# POLITECNICO DI TORINO

**Internship report for bachelor's Degree in Electronics Engineering**

# Development of a remote control application via the internet for a ROS-based robot


Supervisors

Prof. Stefano Alberto MALAN

Tutor: Ilaria BOSI

Candidate

Antonino CACICIA


**OCTOBER 2020**

# Summary

Working with robotics has been my main professional goal since I was a kid, which led me to undertake this educational path in Politecnico di Torino. Before March, the starting date, I had two personal projects developed in adolescence that I am very proud of; one of them was an Arduino-based wheeled obstacle avoidance, the other is a semi-humanoid robot, with arms, head and little tires for locomotion, controlled via Bluetooth. Despite having requested an IoT-oriented internship, after a brief "mail-interview" about me, I was given the chance to work with robotics.

The intent of this report is to expose some state of the art, methods and skills acquired during my internship at LINKS Foundation, which led to the implementation of an application for remote control of a ROS-based robot.

The entire internship path was centered on ROS, the Robot Operative System, an high-level paradigm for robotics application, but for desired purposes, I learned and reinforced many fields of information technology such as DDS (Data Distribution Service), MQTT, ASGI Server (Uvicorn), Qt5 and OpenCV libraries and more. In the first part of the internship I focused mainly on the study of the latest various technologies and protocols I used, after which the focus shifted to implementing and testing the different software modules (mostly written in Python). My previous good skills with GNU/Linux operative system helped me a lot with developing and testing.

First of all, I learned how to deal with ROS(2), studying how it works, how to create and develop packages, also using its default simulator, Gazebo; then I started working on a ROS lane detecting software, using OpenCV.
The first objective of this internship has been to find a way to develop Python $ROS2_{dashing}$ applications without the ROS 2 ecosystem itself. It's possible to make and run "external" support-ros-programs only if it's possible to communicate with "topics"; Advanced Image Processing and data visualization are good examples of this operative way. To make this possible, the $ROS2_{dashing}$' s DDS default middleware ($FastRTPS$) would be necessarily binded from C++ to Python.

Consequentially the problems with Python bindings, the choice fell on a RTI Connext solution, who natively implement a method to use its C++ based middleware with pure Python, using its **rti_connector** module. This is possible thanks to the interoperability among FastRTPS and RTIConnext, because it adapts to the *OMG* standard. For this reason, the final project it's meanted to provide a demonstration of the potentialities of this choice.

The report begins with a general overview of the communication protocols, developing tools and libraries used for this project, followed by an introduction to the implemented remote control application and finally some snippets of code will be extrapolated and described in order to better understand the deployment of the implemented scenario.

# Table of Contents

# Acronyms

**OMG**
: Object Management Group

**OSFR**
: Open Source Robotics Foundation

**ROS**
: Robot Operating System

**SBC**
: Single Board Computer

**DDS**
: Data Distribution Service

**GUI**
: Graphical User Interface

**CLI**
: Command Line Interface

**rmw**
: ROS middleware, DDS

**IR**
: Infra Red (sensor)

**LIDAR**
: Laser Imaging Detection and Ranging

**SVG**

Scalable Vector Graphics

**QoS**

Quality of Service

**MQTT**

Message Queue Telemetry Transport

**TLS**

Transport Layer Security

**SSL**

Secure Sockets Layer

**ASGI**

Asynchronous Server Gateway Interface

**WSGI**

Web Server Gateway Interface

**POSIX**

Unix-like operating systems, standard IEEE 1003

**SAR**

Search And Rescue

# Chapter 1

# Adopted Technologies

In this chapter will be exposed the state of the art and the premises useful to well understand the next sections

## 1.1 ROS, the Robot Operating System

Robot Operating System (ROS)[1] is a collection of software frameworks for robot software development. ROS is not a real operating system, (needing GNU/Linux Ubuntu as officially supported OS) but provides services such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management.

Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, post and multiplex sensor data, control, state, planning, actuator, and other messages. Despite the importance of reactivity and low latency in robot control, ROS itself is not a real-time OS (RTOS). It is possible, however, to integrate ROS with real-time code. The lack of support for real-time systems has been addressed in the creation of ROS 2.0 (the chosen one for our purposes), a major revision of the ROS API which will take advantage of modern libraries and technologies for core ROS functionality and add support for real-time code and embedded hardware.

All the ROS softwares, tools and the main client libraries (C++ and Python) are released under the terms of the BSD[2] or Apache[3] license, very interesting for both commercial and research use.

Software in the ROS Ecosystem can be separated into three groups:

- ROS client library implementations such as rclcpp for C++ and rclpy for Python, but other languages are supported by the community;
- tools used for building ROS-based software and software for debugging and simulation ;
- packages containing application-related code which uses one or more ROS client libraries.

### 1.1.1 ROS libraries

As mentioned before, both Python and C++ is supported by default on ROS. OSRF provides the libraries rclcpp for C++ and rclpy for Python who implements all the necessary abstraction, from Nodes to functions for interface with DDS. All ROS-compatible software must be organised in packages with a specific organization, essentially an xml file named `package.xml` who contain all the package informations (author's infos, version, dependencies), a `CMakeLists.txt` / `setup.py` for building and the source code itself. The ROS documentation provides an exhaustive explanation for the package organization[4].

### 1.1.2 ROS tools

The tools provided by ROS are shown below (referred to `ros-foxy-desktop` package)

#### CLI Tools

Here the CLI tools; for other info add -h to the desidered one

| | |
|---|---|
| ros2 launch | Launching multiple programs |
| ros2 run | Run a single program |
| ros2 topic | Publish/Subscribe on/to topics |

| | |
|---|---|
| ros2 node | Nodes utility |
| ros2 service | Services utility |
| ros2 pkg | Package utility |

#### GUI Tools

- RVIZ, a 3D visualization environment [5].
- rqt, a multiple tool for reading/sending various data, node visualization and more [6].
- Gazebo [7], the official ROS robot simulator, which deserves a separate discussion.

### 1.1.3 ROS Packages

In addition to made-from-scratch packages, it's also possible to find more online; functionality and applications such as hardware drivers, robot models, datatypes, planning, perception, simultaneous localization and mapping, simulation tools, and other algorithms could be available on wiki.ros.org or on github.com.
Every ROS package (based on Python or C++) must be first builded with colcon; before running it is required to execute the installation script.
In root package folder type:

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

## 1.2 Gazebo, the simulator

A fundamental component of the ROS complete ecosystem is its great simulator, Gazebo [8]. With Gazebo, in addition to a working 3D robot model, it's possible to test and debug all the in-developing packages. LIDARs, cameras, IR sensors,

motors could be emulated on this virtual environment.
Inputs and Outputs, each one with its type, are subscribed/published into topics, managed by `rmw_<dds_vendor>`.

Not only a raw simulator, Gazebo provides a good tool for building scenarios out of an house plan, Building Editor, down to the Edit section of the drop-down menu of the window.

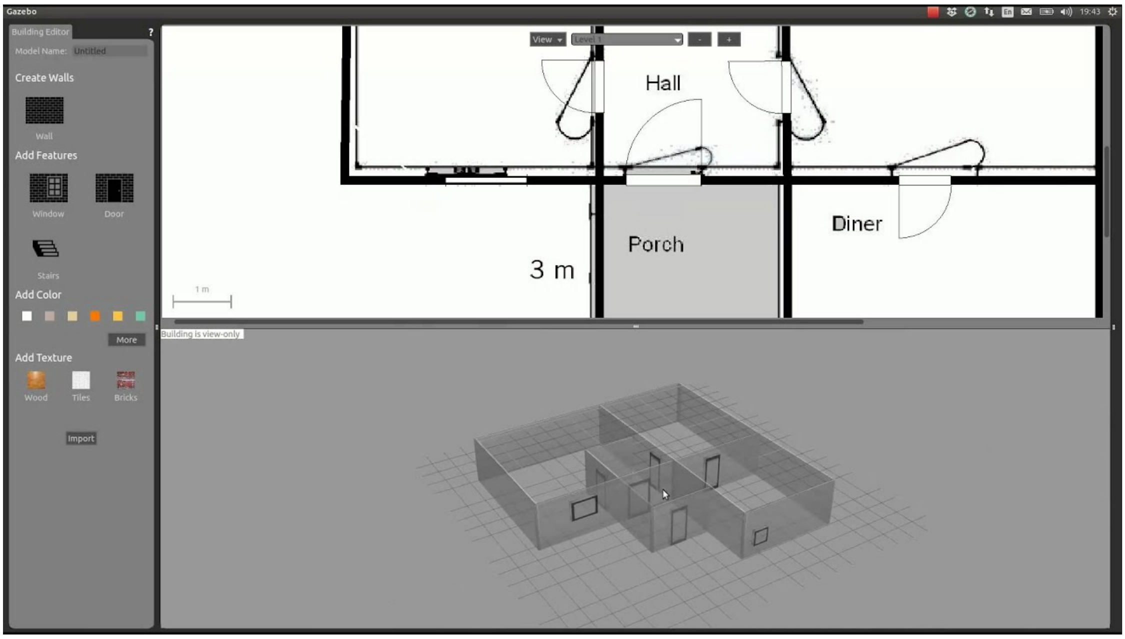


**Figure 1.1:** Building Editor on Gazebo

It is also possible to import more complex objects, that could be needed for any reason; Model Editor is the subprogram needed for this purpose:

- SVG files are supported (Inkscape, CAD 2D) to be extruded.

- Spheres, Cubes and Cylinders could be added in-program by default.
- 3D files with `.dae` `.stl` `.obj` format could be imported (Fusion360, Blender, FreeCAD)

A 3D object imported as a polygonal mesh needs some adjustments. Various parameter could be set as density, mass, color, physics parameters and more. With a simple manipulation, also textures could be added.

Various robot models are already available by default (also some NASA Robots). During the all internship Turtlebot3 is the chosen one.

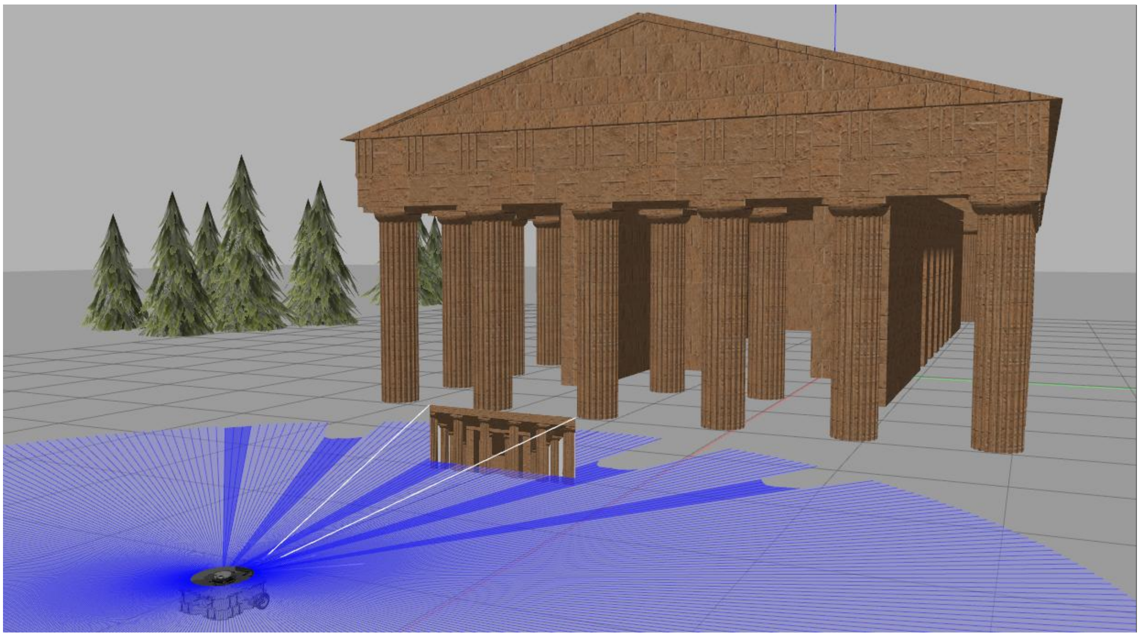

**Figure 1.2:** Turtlebot3 simulation with custom 3D building (Tempio della Concordia, Agrigento)

## 1.3 TurtleBot

TurtleBot [9] is a low-cost, personal robot kit with open source software. TurtleBot was created at Willow Garage (the same company that initially supported OpenCV) in November 2010. The TurtleBot kit consists of a mobile base, 3D

Sensor, computer system and the TurtleBot mounting hardware kit. TurtleBot is designed to be easy to buy, build, and assemble, using off the shelf consumer products and parts that easily can be created from standard materials. As an entry level mobile robotics platform, TurtleBot has many of the same capabilities of the company's larger robotics platforms. With TurtleBot, users can drive around and map their environment, see in 3D, and have enough power to create their own applications.

The chosen version of this series was TurtleBot3 WafflePi, which uses a Raspberry Pi 3 Model B as SBC.

| SBC | Sensors | Embedded Board | Actuators |
|---|---|---|---|
| Raspberry Pi 3<br>(Model B/B+) | LIDAR LDS-01<br><br>Raspberry Pi Camera<br>(IMX219PQH5) | OpenCR1.0<br>(STM32 based) | Dynamixel XM430 |

ROBOTIS provides a good model for this robot, well used in Gazebo environment during all the whole internship.

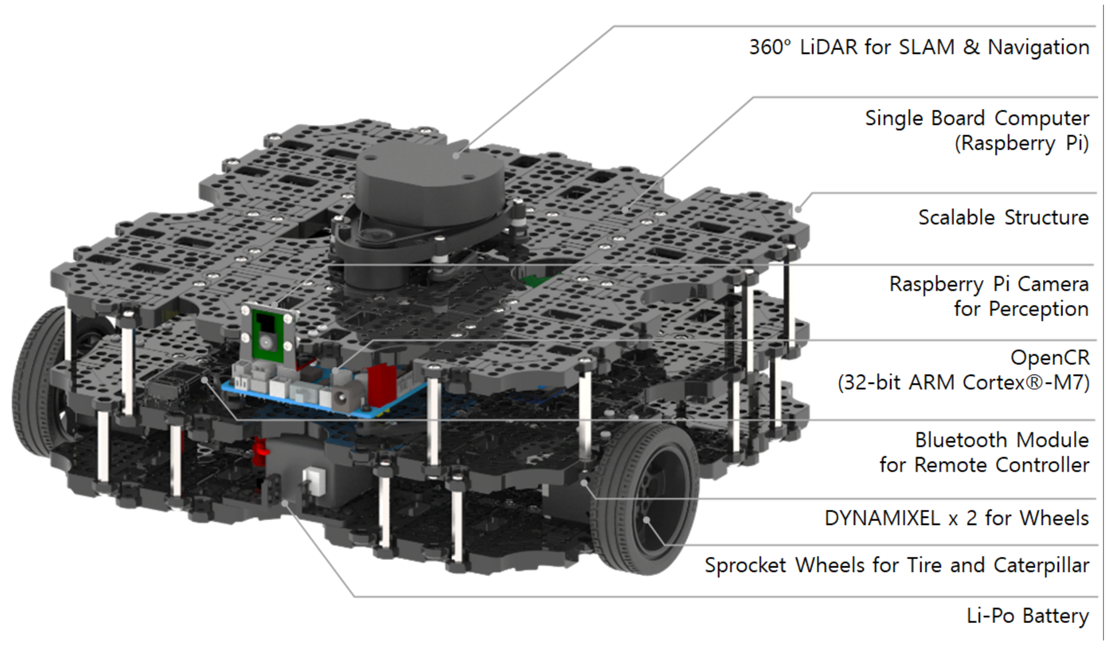


**Figure 1.3:** The TurtleBot3 Waffle Pi robot

## 1.4 DDS, Middleware

Data Distribution Service (DDS) for real-time systems is a middleware standard that aims to enable dependable, high-performance, interoperable, real-time, scalable data exchanges using a publish–subscribe pattern.

DDS addresses the needs of applications like robotics, aerospace and defense, air-traffic control, autonomous vehicles, medical devices, simulation and testing, transportation systems, and other applications that require real-time data exchange.

DDS is a networking middleware that simplifies complex network programming. It implements a publish–subscribe pattern for sending and receiving data, events, and commands among the nodes. Nodes that produce information (publishers) create "topics" (e.g., temperature, location, pressure) and publish "samples". DDS delivers the samples to subscribers that declare an interest in that topic. Any node can be a publisher, subscriber, or both simultaneously.
Various std types of data are supported:

- char, wchar
- short, unsigned short
- long, unsigned long
- float
- double, long double
- boolean
- string
- enum

From these std types it is possible to derive also more complex structures, basically declaring it in .idl files; Down below some examples of composted DDS .idl$_s$ used in ROS:

- Vector3, out of 3 double
- Twist, out of 2 Vector3
- Image, out of $float_s$ and other derived types
- LaserScan
- Imu

...

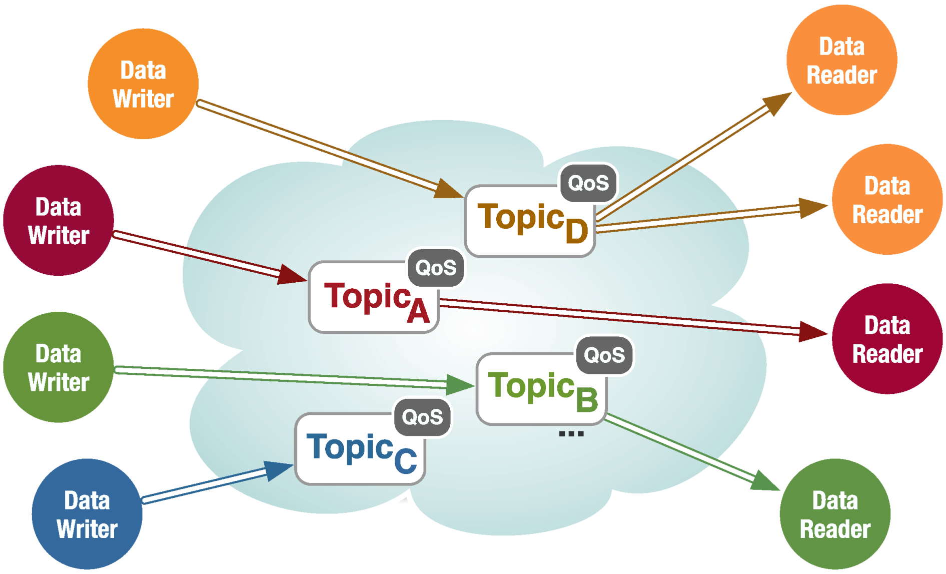


**Figure 1.4:** DDS scheme visualization

DDS allows the user to specify quality of service (QoS) parameters to configure discovery and behavior mechanisms up-front. By exchanging messages anonymously, DDS simplifies distributed applications and encourages modular, well-structured programs. DDS also automatically handles hot-swapping redundant publishers if the primary fails. Subscribers always get the sample with the highest priority whose data is still valid (that is, whose publisher-specified validity period has not expired). It automatically switches back to the primary when it recovers, too.

As mentioned before, OSRFoundation chosed FastRTPS as default for ROS 2. It is possible to change the default DDS middleware of ROS as shown in documentation [10].

## 1.5 MQTT

MQTT (Message Queuing Telemetry Transport) is a standard (ISO) for lightweight, publish/subscribe network protocol that transports messages between devices. This protocol usually runs over TCP/IP. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited. For this peculiarity it very used in many IoT applications, from Amazon Dash, Facebook Messenger to smart factories.

The MQTT protocol defines two types of network entities: a message broker (server) and a number of clients. An MQTT broker receives all messages from the clients and then routes the messages to the appropriate destination. An MQTT client is any device (from a micro controller like ESP32 up to a full-fledged server) that runs an MQTT library and connects to an MQTT broker over a network. For broker purposes **Mosquitto** [11] was good. As Python library is used `paho-mqtt` [12].

Information is organized in a hierarchy of topics. When a publisher has a new item of data to distribute, it sends a control message with the data to the connected broker. The broker then distributes the information to any clients that have subscribed to that topic. The publisher does not need to have any data on the number or locations of subscribers, and subscribers, in turn, do not have to be configured with any data about the publishers.

If a broker receives a message on a topic for which there are no current subscribers, the broker discards the message unless the publisher of the message designated the message as a retained message. A retained message is a normal MQTT message with the retained flag set to true. The broker stores the last retained message and the corresponding QoS for the selected topic. Each client that subscribes to a topic pattern that matches the topic of the retained message receives the retained message immediately after they subscribe. The broker stores only one retained message per topic. This allows new subscribers to a topic to receive the **most current value** rather than waiting for the next update from a publisher.

When a publishing client first connects to the broker, it can set up a default message to be sent to subscribers.
A minimal MQTT control message can be as little as two bytes of data. A control message can carry nearly 256 MB of data if needed; as first implementation for a component of the software project, for educational purposes, I tried to carry a string encoded Image from simulation, with very poor results... So I choose another

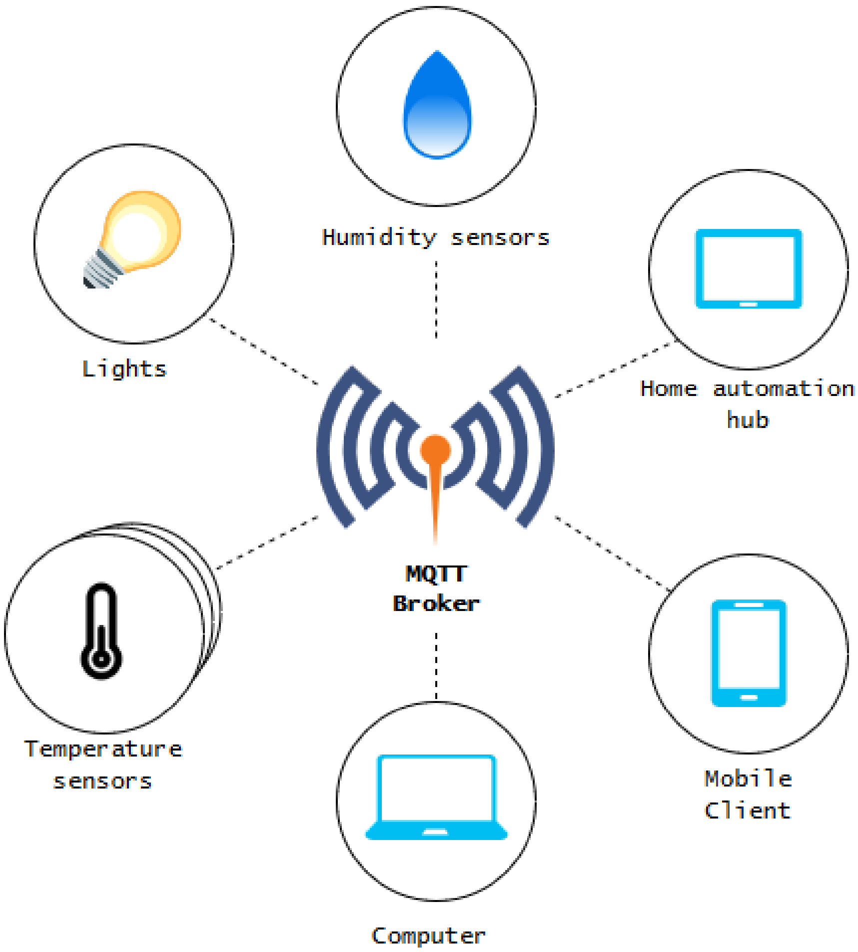


**Figure 1.5:** MQTT protocol architecture scheme

technology for it.

MQTT sends connection credentials in plain text format and does not include any measures for security or authentication. This can be provided by using TLS/SSL to encrypt and protect the transferred information [13]. By default the unencrypted MQTT port is 1883. The encrypted port is 8883.

There are also implementations for non TCP based networks as Bluetooth or ZigBee (MQTT-SN).

## 1.6 ASGI, Uvicorn

During the developing of the project, as mentioned before, it was necessary to find a way to broadcast a video stream; For this purpose I used the web framework Flask (WSGI), but I obtained poor results for my desidered standard, with a medium latency of 3 5 seconds.
I needed something more performing, so ASGI[14] was the best choice.
ASGI (Asynchronous Server Gateway Interface) is a improvement of WSGI, intended to evolve it providing a standard interface between async-capable Python web servers, frameworks, and applications. Where WSGI provided a standard for synchronous Python apps, ASGI provides one for both asynchronous and synchronous apps, sensibly improving performances.
Uvicorn[15] was chosen as server implementation, while Starlette[16] as toolkit.

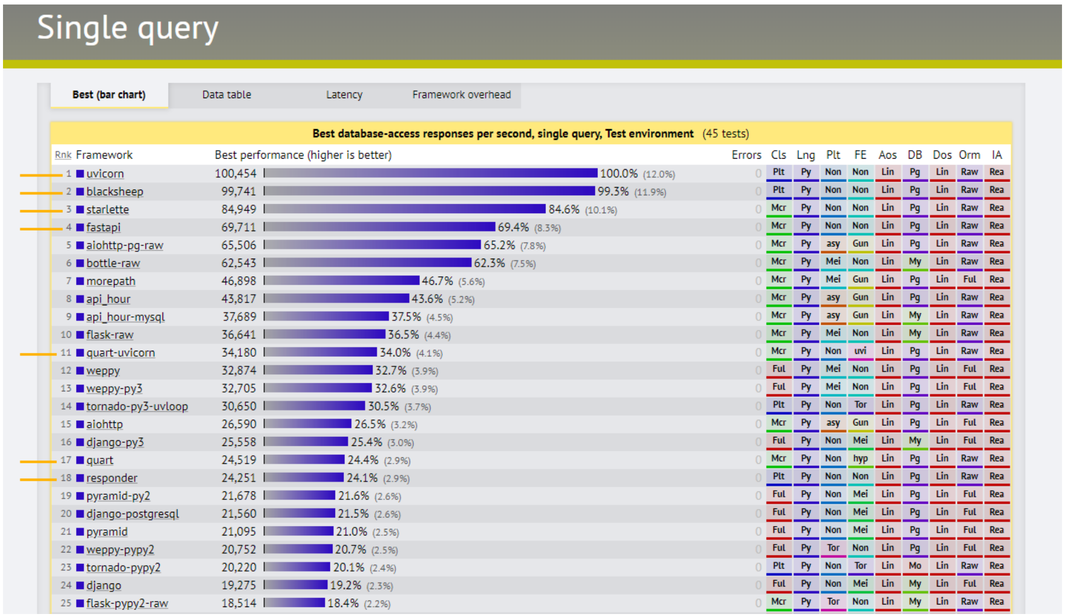


**Figure 1.6:** Comparison of various Python Web Frameworks (ASGI in yellow)

## 1.7 Qt Libraries

Qt [17] is a free and open-source widget toolkit for creating graphical user interfaces as well as cross-platform applications that run on various software and hardware platforms such as GNU/Linux, Windows, macOS, Android or embedded systems.

Written in C++, it implements a wide range of different features that could be needed in a variety of cases. There is also an official Qt implementation for Python, which made his choice particularly interesting for the my final project. There are other programming language unofficial bindings supported by the community [18].

Today is very popular among GUI developers, used by both FOSS projects like VLC, KDE, OBS and proprietary software as EAGLE(EDA), VirtualBox, Google Earth.

#### Why Qt

Qt was chosen because it offers the following advantages:

- **Maturity**
  Qt framework has been around for more than 20 years while QML itself has appeared almost 10 years ago. During that time the framework has gone through quite a few iterations and was constantly improved. Being used that much in a production also gave the needed testing from which comes stability and that you cannot get with a brand new framework which might get completely rewritten or abandoned next year.

- **Documentation**
  Comparing with others GUI libraries, it has one of the best documentations available. Every part of the framework is covered, even some video tutorials and examples are provided by the Qt alliance.

- **Performances**
  Qt is after all written in C++ and it runs natively on supported platforms. Also web app are well-supported, thanks to Qt WebEngine which integrates chromium engine.

- **Cross-platform development**
  As developers it's more desirable to make a software for the majority of

platforms in order to increase the user base. Qt officially supports GNU/Linux, Windows, MacOS, Android, iOS,<tv>OS. No needing to rewrite the code to adapt the various O.S. could encourage companies to release multiple versions of the same software virtually with no costs, unbinding the user to a particular operating system.

- **QtCreator**
  Qt SDK full package comes with QtCreator as it's own IDE that can be used for Qt/QML development. It has great integration with the whole Qt platform, programming auto-completion, syntax highlighting and debugger.

## 1.8 OpenCV Library

Another library worthy of mention is OpenCV[19], which includes many features related to image processing and utilities for image manipulation.

Methods like `VideoCapture()` or `imshow()` were very useful in developing and debugging both for ROS and for the final project.

Some uses of this powerful library will be shown in the next chapter, where will be exposed little snippets of code.

# Chapter 2

# Development

This chapter is dedicated to present the final GUI control program and the back-end software necessary for its correctly functioning.
Snippets of example code will be exposed for better explaining the system flow.
The primary focus of the work was to write a Python CLI program able to standalone interact with ROS topics, without having to build ROS packages and sourcing it.

After a short feasibility study, I developed a remote control software with a graphical interface and a built-in window showing the robot's video stream.

## 2.1 Overview

Since it was not possible to develop directly on the physical robot, everything was made thanks to the use of its ideal model on the Gazebo simulation environment.
All the system consists of various elements:

**Client Side:**

- DDS Qomunicator - GUI client program

**Robot/Simulation Side:**

- camera_subscriber_writer (simulation only) - ROS package that sniff from the Image topic and write the data on virtual device
- daemon.py - fetch signals from MQTT server and write navigation data on topic /cmd_vel bypassing ROS ecosystem

- ASGI Server - take /dev/video3 and make a video stream

**Server Side:**

- Mosquitto Server - MQTT Broker (for security and remote telemetry)

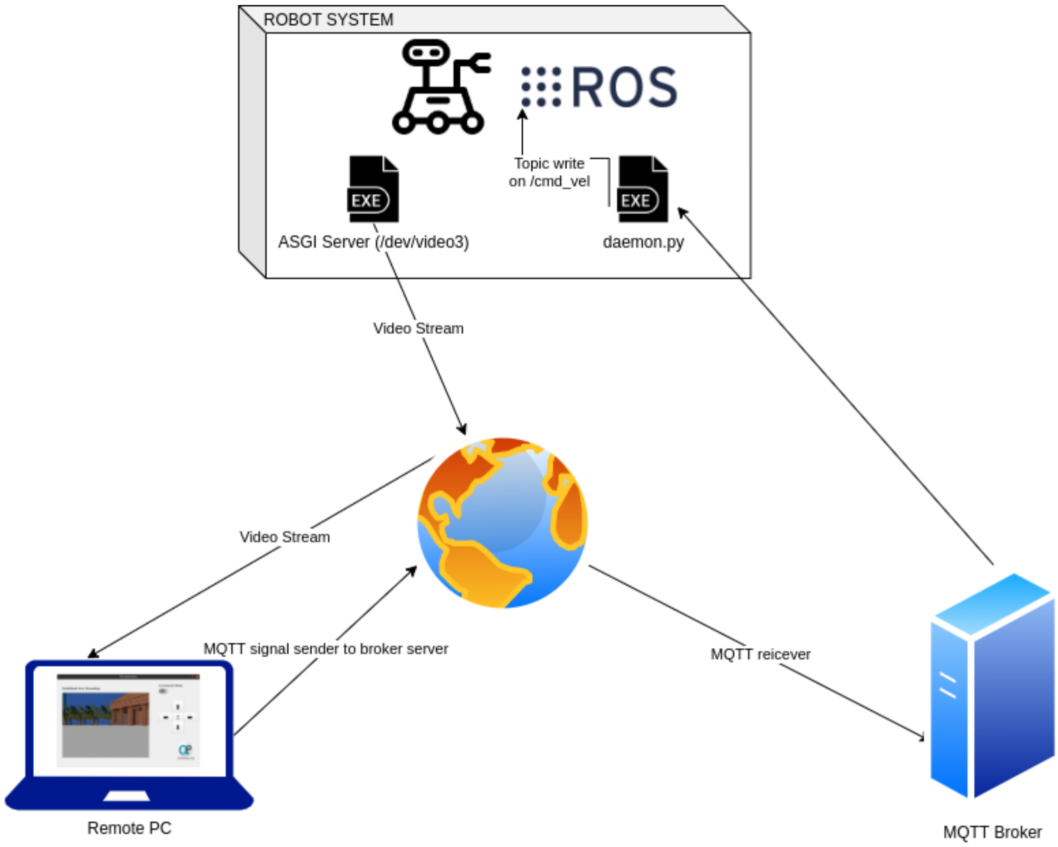


**Figure 2.1:** Block diagram of the system

## 2.2 DDS Qonnector

DDS Qonnector is the main user side program. It is intended as the only program needed to control remotely the robot. It connects to a remote MQTT broker, sending control data. With a simple UI, it has a built-in web browser windows showing the robot video stream and an arrow style controller on the right. Just up the arrows there is a switch capable of enable/disable the "Incremental Mode"; by default this functionality is OFF.

**Incremental Mode: OFF**
The arrows simply determine the motion. This is the safer mode, because it acts one movement by at a time. If the right arrow is pressed just after the up one, the robot stops its forward movement and start to rotate clockwise.

**Incremental Mode: ON**
Here the arrows acts as adder/subtractor for the linear and angular velocity, allowing to compose the vectors.
When switch is pressed, the robot stops.

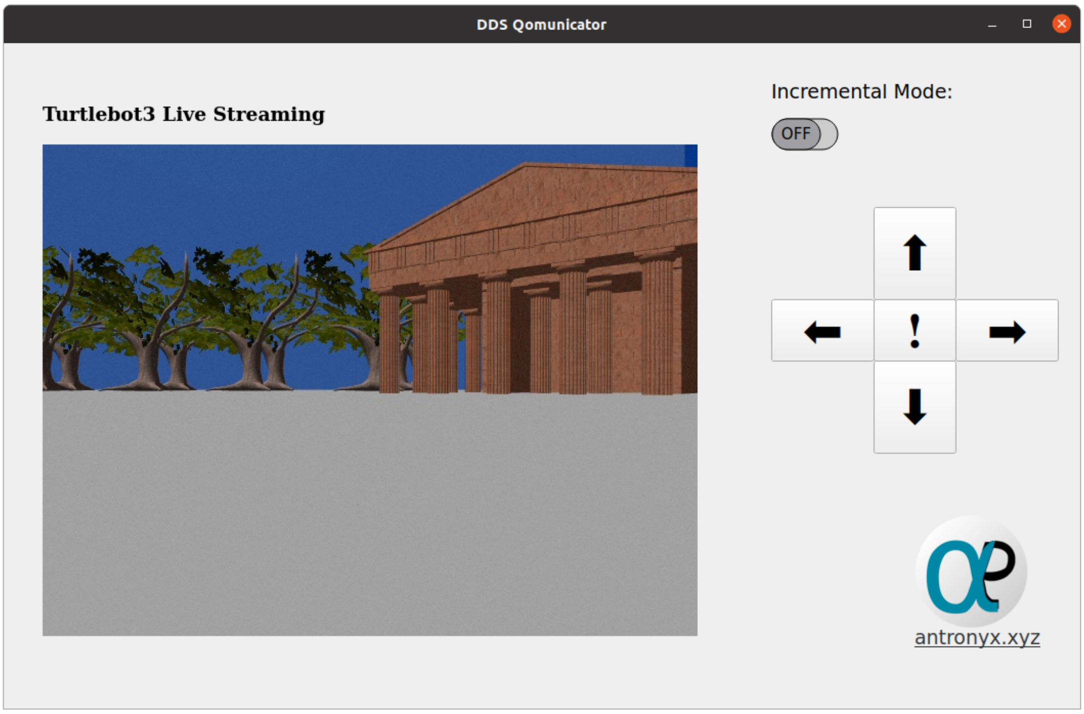


**Figure 2.2:** End-User Program with camera visualization and controls

### 2.2.1 Qt Interface

In Python GUI developing, Qt5 is a professional and wise choice. The use of this library will now be presented with an example, a simple windows with 3 pushButtons. Through an XML-like file with a .ui extension, properly processed by a software, it can be created a .py module that can be imported in a main program, providing the graphical interface.

All the main window elements (PushButtons, RadioButtons, TextBox etc) are included as classes in PyQt, and are called widget. A widget has various properties such as an object name, a position, a dimension, eventually also a text on it. Here the XML for this example code:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>Dialog</class>
 <widget class="QDialog" name="Dialog">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>500</width>
    <height>300</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Dialog</string>
  </property>
  <widget class="QPushButton" name="pushButton">
   <property name="geometry">
    <rect>
     <x>80</x>
     <y>100</y>
     <width>80</width>
     <height>100</height>
    </rect>
   </property>
   <property name="text">
    <string>1</string>
   </property>
  </widget>
  <widget class="QPushButton" name="pushButton_2">
   <property name="geometry">
    <rect>
     <x>200</x>
     <y>100</y>
     <width>80</width>
     <height>100</height>
    </rect>
```

```
   </property>
   <property name="text">
    <string>2</string>
   </property>
  </widget>
  <widget class="QPushButton" name="pushButton_3">
   <property name="geometry">
    <rect>
     <x>320</x>
     <y>100</y>
     <width>80</width>
     <height>100</height>
    </rect>
   </property>
   <property name="text">
    <string>3</string>
   </property>
  </widget>
 </widget>
 <resources/>
 <connections/>
</ui>
```

Once the XML-like file is ready, it must be processed by the pyuic utility, from the QtDevTools package.

```
$ pyuic5 -x Dialog.ui -o Dialog.py
```

After a first setting of the window and definition/arrangement of the graphic components, it's necessary to connect GUI actions like a click to a desired function of the program. It's important to know that there are different ways to interact with a graphical widget. On the pushButton you can Click, press, release, etc. Here we use clicked() way, the simplest. Main example program:

```
#!/usr/bin/python3
## File main.py ##
###################
## Example code for Qt5
from PyQt5.QtWidgets import QApplication, QWidget
from Dialog import Ui_Dialog #this is the generated file

def one():
```

```
    print("You pressed one")
def two():
    print("You pressed two")
def three():
    print("You pressed three")

app=QApplication([])
window=QWidget()
dialog=Ui_Dialog()
dialog.setupUi(window)
# when a clicked() event is detected
# on object pushButtonX, connect a
# custom function
# ####################################
# from dialog object, on the pushButtonX
# at the event clicked, connect
# <foo>
dialog.pushButton.clicked.connect(one)
dialog.pushButton_2.clicked.connect(two)
dialog.pushButton_3.clicked.connect(three)
window.show()
app.exec()
```

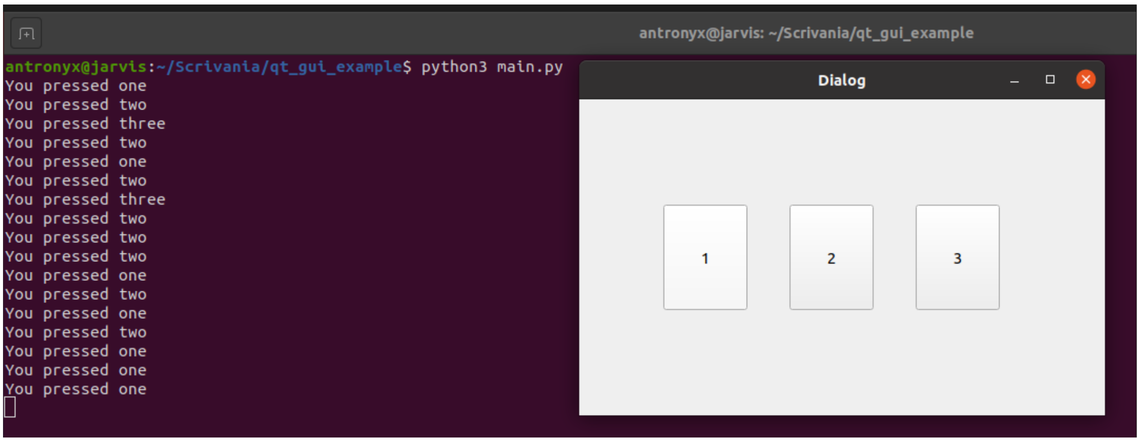


**Figure 2.3:** Qt Example program

### 2.2.2 MQTT publisher

DDS Qonnector has a MQTT publisher on it, sending navigation controls to a remote server; more will be better treated in the next chapters. At the pressure of a button, the main program sends the relative signal to the server. Here an example for the code that manage this purpose:

```python
import time
import paho.mqtt.client as mqtt

# Broker Server IP and topic name
Broker = "Broker Server IP es 95.12.24.48"
topic = "Qonnector/keys"

# Do this every time the program connects to the server
def on_connect(client, userdata, flags, rc) :
    print("Client who send message with code: " + str(rc))

# Connect the client to the Broker at the 1883 port
client = mqtt.Client()
client.on_connect = on_connect
client.connect(Broker, 1883, 60)
client.loop_start()

i = 0
while True:
    i = i + 1
    client.publish(topic, "Test sample no: " + str(i))
    time.sleep(0.001)
```

## 2.3 camera_subscriber_writer

Since a video streaming is needed but there's no physical camera available on Gazebo simulation, this package makes possible to emulate it as a device.

On a physical POSIX system the **/dev** directory contains the special device files for all the devices. ROS relies on Ubuntu GNU/Linux, so every camera connected is normally represented in the special file `/dev/videoN`.

For this reason is better to stream directly from, for example, `/dev/video0` rather than subscribing to the video topic and let send its data by a server.

So, basically, the `camera_subscriber_writer` ROS package allows you to work exactly as you would work with a real robot.
In that package it's used the pyfakewebcam Python library.

To install it:

```
$ pip3 install pyfakewebcam
```

Here the main code for the package:

```
    class DevWriter(Node):
    def __init__(self):
        # Creating the "camera_subscriber ROS Node"
        super().__init__('camera_subscriber')
        # Connector between ROS and OpenCV images
        self.bridge=CvBridge()
        # Creating a subscription to the camera topic...
        # the create_subscription requires a callback,
        # called listener_callback.
        self.img_sub = self.create_subscription(
            Image,
            '/camera/image_raw',
            self.listener_callback,
            qos_profile_sensor_data)
        # Making the virtual device with resolution 640x480
        self.fakecamera=pyfakewebcam.FakeWebcam('/dev/video3',640,480)

    def listener_callback(self, ros_image):
        try:
            frame = self.bridge.imgmsg_to_cv2(ros_image, "bgr8")
        except CvBridgeError as e:
            print(e)

        # We need an RGB image...
        # Since ROS works with Blue Green Red image by default,
        # We need to invert the values in order to arrange correctly
        b,g,r = cv2.split(frame)        # get b,g,r
        rgb_img = cv2.merge([r,g,b])      # switch it to rgb

        # Sending the processed image to the virtual device
        self.fakecamera.schedule_frame(rgb_img)
        time.sleep(1/30.0)


def main(args=None):
    # Initialization before creating the node
    rclpy.init(args=args)
    # Creating ROS node called "camera_subscriber"
    camera_subscriber = DevWriter()
```

```

    # Execute work and block until the context
    # associated with the executor is shutdown.
    rclpy.spin(camera_subscriber)

    # Destroy the node explicitly
    # This is optional, otherwise it will be done
    #automatically by the garbage collector    camera_subscriber.destroy_node()
    rclpy.shutdown()

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

## 2.4 MQTT Server

Although not strictly necessary for this basic driving purpose, an external server was used as MQTT broker. Imagine a scenario in which an expensive robot with a critical task and environment must be remote controlled. An external server could act as a remote telemetry; as an example, space agencies as NASA, ESA and Jaxa uses remote systems to collect data from rovers, satellites and space vehicles. Telemetry is fundamental because the system could be destroyed after or during the test. Engineers need critical parameters to analyze and improve system performance. Without telemetry, this data would often not be available.

Another scenario could be the use of robots in military or SAR, where could be necessarily to know the correct and legit use of the robot by the operator, in order to prevent abuses or misuse; an external server, with a log program could record logs, inaccessible by the user, but by the maintainer.

Also, it is possible to implement a security system, as mentioned in previous paragraph.

To install it:

```
$ sudo apt install mosquitto
```

To make it reachable from the internet the router `port 1883 (default) as TCP` must be opened

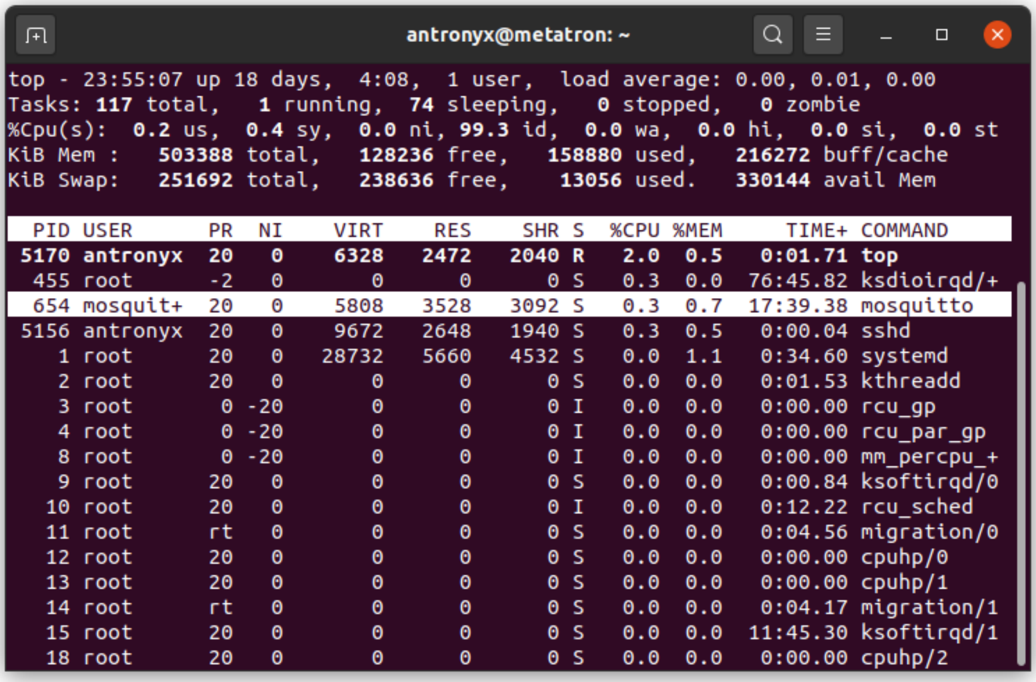


**Figure 2.4:** Mosquitto running on Raspberry-Pi Server

## 2.5 daemon.py

As a *dáimon* in classical mythology, `daemon.py` it's put in the middle between the MQTT Server and the robot. It connects itself to the broker and on new message, it updates the **/cmd_vel** topic who cares about linear and angular velocity, using the Twist data type.

Briefly Twist consists of two $\mathbb{R}^3$ vectors with it's relatives axis **x**, **y**, **z**.

```
Twist
├── linear
│   ├── x
│   ├── y
│   └── z
└── angular
    ├── x
    ├── y
    └── z
```

To drive this robot, linear(x) and angular(z) are involved. Positive linear velocity to move forward, negative velocity to move backward. For rotation around his normal axis, **z**, the value convention reflect the right-hand rule for curve orientation: negative values for clockwise rotation, positive values for counterclockwise.

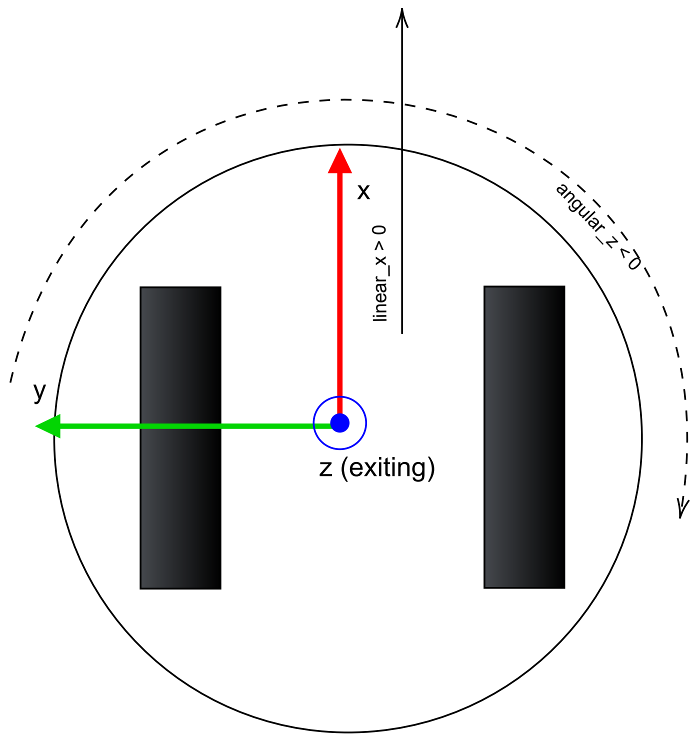


**Figure 2.5:** Axis convention for the robot with velocity value sign representation

Due confidentiality issues,the operative code will not be shown, but for educational purposes an ad-hoc fictitious scenario has been created.

Custom example type:

```
DimensionalTravel
 ├─ name
 ├─ age
 ├─ start_dimension
 └─ arrival_dimension
```

This custom data type has been included in ROS system as a ROS package, so it's possible to develop on it.

It's not shown how to make it possible in this report, but here we show the package file where the type is defined for better understanding:

```
$ cat ~/Scrivania/ms_ws/custom_msgs/msg/DimensionalTraveler.msg
string name
uint16 age
string start_dimension
string arrival_dimension
$
```

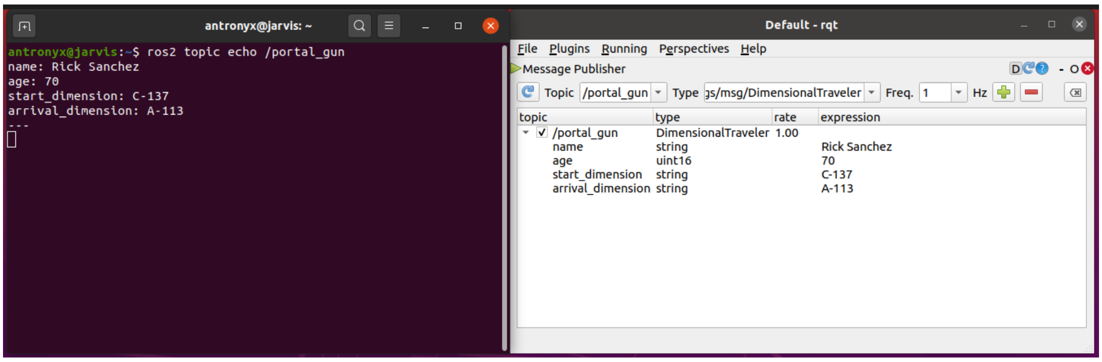


**Figure 2.6:** Using rqt to check the correct working of DimensionTraveler type

### 2.5.1 rticonnextdds_connector

RTI Connext DDS is a software connectivity framework for real-time distributed applications. It uses the publish-subscribe communications model to make data distribution efficient and robust.

Python RTI Connector is an API for publishing and subscribing to the Connext DDS Databus, written in C++

In Connector, the DDS system is defined in XML. This includes the DDS entities and their data types and QoS. Applications instantiate a Connector object that loads an XML configuration and creates the entities that allow publishing and subscribing to DDS Topics.
As mentioned before, Connector works good with any other DDS applications, FastRTPS and ROS topics included.

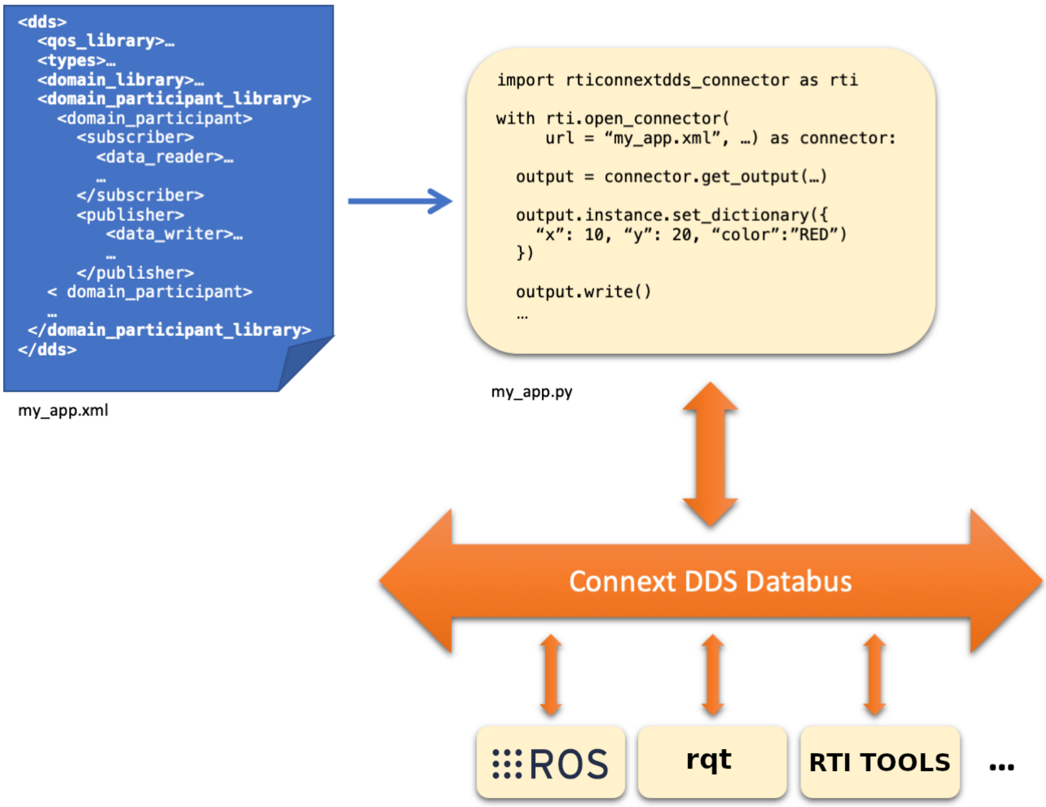


**Figure 2.7:** RTI Connector scheme

The best way to obtain RTI Connector for Python is installing it with pip:

```
$ pip3 install rticonnextdds_connector
```

### 2.5.2 Defining DDS system in XML

Connector loads the definition of a DDS system from an XML configuration file that includes the definition of domains, DomainParticipants, Topics, DataReaders and DataWriters, data types and quality of service.

The following table summarizes the XML tags, the DDS concepts they define, and how they are exposed in the Connector API:

| **XML Tag** | **DDS Concept** | Connector API |
|---|---|---|
| <types> | DDS data types (the types associated with Topics) | Types used by inputs and outputs . |
| <domain_library>, <domain>, <register_type> and <topic> | DDS Domain, Topic | Defines the domain joined by a Connector and the Topics used by its inputs and outputs. |
| <domain_participant_library> and <domain_participant> | DomainParticipant | Each Connector instance loads a <domain_participant>. |
| <publisher>and <data_writer> | DomainParticipant | Each <data_writer> defines an Output |
| <subscriber>and <data_reader> | Subscriber and DataReader | Each <data_reader> defines an Input. |
| <qos_library>and <qos_profile> | Quality of service (QoS) | Quality of service used to configure Connector, Input and Output. |

Here the xml code for DDS system:

```
<?xml version="1.0"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://community.rti.com/schema/6.0.0/rti_dds_profiles.xsd"
    version="6.0.0">
    <!-- Qos Library -->
    <qos_library name="QosLibrary">
        <qos_profile name="DefaultProfile"
                     base_name="BuiltinQosLib::Generic.KeepLastReliable"
                     is_default_qos="true">
            <participant_qos>
                <participant_name>
                    <name>Dimensional Traveler DDS</name>
                </participant_name>
            </participant_qos>
```

```
        </qos_profile>
    </qos_library>

    <!-- types -->
<types>
  <module name="custom_msgs">
  <module name="msg">
    <module name="dds_">
      <struct name= "DimensionalTraveler_">
        <member name="name_" stringMaxLength="255" type="string"/>
        <member name="age_" type="uint16"/>
        <member name="start_dimension_" stringMaxLength="255" type=
"string"/>
        <member name="arrival_dimension_" stringMaxLength="255"
type="string"/>           </struct>
    </module>
  </module>
  </module>
</types>
    <!-- Domain Library -->
    <domain_library name="MyDomainLibrary">
        <domain name="MyDomain" domain_id="30">
            <register_type name="custom_msgs::msg::dds_::
DimensionalTraveler_" type_ref="custom_msgs::msg::dds_::
DimensionalTraveler_" />
            <topic name="rt/portal_gun" register_type_ref="
custom_msgs::msg::dds_::DimensionalTraveler_"/>
        </domain>
    </domain_library>

    <!-- Participant library -->
    <domain_participant_library name="MyParticipantLibrary">
        <domain_participant name="MyPubParticipant" domain_ref="
MyDomainLibrary::MyDomain">
            <publisher name="MyPublisher">
                <data_writer name="MyDataWriter" topic_ref="rt/
portal_gun"/>
            </publisher>
        </domain_participant>

```

```
        <!-- We use separate participants because we run the writer and the reader
            on different applications, and wouldn't want to create the reader
            in writer.py, or the writer in reader.py -->
        <domain_participant name="MySubParticipant" domain_ref="MyDomainLibrary::MyDomain">
            <subscriber name="MySubscriber">
                <data_reader name="MyDataReader" topic_ref="rt/portal_gun"/>
            </subscriber>
        </domain_participant>

        <!-- You can also define the reader and writer inside the same
            connector instance if you plan to use both in the same application.
        -->
        <domain_participant name="MyParticipant" domain_ref="MyDomainLibrary::MyDomain">
            <publisher name="MyPublisher">
                <data_writer name="MyDataWriter" topic_ref="rt/portal_gun"/>
            </publisher>
            <subscriber name="MySubscriber">
                <data_reader name="MyDataReader" topic_ref="rt/portal_gun"/>
            </subscriber>
        </domain_participant>
    </domain_participant_library>
</dds>
```

Now a step-by-step explanation.

### Quality of Service

All DDS entities have an associated QoS[20], that describes the performance constraints of a communication service.

"Generic.KeepLastReliable" was the best choice, because is the default QoS seleted for ROS. It enables keep-last reliability, delivering samples by order of sending. However, new data can overwrite data that has not yet been acknowledged by the reader, therefore causing possible sample loss.

### Types

The <types> tags define the data types associated with the Topics to be published or subscribed to. In this example whe have only a simple structure composed only of standard data types, so we don't need the $< includefile = "./FILE.xml"/ >$ instruction.

```
<types>
  <module name="custom_msgs">
  <module name="msg">
    <module name="dds_">
      <struct name= "DimensionalTraveler_">
        <member name="name_" stringMaxLength="255" type="string"/>
        <member name="age_" type="uint16"/>
        <member name="start_dimension_" stringMaxLength="255" type=
   "string"/>
        <member name="arrival_dimension_" stringMaxLength="255"
   type="string"/>
       </struct>
    </module>
  </module>
  </module>
</types>
```

The `<module>` tag reflect the folder representation of the referred type, custom or ROS standard.

Just before the main structure is needed the additional `<module>` block with **"dds_"** as name.
Under the `<struct>` tag, the core of the complex data type.
With **<member name="" ... type ="string"/>** specify the parameter and the std type of the struct member;
Every member name and the main structure itself must finish with **"_"**

If is not string, integer, float, must be specified is a "non basic variable" using:
**type="nonBasic" nonBasicTypeName="some_path::dds_::SomeNoBasic_"**

### Domain

A DDS domain is a logical network of applications: only applications that belong to the same DDS domain may communicate with each other. A DDS domain has a unique numerical value, called id.
For every ROS installation is proper to define a particular id at the start of the system, setting an environment variable:

```
$ export ROS_DOMAIN_ID=30
```

Or simply append it to `/.bashrc`.

Also, it can be determined a set of registered types and topics, making a pass/stop filter for both.

```
<domain_library name="MyDomainLibrary">
        <domain name="MyDomain" domain_id="30">
            <register_type name="custom_msgs::msg::dds_::DimensionalTraveler_" type_ref="custom_msgs::msg::dds_::DimensionalTraveler_" />
            <topic name="rt/portal_gun" register_type_ref="custom_msgs::msg::dds_::DimensionalTraveler_"/>
        </domain>
    </domain_library>
```

As in code snippet the domain it's restricted to one only topic and to one only data type, the customised one.

### Participant

A (Domain)Participant joins a domain and contains Publishers and Subscribers, which contain DataWriters and DataReaders. Multiples DomainParticipants can be declared.

```

<domain_participant_library name="MyParticipantLibrary">
    <domain_participant name="MyPubParticipant" domain_ref="MyDomainLibrary::MyDomain">
        ...
    </domain_participant>
```

DomainParticipant must have it's own name, callable in after definitions. At the definition of subscriber/publisher is fundamental to declare the topic name.
If desired to work with rmw, it's required to add the **rt/** prefix to the topic name.

```xml
<domain_participant name="MyParticipant" domain_ref="
    MyDomainLibrary::MyDomain">
    <publisher name="MyPublisher">
        <data_writer name="MyDataWriter" topic_ref="rt/portal_gun"/
    >
    </publisher>
    <subscriber name="MySubscriber">
        <data_reader name="MyDataReader" topic_ref="rt/portal_gun"/
    >
    </subscriber>
</domain_participant>
```

### 2.5.3 Python Connector

Since everything is declared in the XML file, the code is not so complex. A connector class is instantiated by receiving the **DDS_setting_file.xml** and the desired Domain Participant.

```python
connector = rti.Connector("MyParticipantLibrary::MyParticipant", "MyDDS.xml");
```

When declaring a Connector, the chosen DDS DomainParticipant and all its sub-entities (Topics, Subscribers, DataReaders, Publishers, DataWriters) are created.
Open and close a connector using respectively `connector=rti.Connector(...)` and `connector.close()`
Alternatively, the `open_connector()` method automatically open and close the connector

```python
with rti.open_connector("MyParticipantLibrary::MyParticipant", "MyDDS.xml") as
    connector.
```

For creation of a Publisher, an object **output** must be created from connector, loading a data writer defined before in XML file.

```
output = connector.get\_output("MyPublisher::MyDataWriter")
```

After the creation, it's possible to set the output object's parameters:

```
output.instance.set\_string("name\_", "Rick Sanchez")
```

Once everything is setted, use the method `write()` to send it; If a delay is desired, the library provides the method `wait(<TIME_MILLIS>)`. Here the example code for topic writer (Python):

```
########## writer code #########

import rticonnextdds_connector as rti

with rti.open_connector("MyParticipantLibrary::MyParticipant", "MyDDS.xml") as
    connector:
  # Use connector
  while True:
    # Setting all the various parameters
    output = connector.get_output("MyPublisher::MyDataWriter")

    # Setting Parameters:
    # the object "output" has the previous introduced
    # attributes...
    #
    # set_string()/set_number() for string/numerical values

    output.instance.set_string("name_", "Rick Sanchez")
    output.instance.set_number("age_", 70)
    output.instance.set_string("start_dimension_", "C-137")
    output.instance.set_string("arrival_dimension_", "A-185")

    # Once everything is set, send the message
    output.write()
    print("DEBUG: Sended")

    # Introducing a delay in sender
    output.wait(1000)
```

For completeness of information, here an example code also for subscriber:

```python
########## reader code #########

import rticonnextdds_connector as rti

with rti.open_connector("MyParticipantLibrary::MySubParticipant","MyDDS.xml") as
    connector:
    input=connector.get_input("MySubscriber::MyDataReader")
    print("Starting Subscriber: Waiting for data...")
    for i in range(1, 500):
        input.wait() # wait for data on this input
        input.take()
        for sample in input.samples.valid_data_iter:
            data = sample.get_dictionary()
            name = data['name_']
            age = data['age_']
            start_dimension=data['start_dimension_']
            arrival_dimension=data['arrival_dimension_']
            print("name: " + name)
            print("age: " + repr(age))
            print("start dimension: " + start_dimension)
            print("arrival dimension: " + arrival_dimension)
            print("-------")
```

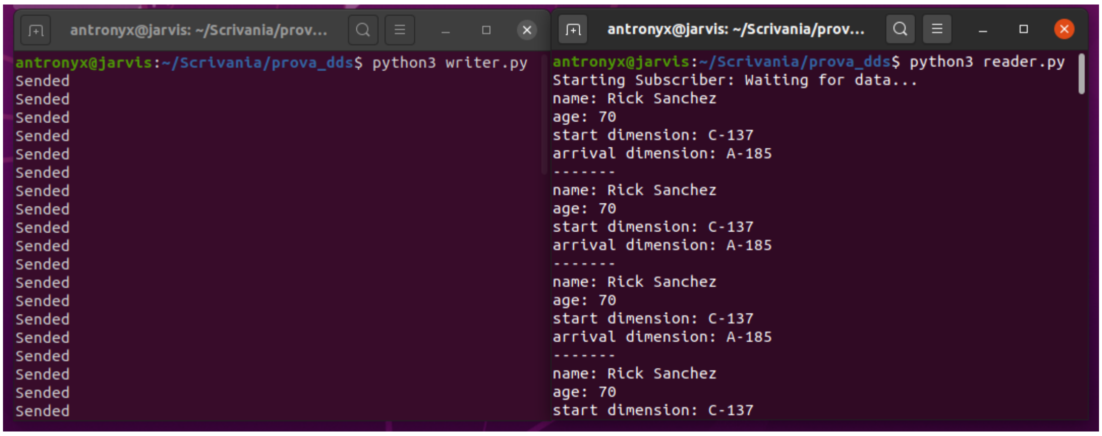


**Figure 2.8:** Publisher and Subscriber working correctly

### 2.5.4 MQTT receiver

This ROS-independent drive program needs to connect to the already introduced MQTT server. The chosen Python library for this purpose is paho-mqtt.

```
$ pip3 install paho-mqtt
```

Briefly the MQTT client code of `daemon.py` subscribes to the remote server topic and, multiplexing the inputs, it writes on the ROS topic /cmd_vel in order to control the robot in the environment.
Here an code example that show how to library can be implemented as a client.

```python
import time
import paho.mqtt.client as mqtt

# broker IP address
Broker = "Remote Server IP e.g. 93.123.45.67"
# subscribe topic
topic = "Qonnector/keys"

# on connect function
def on_connect(client, userdata, flags, rc) :
    print("MQTT Qonnect Server up with code " + str(rc))
    client.subscribe(topic)

# on message function
def on_message(client, userdata, msg) :
    print("MQTT topic contains " + str(msg.payload))
    if msg.payload==b'u':
        ... do something
    if msg.payload==b'd':
        ... do something

# instantiate paho MQTT client
client = mqtt.Client()

# add on_connect and on_message functions to client events
client.on_connect = on_connect
client.on_message = on_message

# connect paho client to mosquitto broker (IP, port, timeout)
client.connect(Broker, 1883, 60)

# client loops forever
client.loop_forever()
```

## 2.6 ASGI Video Stream

A software for remote robot control is useless without the visualization on what is happening to the other part of the world. For this reason a video stream is implemented on the robot side. To extremely minimize latency, an Asynchronous Server Gateway Interface has been used.

Two dependencies must be installed:

```
$ pip3 install uvicorn
```

```
$ pip3 install starlette
```

To implement the streaming, first a `templates/index.html` file is needed:

```html
<html>
  <head>
      <title>Turtlebot3 Live Streaming</title>
  </head>
  <body>
    <!-- HTML color to match the DDS Qonnector's window background -->
    <body style="background-color:#efefef">
      <h3 class="mt-5">Turtlebot3 Live Streaming</h3>
    <img src="/stream/">
  </body>
</html>
```

Here the video stream Python program; to write it I based much of my work relying toughly on documentation [21] [22]:

```python
import cv2
import asyncio
import uvicorn
from starlette.applications import Starlette
from starlette.routing import Route, Mount
from starlette.templating import Jinja2Templates

# Linking the folder ./templates/
templates = Jinja2Templates(directory="templates")

class Camera:
    def __init__(self):
        # camera on /dev/video3
```

```
        self.video_source = 3

    # This function takes bites of video stream,
    # converting to bytes ready to be transmitted by the server
    async def frames(self):
        # Handling video from /dev/video3
        video = cv2.VideoCapture(self.video_source)

        # Handling video streaming error
        if not video.isOpened():
            raise RuntimeError("Could not start video.")

        # return the number of frames in video file
        frame_total = int(video.get(cv2.CAP_PROP_FRAME_COUNT))
        frame_count = 0

        while True:
            if frame_count == frame_total:
                frame_count = 0
                video = cv2.VideoCapture(self.video_source)

            ret, frame = video.read()
            frame_count += 1


            frame_bytes = cv2.imencode(".jpg", frame)[1].tobytes()
            yield frame_bytes
            await asyncio.sleep(0.01)


async def homepage(request):
    # Connecting to webpage showed before
    return templates.TemplateResponse("index.html", {"request": request})

async def stream(scope, receive, send):
    # Server side code

    message = await receive()
    camera = Camera()

    if message["type"] == "http.request":
        await send(
            {
                "type": "http.response.start",
                "status": 200,
```

```
                "headers": [
                    [b"Content-Type", b"multipart/x-mixed-replace; boundary=frame"]
                ],
            }
        )
        while True:
            async for frame in camera.frames():
                data = b"".join(
                    [
                        b"--frame\r\n",
                        b"Content-Type: image/jpeg\r\n\r\n",
                        frame,
                        b"\r\n",
                    ]
                )
                # Sending results to net
                await send(
                    {"type": "http.response.body", "body": data, "more_body": True}
                )

routes = [Route("/", endpoint=homepage), Mount("/stream/", stream)]
app = Starlette(debug=True, routes=routes)

if __name__=="__main__":
    # Server running the framework (uvicorn)
    uvicorn.run("app:app", host="<robot_ip>", port=5000, log_level="info")
```

# Chapter 3

# Conclusions

During this period of internship (taking advantage of the smart and agile working mode with the colleagues of LINKS Foundation), I surely acquired the basis of **real** robotics, that opened up a whole new world for me to explore.
Not only ROS and programming, but also 3D design, web protocols, GNU/Linux system administration are new or reinforced fields.
Certainly, although it was very gratifying, there was no lack of moments of discouragement, doubts and frustration; A lot of effort went into finding bugs and fixing them, spending many hours, days and sometimes a few sleepless nights in this process, searching for documentation and test improvements.

The use of robots is increasing in recent years to carry out risky tasks or in critical and difficult to access areas. The first robots to be used to explore areas inaccessible to people and to report useful information on the state of things were the crawler robots that entered the World Trade Center on 11 September 2001, and the same was for the Fukushima Daiichi nuclear disaster in 2011. In earthquake of Amatrice (Italy) in 2016 the use of drones helped a lot evaluating the impact and to ensure the safety of operators in the field, and during this year it's helping us fight the COVID-19 pandemic [23].
With the next generation network, known as 5G, we will have an almost real-time communication among devices even very distant from each other, which will make telesurgery just another way to operate on a patient [24].

The desired aim of the work is to estimate the potential of remote-controlled robotic systems as much as possible in real time, in order to allow their operation from distant areas, with particular attention to video latency, as the MQTT protocol satisfies very well the reactivity issue. Choosing to use an intermediate MQTT server without having significant delays (in any case much less than video streaming) is linked to hypothetical security issues as explained in the course of the report, primarily linked to the possibility of having a remote "black box". A smooth and reactive video stream was the hardest task to satisfy. At first, the popular Flask framework wasn't enough for it, but the choice of using an asynchronous server as uvicorn-starlette was a wise choice, which allowed for a very short delay overall.

Since ROS could probably be the future standard for robotics as GNU/Linux for servers, the needing to develop compatible software with this platform is growing more and more, I personally believe that the results obtained from my project are a useful starting point for more complex programs for remote control of robots. The autonomy of this program from ROS environment allows to use the same code even with other robotics/automation systems without having to change or modify the software, as long as the system relies on a DDS an use the same naming conventions.

The project that I leave is working properly, but the best would be to test the system on the physical robot, right now in the laboratory. As a demo, the project can only be extended; I thought some improvements as labels on DDS Qonnector showing velocity info, sensor values, network statistics, and by also server side it could be implemented a security and an analytics system.

Despite the COVID-19 pandemic issue, the Links Foundation was made possible to carry out the internship without particular problems, extending what was an exceptional practice to normality. Skype calls with my mentor were very frequent, in which my work was communicated day by day and each time I got useful advice and knowledge for my works. During those calls I learned how is the real work of a programmer, business dynamics and how large software projects are organised and maintained.
Following timetables, chasing deadlines, I felt part of the company and I got a general idea of how to deal with new hard problems, how to "workaround" and "think out the box", discovering in me a new dedication to the pleasure of resolution and the search for new knowledge, hoping finding these challenges in future jobs.

# Bibliography


[1] Open Robotics Foundation. *ROS 2 Documentation*. https://index.ros.org/doc/ros2/ (cit. on p. 1).

[2] Berkeley Regents of the University of California. *BSD-2 License*. https://opensource.org/licenses/BSD-2-Clause (cit. on p. 1).

[3] Apache Software Foundation. *Apache License 2.0*. https://www.apache.org/licenses/LICENSE-2.0 (cit. on p. 1).

[4] Open Robotics Foundation. *ROS 2 Package Documentation*. https://index.ros.org/doc/ros2/Tutorials/Developing-a-ROS-2-Package/ (cit. on p. 2).

[5] Open Robotics Foundation. *ROS 2 Documentation*. https://wiki.ros.org/rviz (cit. on p. 3).

[6] Open Robotics Foundation. *ROS 2 rqt Documentation*. http://wiki.ros.org/rqt (cit. on p. 3).

[7] Open Robotics Foundation. *Gazebo Simulator*. http://gazebosim.org/ (cit. on p. 3).

[8] Open Robotics Foundation. *Gazebo Documentation*. http://gazebosim.org/tutorials (cit. on p. 3).

[9] ROBOTIS. *TurtleBot3 Documentation*. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/ (cit. on p. 5).

[10] Open Robotics Foundation. *ROS 2 Middleware Documentation*. https://index.ros.org/doc/ros2/Tutorials/Working-with-multiple-RMW-implementations/ (cit. on p. 8).

[11] Eclipse Foundation. *Mosquitto Website*. https://mosquitto.org/ (cit. on p. 9).

[12] Eclipse Foundation. *paho-mqtt PyPi Page*. https://pypi.org/project/paho-mqtt/ (cit. on p. 9).

[13] Eclipse Foundation. *Mosquitto Security Documentation*. https://mosquitto.org/man/mosquitto-tls-7.htm (cit. on p. 10).

[14] From ReadTheDocs.io. *ASGI Documentation*. https://asgi.readthedocs.io/ (cit. on p. 11).

[15] Encode.io. *Uvicorn Website*. http://www.uvicorn.org/ (cit. on p. 11).

[16] Encode.io. *Starlette Website*. https://www.starlette.io/ (cit. on p. 11).

[17] The Qt Company. *Qt Documentation*. https://www.qt.io/developers (cit. on p. 11).

[18] The Qt Company. *Qt Language Bindings*. https://wiki.qt.io/Language_Bindings (cit. on p. 12).

[19] OpenCV Developers. *OpenCV Documentation*. https://docs.opencv.org/master/ (cit. on p. 13).

[20] Real Time Innovation. *RTI QoS*. https://community.rti.com/rti-doc/510/ndds.5.1.0/doc/html/api_cpp/group__DDSBuiltinQosProfilesModule.html (cit. on p. 30).

[21] Encode.io. *Starlette Requests Documentation*. https://www.starlette.io/requests/ (cit. on p. 37).

[22] Encode.io. *Starlette Templates Documentation*. https://www.starlette.io/templates/ (cit. on p. 37).

[23] V.B. Manjunath Gandudi R. Murphy J. Adams. «Robots have demonstrated their crucial role in pandemics - and how they can help for years to come». In: *World Economic Forum* (May 6, 2020). URL: https://www.weforum.org/agenda/2020/05/robots-coronavirus-crisis/ (cit. on p. 41).

[24] Caroline Frost. «5G is being used to perform remote surgery from thousands of miles away, and it could transform the healthcare industry». In: *Business Insider* (Aug. 16, 2019). URL: https://www.businessinsider.com/5g-surgery-could-transform-healthcare-industry-2019-8?IR=T (cit. on p. 41).