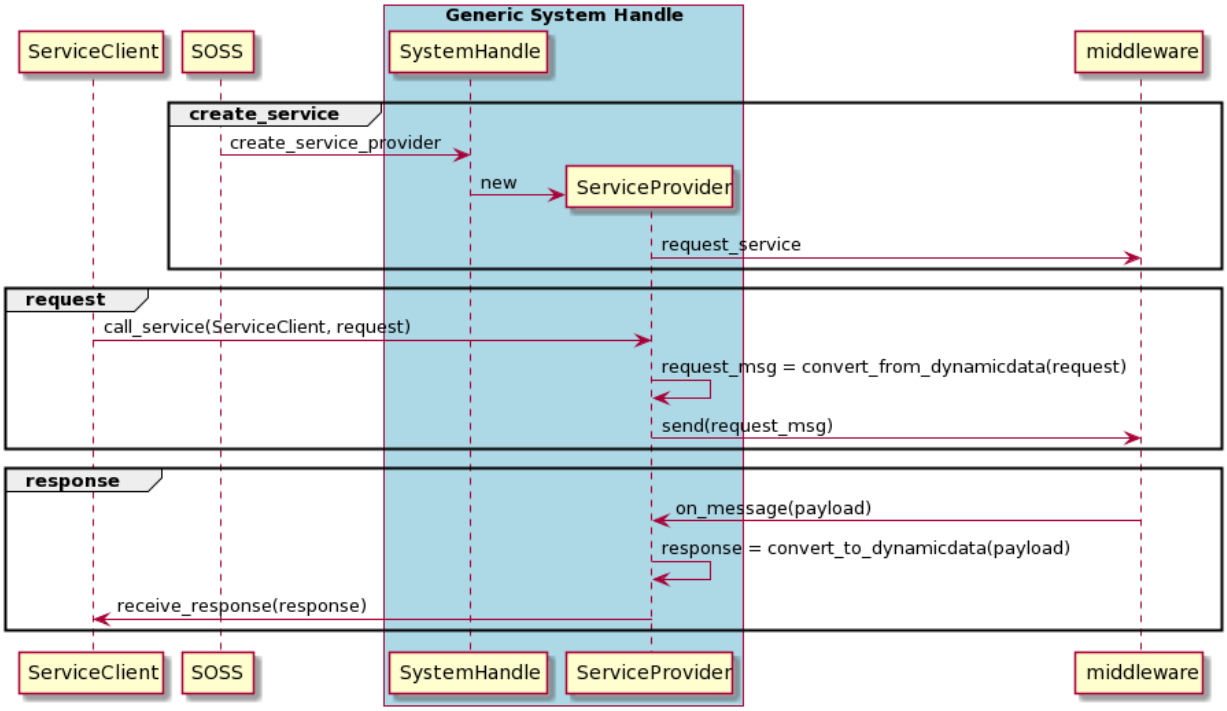
ServiceClient flow

Note that a `ServiceClient` acts as a client for *SOSS* and as a server for the **middleware**.



ServiceProvider flow

Note that a `ServiceProvider` acts as a server for *SOSS* and as a client for the **middleware**.



2.2.2 YAML Configuration

SOSS 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 *SOSS* to transmit messages.
  - `idl`: IDL content.
- `systems`: specifies the middlewares involved in the communication, allowing to configure them.
- `routes`: specifies which bridges SOSS needs to create.
  - `from-to`: publisher/subscriber communication.
  - `server-clients`: server/client communication.
- `topics/services`: specify the topics exchanged over the `routes` above in either publisher/subscriber or client/server type communications, allowing to configure them.
  - `type`: type involved in the communication.
  - `route`: communication bridge to apply.
  - `remap`: allows to establish equivalences between `topic` names, `types`, and custom configurations.

A generic YAML communicating two systems has the following structure:

```
types:
    idl: <idl_content>
    paths: [idl_include_path_1, id_include_path_2 ]
systems:
    <system_1_name>: {
        type: <system_1_type>,
        types-from: <other_system_name>,
        <system_1_config>
    }
    <system_2_name>: {
        type: <system_2_type>,
        types-from: <other_system_name>,
        <system_2_config>
    }
routes:
    <route_name>: { from: <system_1_name>, to: <system_2_name> }
    <service_route_name>: {
        server: <system_1_name>,
        client: [<system_2_name>, <other_system_name>]
    }
topics:
    <topic_name>:
        type: <type_name>
        route: <route_name>
        remap:
            <system_1_name>: {
                type: <type_remap_name>,
                topic: <topic_remap_name>
            }
        <custom_topic_key>: <custom_topic_config>
services:
    <service_name>:
        type: <type_service_name>
        route: <service_route_name>
```

(continues on next page)

(continued from previous page)

```
        remap:
            system_2_name>: {
                type: <type_remap_name>,
                topic: <topic_name>
            }
        <custom_service_key>: <custom_service_config>
```

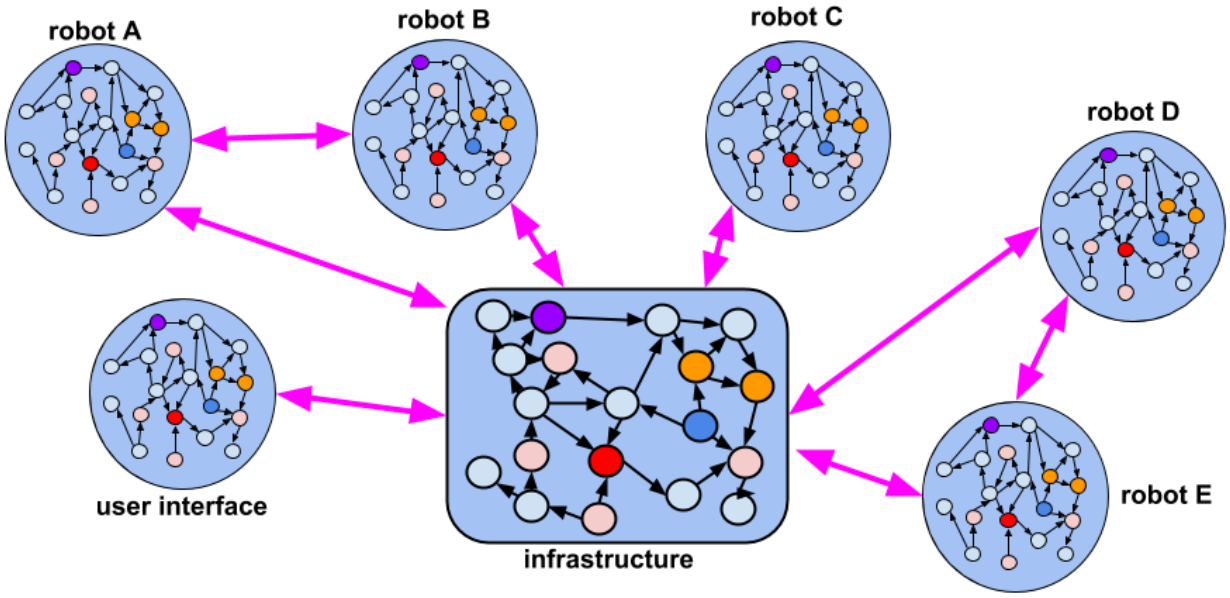
Here is a nontrivial example, which translates a number of topics and some service clients between *WebSocket+Rosbridge_v2*, *ROS2*, and a (fictitious) automated door-opening firmware:

```
systems:
    web: { type: websocket_server_json, types-from: ros2, port: 12345 }
    robot: { type: ros2 }
    door:
        type: veridian_dynamics_proprietary_door_firmware
        types-from: ros2
        serial: 1765TED

routes:
    web2robot: {from: web, to: robot}
    robot2web: {from: web, to: robot}
    door_broadcast: {from: door, to: [web, robot]}
    web_service: {server: web, clients: robot}
    door_service: {server: door, clients: [web, robot]}

topics:
    videocall_signalling_tx:
        type: "rmf_msgs/SignallingMessage"
        route: web2robot
    videocall_presence: { type: "std_msgs/String", route: web2robot }
    call_button_state_array:
        type: "rmf_msgs/CallButtonStateArray"
        route: robot2web
    videocall_signalling_rx:
        type: "rmf_msgs/SignallingMessage"
        remap: {
            robot:
                type: {"videocall_signalling_rx/{message.message_to}" }
        }
        route: robot2web
    door_status:
        type: "rmf_msgs/DoorStatus"
        route: door_broadcast

services:
    get_video_callers:
        type: "rmf_msgs/GetVideoCallers"
        route: web_service
    reserve_robot: { type: "rmf_msgs/ReserveRobot", route: web_service }
    release_robot: { type: "rmf_msgs/ReleaseRobot", route: web_service }
    open_door: { type: "rmf_msgs/OpenDoor", route: door_service }
    close_door: { type: "rmf_msgs/CloseDoor", route: door_service }
```

The idea is that each system plays some role in the overall system of systems, and the user needs to specify the channels that these systems are expected to communicate over, as well as the direction that information should flow over those channels. Topics can be many-to-many, one-to-many, or many-to-one. Additionally, service-client routes can be provided. Services must always designate one service provider, but may have one or more clients. Some

systems may have a different name for a topic or a service, so the `remap` dictionary allows the configuration file to specify a different name that *SOSS* should use for each system.

Here is a diagram that illustrates the concept:



In the diagram, Robot A has a bunch of internal topics and services. It wishes to export some (but not all) of them to a much larger collection of other topics and services. In the process, some topic/service names will need to change, and perhaps some other filtering will occur (for example, the rate of publishing of its location will only be 1 Hz instead of 100 Hz, or its camera image will be dramatically down-sampled, etc.). The *SOSS* configuration file will specify the topics within Robot A that the robot needs to export, as well as what system middlewares each exported topic needs to be forwarded to.

Types definition

Some **System-Handles** have the ability to inform *SOSS* of the types definition (using XTypes) that they can use. The **System-Handles** of *ROS1* and *ROS2* are examples of this. Nevertheless, there are cases where the **System-Handle** is not able to retrieve the type specification (*websocket*, *mock*, *dds*, *fiware*, ...) that it needs for the communication.

In those cases, there are two ways to pass this information to the **System-Handle**:

- Using the `types-from` property, that *imports* the types specification from another system.
- Specifying the type yourself by embedding an IDL into the YAML.

Regarding the second option, the IDL content can be provided in the YAML either directly, as follows:

```
types:
    idls:
        - >
            struct name
            {
                idl_type1 member_1_name;
                idl_type2 member_2_name;
            };
```

or by inclusion of a `paths` field, that can be used to provide the preprocessor with a list of paths where to search for IDL files to include into the IDL content. The syntax in this case would be:

```
types:
    idls:
        - >
            #include <idl_file_to_parse.idl>

        paths: [ idl_file_to_parse_path ]
```

Notice that these two approaches can be mixed.

The name for each type can be whatever the user wants, with the two following rules:

1. The name cannot have spaces in it.
2. The name must be formed only by letters, numbers and underscores.

Note: a minimum of a structure type is required for the communication.

For more details about IDL definition, please refer to IDL documentation.

The following is an example of a full configuration defining a `dds-fiware` communication using the types definition contained in the `idls` block.

```
types:
    idls:
        - >
            struct Stamp
            {
                int32 sec;
                uint32 nanosec;
            };

            struct Header
            {
                string frame_id;
                stamp stamp;
            };

systems:
    dds: { type: dds }
    fiware: { type: fiware, host: 192.168.1.59, port: 1026 }

routes:
    fiware_to_dds: { from: fiware, to: dds }
    dds_to_fiware: { from: dds, to: fiware }

topics:
    hello_dds:
        type: "Header"
        route: fiware_to_dds
    hello_fiware:
        type: "Header"
        route: dds_to_fiware
```

Systems definition

A **System-Handle** may need additional configuration that should be defined in its `systems` entry as a YAML map. Each entry of this section represents a middleware involved in the communication, and corresponds to an instance of a **System-Handle**. All **System-Handles** accept the `type` and `types-from` options in their `systems` entry. If `type` is omitted, the key of the YAML entry will be used as `type`.

```
systems:
    dds:
    ros2_domain5: { type: ros2, domain: 5, node_name: "soss_5" }
    fiware: { host: 192.168.1.59, port: 1026 }
```

The snippet above will create three **System-Handles**:

- A *DDS* **System-Handle** or **SOSS-DDS** with default configuration.
- A *ROS2* **System-Handle** or **SOSS-ROS2** named `ros2_domain` with `domain = 5` and `node_name = "soss_5"`.
- A *Fiware* **System-Handle** or **SOSS-FIWARE** with `host = 192.168.1.59` and `port = 1026`.

The **System-Handles** currently available for *SOSS* are listed in a table that you can find in the *Related Links* section of this documentation.

A new **System-Handle** can be created by implementing the desired `SystemHandle` subclasses to add support to any other protocol or system. For more information consult the *System-Handle Creation* section.

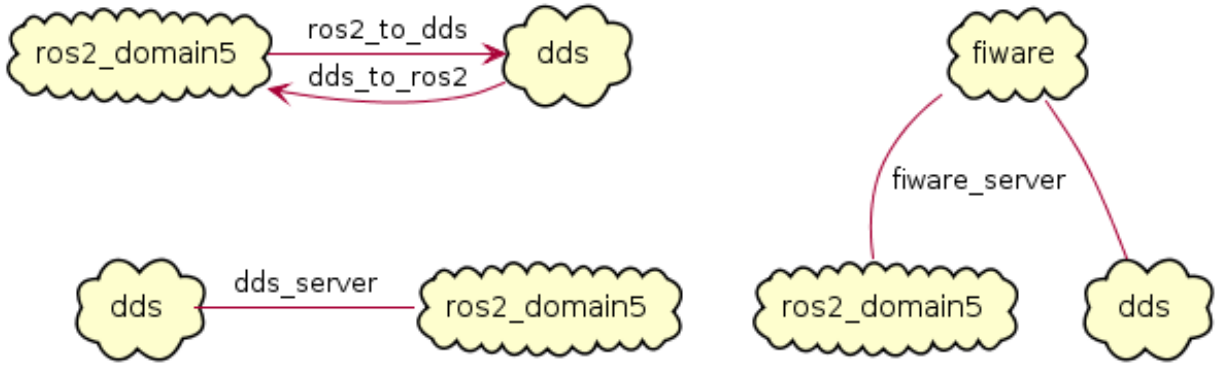
Routes definition

This section allows enumerating the bridges between the systems that *SOSS* must manage. To achieve bidirectional communication, both ways must be specified.

`routes` definition keywords are specific depending on whether the route is defining a *publisher/subscriber* path (`from-to`) or a *service/client* communication path (`server-client`). For example:

```
routes:
    ros2_to_dds: { from: ros2_domain5, to: dds }
    dds_to_ros2: { from: dds, to: ros2_domain5 }
    dds_server: { server: dds, clients: ros2_domain5 }
    fiware_server: { server: fiware, clients: [ dds, ros2_domain5 ] }
```

This YAML defines the following routes:



- The route `ros2_to_dds` defines a `ros2_domain5` publisher with a `dds` subscriber.

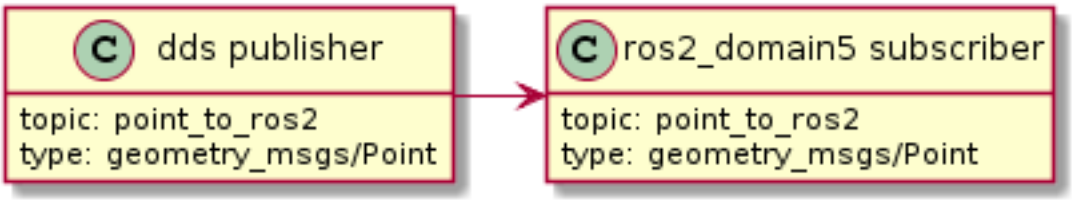
- The route `dds_to_ros2` defines a `dds` publisher with a `ros2_domain5` subscriber.
- Having the routes `ros2_to_dds` and `dds_to_ros2` results in a bidirectional communication between the `ros2_domain5` and `dds` systems.
- The route `dds_server` defines a `dds` server with only one client: `ros2_domain5`.
- The route `fiware_server` defines a `fiware` server with two clients: `ros2_domain5` and `dds`.

Topics definition

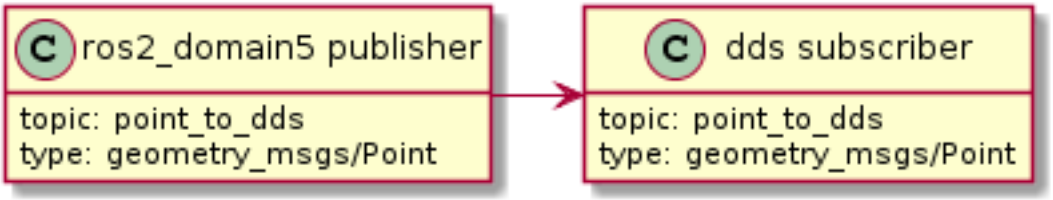
Each `system` is able to *publish/subscribe* to each other's `topics`. These *publish/subscription* policies are set directly in the YAML configuration file by specifying the topic `type` and its `route` (which `system` is the publisher and which is the subscriber) as the main parameters:

```
topics:
    point_to_ros2:
        type: "geometry_msgs/Point"
        route: dds_to_ros2
    point_to_dds:
        type: "geometry_msgs/Point"
        route: ros2_to_dds
```

- The topic `point_to_ros2` will create a `dds` publisher and a `ros2_domain5` subscriber.



- The topic `point_to_dds` will create a `ros2_domain5` publisher and a `dds` subscriber.



If a custom **System-Handle** needs additional configuration regarding the `topics`, it can be added to the topic definition as new map entries.

Services definition

`service` definition is very similar to `topics` definition, with the difference that in this case `routes` can only be chosen among the ones specified with the *server/client* syntax; also, the `type` entry for these fields usually follows the *request/response* model, pairing each of them with the corresponding `route`, depending on which `system` acts as the server and which as the client(s).

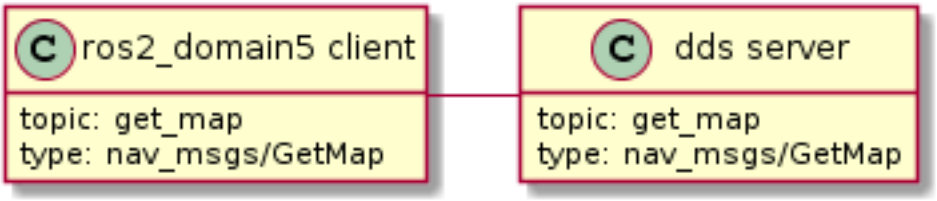
```
services:
    get_map:
        type: "nav_msgs/GetMap"
```

(continues on next page)

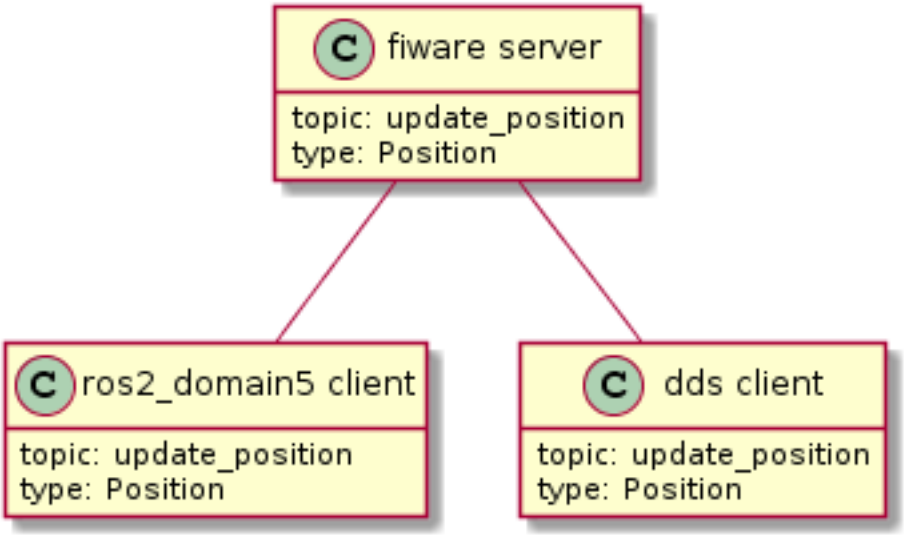
(continued from previous page)

```
        route: dds_server
    update_position:
        type: "Position"
        route: fiware_server
```

- The service `get_map` will create a `dds` server and a `ros2_domain5` client.



- The service `update_position` will create a `fiware` server, and `dds` and `ros2_domain5` clients.



If a custom **System-Handle** needs additional configuration regarding the `services`, it can be added in the service definition as new map entries.

Remapping

Sometimes, `topics` or `types` from one system are different from those managed by the systems with which it is being bridged. To solve this, *SOSS* allows to remap `types` and `topics` in the *Topics definition* and in the *Services definition*.

```
services:
    set_destination:
        type: "nav_msgs/Position"
        route: dds_server
        remap:
            dds:
                type: "dds/Destination"
                topic: "command_destination"
```

In this `services` entry, the `remap` section defines the `type` and the `topic` that must be used in the `dds` system, instead of the ones defined by the service definition, which will be used by the `ros2_domain5` system.