ROS2 Tutorial

Roberto Canale, Filip Hesse, Justin Lee, Daniel Nieto, Steven Palma, Josep Rueda

Department of Computer Science, Bioengineering, Robotics and System Engineering (DIBRIS)

University of Genova

Supervisor

Fulvio Mastrogiovanni

In partial fulfillment of the requirements for the degree of

Master of Science in Robotics Engineering

July 31, 2020

Contents

Chapter 1

Introduction

1.1 ROS1 Design Parameters

ROS1 was designed with a couple of characteristics in mind:

- single robot
- workstation-class computational resources on board
- no real time requirements
- excellent network connectivity
- research applications
- maximum flexibility

It fulfilled these requirements very well and it became much more robust than the designers initially intended. However, it still has some limitations which ROS2 tries to address.

1.2 ROS2 Design Parameters

ROS2 is designed to handle some new use cases:

- Standardize Implementation of Multi-Robot Systems
- Embedded Systems (Raspberry Pi, Arduino, etc..)
- Real-time Systems
- Network Degradation
- Product Applications (non research)
- Create patterns for building and structuring systems while keeping flexibility

1.3 New Technologies

ROS2 incorporates a lot of new technologies which are open source. There are several benefits to utilising open source libraries:

- Maintain less code
- Take advantage of features in those libraries

Some of the new technologies ROS2 uses are:

- Zeroconf
- Protocol Buffers
- Zero MQ (and other message queues)
- Redis

- Web Sockets
- Data Distribution Service (DDS)

We will go more in depth into DDS as it creates huge changes in ROS2 compared to ROS 1.

1.4 Architecture

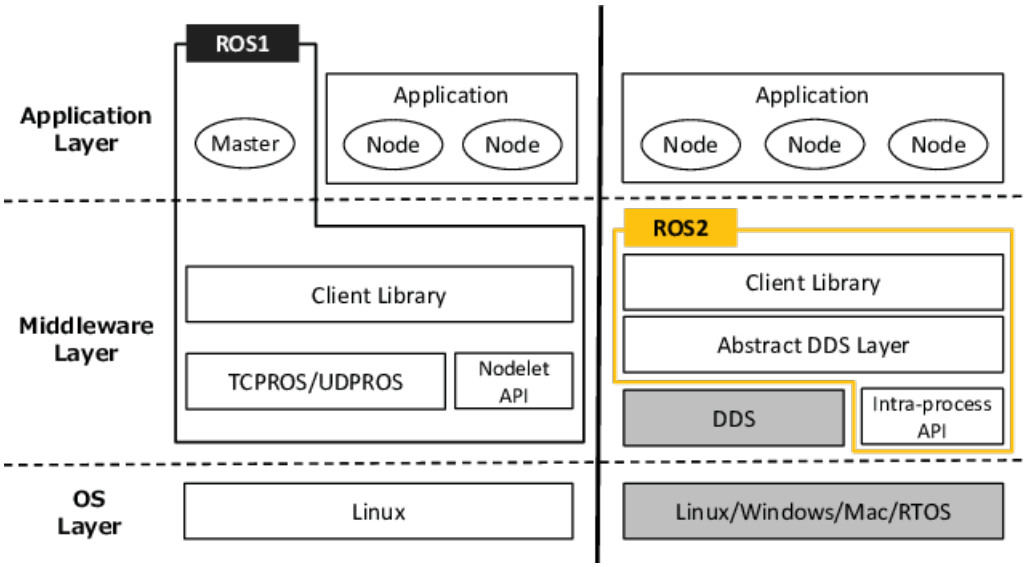


Figure 1.1: ROS1 and ROS2 Architecture

We will talk about some key differences between the ROS1 and ROS2 Architectures, which can be seen in Fig. 1.1.

1.4.1 ROS Master

In ROS1, a ROS Master node is required before starting any other nodes. This is because the ROS Master informs the other nodes about which topics they are publishing/subscribing to as well as services. It helps different nodes find each

other. This is not necessary in ROS2 because of the implementation of DDS, which will be explained later.

1.4.2 Inter-Process Communication

In ROS1, inter-process communication is handled by custom protocols called TCPROS and UDPROS, which are based off of TCP and UDP sockets. In ROS2, this is handled by the DDS implementation.

1.4.3 DDS

Data Distribution Service (DDS) is a publish subscribe transport model which uses an Interface Description Language (IDL) for message definition and serialization. Both DDS and its IDL are defined by the Object Management Group (OMG). It is meant to enable dependable, high-performance, interoperable, real-time, scalable data exchanges. It uses a request-response method of communication, which is similar to a ROS service. One of the main advantages of DDS for ROS2 is its distributed discovery system. This means that two programs can communicate with each other without the need of a centralized program (like ROS Master). This is more fault tolerant and flexible than a static discovery system. It is important to note that DDS is a protocol, and there are many different implementations of DDS. DDS is also widely used in many fields, like military, air traffic control, autonomous vehicles, medical devices, etc...

1.4.4 RMW

As stated before, there are multiple DDS implementations. The way that ROS2 deals with these different implementations is through the ROS Middleware Interface (RMW). The RMW creates a layer of abstraction between ROS2 and the

DDS implementation. This is important as it makes sure that the user doesn't have to deal with any DDS specifics due to its complexity. Each DDS implementation will have its own abstract layer to bridge the gap between the implementation and ROS2. User nodes will create ROS messages and the client library will send them to the RMW. The RMW passes them onto the DDS implementation, which then converts it to a DDS sample and publishes it. The opposite happens when the node receives a DDS data packet. It converts the data packet to a ROS message and invokes a subscriber callback. The ROS client library will then send a callback passing the ROS message to the user node. In ROS2, the default implementation is eProsima Fast RTPS.

1.5 Supported OS

ROS 1 only supported Linux OS. ROS2 supports Linux, Mac OS, Windows, and their real time variants like VxWorks. In our experience it is easier to install it on Linux as opposed to Windows as the number of dependencies for Windows are much higher than Linux.

1.6 Supported Languages

ROS1 supports Python2 and C++03 while ROS2 requires a minimum of Python 3.5 and is largely written in C++11 extensively while using some aspects of C++14. This is a huge benefit because both C++03 and python2 are not supported anymore and do not benefit from a lot of the upgrades the other versions have. Most new libraries are also not supporting these old languages.

Chapter 2

ROS2 Basic Elements

2.1 Introduction

The present chapter is a collection of the very basic elements that make up a ROS2 system. Each subsection correspond to a different element and it is composed by a brief summary of its description, important commands or steps for the reader to refer to and a list of useful resources for deeper information of each topic. The reader will find that most of the times the concepts and the important commands are very similar to those in ROS1; however, they are included anyways for sake of completeness. Moreover, the reader can also find in the tutorial repository practical examples to help him get used to the ROS2, its elements and the necessary tools to interact with them.

2.2 ROS2 Nodes

2.2.1 Concept

A ROS2 node, like a ROS1 node, is a fundamental element that serves a single, modular purpose in a robotics system. That is, if a ROS2 application system is

composed by multiple tasks like for example, a path planner, a data acquisition and a localization process, then a ROS2 node will take care of each executing each individual process. Nodes can send and receive data between each other via topics, services, actions or parameters (which will be addressed later). Therefore, a full robotic system designed with ROS2 is comprised of many nodes working in concert.

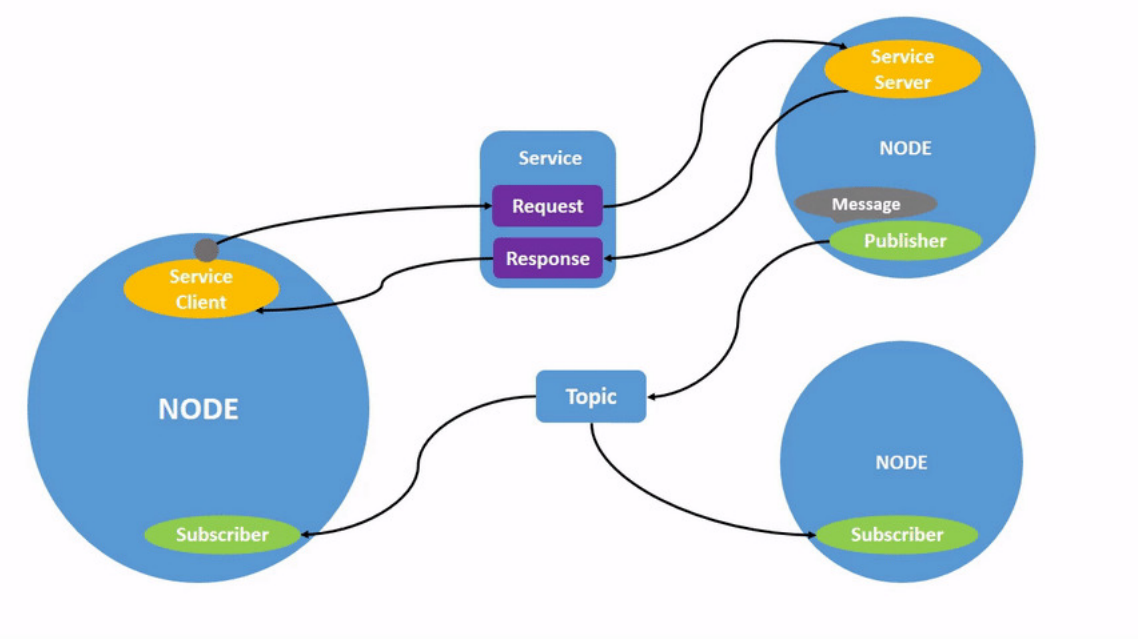


Figure 2.1: ROS2 Nodes Intuition

As it is shown in the Fig.2.1, the nodes communicate with other nodes using different interfaces while they execute a specific task. Nodes may be located in the same process, in different processes, in the same machine or on different machines. Also, in ROS2, a node can be written both in C++ and Python language.

2.2.2 Useful commands

Launches an executable from a package:

```
ros2 run <package_name> <executable_name> (--ros-args)
```

Shows the names of all running nodes:

```
ros2 node list
```

Returns a list of subscribers, publishers, services, and actions that interact with the node:

```
ros2 node info <node_name>
```

2.3 ROS2 Topics

2.3.1 Concept

As it was stated in the past subsection, nodes can send and receive data using topics. A topic is a vital element of ROS2 that act as a bus for nodes to exchange messages. Nodes can publish information over topics, which allows any number of other nodes to subscribe to and access that information.

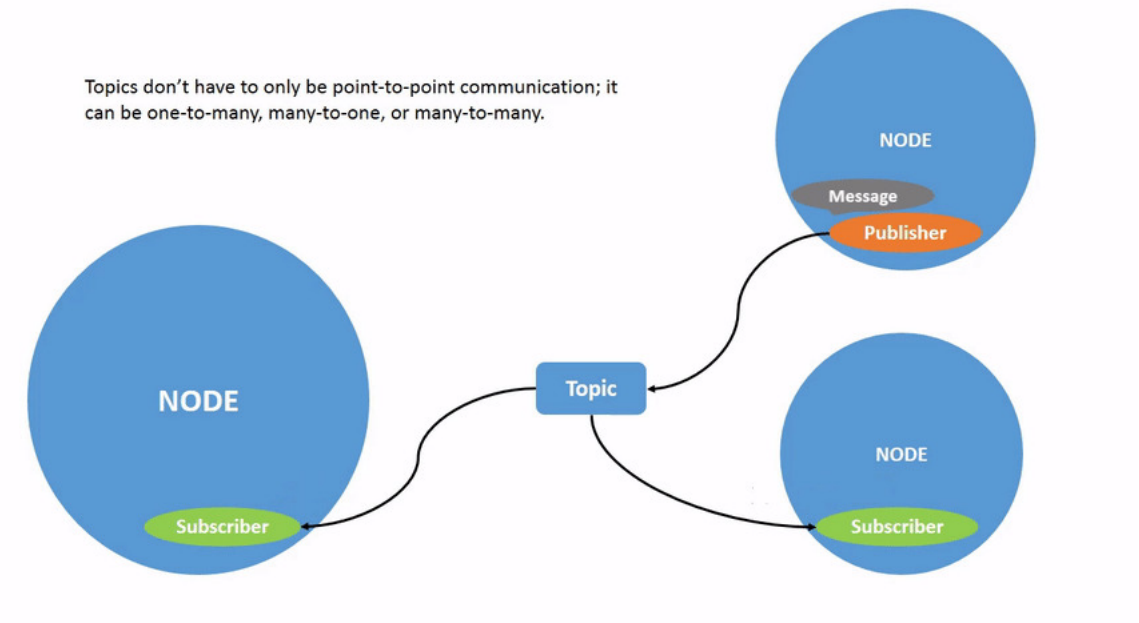


Figure 2.2: ROS2 Topics Intuition

As it can be seen in the Fig.2.2, the upper-right node is constantly publishing messages to a topic called "Topic", then the other two nodes are constantly receiving the messages sent onto the topic "Topic" to use that data in their own process. Even tho topics can be one-to-many or many-to-many, the nodes publishing and the nodes subscribed must send and receive the same type of message to communicate properly. Later in this chapter the definition of message will be clarified; however, for now the user can think of the message to be just data of a certain type like for example, an integer number or a string of characters.

2.3.2 Useful commands

Returns a list of all the topics currently active in the system

```
ros2 topic list (-t)
```

Displays the data being published on a topic

```
ros2 topic echo <topic_name>
```

Displays the type of the topic and its publisher and subscriber count:

```
ros2 topic info <topic_name>
```

Publishes data onto a topic directly from the command line:

```
ros2 topic pub <topic_name> <msg_type> '<args>'
```

Reports the rate at which data is published in the topic

```
ros2 topic hz <topic_name>
```

2.4 ROS2 Interfaces

ROS2 does not have a master node, and nodes communicate between each other through DDS at lower levels, while at a higher level the types of communications and data exchanges are called **Interfaces**. These can be of 3 kinds:

- **Messages**: Allows the creation of custom messages, used by topics in the Publisher-Subscriber Model Datatypes custom to your application, synchronous.
- **Services**: Allows the creation of a Request-Response interface between nodes, any type of data can be sent both as a Request and as a Response, asynchronous.
- **Actions**: Made of Clients and Servers, that provide constant feedback to clients on their request, asynchronous.

2.4.1 Data Types

In general, these interface work similarly, to ROS1, but Actions are now more supported and are expected to be used much more by developers. ROS2 allows the user to create interfaces with many data types, that can be found here:

Type name	C++	Python	DDS type
bool	bool	builtins.bool	boolean
byte	uint8_t	builtins.bytes*	octet
char	char	builtins.str*	char
float32	float	builtins.float*	float
float64	double	builtins.float*	double
int8	int8_t	builtins.int*	octet
uint8	uint8_t	builtins.int*	octet
int16	int16_t	builtins.int*	short
uint16	uint16_t	builtins.int*	unsigned short
int32	int32_t	builtins.int*	long
uint32	uint32_t	builtins.int*	unsigned long
int64	int64_t	builtins.int*	long long
uint64	uint64_t	builtins.int*	unsigned long long
string	std::string	builtins.str	string
wstring	std::u16string	builtins.str	wstring

Every built-in-type can be used to define arrays:

Type name	C++	Python	DDS type
static array	std::array<T, N>	builtins.list*	T[N]
unbounded dynamic array	std::vector	builtins.list	sequence
bounded dynamic array	custom_class<T, N>	builtins.list*	sequence<T, N>
bounded string	std::string	builtins.str*	string

Figure 2.3: ROS2 Data Types

ROS2 however expands the available Data Types and provides 3 new data types that are unavailable in ROS1:

- **Bounded Arrays** arrays with a size upper bound.
  - **Example** int32[<=5];
- **Bounded Strings** strings with a a size upper bound.
  - **Example** string<=5;
- **Default Values** allows to set default values

Also, it is possible to fix constants in Interfaces, but they need to be saved in CAPITAL LETTERS when initialized in their respective files.

2.4.2 Useful Commands

To use an interface, the user can use the following in the nodes to interface with it:

```
<package_name::interface_type::interface_name>
```

Also, it is possible to check the available interfaces by issuing the following command in the terminal:

```
ros2 interface show <package_name>/<interface_type>/<
   interface_name>
```

2.4.3 Further Capabilities and Remarks

It is also worth noting some further capabilities of ROS2 interfaces. In particular, it is possible to import custom interfaces from one package to another and use them. Let's say package1 contains a message that package2 wants to use, then it is necessary to update the following files with the following code:

CMakeLists.txt

```
find_package(package1 REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  ${msg_files}
  DEPENDENCIES package1
```

package.xml

```
<build_depend>package1</build_depend>

<exec_depend>package1</exec_depend>
```

2.5 ROS2 Messages

Messages allows for the creation of custom messages to be used between nodes. To create a custom message, the package also needs to be able to create custom interface. It is therefore necessary to insert the following in the *package.xml*:

```
<build_depend>rosidl_default_generators</build_depend>

<exec_depend>rosidl_default_runtime</exec_depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

The next steps require the user to create a *msg* folder in the package directory and then create a *MyCustomMessage.msg* into such directory. Next, **write the custom message** with the desired data types.

The last step is to modify the *CMakeLists.txt* file with the following lines, in order to properly build the custom message:

```
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}

"msg/MyCustomMessage.msg"
)
```

It is now possible to use the custom message in the code with the following structure:

```
<package_name::msg::msg_name>
```

Do not forget to include the header inside your nodes by adding the following line:

```
#include "package_name/interface_type/interface_name.hpp"
```

2.6 ROS2 Services

2.6.1 Concept

Nodes can also communicate using ROS2 services. Services are based on a call-and-response model, so a service will only pass information to a node if that node specificalle request it, and will only do so once per request (not in a continuos streams). In contrast to the use of topics for communication, a service generally isn't used for continuous calls, but for a precise and asynchronous call.

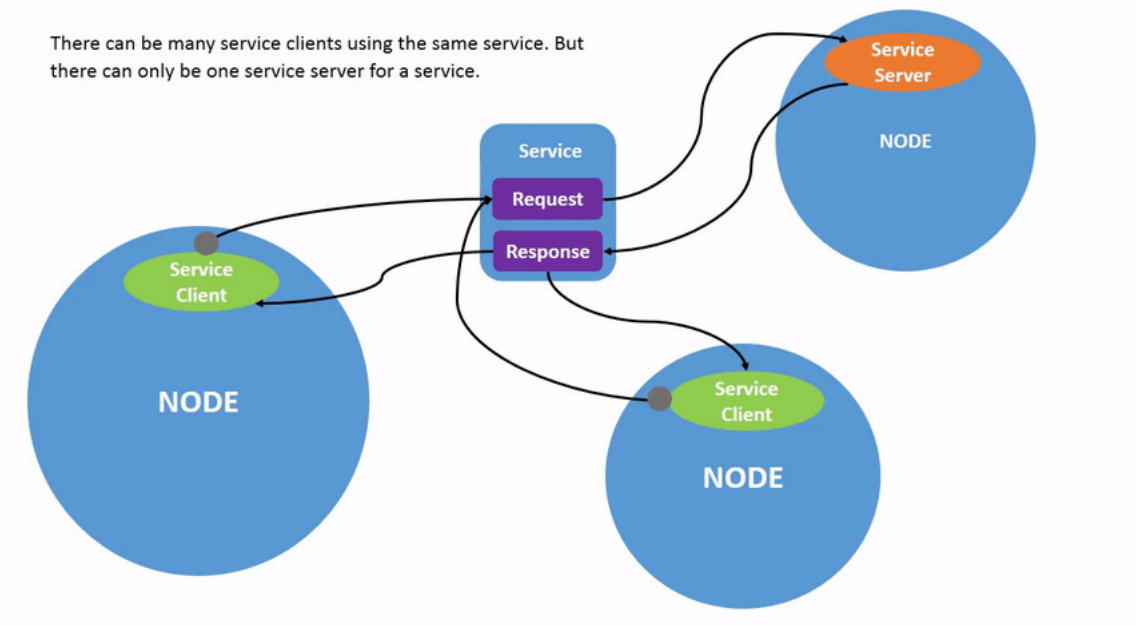


Figure 2.4: ROS2 Services Intuition

AS it is shown in the Fig.2.4 not only It is needed to declare a service with Its own type of request and response interface but It is also needed to declare a

node as the server or the service. This node will be the responsible for treating, resolving and answering back each request sent by any client node.

2.6.2 Useful commands

Return a list of all the services currently active in the system:

```
ros2 service list (-t)
```

Finds out the type of a service:

```
ros2 service type <service_name>
```

Finds all the services of a specific type:

```
ros2 service find <type_name>
```

Calls a service from terminal:

```
ros2 service call <service_name> <service_type> <arguments>
```

2.6.3 Hands-on activity: Services

The reader can find in the tutorial repository a practical example for the use in ROS2 of services written both in Python and C++ language. However; it is recommended to follow this hands-on activity after going through the Chapter 3 on how to manage a ROS2 application from scratch and after following the more in-detail and guided hands-on activity for the publisher and subscriber explained in Chapter 4.

After configuring the ROS2 environment on terminal, creating a new workspace, cloning the repository and building properly the packages, the reader can follow the next steps.

In the ROS2_tutorial/ros2_ws directory,

```
. install/local\_setup.bash
```

Then, for starting the server node written in Python:

```
ros2 run py\_srvcli service
```

Open a new terminal and then run:

```
. install/local\_setup.bash
```

Then, for starting the client node written in Python:

```
ros2 run py\_srvcli client 2 3
```

The reader should then see the message displayed by both nodes indicating that the request of the service was successful. To stop the nodes the reader can press CTRL+C while being in the terminal. Same but now for the server and client node written in C++, the reader can follow the same steps but this time running the following commands.

To start the server node written in C++:

```
ros2 run cpp\_srvcli server
```

To start the client node written in C++:

```
ros2 run cpp\_srvcli client 2 3
```

Notice that it is possible to use a combination of both. That is, It is possible to call the python server node with the c++ client node and so on. The reader is encourage to try different combinations and to analyze why this is possible.

2.7 ROS2 Actions

2.7.1 Concept

Actions are other type of asynchronous communication between nodes in ROS2 specifically intended for long running tasks given by a goal. They consist of three main parts: a goal, a result, and a feedback. In terms of functionality they are very similar to services, since an "action client" node sends a goal to an "action server" node that acknowledges the goal and returns a stream of feedback and a

result. However; actions are not only different from services only because they provide steady feedback onto a topic to the client node, but also because actions are preemptable. That means that the user can cancel the execution of the action while executing.

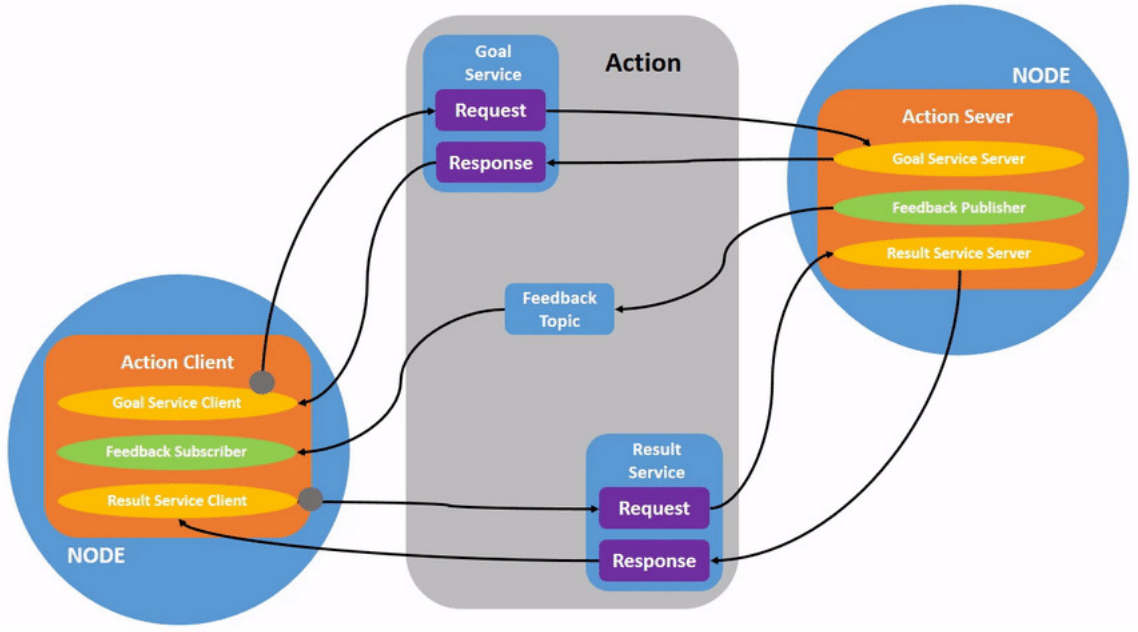


Figure 2.5: ROS2 Actions Intuition

Actions then become very useful for long running tasks as said before. For example, lets say that there is a robotic manipulator that needs to change its orientation. By using a simple ROS2 service, the client node will send a request to the server node indicating the new desired orientation, then the server will compute all the necessary steps in order to do it, including the actual progressive movement from one position to another. However; in that process the client node is clueless about what the server is doing and about the overall progress of the task, that is, the actual -but intermediate- orientation of the manipulator, also if for some reason the request or the computations of the server fails and the manipulator starts moving in the wrong direction the system will have to be

stopped and this is very inconvenient. On the other hand, these problems don't arise if actions were used. That is because actions offer a feedback topic to the client; for example, in this case the client would know the progress and actual position of the manipulator, maybe just for checking if its doing it right or maybe for using this data to process another task, and if the process isn't being what it is required or excepted, the task can be stopped without having to kill the nodes.

2.7.2 Useful commands

Identifies all the actions in the ROS graph:

```
ros2 action list (-t)
```

Returns the action clients and servers:

```
ros2 action info <action_name>
```

Sends an action goal from the command line:

```
ros2 action send_goal <action_name> <action_type> <values> (--
   feedback)
```

Checks if the definition of the action exists:

```
ros2 interface show <package_name>/action/<action_name>
```

2.7.3 Hands-on activity: Actions

Just as for services, the reader can find in the tutorial repository a practical example for the use in ROS2 of actions written both in Python and C++ language. However; -again- it is recommended to follow this hands-on activity after going through the Chapter 3 on how to manage a ROS2 application from scratch and after following the more in-detail and guided hands-on activity for the publisher and subscriber explained in Chapter 4.

After configuring the ROS2 environment on terminal, creating a new workspace, cloning the repository and building properly the packages, the reader can follow the next steps.

In the ROS2_tutorial/ros2_ws directory,

```
. install/local\_setup.bash
```

Then, for starting the action server node written on Python:

```
ros2 run action_tutorials_py fibonacci_action_server
```

Open a new terminal and run:

```
. install/local\_setup.bash
```

Then, for starting the action client node written on Python:

```
ros2 run action_tutorials_py fibonacci_action_client
```

The reader should then see a message displayed by both nodes indicating that the goal given by the client is being processed while the server is constantly publishing feedback of the task. To stop the nodes the reader can press CTRL+C while being in the terminal. Same but now for the action server and action client node written in C++, the reader can follow the same steps but this time running the following commands.

To start the action server node written in C++:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

To start the action client node written in C++:

```
ros2 run action_tutorials_cpp fibonacci_action_client
```

Notice that -again- it is possible to use a combination of both. That is, It is possible to call the python action server node with the c++ action client node and so on. The reader is encourage to try different combinations and to analyze why this is possible.

2.8 Launch Files in ROS2

Launch Files in ROS2 have the same goal and purpose as ROS1: launching and monitoring multiple nodes at the same time. Like ROS1, these launch files allow us to set parameters, environment variables, remap topics and send arguments to the various packages that are being launched. However, ROS2 launch files have two major differences:

- ROS2 launch files are written in *Python 3.5* while ROS1 launch files are written in *XML*.
- ROS2 launch files have more **TAGS** that can be used and set.

The following table sums up the major key differences:

ROS1	**ROS2**
XML	Python 3.5
Node	!NEW TAGS! (+all the ones already in ROS)
Param *//parameter to a node*	Set_env and unset_env
Rosparam *//parameter to yaml*	Push-ros-namespace
Remap *//remapping topics*	Let *//replaces Arg*
Include *//include other launchfiles*	Executable *//for executables*
Arg *//launch argument*	
Env *//environment variables*	
Group	

2.8.1 How to create a Launch File

Creating launch files in ROS2 is quite straightforward and very similar to ROS1. In the package folder, it is necessary to create a *launch* directory and then create a launch file in it. However, there are small differences between C++ and Python Packages

C++

Name the file *my_launchfile.py* and add to the *CMakeLists.txt* the following lines:

```
install(DIRECTORY
launch
DESTINATION share/\${PROJECT\_NAME}/
)
```

Python

It is necessary to name the launch file as *my_launchfile.launch.py*

In the *setup.py*, add the following lines:

```
import os
from glob import glob
from setuptools import setup

package_name = 'my_package'

setup(
    # Other parameters ...
    data_files=[
        # ... Other data files
        # Include all launch files. This is the most
   important line here!
        (os.path.join('share', package_name), glob('launch
   /*.launch.py'))
    ]
)
```

Next, it is necessary to write the launch file: a code snippet to add to the launch file is provided below, where we can see how to initialize each node and how to remap a topic:

```
from launch import LaunchDescription
from launch_ros.actions import Node     //And others if needed

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='package_name',
            node_namespace='namespace',
            node_executable='name_executable',
            node_name='name_of_the_node',
            remappings=[('/old/topic', '/new/topic')])
 ])
```

Lastly, we need to know how to launch our packages through a launch file, after the overlay and underlay are sourced. In your launch directory, it is possible to execute the following command (without the need of a ROSMASTER) to launch your nodes:

```
ros2  launch <package_name> <launch_file_name>
```

2.9 Useful sources

Even if the present document has a well written bibliography at the end, this section has the objective of serving as a quick and clean point of reference for useful sources about the elements treated in the present chapter.

For more information about nodes:

- `https://bit.ly/2BI0Vly`
- `https://bit.ly/30dIuPl`

For more information about topics:

- `https://bit.ly/3gfsQII`

For more information about services:

- https://bit.ly/3fbWixJ

For more information about actions:

- https://bit.ly/39KDXXM
- https://bit.ly/2EBuGpp
- https://bit.ly/2CVlfAN

Chapter 3

ROS2 Application Managment

3.1 Introduction

After reading about the basic ROS2 elements that can be used in order to develop a robotic software application, it is natural to wonder how can all those elements work and stick together in a specific robotic system. In the present chapter, the reader will find the necessary concepts and steps in order to create from scratch the right environment and workspace for his ROS2 application.

3.2 Configuring the ROS environment

3.2.1 Useful steps

In order to configure the ROS2 environment in the user's machine, the next steps are usually required.

Run this command on every new shell that its opened to have access to the ROS 2 commands:

```
source /opt/ros/<distro>/setup.bash
```

Optionally, display information about the environment setup by running:

```
printenv | grep -i ROS
```

3.3 Creating a workspace

3.3.1 Concept

A workspace is essentially a directory in the machine that contains the ROS2 packages (next section) needed for the robotic application, it is the location on the machine where the user is developing with ROS2. Usually the workspace folder must have a src directory in it where the developed packages are placed. A good practice is to create a new directory for every new workspace and application.

With ROS2, the user has the option of sourcing overlays and underlays. The core ROS 2 workspace or a source installation is called the underlay, subsequent local workspaces are called overlays. An overlay is a secondary workspace where the user can add new packages without interfering with the existing ROS 2 underlay workspace and the underlay workspace must must contain the dependencies of all the packages in the overlay. It is possible to have several layers of underlays and overlays, with each successive overlay using the packages of its parent underlays. Also, packages in the overlay will override packages in the underlay. Using overlays is recommended for working on a small number of packages, so the user doesn't have to put everything in the same workspace and rebuild a huge workspace on every iteration.

3.3.2 Useful steps

In order to configure a ROS2 workspace, the next steps are required.

First, create a new directory by running in a terminal:

```
mkdir -p ~/<workspace_name>/src
cd ~/<workspace_name>/src
```

Then, create a new package (next section) or clone an existing one in the src directory.

Before building the workspace, it is usually needed to resolve package dependencies by running:

```
rosdep install -i --from-path src --rosdistro <distro> -y
```

From the root of the workspace directory, build the packages using the command:

```
colcon build
```

Lastly, source an overlay in every terminal to be used by executing:

```
. install/local_setup.bash
```

3.4 Creating a package

3.4.1 Concept

A package can be considered a container for your ROS2 code. If the workspace is the game room, then the package is a box in it full of toys and the nodes are the toys. A single workspace can contain as many packages as you want, each in their own folder. By designing a ROS2 application using packages, the work can be easily released and shared to others and used by others.

3.4.2 Useful steps

In order to create a ROS2 package, the next steps are required.

The user can decide if he wants to create a package using the CMake build tool or the Python build tool. Once in the proper src directory of the workspace created, the command syntax for creating a new ROS2 package is: Using CMake:

```
ros2 pkg create --build-type ament_cmake <package_name>
```

Using Python:

```
ros2 pkg create --build-type ament_python <package_name>
```

To run a node inside a package:

```
ros2 run my_package my_node
```

After creating the package, the user should modify the file 'package.xml' automatically generated to fill the details of the package description. Notice that with these steps an empty ROS2 package is created, if the user wants to add his own nodes and customize deeper his package, he will have to edit the 'CmakeList.txt' or the 'setup.py' file depending if the package was created using the CMake or the Python tool. To know the details about these modifications to these files, the user is encourage to check the sources cited below for creating a package.

3.5 Useful sources

Even if the present document has a well written bibliography at the end, this section has the objective of serving as a quick and clean point of reference for useful sources about the topics treated in the present chapter.

For more information about configuring the ROS2 environment:

- `https://bit.ly/2P6Vj7F`

For more information about creating a workspace:

- `https://bit.ly/3gg7JWy`

For more information about creating a package:

- `https://bit.ly/2XbK2XT`
- `https://bit.ly/2D3894j`

Chapter 4

Hands on: Create a basic Publisher and Subscriber

In this part of the tutorial, we will get practical and examine the code for a basic publisher and subscriber in ROS2. A publisher periodically publishes a messages to a topic. Each subscriber, that subscribes to that topic, will receive those messages. In this most basic example the subscriber will simply output the received string to the terminal. We stay close to the official ROS2 tutorial [1], but we will especially emphasize the differences with ROS1. Just like ROS1, ROS2 has native support for C++ and Python. The code we will mainly look at is written in python, the corresponding C++ code will just briefly be commented afterwards. In order to compare more easily ROS2 code with ROS1 code, both versions will be shown first, and then analyzed in the following.

ROS2 requires at least Python version 3.5, which is different to many ROS1 versions that only support Python 2. Still, this feature is not completely new, because ROS1 Noetic (release date May 2020) also requires Python 3.

All source code files can be found in the corresponding Repository on Github [2].

4.1 Publisher (Python)

4.1.1 ROS1 - Code

```
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

ros1_ws/src/tutorial_py/scripts/talker.py

4.1.2 ROS2 - Code

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1


def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

ros2_ws/src/tutorial_py/tutorial_py/talker.py

4.1.3 Explanations

Before even considering the content of the files, we observe the location of the talker.py file: Python scripts must appear in a python package which is located in the directory of the ROS2 package and has the same name as the ROS2 package. This is why the scripts in our case appear in the directory ros2_ws/src/tutorial_py/tutorial_py/. In ROS1, script files were allowed to be anywhere in the ROS package, usually they would be located inside a scripts/ or src/ folder. That change was necessary for the new build process for python files that will be explained later.

In the first line of both codes, the first difference appears: In ROS2 we import rclpy instead of rospy. A completely different python-package is needed to access ROS2 features, because ROS2 communication mechanisms internally work different from the mechanisms in ROS1.

Comparing the ROS1 and ROS2 codes, one can notice that the ROS2 code for that minimal example has many more lines than the ROS1 code. This mainly has to do with the much more object oriented approach of ROS2: Each node is represented as an object of a self defined class, which has to be derived from the "Node" baseclass from rclppy.node. The source code for an entire node consists of importing packages, the node class definition and the definition of the main function. In ROS1 the entire code for a node could be written inside a single function.

Node class definition

The requirements for the definition of a publisher node class are:

- Derive from Node: `class MinimalPublisher(Node):`
- Define constructor (`def __init__ (self):`) with
  - Initialization of base class with node name:

`super().__init__('minimal_publisher')`

- Construct a private publisher member specifying the type of the messages, the name of the topic (in our case 'topic') and the queue-size for old messages in case the subscriber can not receive the messages fast enough:
  `self.publisher_ = self.create_publisher(String , 'topic ', 10)`
- Construct a timer member specifying a period for publishing and a callback function:
  `self.timer = self.create_timer(timer_period, self.timer_callback)`

- Define the callback function for the timer: `def timer_callback( self)` This function gets called periodically with the specified timer period. This function has to include:
  - Publishing the specified message:
    `self.publisher_.publish(msg)`

Main function definition

A ROS2 program, independent if it represents a publisher or subscriber typically consists of the following operations:

- Initialization: `rclpy.init(args=args)`
- Create one or more ROS nodes:
  `minimal_publisher = MinimalPublisher ()`
- Process node callbacks: `rclpy.spin(minimal_publisher)`
- Shutdown: `rclpy.shutdown ()` The biggest difference to ROS1 in this part is, that publishers work with callbacks of timer objects now.

This structure allows the timer object to handle explicitly periodic tasks. That is especially interesting for the implementation of ROS2 for real time systems, because this structure is common for the creation of periodic tasks.

4.2 Subscriber (Python)

4.2.1 ROS1 - Code

```
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

ros1_ws/src/tutorial_py/scripts/listener.py

4.2.2 ROS2 - Code

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)


def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

ros2_ws/src/tutorial_py/tutorial_py/listener.py

4.2.3 Explanations

The subscriber code in ROS2 again has many differences to the ROS1 code, but this time it is *very* similar to the ROS2-version of the publisher. The structure is again a class definition with a constructor and a callback function and a main function with initialization, node creation, callback processing and shutdown.

This entire unification of code structure makes the code much cleaner and more readable provided the programmer is familiar with object oriented programming.

4.3 C++ Versions of code

For sake of readability the C++ code is not included in this text, but it can be found in the provided repository. Similarly to the python version, there is also a new library to be included: "rclcpp/rclcpp.hpp" instead of "ros/ros.h". This library is written in C++ 11 instead of C++ 03 and extensively makes use of modern C++-features such as smart pointers, STL-containers and standard algorithms. These features are also required in the rclcpp-interface. So in order to code for ROS2 in C++, the programmer has to be familiar with features of modern C++ as well as object oriented programming. The user source code can be written even in C++ 14. For programming ROS1, much more rudimentary C++ knowledge was sufficient.

Apart from that, the code for publishers and subscribers are structurally identical to the python version and differ from the ROS1 versions in the same aspects.

Chapter 5

Building and compiling nodes

In this chapter we are going to build and execute the nodes we have created in the previous chapter. Because the packaging tool is ament now and the build tool is colcon (Collective Construction) instead of catkin, there are several differences in the building and executing process with respect to ROS1.

5.1 Edit package.xml

The first step has not changed from ROS1 to ROS2: The file Package.xml needs to be edited adding the dependencies rclpy an std_msgs. This could have been done already when creating the package using the argument `--dependencies rclpy std\_msgs`. Note that over time three different formats of the package.xml emerged. ROS2 only support format 2 and higher, while recent ROS1 versions support all of them. The only difference with ROS1 are the names of the libraries the project depends on: For python we use rclpy instead of rospy and rclcpp instead of roscpp. In our case the following lines need to be added:

Python

```
<exec_depend>rclpy</exec_depend>
```

```
<exec_depend>std_msgs</exec_depend>
```

C++

```
<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

5.2 Edit setup.py and CMakeLists.txt

The second step differs a lot between building a python package and building a C++ package. There is also the possibility to build packages which contain C++ code and Python code, but that case is a little bit more complicated and will not be covered here.

Python

To build Python packages the standard module distribution process of python is used: setuptools. This tool is used to provide some metadata about the package and define entry points for execution. Setuptools will also create executables such that later we will not execute a *.py file but an executable. This is why we do not necessarily need to run chmod +x on the python scripts. All the necessary settings are made in the setup.py file, that can be found inside the package directory. It is the file that corresponds to the CMakeLists.txt from C++ packages (CMakeLists.txt does not exist in pure python packages anymore!). The relevant changes have to be made in the entry_points section. We add the entry points in the following form: <nodename> = <package>.<scriptname>:<entryfunction>

```
entry_points={
        'console_scripts': [
                'talker = tutorial_py.talker:main',
                'listener = tutorial_py.listener:main',
        ],
},
```

Now we have defined the entry function for a python-script, which makes the last two lines of our scripts obsolete (`if __name__ == '__main__': main()`)

C++

Due to the new build tool colcon and the packaging tool ament, there are plenty small differences in the CMakeLists.txt. The code snipped below shows, what needs to be added to the CMakeLists.txt for ROS2. Mainly some keywords change with respect to ROS1 and the last `install(Targets ...)` part is added.

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(talker src/talker.cpp)
ament_target_dependencies(talker rclcpp std_msgs)

add_executable(listener src/listener.cpp)
ament_target_dependencies(listener rclcpp std_msgs)

install(TARGETS
  talker
  listener
  DESTINATION lib/${PROJECT_NAME})
```

5.3 Build and run

At this point we have prepared everything to build and run our code. From now on, we do not have to differentiate between C++ and Python packages anymore. Just follow the following steps:

1. It is good practice but not mandatory to run the following command to install missing dependencies, so we do not have to analyze build errors before

installing dependencies. If the following command does not work, install python3-rosdep2 first (via apt):

```
rosdep install -i --from-path src --rosdistro foxy -y
```

The new command to build a package is the following (equivalent to ROS1 catkin_make):

```
colcon build
```

If we did not make any mistakes before, the build process should be successful. Before finally running our nodes we first need to source the setup file in the "install/" (not devel/) directory of the workspace in each terminal.

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

Now we are finally ready to run our nodes. Note, that we do NOT need to launch the ros master, so the command `roscore` is not necessary! We start by running the talker node with the following command (not rosrun tutorial_py talker.py):

```
ros2 run tutorial_py talker
```

We do not have to write the suffix `.py` behind the call of our talker script, although we are using python. The reason is, that we used setuptools for python packages which created exacutables, that we are calling now. In another terminal run the listener code (after sourcing install/setup.bash):

```
ros2 run tutorial_py listener
```

Now we should see the talker publishing and the listener receiving the messages.

Chapter 6

ROS Bridge

6.1 Introduction

ROS Bridge (or ros2/ros1_bridge) is a ROS2 package that enables communication between ROS1 and ROS2 nodes. It provides support for the bidirectional exchange of both messages and services between the two versions of the Robot Operating System, with the pre-compiled binaries supporting common interfaces such as tf2_msgs.

The package's main advantage relies on the fact that ROS2 is still relatively new, and porting the code from one version to the other takes tremendous amounts of effort. By using ROS Bridge, developers are given the possibility to use the already working and tested ROS1 packages, while exploiting the new features and capabilities that ROS2 has to offer.

As already mentioned, ROS Bridge can be installed through pre-compiled binaries. However, this has certain limitations, as packages providing custom messages and services cannot be used out of the box. In this part of the tutorial, a workflow will be covered which will allow the reader to use any ROS1 package with ROS2, regardless of whether it contains custom messages or not.

6.2 Standard Messages

Before diving into the workflow that will be used to get any ROS1 package to work with ROS2, it is important to understand what ROS Bridge does. For this reason, the current section of the tutorial will guide the reader through the basic steps needed to install ROS Bridge and use it in their application.
This section will build upon the contents of chapter 4 by using the basic publisher and subscriber. For this particular section, we ask the reader to clone the repository for this tutorial. However, the repository's folder should be renamed in order for it not to interfere with the folder we will be using during the next sections, which won't be cloned from the repository but developed step by step. To clone and rename the repository, open up a terminal on your desired location and run the following commands:

```
git clone https://www.github.com/FilipHesse/ROS2_Tutorial
mv ROS2_Tutorial/ TutorialFiles/
```

Now, let us install ros2/ros1_bridge from its pre-compiled binaries (this will not interfere with the subsequent source installation due to it overlaying the binary one):

```
sudo apt install ros-<ros2_distro>-ros1-bridge
```

We will now illustrate how the bridge works by running the publisher node in ROS1 and the subscriber in ROS2, with ROS Bridge bridging the connection between the two. In order to do so, open up four different terminals in the TutorialFiles folder and type the following commands in each one (in order):
Terminal 1 (ROS1):

```
source /opt/ros/<ros1_distro>/setup.bash
roscore
```

Terminal 2(ROS1);

```
source /opt/ros/<ros1_distro>/setup.bash
cd ros1_ws
```

```
source devel/setup.bash
rosrun tutorial_cpp talker
```

Terminal 3(ROS2):

```
source /opt/ros/<ros2_distro>/setup.bash
cd ros2_ws
source install/local_setup.bash
ros2 run tutorial_cpp listener
```

Terminal 4(Bridge);

```
source /opt/ros/<ros1_distro>/setup.bash
source /opt/ros/<ros2_distro>/setup.bash
ros2 run ros1_bridge dynamic_bridge --bridge-all-topics
```

After typing the last command, you should see the ROS2 program in Terminal 3 displaying the messages received from the ROS1 program. You can do this the other way around too, simply replace talker with listener and vice-versa in the above commands.

6.3 Custom Messages and Services

Throughout this section, the process needed to use packages providing custom messages and services in conjunction between ROS1 and ROS2 will be explored. In order to do so, ROS Bridge will have to be installed from source.

6.3.1 Workspace Creation

The first important thing to do is to create three different workspaces. One of them will be destined towards ROS1 code, the other towards ROS2 code, and the third one towards the ROS Bridge installation. For more information on how to create packages in ROS2, see chapter 3. The following folder structure must be created:

```
ROS2_Tutorial
├── ros1_ws
│   └── src
└── ros2_ws
```

```
│   └── src
└── bridge_ws
    └── src
```

To create these workspaces, open up a terminal and type:

```
mkdir ROS2_Tutorial
cd ROS2_Tutorial
mkdir -p ros1_ws/src
mkdir -p ros2_ws/src
mkdir -p bridge_ws/src
```

6.3.2 Package Creation

Now, the ros2/ros1_bridge repository must be cloned into the bridge_ws workspace by typing into the terminal:

```
cd bridge_ws/src
git clone https://www.github.com/ros2/ros1_bridge
```

We will now create two packages, one in ROS1 and another one in ROS2. For more information on how to create packages in ROS2, see chapter 3. It is important to note that both packages must have the same name. First, open up a terminal and navigate to the ROS2_Tutorial directory. You will use this terminal for ROS 1. Inside the terminal, source the ROS1 installation and create a new package called tutorial_msgs, as follows:

```
source /opt/ros/<ros1_distro>/setup.bash
cd ros1_ws/src
catkin_create_pkg tutorial_msgs
```

Open up a different terminal, which you will use for ROS2. Navigate towards the ROS2_Tutorial directory and type in the following lines to source your ROS2 installation and create a new package with the same name as its ROS1 counterpart:

```
source /opt/ros/<ros2_distro>/setup.bash
cd ros2_ws/src
ros2 pkg create --build-type ament-cmake tutorial_msgs
```

6.3.3 Custom Message Definition

Let us now define a custom message that could be used to output the results from an image classifier. To simplify the process, we will start by using the exact same file names for both ROS1 and ROS2 packages, however, this is not necessary. We will later discuss how to use different file or field names. For more information on how to define custom messages, services, and actions in ROS2, see chapter 2. In both the ROS1 and ROS2 terminals you had opened up, run the following commands to create a msg directory in the packages you just created and create a new file called CustomMessage.msg containing the message definition:

```
cd tutorial_msgs
mkdir msg
cd msg
touch CustomMessage.msg
```

Open the message definition file in both the ROS1 and the ROS2 packages and copy the following code to it:

```
string class_name
float32 confidence
uint16[4] bounding_box_coords
```

6.3.4 Nodes

Now, we need a node running in ROS2 and another node running in ROS1 that will communicate with each other using a topic carrying messages of this new type. Create a new ROS1 package inside the ros1_ws/src directory called tutorial_cpp. Inside the src directory of the new package, create a C++ source file called ros_bridge.cpp and paste the following code into it:

```
#include "ros/ros.h"
#include <tutorial_msgs/CustomMessage.h>

#include <ctime>
```

```
 int main(int argc, char **argv) {
   srand((unsigned) time(0));

   ros::init(argc, argv, "ros_bridge_1");
   ros::NodeHandle nh;

   ros::Publisher classifier_pub = nh.advertise<
tutorial_msgs::CustomMessage>("classification", 1000);

   ros::Rate loop_rate(10);

   std::string classes[3] = {"Apple", "Banana", "Watermelon
"};

   int i;

   while(ros::ok()){
     tutorial_msgs::CustomMessage msg;

     i = rand() % 3;
     msg.class_name = classes[i];

     msg.confidence = static_cast<float>(rand()) /
static_cast<float>(RAND_MAX/100);
     for(i = 0; i < 4; i++){
       msg.bounding_box_coords[0] = rand() % 125;
       msg.bounding_box_coords[1] = rand() % 125;
       msg.bounding_box_coords[2] = (rand() % 125) + 125;
       msg.bounding_box_coords[3] = (rand() % 125) + 125;
     }

     classifier_pub.publish(msg);

     ros::spinOnce();
       loop_rate.sleep();
   }

   return 0;
 }
```

The above node makes use of the custom message we just created and publishes random values for all the fields, with three different fruit (Apple, Banana,

Watermelon) chosen as the classes "detected" by the node.

Inside the src directory of the ros2_ws, create a package called tutorial_cpp using the commands from before for ROS2. Navigate to the package and paste the following code into a file named ros_bridge.cpp located inside the src directory of the new package:

```
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "tutorial_msgs/msg/custom_message.hpp"
using std::placeholders::_1;

class ROS1Subscriber : public rclcpp::Node
{
  public:
    ROS1Subscriber()
    : Node("ros_bridge_2")
    {
      subscription_ = this->create_subscription<
tutorial_msgs::msg::CustomMessage>(
      "classification", 10, std::bind(&ROS1Subscriber::
class_callback, this, _1));
    }

  private:
    void class_callback(const tutorial_msgs::msg::
CustomMessage::SharedPtr msg) const
    {
      RCLCPP_INFO(this->get_logger(), "Class detected: '%s
'", msg->class_name.c_str());
    }
    rclcpp::Subscription<tutorial_msgs::msg::CustomMessage
>::SharedPtr subscription_;
};

int main(int argc, char *argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<ROS1Subscriber>());
  rclcpp::shutdown();
  return 0;
}
```

The above node subscribes to messages of the custom type and outputs the class of the received messages to the screen. To better understand the code for this node and the one above it, see chapter 4.

6.3.5 Package Set-Up

What is left to do before we can make these two nodes communicate is compile and run the packages. You can see more information on the configuration of the CMakeLists.txt and package.xml files for custom messages on chapters 2 and 5, but the corresponding files are nevertheless added here for completeness.
Let us first change the package.xml and CMakeLists.txt files for each of the two packages in the ROS1 workspace. First, add the following lines to the tutorial_msgs package.xml:

```
<build_depend>message_generation</build_depend>
<build_depend>std_msgs</build_depend>
<exec_depend>message_runtime</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

Edit the CMakeLists.txt on the same package so that it looks like this:

```
cmake_minimum_required(VERSION 3.0.2)
project(tutorial_msgs)

find_package(catkin REQUIRED COMPONENTS
    message_generation
    std_msgs
)

add_message_files(
    FILES
    CustomMessage.msg
)

generate_messages(
  DEPENDENCIES
  std_msgs
)
```

```
catkin_package(
    CATKIN_DEPENDS
    message_runtime
    std_msgs
)
```

Now moving on to the tutorial_cpp ROS1 package, add these lines to the package.xml manifest:

```
<build_depend>roscpp</build_depend>
<build_depend>tutorial_msgs</build_depend>
<build_export_depend>roscpp</build_export_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>tutorial_msgs</exec_depend>
```

Edit the CMakeLists.txt from the same package such that it looks like this:

```
cmake_minimum_required(VERSION 3.0.2)
project(tutorial_cpp)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  tutorial_msgs)

catkin_package()

add_executable(ros_bridge_1 src/ros_bridge.cpp)
add_dependencies(ros_bridge_1 ${catkin_EXPORTED_TARGETS})
target_link_libraries(ros_bridge_1 ${catkin_LIBRARIES})
```

Let us now configure the package.xml and CMakeLists.txt files for the ROS2 packages. Starting with the tutorial_msgs package, add the following lines to its package.xml:

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Edit its CMakeLists.txt such that it looks like this:

```
cmake_minimum_required(VERSION 3.5)
project(tutorial_msgs)
```

```
  if(NOT CMAKE_C_STANDARD)
    set(CMAKE_C_STANDARD 99)
  endif()

  if(NOT CMAKE_CXX_STANDARD)
    set(CMAKE_CXX_STANDARD 14)
  endif()

  if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID
 MATCHES "Clang")
    add_compile_options(-Wall -Wextra -Wpedantic)
  endif()

  find_package(ament_cmake REQUIRED)
  find_package(rosidl_default_generators REQUIRED)

  set(msg_files
    "msg/RobotData.msg"
    "msg/CustomMessage.msg"
  )

  set(srv_files
    "srv/Activate.srv"
  )

  rosidl_generate_interfaces(${PROJECT_NAME}
    ${msg_files}
    ${srv_files}
  )

  ament_export_dependencies(rosidl_default_runtime)

  if(BUILD_TESTING)
    find_package(ament_lint_auto REQUIRED)
    ament_lint_auto_find_test_dependencies()
  endif()

  ament_package()
```

Finally, let us configure the package.xml and CMakeLists.txt for the tutorial_cpp package. Starting with the package.xml, add the following lines to it:

```
<depend>rclcpp</depend>
```

```
<depend>tutorial_msgs</depend>
```

Edit the CMakeLists.txt such that it looks like this:

```
cmake_minimum_required(VERSION 3.5)
project(tutorial_cpp)

if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID
MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(tutorial_msgs REQUIRED)

add_executable(ros_bridge_2 src/ros_bridge.cpp)
ament_target_dependencies(ros_bridge_2 rclcpp
tutorial_msgs)

install(TARGETS
  ros_bridge_2
  DESTINATION lib/\${PROJECT_NAME})

ament_package()
```

6.3.6 Compilation Workflow

We now present the workflow which must be followed to compile in a way such that ros1_bridge can be used. In order to compile all these packages, you will need to open three different terminals in the ROS2_Tutorial folder. In one terminal, you will compile ROS1 code; in another one, ROS2 code; in the last one, ROS Bridge. Use the following commands on each terminal to compile (the third terminal MUST go last):

Terminal 1 (ROS1):

```
source /opt/ros/<ros1_distro>/setup.bash
cd ros1_ws
catkin_make_isolated --install
```

Terminal 2 (ROS2):

```
source /opt/ros/<ros2_distro>/setup.bash
cd ros2_ws
colcon build
```

Terminal 3 (ROS Bridge)

```
source /opt/ros/<ros1_distro>/setup.bash
source /opt/ros/<ros2_distro>/setup.bash
source ros1_ws/install_isolated/setup.bash
source ros2_ws/install/local_setup.bash
cd bridge_ws
colcon build --packages-select ros1_bridge --cmake-force-
configure
```

To test whether you ran all the commands successfully, run the following commands on the third terminal. You should see the association between the two message interfaces in the ROS1 and ROS2 packages listed:

```
source install/local_setup.bash
ros2 run ros1_bridge dynamic_bridge --print-pairs | grep
tutorial_msgs
```

6.3.7 Running Workflow

Keep open the three terminals you had opened before. Open up a new one, in which you will run the ROS1 master as follows:

```
source /opt/ros/<ros1_distro>/setup.bash
roscore
```

On the other three, enter the following commands (go terminal by terminal): Terminal 1 (ROS1):

```
source install_isolated/setup.bash
rosrun tutorial_cpp ros_bridge_1
```

Terminal 2 (ROS2):

```
source install/local_setup.bash
ros2 run tutorial_cpp ros_bridge_2
```

Terminal 3 (ROS Bridge):

```
ros2 run ros1_bridge dynamic_bridge --bridge-all-topics
```

You will see immediately after running the last command that the ROS2 node is echoing the commands being sent to it by the ROS1 node. You have successfully used the ros1_bridge package with custom messages, the next time you need to connect two nodes in different ROS versions you will be able to do so by following the exact same workflow laid out in these last two steps.

6.4 Bridging a Package Installed Through Binaries

It might happen that the package you want to use along with ROS2 is not open-source. If that is the case, the workflow is slightly different from the one explained before. In order to create a copy of the custom msg and srv files of the package you will need to inspect the interface definitions via the terminal.

Let us say you want to bridge the turtlesim package. In ROS1, you can inspect which custom messages and services turtlesim has by using the following two commands:

```
rosmsg package turtlesim
rossrv package turtlesim
```

In ROS2, you can do it using a single command (which will also show the ROS2 action definitions), as follows:

```
ros2 interface package turtlesim
```

Once you have the names of the messages and services, you can inspect the interface of a specific message/service. Let us assume you'd like to know the mes-

sage and service definitions for turtlesim/Color and turtlesim/Spawn in ROS1. In order to do so, you can use the following two commands:

```
rosmsg show turtlesim/Color
rossrv show turtlesim/Spawn
```

To find the definition for the above message and service in ROS2, you can use a single command too, as follows:

```
ros2 interface show turtlesim/msg/Color
```

If you do this with all messages, you can create a package named turtlesim either in ROS1 or ROS2, in which you will copy these definitions to the corresponding msg and srv folders. Once you do so, the steps to compile and run are almost identical to the ones we followed before. Pay attention to the package.xml and CMakeLists.txt, as those files might differ from the ones here. Follow the tutorials about custom messages and services in both ROS1 and ROS2 to configure those files, for ROS2 you can find those in chapter 2.
The process now involves only two terminals, one for the ROS version in which you don't have the package installed, and one for the ROS bridge. The commands used in the corresponding two terminals before are the ones that should be used in this case as well.

Chapter 7

Real Time

7.1 Introduction to Real Time Computing

Real-time computing is a key feature of many robotics systems, particularly safety and mission-critical applications. ROS2 has been designed with real-time performance constraints, since this is a requirement that was not considered in the early stages of ROS1. The idea is to consider ROS2 as a real-time friendly environment.

7.2 Real Time Computing in ROS2

7.2.1 Jitter

To make a real-time computer system, our real-time loop must update periodically to meet its deadlines. ROS2 can only tolerate a small margin of error on these deadlines. The time between the deadline of a task and its completion is called jitter. ROS2 defines a maximum allowable jitter. The completion of a task over the maximum allowable jitter is considered a severe mistake.

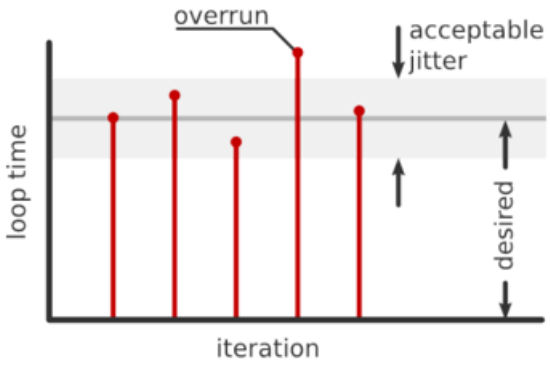


Figure 7.1: Graphical representation of the jitter

7.2.2 Overrun

Missing a deadline is considered a system failure. In some systems, it may lead to loss of life or financial damage. To avoid these, we must avoid non-deterministic operations in the execution path, things like: Blocking synchronization primitives Memory allocation and management Network access, especially TCP/IP Etc... Also, to make sure ROS2 doesn't miss any deadline, when performing Real-Time operations, it upgrades the threads to high priority threads.

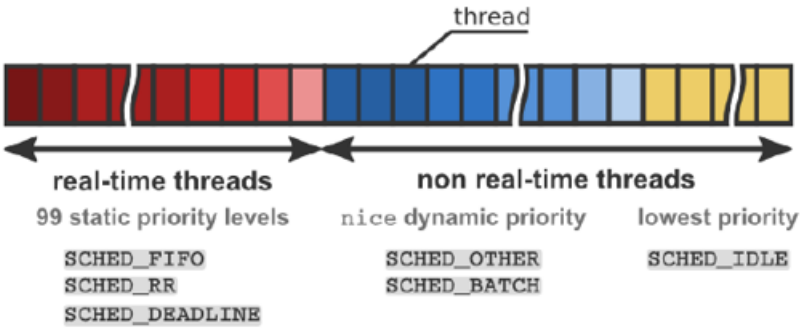


Figure 7.2: Real-Time threads are considered high priority threads in ROS2

7.3 Example: Task Priority Test

We will run a task that makes periodic loops using ROS2 and tries to keep up with the deadlines.

We will compare the results with the same task run if we manually modify its priority with a higher one and also allocate a permanent memory before running it to make it even faster.

First we will download and install the pendulum package from our repository:

```
$   cd ~
$   git clone https://github.com/FilipHesse/ROS2_Tutorial.git
$   cd ~/ROS2_Tutorial/ros2_ws
$   colcon build --symlink-install
$   cd ~/ROS2_Tutorial/ros2_ws/src/scripts/REAL-TIME_TEST
```

We will open 3 terminals and run the following commands:

Terminal 1:

```
$   bash 1.1.sh
        source ~/ROS2_Tutorial/ros2_ws/install/local_setup.
         bash
        ros2 run pendulum_demo pendulum_demo  --pub-stats
```

Terminal 2:

```
$   bash 2.sh
        source ~/ROS2_Tutorial/ros2_ws/install/local_setup.
         bash
        ros2 run pendulum_manager pendulum_manager
```

We will type a 0 to let the manager know he can activate the Real-Time nodes.

Terminal 3:

```
$   bash 3.sh
        source ~/ROS2_Tutorial/ros2_ws/install/local_setup.
         bash
        ros2 topic echo /controller_statistics
```

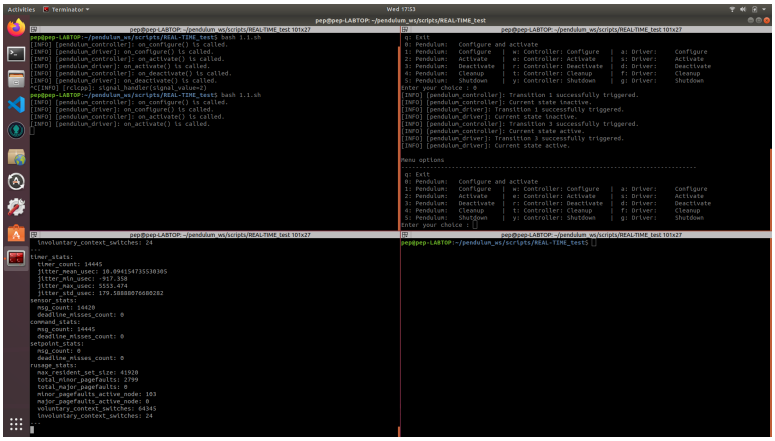


Figure 7.3: Run program with normal priority

As we can see, there are no "major pagefaults" what means that no deadline was missed, even though the mean timings between executions might seem high. Now we will send a 3 to the manager to stop the nodes, and we will signal the Terminal 1 to stop it. Next we will run the following code on the Terminal 1 and we will send a 0 to the manager to activate again the main program:

```
$   bash 1.2.sh
        source ~/ROS2_Tutorial/ros2_ws/install/local_setup.
         bash
        ros2 run pendulum_demo pendulum_demo --priority 90
         --pub-stats --lock-memory-size 1000
```

As we can see, there are also no "major pagefaults" but the execution is extremely fast. The negative numbers mean that the execution of a loop, was under the desired time, which means a very good performance.

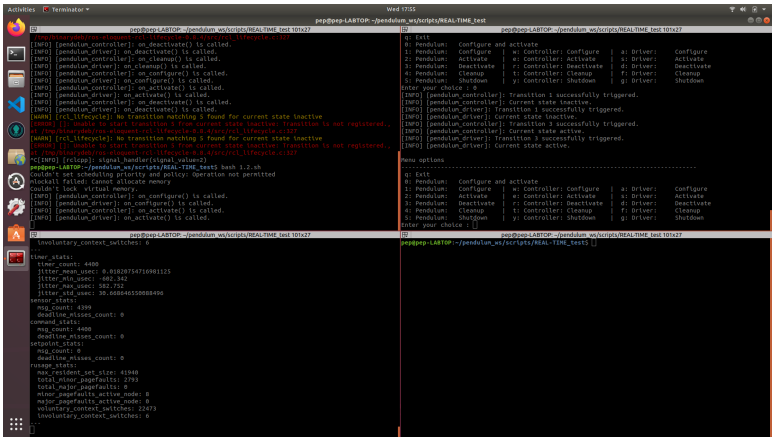


Figure 7.4: Run program with higher priority and pre-allocated memory

We can conclude, that if we establish manually high priorities for given tasks, we can make them go even faster than desired. But when we run a Real-Time application, by default ROS2 will make it respect the desired time constrains.

7.4 Example: Inverted Pendulum Control System

A classic example of a control problem commonly solved by real-time computing is balancing an inverted pendulum. If the controller is blocked for an unexpectedly long time, the pendulum would fall down or become unstable. But if the controller reliably updates at a rate faster than the motor controlling the pendulum can operate, the pendulum will successfully react to sensor data and balance the pendulum.

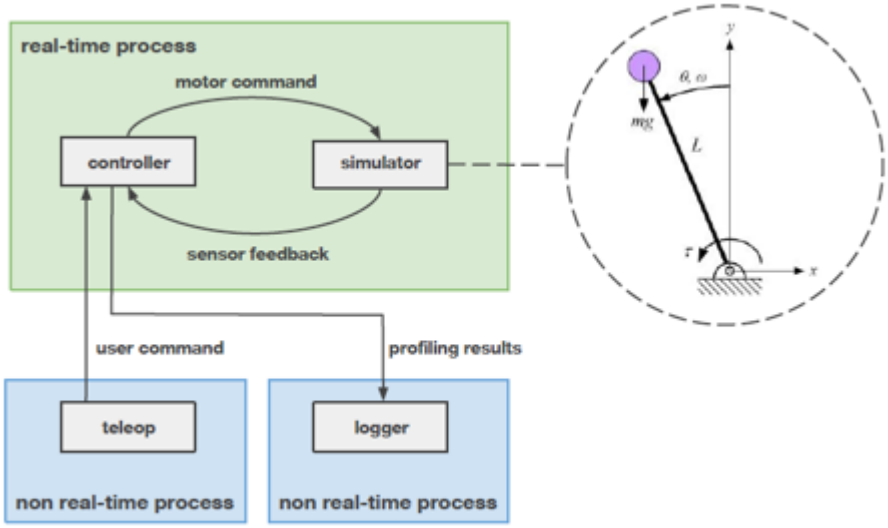


Figure 7.5: Real-time and non Real-Time parts of the program

The program is split into Real-Time nodes and non Real-Time nodes. Real-Time nodes have high priority and have a high impact on the processor, so we will use the "manager" to activate them at the start of the simulation and deactivate them when we finish.

First we will download and install the pendulum package from our repository:

```
$   cd ~
$   git clone https://github.com/FilipHesse/ROS2_Tutorial.git
$   cd ~/ROS2_Tutorial/ros2_ws
$   colcon build --symlink-install
$   cd ~/ROS2_Tutorial/ros2_ws/src/scripts/PENDULUM_RVIZ
```

We will open 3 terminals and run the following commands:

Terminal 1:

```
$   bash 1.sh
        source ~/ROS2_Tutorial/ros2_ws/install/local_setup.
         bash
        ros2 launch pendulum_bringup pendulum_bringup.launch.
         py
```

Terminal 2:

```
$   bash 2.sh
        source ~/ROS2_Tutorial/ros2_ws/install/local_setup.
         bash
        ros2 run pendulum_manager pendulum_manager
```

We will type a 0 to let the manager know he can activate the Real-Time nodes.

Terminal 3:

```
$   bash 3.sh
        source ~/ROS2_Tutorial/ros2_ws/install/local_setup.
         bash
        export LC_NUMERIC="en_US.UTF-8"
        ros2 run rviz2 rviz2 -d `ros2 pkg prefix
         pendulum_bringup --share`/config/pendulum.rviz
```

RVIZ will open and we will be able to see a simulation of a car holding an inverted pendulum.

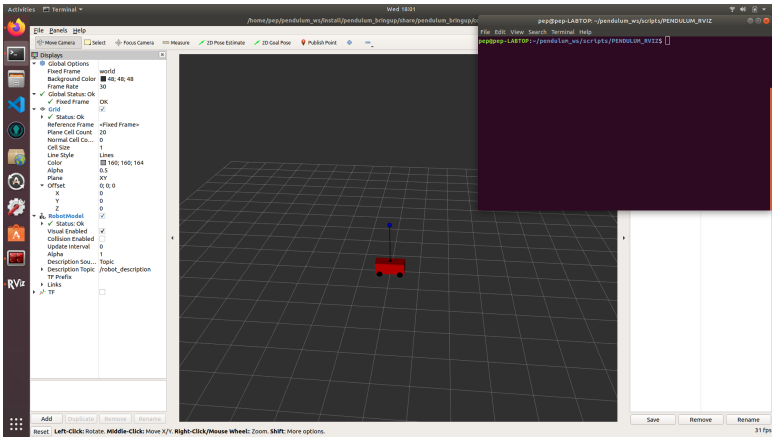
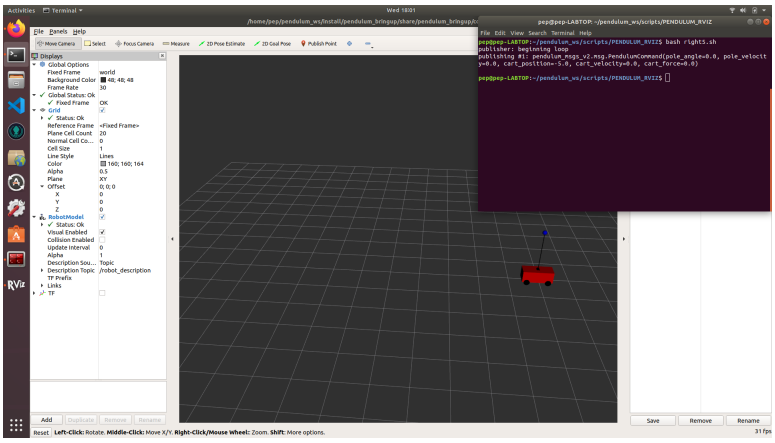


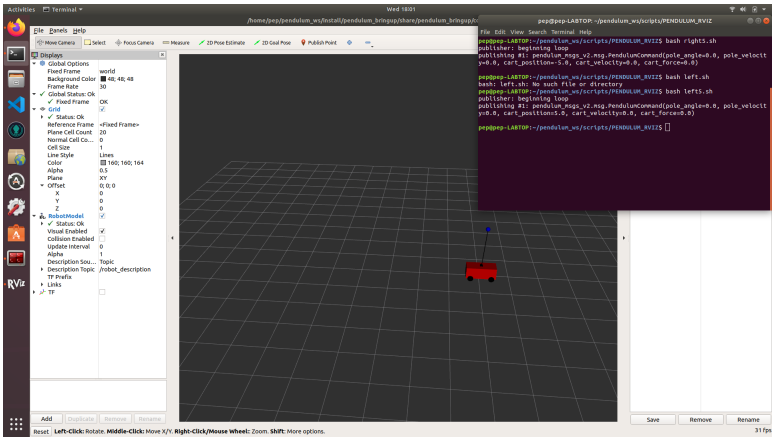
Figure 7.6: RVIZ simulation of the inverted pendulum

Using some commands like these we will be able to make the car move:

```
$   bash right5.sh
        source ~/ROS2_Tutorial/ros2_ws/install/local_setup.
         bash
        ros2 topic pub -1 /pendulum_setpoint pendulum_msgs_v2/
         msg/PendulumCommand "cart_position: -5.0"
```

```
$   bash left5.sh
        source ~/ROS2_Tutorial/ros2_ws/install/local_setup.
         bash
        ros2 topic pub -1 /pendulum_setpoint pendulum_msgs_v2/
         msg/PendulumCommand "cart_position: 5.0"
```





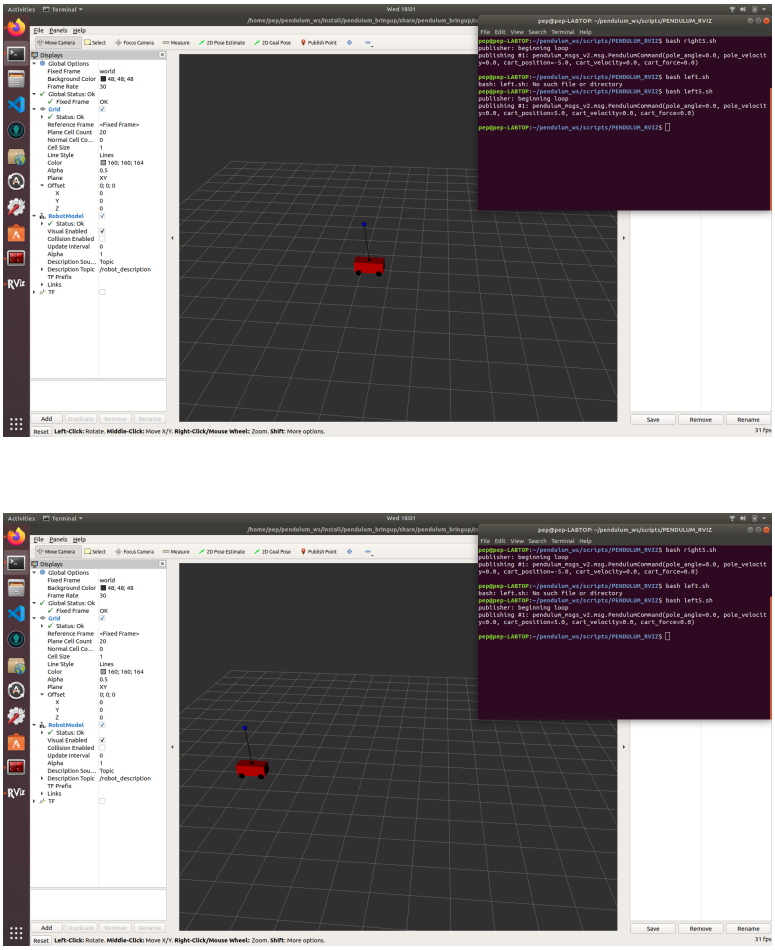


Figure 7.7: Movement examples

As said before, a system like this would not be able to keep the pendulum straight if the Real-Time constrains weren't met.

Chapter 8

SROS

8.1 Introduction to SROS

SROS is a software that provides the tools to use ROS2 on top of DDS Security. Data Distribution Service (DDS) is an Object Management Group for real-time systems used for systems that require real-time data exchange.
DDS-Security enables out-of-the box security and interoperability between compliant DDS applications. A user is able to customize the behavior of the DDS implementations through specifying different technologies for authentication.
There are different DDS providers, some of them free, some of them require a license, and some of them are open source. We will use "RTI Connext DDS" which has full compatibility with SROS and is recommended by the ROS2 developers.

8.2 SROS using DDS

DDS can perform 3 main actions in SROS: Authentication, Access Control and Cryptographic.

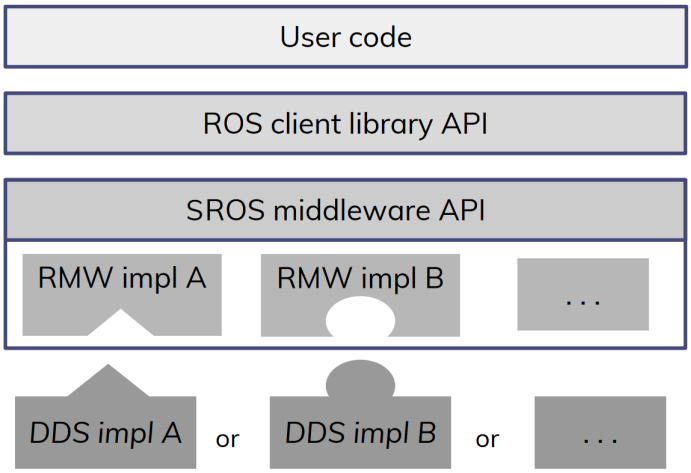


Figure 8.1: Architecture of SROS

8.2.1 Authentication

The Authentication plugin provides the concept of confirming the identity of nodes (e.g. it would be awfully hard to make sure a given ROS node could only access specific topics if it was impossible to securely determine which node it was).

8.2.2 Access Control

The Access Control plugin deals with defining and enforcing restrictions on the DDS-related capabilities of a given domain participant. It limits a particular participant to a specific DDS domain, or only allows the participant to read from or write to specific DDS topics.

8.2.3 Cryptographic

The Cryptographic plugin is where all the cryptography-related operations are handled: encryption, decryption, signing, etc. Both the Authentication and Access control plugins utilize this plugin to verify signatures. This is where the

functionality to encrypt DDS topic communication resides.

8.3 SROS Utilities

SROS 2 utilities are the following ones:

- Creation of keys for the Authentication and Access control plugins.
- Creation of a directory tree which contains all security files (keypairs, governance and permission files, etc.)
- Creation of a new identity for a given node instance, generating a keypair (it's signing it's x.509 certificate)
- The support specified permissions in familiar ROS terms are automatically converted into low-level DDS permissions.
- The support also automatically discovers the required permissions from a running ROS system.

8.4 SROS Main Commands

```
create_key          Create key
create_keystore     Create keystore
create_permission   Create permission
distribute_key      Distribute key
generate_artifacts  Generate keys and permission files from
 a list of identities and policy files
generate_policy     Generate XML policy file from ROS graph
 data
list_keys           List keys
```

We have to remember that we group keys in stores. All nodes sharing a different key of the same store, can access to the same topics and interact with each other.

8.5 Example: Run Nodes using a Key

Here we will show how our system behaves while running a simple publisher subscriber with, and without using a key. First we will download the folder with scripts from our repository:

```
$   cd ~
$   git clone https://github.com/FilipHesse/ROS2_Tutorial.git
$   cd ~/ROS2_Tutorial/ros2_ws/src/scripts/SECURITY
```

Then we will run in two separate terminals the following commands:

```
$   bash 1.sh
        cd ~
        mkdir -p store
        export ROS_SECURITY_ROOT_DIRECTORY=~/store
        export ROS_SECURITY_ENABLE=true
        export ROS_SECURITY_STRATEGY=Enforce
        export ROS_DOMAIN_ID=10
        ros2 security create_keystore store
        ros2 security create_key store /talker
        ros2 run demo_nodes_cpp talker
```

```
$   bash 2.sh
        cd ~
        mkdir -p store
        export ROS_SECURITY_ROOT_DIRECTORY=~/store
        export ROS_SECURITY_ENABLE=true
        export ROS_SECURITY_STRATEGY=Enforce
        export ROS_DOMAIN_ID=10
        ros2 security create_keystore store
        ros2 security create_key store /listener
        ros2 run demo_nodes_cpp listener
```

As we can see, they communicate with each other without any problem.

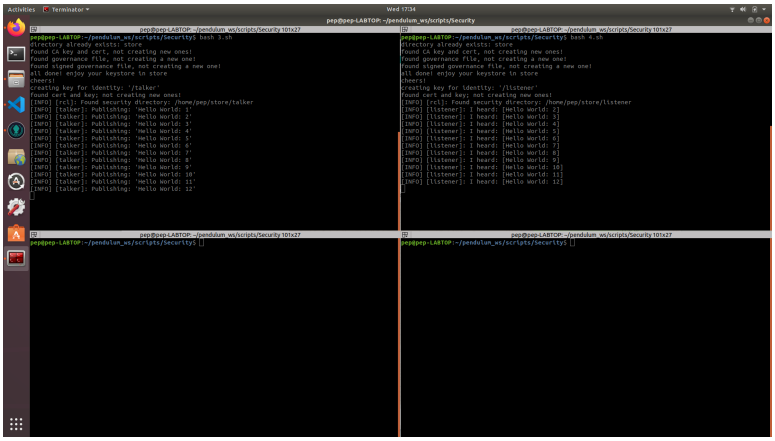


Figure 8.2: Key talker and listener

If we run the following command in another shell we will see that the topic they are using to communicate can't be seen:

```
ros2 topic list
```

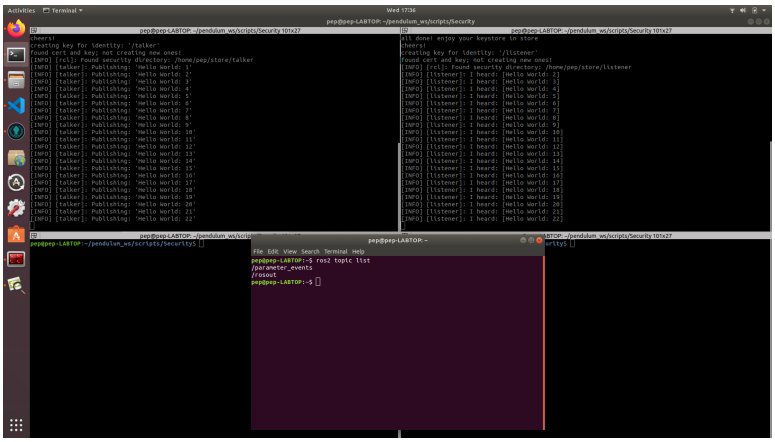


Figure 8.3: We can't see the used topic in the topic list

Now, instead, we run into two new terminals the following codes:

```
ros2 run demo_nodes_cpp talker
```

```
ros2 run demo_nodes_cpp listener
```

And we will see that now the topic does appear in the shell:

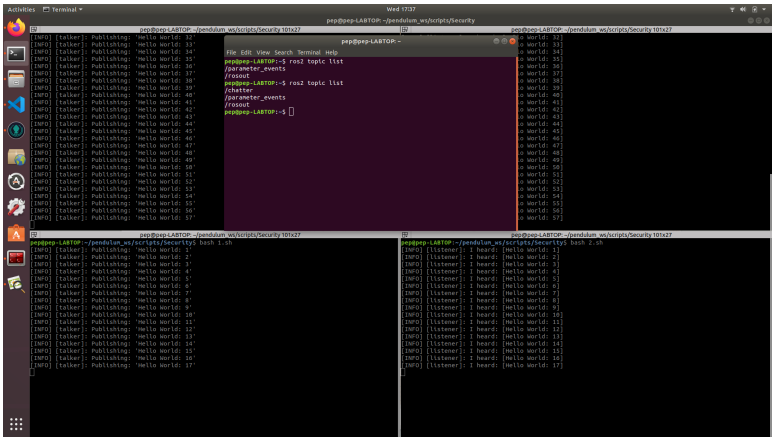


Figure 8.4: List topics being run without key

And using the next command we can see the messages inside the topic:

```
ros2 run topic echo /chatter
```

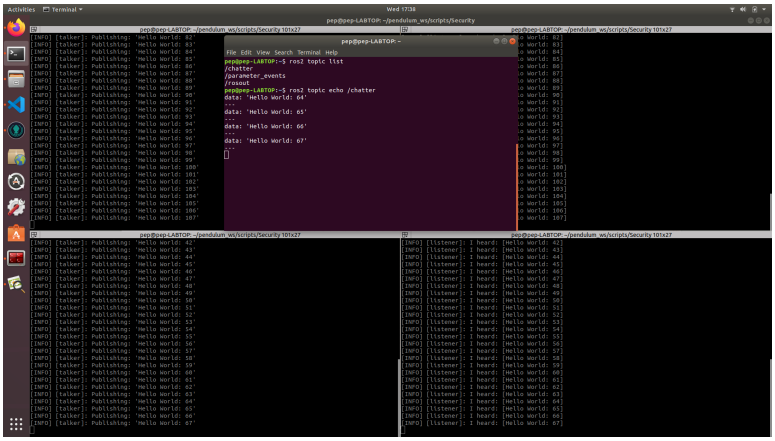


Figure 8.5: Messages sent in the chatter topic

We can conclude from this experiment that topics being used by a node run with a key can't be read or written by any other instance that doesn't have the key. Also, topics run with a key and without it, don't interfere with each other.

Chapter 9

Conclusions

In conclusion, ROS2 is a new and dynamic middleware for creating and developing robotic systems. It incorporates the best parts of ROS1 and also integrates a lot of new features to address some of its shortcomings. It is an incredibly dynamic system that is changing very rapidly, which means problems experienced today will be fixed in a relatively quick time.

References

[1] ROS2 Community, *"Writing a simple publisher and subscriber"*. https://index.ros.org/doc/ros2/Tutorials/Writing-A-Simple-Py-Publisher-And-Subscriber/. 28

[2] ""ros2_tutorial repository"." https://github.com/ros2/sros2/blob/master/SROS2_Linux.md. 28

[3] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *The 13th International Conference*, pp. 1–10, 10 2016.

[4] ROS Index, *"Real-time programming in ROS 2"*. https://index.ros.org/doc/ros2/Tutorials/Real-Time-Programming/.

[5] L. U. S. Juan, *"Inverted pendulum demo"*. https://design.ros2.org/articles/ros2_dds_security.html.

[6] ROS 2 Design, *"ROS 2 DDS-Security integration"*. https://design.ros2.org/articles/ros2_dds_security.html.

[7] M. Arguedas, *"Try SROS2 in Linux"*. https://github.com/ros2/sros2/blob/master/SROS2_Linux.md.

[8] B. Gerkey, *"Why ROS2?"*. https://design.ros2.org/articles/why_ros2.html.

[9] ROS 2 Design, *"About different ROS 2 DDS/RTPS vendors"*. https://index.ros.org/doc/ros2/Concepts/DDS-and-ROS-middleware-implementations/#supported-rmw-implementations.

[10] D. Thomas, *"ROS 2 middleware Interface"*. https://design.ros2.org/articles/ros_middleware_interface.html.