Hochschule Zittau/Görlitz
UNIVERSITY OF APPLIED SCIENCES

TURTLEBOT4 AND *ROS2* DOCUMENTATION

By Diego Plumed Beortegui

INDEX

1. – ITRODUCTION

This documentation serves as a comprehensive guide for setting up, programming, and simulating robots using advanced tools such as TurtleBot4, ROS2, and Gazebo. Throughout the chapters, the essential steps for creating efficient and customizable robotic environments, both in real hardware and simulated environments, will be covered.

The first part focuses on the proper setup of TurtleBot4, addressing key aspects such as performing a factory reset for the Raspberry Pi and Create3 in case of communication or battery issues. This section ensures that the physical robot is correctly prepared and operational.

Next, the documentation delves into ROS2 (Robot Operating System 2), a powerful platform that simplifies the programming and management of robotic systems. ROS2 enables the development of complex robotic applications by dividing the code into nodes, allowing efficient communication and interaction between various subprograms. By providing a vast array of libraries and supporting multiple programming languages, ROS2 enhances the flexibility and scalability of robotic systems.

Finally, the guide explores the process of designing custom robots and simulating them in a virtual environment using Gazebo. It covers critical concepts such as managing coordinate frames with the TF library, creating robot descriptions using URDF, and simulating robot behavior in Gazebo. These tools are essential for testing and refining robotic applications before deployment in real-world scenarios.

By following this documentation, users will gain the knowledge and tools necessary to build functional and scalable robotic systems from scratch.

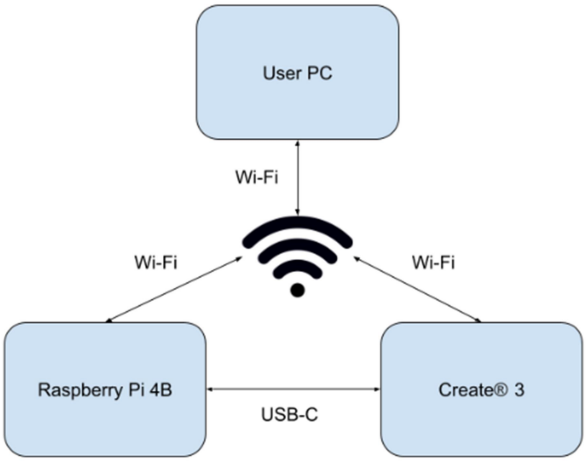
2. – TURTLEBOT4

2.1. – TURTLEBOT4 SETUP

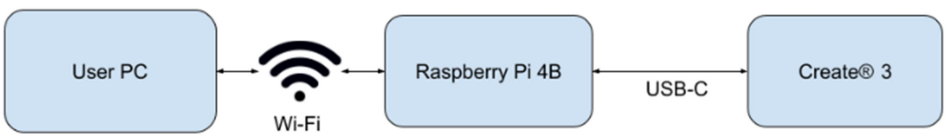
The main of this first part is to properly set up the TurtleBot4. If the Turtlebot4 is not setup yet, since it is new or it was reset for instance, one can find the instructions provided by the developers in the Turtlebot4 User Manual: https://turtlebot.github.io/turtlebot4-user-manual/setup/. One can divide it in two phases: the basic setup (Basic Setup) and the networking configuration (Networking, Simple Discovery and Discovery Server).

With the Basic Setup instructions, one will be able to set up the user PC and robot for basic communication. It is important to note that further setup will depend on the chosen networking configuration. It consists in an easy to follow step by step instruction manual.

Once finished the basic setup, one will notice that there exist two different possible ways to configure the networking, Simple Discovery and Discovery Server. The main difference between both is the way of communication with the Create3: in Simple Discovery the Create3 obtains the information via WiFi, whereas in Discovery Server is through the Raspberry Pi, so there is no need for the Create3 to be connected to any WiFi network.



TurtleBot 4 Simple Discovery configuration



TurtleBot 4 Discovery Server configuration

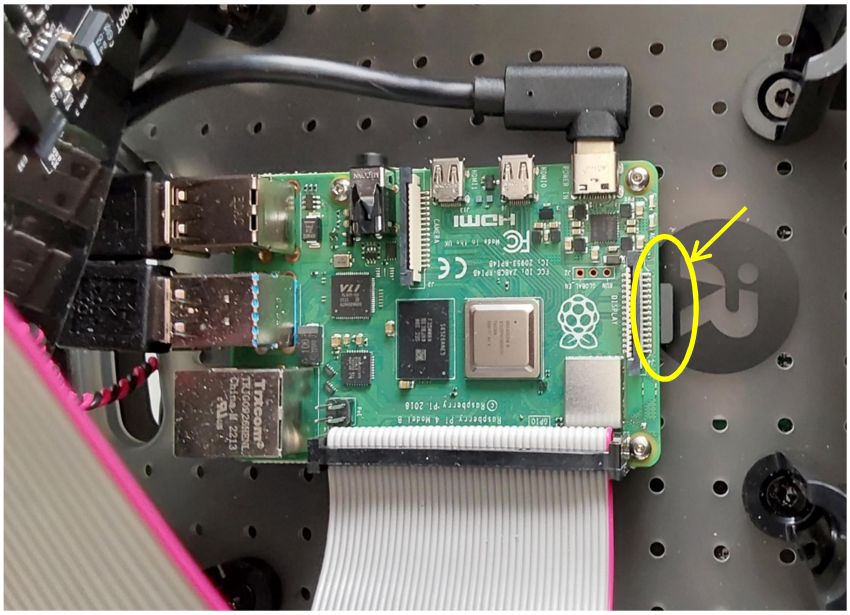
The choice of which to pick is up to the user. However, note that Discovery Server requires extra setup (Check out the Discovery Server documentation for more details). and currently does not support communicating with multiple TurtleBot4's simultaneously from one computer.

In any case, one can obtain the instructions to set them up in the Turtlebot4 User Manual (Simple Discovery and Discovery Server mentioned earlier).

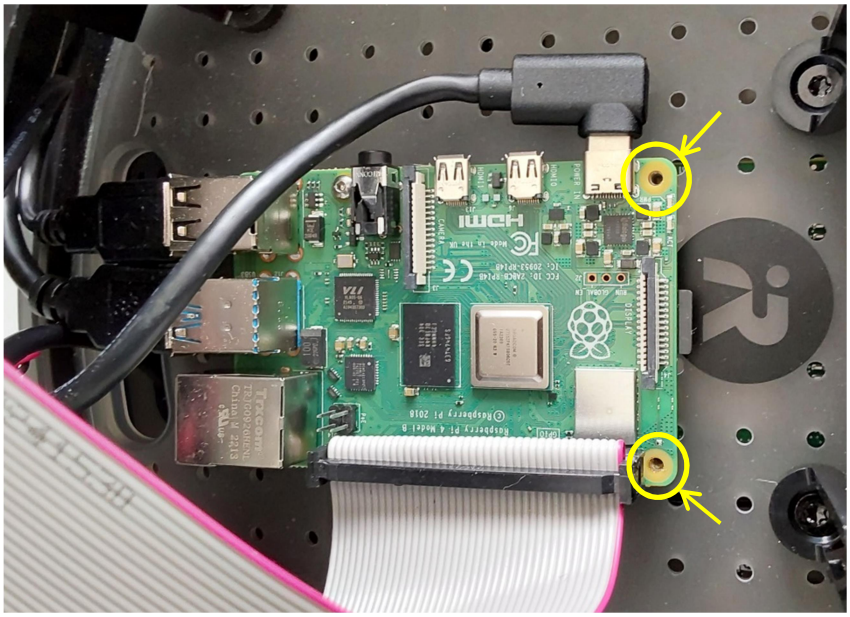
2.2. – TURTLEBOT4 PROBLEM: COMM. and BAT. LEDs issues

In the case where the TurtleBot4 COMM. (communication) and BAT. (battery) LEDs are unable to turn on, a possible solution is to perform a factory reset. To do so, we need to distinguish between the Raspberry Pi and the Create3 factory reset. The steps required are outlined below.

The Raspberry Pi reset is achieved by formatting the integrated SD card. The SD card is placed under the Raspberry Pi.



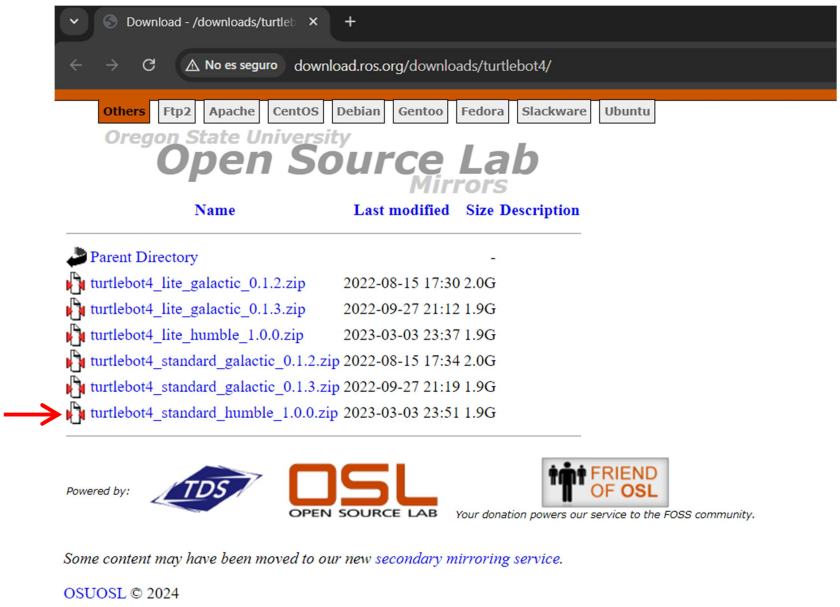
It is recommended to remove, at least, the screws at the right and left side of the SD card so that the release of the card is as safe as possible. Otherwise, it could get damaged.



If the SD card is not empty, it will be necessary to make it so by uninstalling everything on it or formatting it. The formatting can be done using the operating system's formatter or an external one.

Once formatted, the latest Raspberry Pi image will be needed. It can be found at the following link: http://download.ros.org/downloads/turtlebot4/. One should be redirected to a download window.

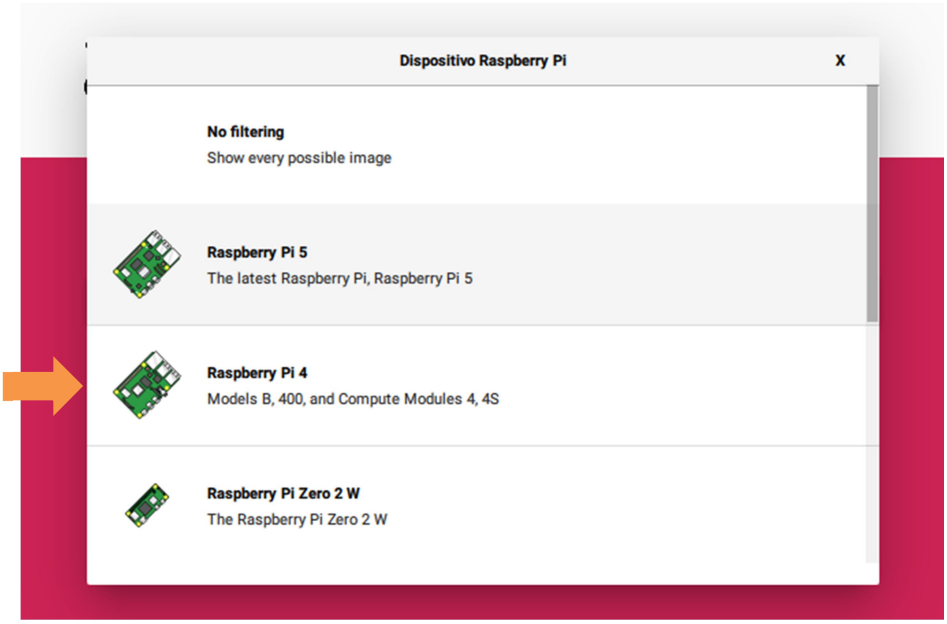
In this case, the Humble ROS2 distribution was used with an standard TurtleBot4 model.



In order to install the OS image on the SD card, the “Raspberry Pi Imager” was used. The executable can be downloaded from here: https://www.raspberrypi.com/software/. Once downloaded, install and open the programme.

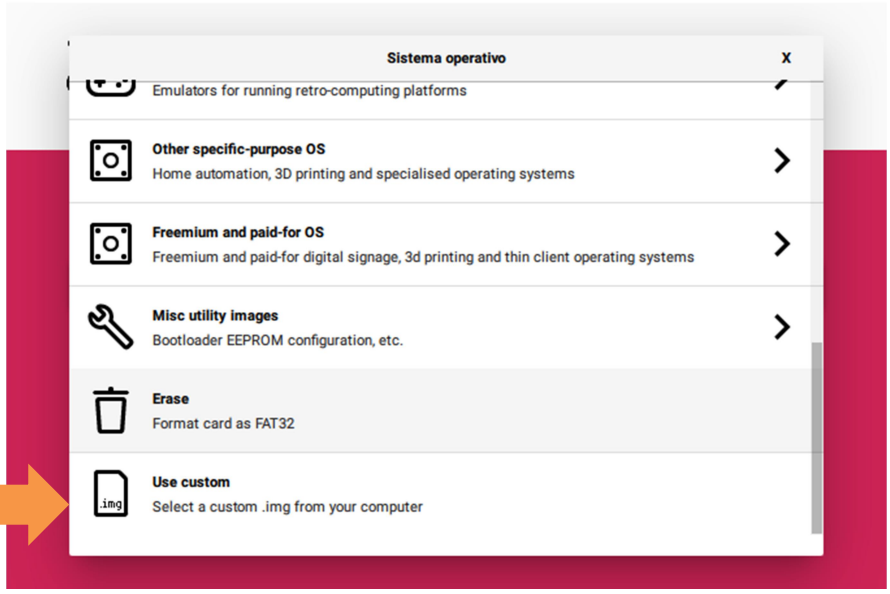
First, the Raspberry device has to be chosen

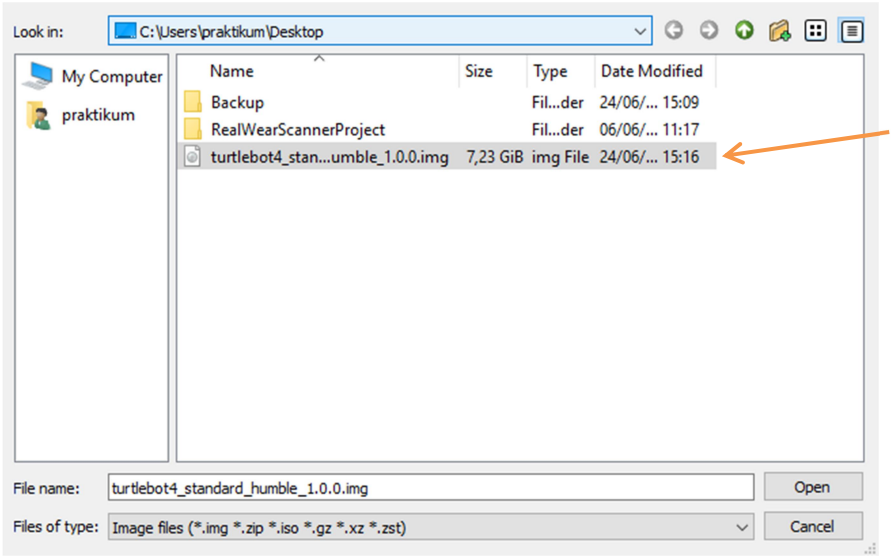
In this case, it is being dealt with Raspberry Pi 4



Then, the corrisponding operating system

At the end, scrolling down, lies the option for custom images

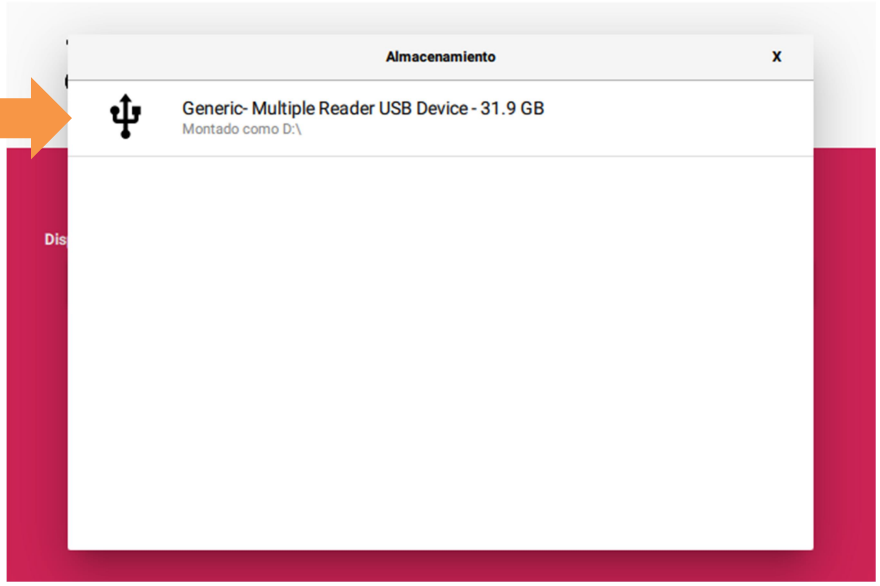




There, it will be found the image previosly downloaded

Finally, the storage device must be chosen

Among the displayed options, the required one will be found

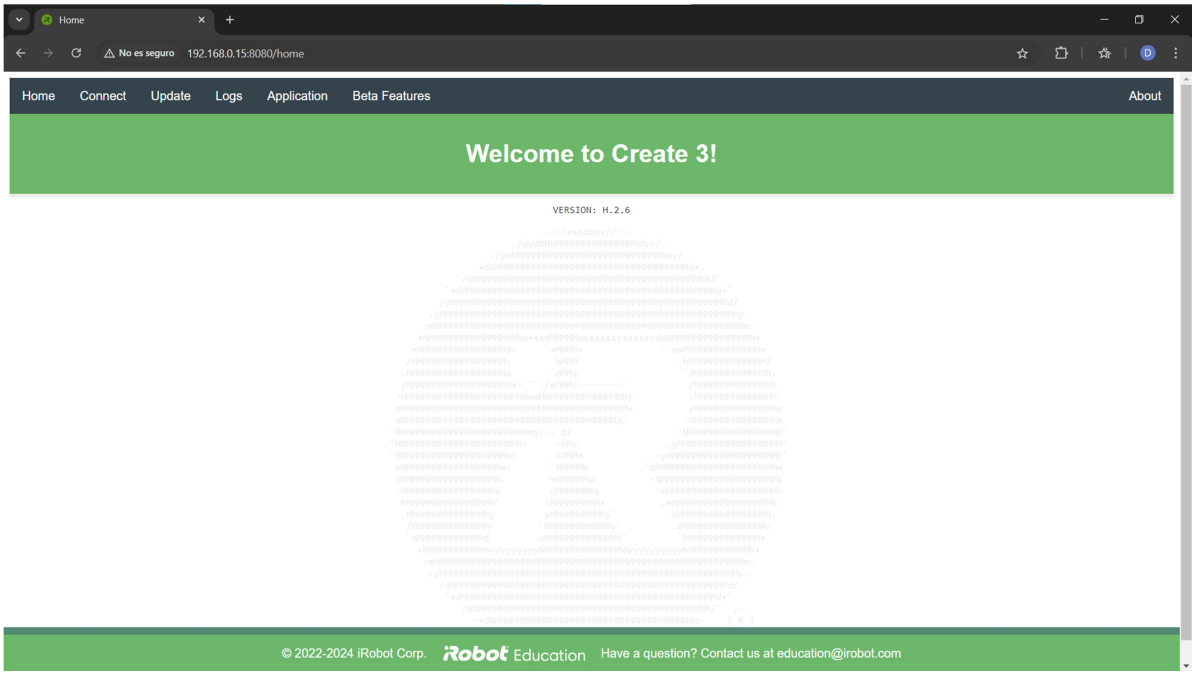


When pressed next, the loading will begin

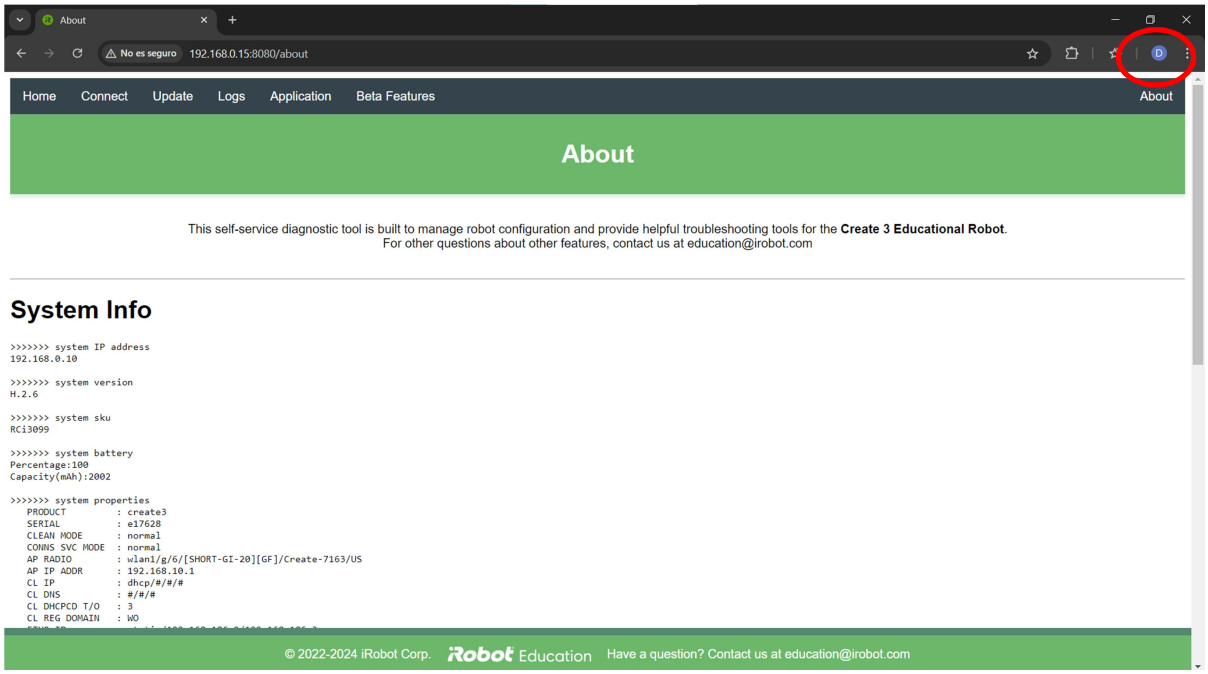
When the installation is finished and the SD card is inserted back into the TurtleBot4, the factory reset of the Raspberry Pi will be complete.

Let's now proceed with the factory reset of the Create3. To do so, the robot must be properly set up.

First, navigate to the Robot3 webpage; one can review how to do it in the TurtleBot4 User Manual (https://turtlebot.github.io/turtlebot4-user-manual/setup/basic.html). Remember, the PC must be connected to the same router as the robot for the webpage to load.



Once here, click on 'About,' located in the top right corner, and scroll down to the bottom of the page.



There lies the hyperlink that allows resetting the Create3.

After following these steps, the Create3 will reboot, and all the LED lights should turn on.

3. – ROBOTIC OPERATING SYSTEM 2

The main goal of ROS2, the Robotic Operating System 2, is to provide a standard for robotic applications, making it much simpler to programme and understand the code of any robot once the basics of ROS2 are known. ROS2 truly shines when one needs to create robot software that requires extensive subprogramme communication or has functionalities beyond simple use cases. This is primarily due to its ability to separate code into blocks (nodes) and the extensive communication tools it provides between them. Additionally, the vast number of specific libraries and the fact that ROS2 is language-agnostic (it can communicate nodes written in C++ with nodes written in Python) facilitate the programming of complex functionalities or interactions within the robot.

Let us delve deeper into the specific tools and functionalities available with ROS2. As mentioned earlier, the key concept of ROS2 is a **node**, a block of code or subprogramme designed to perform a specific task. Nodes can communicate with each other through **topics**; one node can **publish** information to a certain topic, and another can receive it by **subscribing** to the same topic. If the receiving node does not need the data itself but rather needs to perform an action based on that data (e.g., a LED panel that lights up when the battery is below a certain percentage), the publisher would send the information to a **server**, and the receiver would act as a **client** to obtain the conclusion. Hard-coded values can be changed at runtime through **parameters** (e.g., wheel velocity), and programmes made up of various nodes can be configured with **launch files**. Moreover, a wide range of **ROS2 tools** is available to obtain current information about nodes state and communications, or about custom publisher and server messages that may have been created.

With all this, functional and scalable robot applications can be programmed. In the end, a real application case will be presented where all the ROS2 functionalities discussed work simultaneously. Further information and programmes will also be linked.

Notes:

- Visual Studio Code was the coding application used.
- Throughout this topic, “<” and “>” are used to remark the fields that have to be filled by the user (For the package.xml dependencies are not for clarity purposes).

3.1. – ROS2 INSTALLATION

It is important to note that the LTS (long-term support) distribution Humble Hawksbill has been used (it is recommended to always use the latest LTS version). One can find all the ROS2 distributions at the following link: https://docs.ros.org/en/humble/Releases.html. The ones in green are currently in use. Note that it might take from 6 months to 1 year to port all the necessary packages for the newest version to fully support programmes from older distributions. At this other link: https://docs.ros.org/en/humble/Installation.html, you can find the operating systems that support ROS2. In this case, Ubuntu 22.04 was used through a virtual machine. This link leads to a tutorial video on how to install it: https://www.youtube.com/watch?v=1rn7eaEFCoU&t=738s, and this one on how to install and set up ROS2: https://www.youtube.com/watch?v=flT3LIIR5qo (It is already mentioned in the tutorial, but it will be necessary to source the environment every time before using ROS2. Check minute 5:43 of the video).

3.2. – ROS2 PROGRAMMES, NODES

For the purpose of running ROS2 programmes, first it is necessary to install the specific ROS2 building tool: colcon. To do so, type in the terminal the following order:

sudo apt install python3-colcon-common-extensions

Note that the auto completion for colcon is not enabled by default. To activate it, one would have to source the following direction:

source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash

If wanted in every terminal runned (recommended since in ROS2 is common to work with several terminals simultaneously), one could add it at the end of the .bashrc file.

gedit ~/.bashrc

All the programmes of the ROS2 application will be gathered and compiled in a single workspace directory, which is necessary to create and set up.

mkdir ros2_ws (it is a **convention** providing this name to the workspace directory)

Let us set it up.

cd ros2_ws

mkdir src (make source directory)

colcon build (build the workspace)

To enable all the file updates of the workspace, it will be necessary to source the setup.bash file contained in the new install folder of ros2_ws.

source ~/ros2_ws/install/setup.bash

It can be directly done on every new terminal by adding it at the end of the .bashrc file.

gedit ~/.bashrc

Observation: If new files or nodes are added to the workspace, remember to source the environment in all the terminals under use in order to avoid errors.

Before creating a node, one needs to have created a package. Packages will allow separating the code into reusable blocks. For instance, one could have a camera package, a hardware control package and a motion planning package.

Let us create and set up a Python based package:

ros2 pkg create <package_name> --build-type ament_python –dependencies rclpy

To compile the workspace,

cd

cd ros2_ws

colcon build

However, if ran in Humble, a Setuptools error might appear. In that case, the setuptools version will have to be changed,

sudo apt install python3-pip

pip3 list | grep setuptools

pip3 list setuptools==58.2.0

When only is necessary to compile one certain package, then is used the instruction

ros2 colcon build –packages-select <package_name>

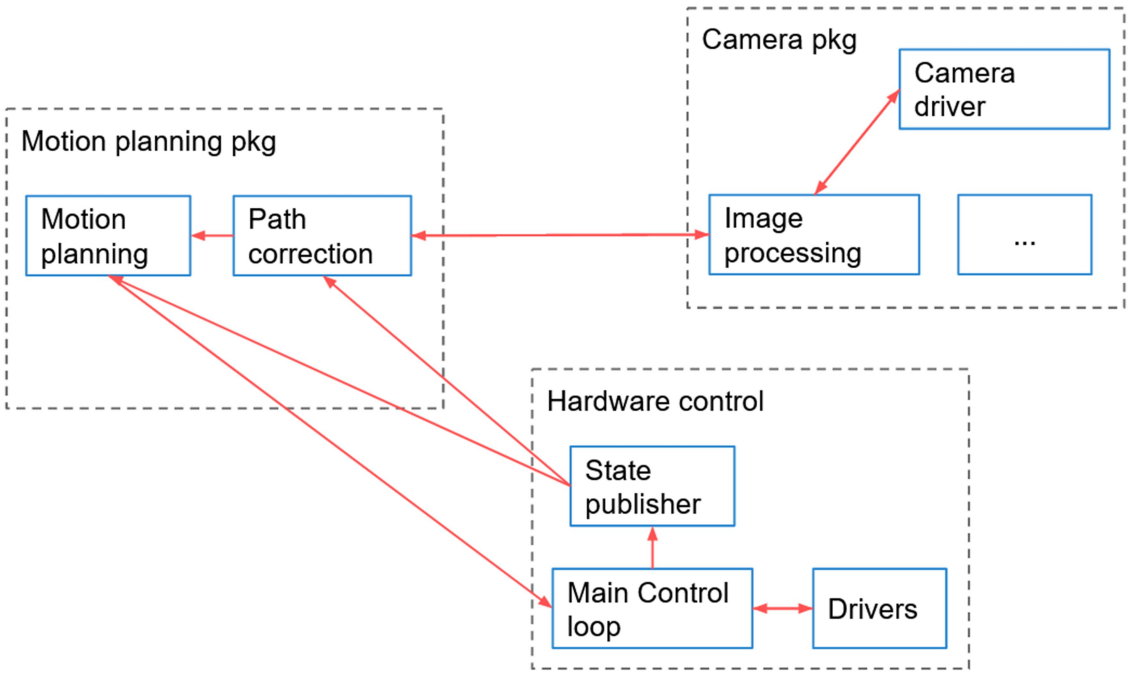
On the other hand, in C++

ros2 pkg create <package_name> --build-type ament_cmake –dependencies rclcpp

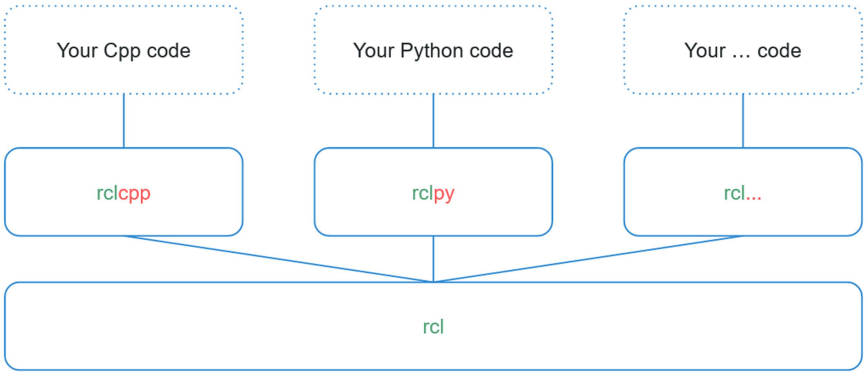
The compilation process is analogous. The first state of *package.xml* and *CMakeList.txt* can be found here:

- *package.xml* : https://drive.google.com/drive/folders/1VMnjS225mrJzT6ZsNDk_4a-sntgxiXYW.
- CMakeList.txt : https://drive.google.com/drive/folders/1kBmvceC1IYtwtfQN5Fl_rJMxqQ4WotLM .

Empty packages will do nothing when compiled nonetheless; they need to be filled with nodes. Nodes are subprogrammes in an application responsible for a single task, and they communicate with each other through topics, services, and parameters. Returning to the example of packages previously given, we could fill those packages with nodes. Here is an example of architecture for a ROS2 application, where one can see packages and nodes, as well as the communications between them



Nodes are written using the appropriate ROS2 client library: rclpy for Python, and rclcpp for C++. Both libraries will provide the same core functionalities. However other programming languages may be used with their appropriate library.



The next two minimal nodes, in Python and C++ respectively, may give an idea of what is a node and how does it look like in OOP:

OOP Python Minimal Node Code

```
1. #!/usr/bin/env python3
2. import rclpy
3. from rclpy.node import Node
4.
5.
6. class MyCustomNode(Node):
7.     def __init__(self):
8.         super().__init__("node_name")
9.
10.
11. def main(args=None):
12.     rclpy.init(args=args)
13.     node = MyCustomNode()
14.     rclpy.spin(node)
15.     rclpy.shutdown()
16.
17.
18. if __name__ == "__main__":
19.     main()
```

OOP C++ Minimal Node Code

```
1. #include "rclcpp/rclcpp.hpp"
2.
3. class MyCustomNode : public rclcpp::Node
4. {
5. public:
6.     MyCustomNode() : Node("node_name")
7.     {
8.     }
9.
10. private:
11. };
12.
13. int main(int argc, char **argv)
14. {
15.     rclcpp::init(argc, argv);
16.     auto node = std::make_shared<MyCustomNode>();
17.     rclcpp::spin(node);
18.     rclcpp::shutdown();
19.     return 0;
20. }
```

Although the Python and the C++ model have been presented, in what follows it will only be taken into account the Python programming structures for ROS2.

After writing the node, it needs to be compiled and the environment re-sourced in order to use it. Nodes are compiled (only for C++, in the CMakeLists.txt), and installed (for both Python and C++, in the package.xml), inside the install/ folder of your ROS2 workspace. They can be directly executed from there, or by using the command line tool

ros2 run <package_name> <executable_name>

Here are left some programmes implementing these concepts: https://drive.google.com/drive/folders/1bfNkplTUc9-tMubq6Rq3aQK5lSoxEU9z.

3.3. – INTRODUCTION TO ROS2 TOOLS

Ros2 includes various useful command lines. Basic examples are:

- *ros2 run <package_name> <executable_name>* : Executes a node with no need of being placed in the corresponding package.
- *ros2 node list* : Lists all the nodes currently running in the environment.
- *ros2 node info /<node_name>* : Displays information about the chosen node.
- *ros2 pkg create <package_name>* : Creates a package named as specified.

Sometimes, may be necessary to run the same node multiple times (e.g. a node to monitor a camera will be needed for every camera of the robot). In that case, it is important to take on board that two nodes cannot have the same name.

- *ros2 run <package_name> <executable_name> --ros-args -r __node:=<new_node_name>* : Executes a node named as specified.

Convention: A node's name always starts with a letter

- *colcon build* : Builds all the packages of the source folder.
- *colcon build --package-select <pkg_name1> <pkg_name2>* : Builds the determined packages from the source folder
- *colcon build --package-select <pkg_name> --symlink-install* : Creates a symlink link to the determined file, so that it will not be necessary to build it again anymore.

Note: The *--symlink-install* flag only works for Python, since the C++ files must be compiled every time. Moreover, it is necessary for the Python file to be an executable. That could be done with the order:

chmod +x <python_file>

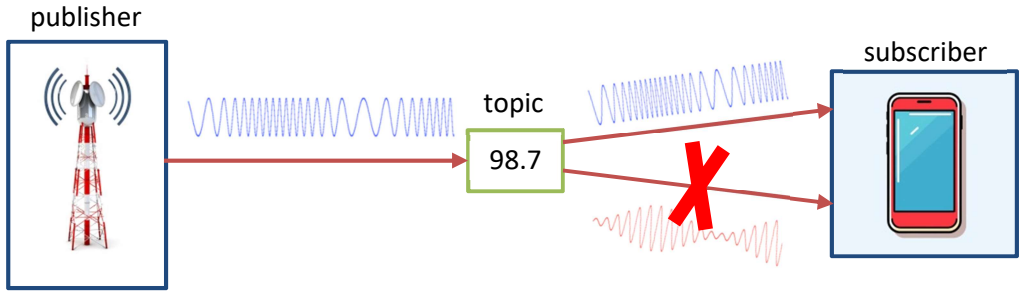
- *rqt* : Runs the *rqt* node.
- *rqt_graph* : Runs the graph plugin form rqt.

rqt is a node provided with a collection of plugins. Among them*, rqt_graph* can be found. *rqt_graph* displays the current nodes graph, so it is a great tool for debugging.

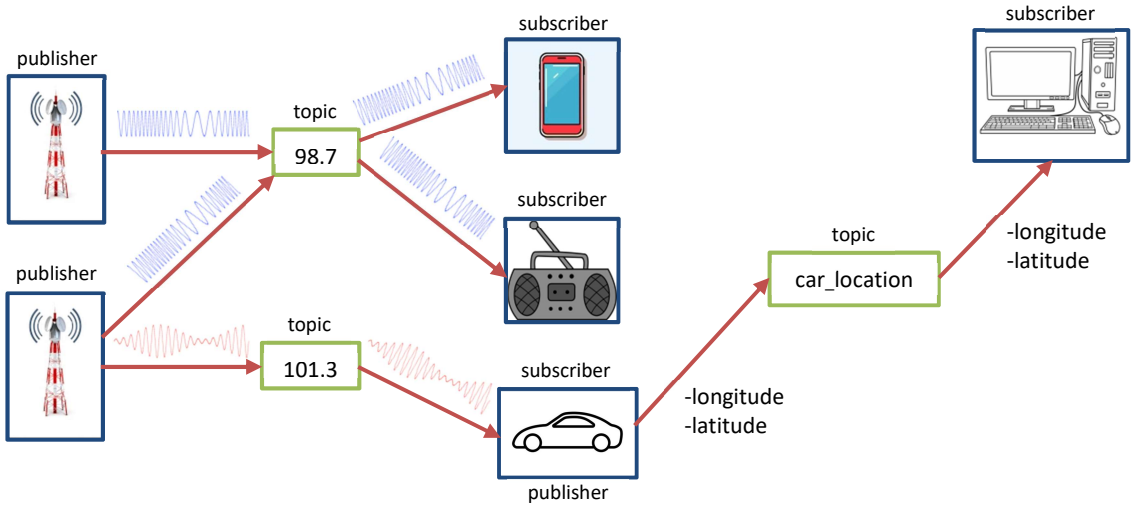
3.4. – PUBLISHERS AND SUBSCRIBERS

Let us introduce the basic communication functionality between nodes: topics, publishers, and subscribers. Their behaviour and interaction can be understood through a real-life example.

Imagine a radio transmitter that sends data on a given frequency, let's say 98.7. At some point, a mobile phone wants to receive that data, so it tunes in to 98.7. In this case, the phone would be a subscriber to the topic. For the communication to be successful, the phone must decode the signal sent by the radio transmitter. Suppose the radio transmitter is sending an FM signal. If the phone tries to decode an AM signal, it would not be able to obtain the data. In other words, the publisher and the subscriber must use the same data structure to communicate effectively. The following picture illustrates this example:



Of course, there are many more possibilities. A topic can have an indefinite number of publishers and subscribers, and a single node can function as both a subscriber and a publisher simultaneously. Considering the blue boxes as nodes and the green ones as topics, we could have the following situation:



Convention: The topic name should start with a letter despite being shown with numbers in this conventional example.

Returning to ROS2, a topic can be defined as a named bus through which nodes send and receive unidirectional data streams. It is important to note that publishers and subscribers are anonymous, meaning that publishers are not aware of who is subscribing, and subscribers do not know who is publishing. To implement topics in a ROS2 application, you first need to have created a node. Inside this node, any number of publishers and subscribers can be created.

Now then, a publisher is created in a node calling:

self.publisher_ = self.create_publisher(MessageType, "<topic_name>", <QoS>)

Quality of service, or QoS, defines the quantity of messages in the que, a standard value for it is "10".

To publish a message on the determined topic,

msg = MessageType()
self.publisher_.publish(msg)

On the other hand, in the case of the subscriber:

```
self.create_subscription(
        <message_type, "<topic_name>", self.callback_of_the_subscription, <QoS>
)
```

Check the programme that follows to obtain a better understanding of these concepts. Before doing so, note that

- Together with ROS2 is installed the *example_interfaces* folder which contains different messages and services data types.
- *self.get_loger.info(<string_msg>)* : Displays the determined string message in the console.
- *self.create_timer (timer_countdown, self.timer_callback)* : Creates a timer which activates the specified callback.

The programmes can be found here: https://drive.google.com/drive/folders/12jPyKADeFmtuxCHPQjqDnhlADV8lpmB-.

Note that publisher and subscriber publish and subscribe respectively to the same topic, and use the same data type. This is fundamental for a successful topic communication.

Once the node is run, there exist several ROS2 command lines which purpose is to facilitate the debug process:

- *ros2 topic list* : Lists all the topics currently running.
- *ros2 topic echo /<topic_name>* : Shows what is the topic receiving.
- *ros2 topic info /<topic_name>* : Displays various information from the specified topic.

- *ros2 interface show <msg_type>* : Shows the exact primitive/s data type that requires the message.
- *ros2 topic hz /<topic_name>* : Shows the actual frequency of the topic.
- *ros2 topic bw /<topic_name>* : Shows the actual bandwidth of the topic.
- *ros2 topic pub -r <publish_frequency> /<topic_name> <msg_type> "{data: <msg_to_publish>}"* : Publishes the written message to the chosen topic at the determinated frequency.
- *ros2 run <pkg_name> <exe_name> --ros2-args -r <topic_name>:=<topic_new_name>* : Renames the chosen topic.

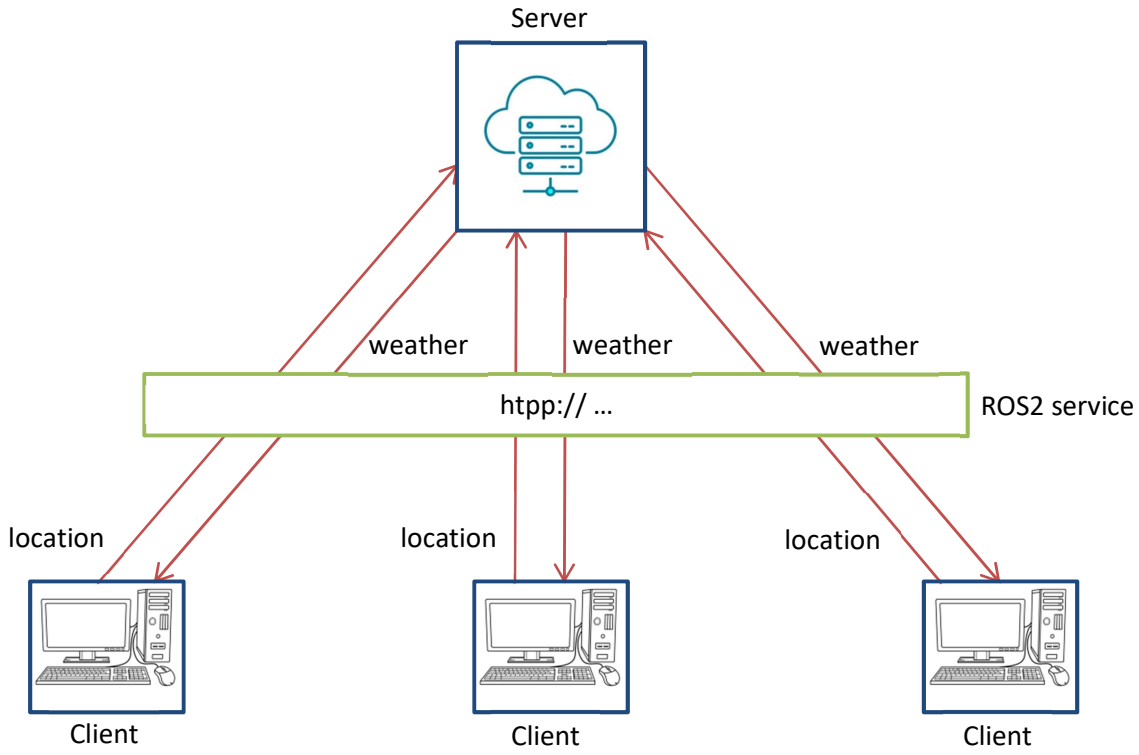
Note: There cannot be two topics with the same name.

Remember that *rqt_graph* is another alternative debug tool which allows observing the different connections between nodes and topics.

3.5. – SERVICES

Again, let us introduce the concept of services with a real-life analogy (although it will not fully correspond to the actual service structure).

Suppose there is an online weather forecast service that provides the local weather for a given location. On the other hand, there is a computer that wants to obtain the weather forecast for its area. In this scenario, the online weather forecast service would be the server, and the computer would be the client. The computer can access the server through an *htpp://…* request followed by the corresponding URL. The URL can be seen as the service name. First, the computer sends a request to the server, such as its location. The server processes that request and sends a response, in this case, the local weather forecast. Note that the request and response are uniquely determined; if the message sent or received is not as expected, the communication will fail. Although many computers (clients) can communicate with the weather forecast server, there must be a unique server for a particular service. This situation is illustrated in the next picture.



In this hypothetical situation, there are three computer nodes, each provided with a service client, and a weather forecast node, provided with a service server. The ROS2 service name would be analogous to the HTTP URL. The computers will call the ROS2 service with a request. The request will be processed by the weather forecast server, which will send back a response through the ROS2 service. Furthermore, this request can be sent either synchronously or asynchronously. As with topics, all clients and servers within nodes are not aware of each other; they only interact through the ROS2 service interface.

Let us see how this is implemented in the code. A service server is created by calling,

self.create_service(

<service_type>, "service_name", self.server_callback

)

On the other hand, to create a service client must be called,

self.create_client(<service_type>, "service_name")

This is applied in these programmes: https://drive.google.com/drive/folders/1xjg7xCDntQJbl2tbumTWArkwZ5lZcW7O.

Note that, to guarantee a successful server/client communication, the service name and the service type must coincide in both server and client.

Remember: Only one server per service can be created, although clients as many as required.

Services are also provided with different ROS2 command lines with the purpose of debugging. The following list will not only gather specific service ROS2 command line but the most useful ones to deal with them.

- *ros2 node info /<node_name>* : It shows, among other information, the services connections of the determined node.
- *ros2 service list* : Lists all the available services.
- *ros2 service type /<service_name>* : It shows the service type of the specified service.
- *ros2 interface show <service_type>* : It diplays both the request and the response interface.
- *ros2 service call /<service_name> <service_type> "{<request>}"* : It calls the specified service with the request given.
    - The definition of the request has to have the following format:
      "{rqst_1: x, rqst_2: y}"
      where was supposed that the service request has two fields, *rqst_1* and *rqst_2*, and *x* and *y* correspond to their desired values.

- *rqt*: In *rqt* → *Plugins* → *Services* → *Service Call* an interactive service call panel is found. Fill in *Expression* the desired parameters and then press *Call.*

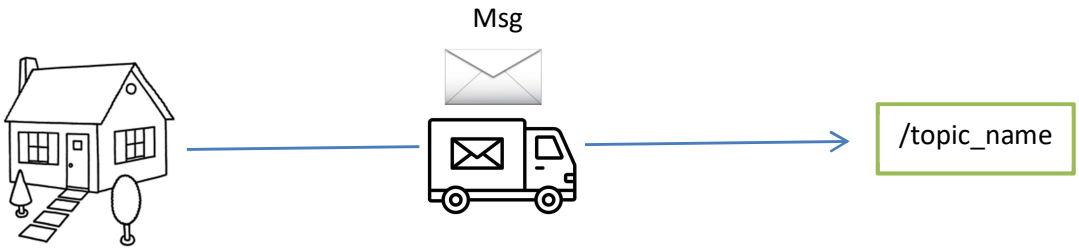
There cannot be two services with the same name, but their name can be changed in runtime:

ros2 run <pkg_name> <exe_name> --ros-args -r <service_name>:=<new_service_name>

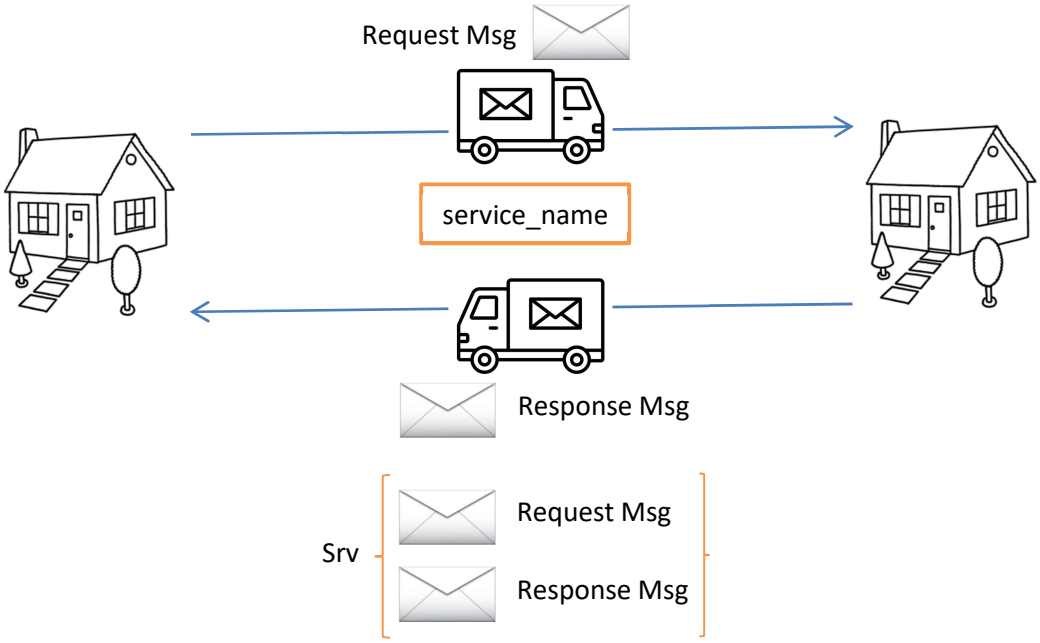
3.6. – CUSTOM INTERFACES

Before studying the concept of the ROS2 interface in more detail, we must first recall that topics are defined by a name, so they can be recognized (e.g., */number_counter*), and by an interface based on a message definition, which is the data structure of the message that is sent (e.g., *example_interfaces/msg/Int64*). On the other hand, services are defined by a name (e.g., /reset_number_count) and an interface that includes two message definitions, one for the request and another for the response (e.g*., example_interfaces/srv/SetBool*). Thus, topics and services can be seen as communication layer tools, while interfaces or messages represent the actual content that is sent. This can be understood through a simple real-life analogy:

When a letter is sent, the mail company carries the letter. The content of that letter is analogous to a ROS2 interface, in this case, using a message definition.



When a response is expected, the letter sent contains a request message, and the one received contains a response message. The combination of these two message definitions constitutes the service definition.



The messages might be in the same package or not. To make a difference between primitive and non-primitive data types, capital letters are used.

Recommendation: Create an alternative package to gather all the custom interfaces created.

ros2 pkg create <robot_name>_interfaces (**conventional** name)

If no build type is specified, by default it will be considered as a C++ package. Thus,

cd <robot_name>_interfaces
rm -rf include
rm -rf src
mkdir msg (folder for messages definitions)
mkdir srv (folder for services definitions)

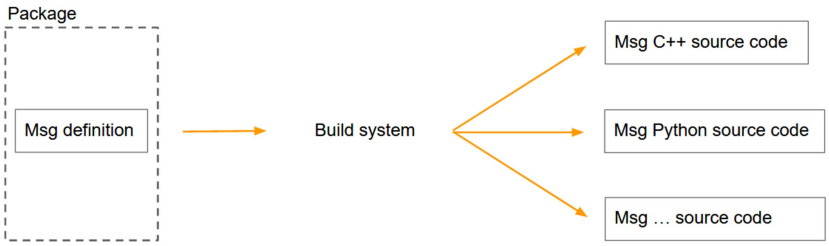
The package setup will be finished after leaving properly the *package.xml* and the *CMakeLists.txt*. They have to end up like these: https://drive.google.com/drive/folders/13dvlNKsTHAKl1p_OysDL_hDlTx0jMBAQ.

The creation of a custom topic or service message consists in:

- creating the file*, .msg* for topic messages and *.srv* for services, in its corresponding folder;
- editing it as wished using primitive, and both local and not local custom data types
- If the custom data type belongs to another package, the dependency will have to be added in both package.xml and CMakeLists.txt;
    - Primitive data types currently available: https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html
    - Github information of the folder example_interfaces: https://github.com/ros2/example_interfaces .
    - Github information about alternative folder, common_interfaces: https://github.com/ros2/common_interfaces .
        - ➢ If the custom data type belongs to another package, the dependency will have to be added in both package.xml and CMakeLists.txt

Convention: To difference primitive and non-primitive data types, capital letters are used in the second one.

- adding it to the *CMakeLists.txt* (as it will be shown in the examples afterwards);
- building the package with *colcon build*;
    - ➢ When the definitions are compiled, new interfaces will be created, along with headers/modules so that they can be included in the nodes



- and, finally, source the workspace in order to use them.

Examples of custom topic and service messages are the following: https://drive.google.com/drive/folders/13dvlNKsTHAKl1p_OysDL_hDlTx0jMBAQ .

Note: The service message structure has to be exactly how is shown in the examples, that is,

<request_msg>

<response_msg>

In Visual Studio Code it is possible to find an error when importing a message. In that case, it will be necessary to add in the *settings.jason* as extra path the direction of the folder where the *__init__.py* of the messages lies.

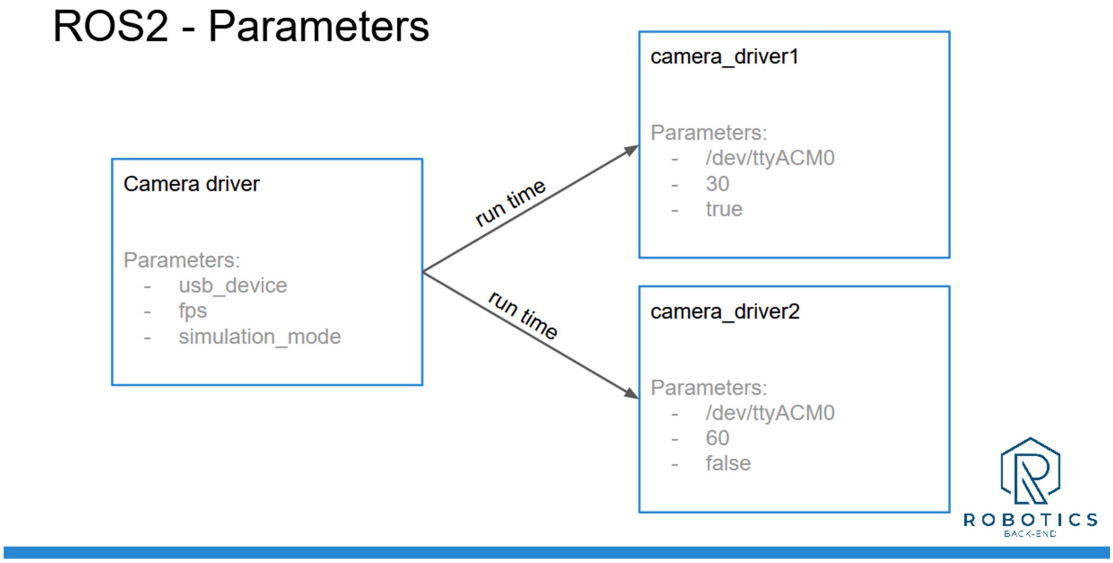
Some useful ROS2 command lines to handle messages and services are:

- *ros2 interface show <msg_type>* : Displays the interface of the message.
- *ros2 interface show <svc_type>* : Displays the interface of the service.
- *ros2 interface list* : Lists all the available interfaces in the environment.
- *ros2 interface package <package_name>* : Displays every message and service contained in the determined package.
- *ros2 node info /<node_name>*
- *ros2 topic list*
- *ros2 topic info /<topic_name>*
- *ros2 service list*
- *ros2 service type /<service_name>* : Specifies the type of the determined type.

3.7. – PARAMETERS

Before explaining what parameters are exactly, let us present their purpose:

Imagine you have a ROS2 package for managing two different cameras. Inside this package, there is a camera driver node that is expected to connect to the camera through a USB and capture images from it. Therefore, inside the node, there will be variables for settings such as the USB device name, the desired FPS for the photos, and whether the node should run in simulation mode or not. However, since each camera is connected through different USBs, need to obtain photos at different resolutions, and one must be run in simulation mode while the other does not, it is not necessary to create another camera driver node with those hardcoded variable values changed. Instead, parameters allow you to set these values at runtime.



Additionally, a parameter is specific to a node; it will only exist while the node is alive. A parameter is defined by a name and a data type, including boolean, int, double, string, and arrays of these types.

Parameters must be previously declared before used. It is achieved by calling in the constructor,

self.declare_parameter("parameter_name", <default_value_of_the_parameter>)

Recommendation: Although *<default_value_of_the_parameter>* is not a mandatory argument, it is saver to always fill it to prevent errors.

Now then, when the node is run the parameter value may be set or changed by making the following addition,

ros2 run <pkg_name> /<node_name> --ros2-args -p <param_name>:=<param_value>

Note: The parameter type does not have to be specified owing to ROS2 type auto-detection.

However, it is also possible to get and use parameters form within a node. To get a parameter from the node is called,

self.get_parameter("<param_name>")

and to get as well its value

self.get_parameter("<param_name>").value

Parameters were added to the *led_panel* programme that can be found here: https://drive.google.com/drive/folders/1AJGVcOwNE4rcvlDW9NGyiCpBeCSu_29n.

Useful command line for parameters is:

- *ros2 param list* : Shows all parameters in the current graph.
- *ros2 param get /<node_name> <parameter_name>* : Shows the type and value of the specified parameter.

Remember: Each parameter is private to the node, many nodes might have the same name for a parameter but these parameters are independent from each other.

3.8. – LAUNCH FILES

It has been seen that nodes, topics, services can be renamed and parameters set. Therefore, the situation becomes messy and complex if several nodes have to be run simultaneously from the terminal. Launch files job is to assist the user in this matter; they allow starting and configuring as many nodes as wanted from one single file.

Launch files are recommended to be centralized in an alternative package to ease their management.

ros2 pkg create <robot_name>_bringup (**conventional** name)

Remember: If no build type is specified, it will be automatically selected the *ament CMake* build type.

rm -rf include/
rm -rf src/
mkdir launch (creation of the directory where launch file will be gathered)

The CMakeList.txt has to be configured as well.

PHOTO OF HOW IT SHOULD BE LIKE

Convention: A launch file name should be *<launch_file_name>.launch.py*

Make it executable to use *--symlink-install*

chmod +x <launch_file_name>.launch.py

Minimal launch file code

```
from launch import LaunchDescription

def generate_launch_description():
    ld = LaunchDescription()


    return ld
```

Note: The name of the function must be that an only that, since when the launch file is installed, the launch functionality will create a new programme which takes this function to the launch application. However, if the function is not found, the lunch file would not run.

Nevertheless, this launch file would do nothing. So as to launch a node, it has to be added,

```
from launch import LaunchDescription
from launch_ros.actions import Node          ##import the package

def generate_launch_description():
    ld = LaunchDescription()

    node_to_launch = Node(                   ##add a node to launch
```

```
            package="<pkg_name>",
            executable="<exe_name>"
        )

        ld.add_action(node_to_launch)          ##instruction to launch the node

        return ld
```

Note: The dependency of every package form which we obtain the nodes have to be added to the *package.xml*.

```
<exec_depend>package_name</exec_depend>
```

Some more lines will have to be added to the constructor of the node in order to fully configure it.

```
node_to_launch = Node(
        package="<pkg_name>",
        executable="<exe_name>",
        name="<new_node_name>",            ##it renames the node
        remapping=[
            ("<topic_name>", "<new_topic_name>")        ##it remaps a topic
            ("<service_name>", "<new_service_name>")    ##it remaps a service
        ],
        parameters=[
            {"<param_name>": <parameter_value>}
        ]

)
```

The ROS2 command line tool that launches packages is the following,

```
ros2 launch <pkg_name> <launch_file_name>.launch.py
```

Here some examples of launch files can be found:
https://drive.google.com/drive/folders/1Yoo9DzGhG5CqrQELyyDvaLjJVPPXONP-.

3. 9. – TURTLESIM CATCH THEM ALL

This is the final section of this topic. Here, a final application that makes use of all the concepts previously explained will be presented. The application was called “Catch Them All” and is based on the *turtlesim* node.

If *turtlesim* it is not installed yet, it can be by running:

sudo apt install ros-humble-turtlesim

turtlesim is a very comprehensive node that allows experimentation with all ROS2 functionalities. The main node of the package is *turtlesim_node*, in fact, the application is based on this node.

“Catch Them All” consists in a simulation making use of *turtlesim_node*. The default spawned turtle becomes the “turtle boss” and its job is to catch all the other turtles that will be randomly spawned in the field. Moreover, the catching order is determined; the closest turtle has to be his primary target. This application can be launched through the *turtlesim_catch_them_all.launch.py* launch file.

Here is left the launch file to run the application: https://drive.google.com/drive/folders/1mfPLZD5Vge_dn8GUyHYf6K4T0vHoNDO3.

4. – ROBOT MODEL AND SIMULATION

The ability to design a custom robot and operate it within a simulated environment is a crucial step in developing a specialized robotic application. This documentation works as a guide to build and simulate a robot from scratch, laying the foundation for any robotics project.

Firstly, it will be explored the **TF** (Transform) library, a vital tool for managing the coordinate frames that define the position and orientation of the different parts of a robot. Understanding TF is essential for ensuring that the components of the robot move and interact correctly. It will also be covered how to visualize these frames in **RViz**, which is used for debugging and refining the design of a robot.

Following that, it will be delved into writing a **Unified Robot Description Format** (URDF) file. URDF is used to describe the robot's physical structure, including its parts and how they move relative to each other. The fundamentals to define **links** and **joints** will be presented, something crucial for accurately modelling a robot.

Once the robot's description is done, the next step will be simulating it in **Gazebo**, a powerful robotics simulator. It will be explained how to control a robot using **Gazebo plugins**, allowing it to interact with a simulated world, including adding sensors and other functional components.

Additionally, it will be covered how to package a robotics application to optimize its management and distribution. This includes enhancing URDF files using **Xacro**, a more flexible and reusable method for defining robot models, and adapting a robot for its integration with Gazebo.

4.1. – PREVIOUS SETUP

First, it is necessary to have Ubuntu installed. In this documentation, as already mention, is used Ubuntu 22.04 alongside with the Humble distribution of ROS2. If Ubuntu is not the main OS installed in the computer, the optimal way to work with ROS2 is installing Ubuntu using a dual boot. In the case it has been impossible to install Ubuntu through a dual boot, a virtual machine could be an alternative. However, virtual machines do not operate well with 3D simulations like Gazebo.

Recommendation: As a VM use VMWare Workstation (free version) and avoid VirtualBox.

Of course, ROS2 has to be installed in the computer. To do so, access the already explained instructions in the page number 14.

Once ROS2 is installed and setup, it can be checked if the computer is powerful enough or that the system is working correctly. By the command

*sudo apt install ros-humble-gazebo**

sudo apt install gazebo

, Gazebo and the ROS2 Gazebo packages will be installed. It may also be needed to source the file */usr/share/gazebo/setup.bash*. So

source /usr/share/gazebo/setup.bash

could be added to the end of the *.bashrc* file to facilitate the process in following times. Now then, by running in the terminal

gazebo

, Gazebo will open. At the right bottom corner the FPS are shown. If the FMS are less than 10, either the computer has not enough resources, or the system is run in a VM and is not powerful enough. In that situation, ROS2 will not be able to perform properly, so alternative ways will have to be searched.

4.2. – INTRODUCTION TO TRANSFORM

Transformations, TFs, are the centerpiece when a robot wants to be run with ROS2, both if it is real or simulated. TFs basically refer to the system ROS2 uses to keep track of multiple coordinates frames over time. Since this concept is a bit ambiguous, let us introduce it with an example.

The lack of robots designs in the system can be solved installing a ROS2 package. This is

sudo apt install ros-humble-urdf-tutorial

source opt/ros/humble/setup.bash

ros2 launch urdf_tutorial display.launch.py model:=/opt/ros/humble/share/urdf_tutorial/urdf/08-macroed.urdf.xacro

Two windows will open; the RViz software and a joint state publisher panel.

On the one hand, RViz is a ROS2 visualization tool capable of computing 3D robots models. The right side panel displays the robot model alongside with its TF. With the mouse it is possible to navigate in it, using the scrolling button and pressing the right and left button. The upper left panel controls the different displays. If *TF* is unchecked, what remains is the Robot Model. Expanding the *RobotModel*, one can check and uncheck the different parts of the robot inside the *Links* tag or change its transparency in the *Alpha* tag. If *RobotModel* is unchecked, the TF of the robot will be displayed. Once again, it is possible to check and uncheck the different TFs, also known as joints, expanding the *Frames* tag inside *TF*.

Note that each joint is represented with a 3D axis where the red, green and blue axis represent, respectively, the x, y and z axis in the positive direction. Its duty is to precise the relative position of the robot links with respect to the others and, if necessary, how their movement behaves relative to each other. This fact could be seen through the joint state publisher window, changing the different values available.

Note: The TF, that is the transformations between the different frames of a robot, is the most important fact for ROS2 in order to properly run a robot.

Furthermore, the origins of the different joint are connected with arrows. These arrows represent the relation between the TFs.

Notation: When a joint is pointed by another one is said to be its father, and the other is said to be the child of the joint.

Note that, the relation order is important; if a parent moves, all its successors will be affected. The set of all robot TFs form what is called the TF tree. In fact, there exists a ROS2 tool that creates a pdf file containing the robot TF tree. First, the existence of the package has to be ensured:

sudo apt install ros-humble-tf2-tools

source /opt/ros/humble/setup.bash

Then, while RViz is still running, the following command will create the pdf file:

ros2 run tf2_tools view_frames

The file will be placed in the directory where the command was run. In the TF tree, the links are represented with circles whereas the transformations with arrow. The TF tree is generated by subscribing to the *tf* topic, the main tool used by ROS2 to handle TFs.

ros2 topic list

ros2 topic echo /tf

A transformation itself consists in a translation and a rotation. In this topic, it is constantly published each one of the transformations data alongside a time stamp. The time stamp is given since; it is not only necessary the relative position and movements between links, but also the exact time when they occurred.

However, the TFs implementation in ROS2 is not done directly, but through URDF files thanks to existing ROS2 packages.

4.3. – UNIFIED ROBOT DESCRIPTION FORMAT (URDF)

A URDF file contains the description of all the elements in a robot; form the links, which will correspond to the physical rigid parts of the robot, to the joint, that represent the different relationship between them. The file format is XML, and can be visualized in RViz, a great debug tool.

Remember: When creating a URDF file, it has to be used the *.urdf* extension and, at the beginning of the file has to be written

```
<?xml version="1.0"?>
```

in order for the format to be XML.

4.3.1 – Links

A basic link code could be the following:

```
<?xml version="1.0"?>
<robot name="robot_name">

  <material name="blue">
    <color rgba="0 0 0.5 1" />
  </material>

  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.6 0.4 0.2" />
        <!—the argument are the box length, width and height respectively -->
      </geometry>
      <origin xyz="0 0 0.3" rpy="0 0 0" />
      <!—
        the default the origin is placed in the middle of the link
        rpy means roll, pitch and yaw:
              roll -> rotation in the x axis
              pitch -> rotation in the y axis
              yaw -> rotation in the z axis
      -->
      <material name="blue" />
    </visual>
  </link>
</robot>
```

The *robot* tag is the main one, which will contain all the information concerning the robot. Its name is necessary to be given as an argument. To create a link is used the *link* tag, which requires a name as an argument. Here is given a simple link with just a *visual* tag. Inside it is specified the geometry of the link, its offset relative to the link origin and the wanted colour for it. Here can be found the complete link tags list: https://wiki.ros.org/urdf/XML/link.

Note: To visualize in RViz a URDF file can be used the tutorial installed previously,

```
ros2 launch urdf_tutorial display.launch.py model:=/absolute_path_of_the_urdf_file
```

4.3.2. – Joints

As it was mentioned, the different links have certain relationships between them given by the so called joints. Let an additional link inside the robot tag be

```
<material name="grey">
   <color rgba="0.5 0.5 0.5 1" />
</material>

<link name="second_link">
   <visual>
      <geometry>
         <cylinder radius="0.1" length="0.2" />
      </geometry>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <material name="grey" />
   </visual>
</link>
```

Note: If

ros2 launch urdf_tutorial display.launch.py model:=/absolute_path_of_the_urdf_file

is tried to be run, an error will trigger. This is due to the lack of relationship between the two links.

Now, let their joint be

```
<joint name="base_second_joint" type="fixed">
   <parent link="base_link" />
   <child link="second_link" />
   <origin xyz="0 0 0" rpy="0 0 0"/>
</joint>
```

The joint tag has as arguments its name and type. Inside it, it is required, at least: the parent link, the child link and its offset relative to the parent origin (the origin link, in this case), more tags might be mandatory depending on its tag.

Convention: The joint is named following the structure *<parent_name>_<child_name>_joint*.

Once the joint is created, it becomes the origin to which the child is placed with respect to. Moreover, the joint origin is offset is relative to the parent

Recommendation: Let the link origin be 0 0 0 until its corresponding joint is correctly placed. This can be checked using RViz.

The procedure to correctly place the *second_link* with respect to the *base_link* would be,

- Change the *base_second_joint* origin to *<origin xyz="0 0 0.1" rpy="0 0 0"/>*
- Change the *second_link* origin to *<origin xyz="0 0 0.2" rpy="0 0 0"/>*

Here can be found all the types available for a joint alongside their requirements: https://wiki.ros.org/urdf/XML/joint. The most relevant ones are:

- *revolute* : to turn around an axis. The following tags must be added:
    - *<axis xyz="0 1 0" />* : by default the rotation is done in the y axis.
    - *<limit lower="radians" upper="radians" velocity="100" effort="100" />.*
- *continuous*: to turn around an axis without limits. The following tags must be added:
    - *<axis xyz="0 1 0" />* : by default the rotation is done in the y axis.
- prismatic: to glide in a certain direction. The following tags must be added:
    - *<axis xyz="0 1 0" />* : by default the rotation is done in the y axis.
    - *<limit lower="radians" upper="radians" velocity="100" effort="100" />.*
    - *<axis xyz="0 1 0" />* : by default the rotation is done in the y axis.

Here is an example of a robot built using different types of joints: https://drive.google.com/drive/folders/1OxE6jL9rvpSaqq4c6FpLPMjgZWD18A8n. Note that the *base_footprint* link completely empty. Its function is the place the robot on the floor, that is on the grid, if it is chosen as *Fixed Frame* in the *Global Options* tag in RViz.

4.4. – BROADCAST TFs

Summarizing so far, URDF files duty is to specify the links and joints of a robot, which afterwards need to be published to the */tf* topic in order for a ROS2 application to work. The publishing process has been done getting advantage of the *urdf_tutorial* package nevertheless, something not practical in a final robotics application. In this section it will be explained how the TFs are really published.

4.4.1. – How is it done?

rqt_graph may be a helpful tool in order to comprehend what happens behind the scenes when publishing to the */tf* topic. By running the following commands in the terminal:

ros2 launch urdf_tutorial display.launch.py model:=/absolute_path_of_the_urdf_file
rqt_graph

, it will be shown in the screen.

It can be seen that the */robot_state_publisher* node, an already existing node in ROS2, is the responsible for publishing into the */tf* topic. Thus, it is necessary to start it and provide it the URDF. Note that, by running

ros2 param list /robot_stKate_publisher

, it is found the parameter *robot_description.* Furthermore, when run

ros2 param get /robot_state_publisher robot_description

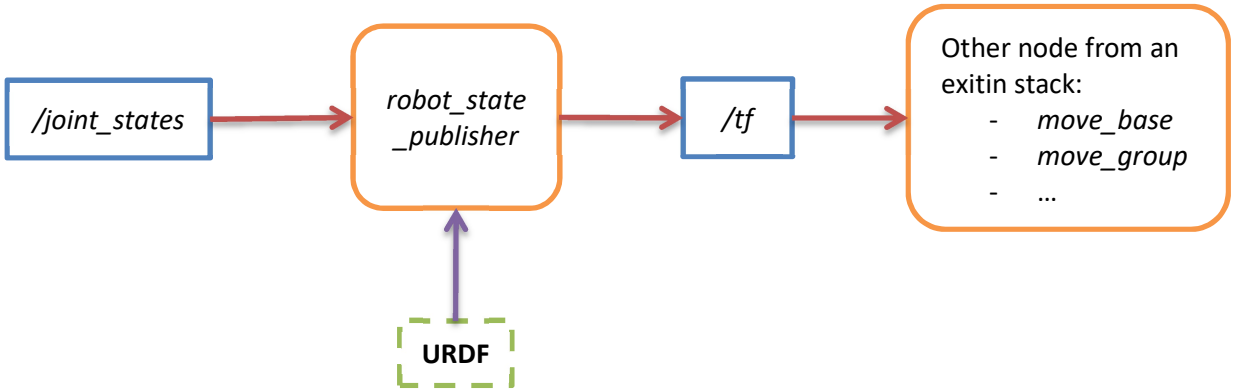
it is displayed the corresponding URDF file. That is, the URDF file is passed to the */robot_state_publisher* through the *robot_description* parameter. Note, additionally, that the */robot_state_publisher* will publish the URDF file to the */robot_description* topic,

ros2 topic echo /robot_description

As shown in *rqt_graph*, the */robot_description_topic* takes as second input the */joint_states*, where is stored the current state of every joint.

ros2 topic echo /joint_states

Therefore, it is necessary to activate the */robot_state_publisher* topic, pass it the URDF and the */joint_states* topic information; so that it will be able to compute the TF and publish it to the */tf* topic.



The command that allows to launch from the terminal the */robot_state_publisher* node with the corresponding URDF parameter is the following:

ros2 run robot_state_publisher robot_state_publisher --ros-args -p robot_description:= "$(xacro <path_of_the_urdf) "

If *xacro* is not installe, it can be done by

sudo apt install ros-humble-xacro

In the case of the */joint_states*, it will have to be run

ros2 run joint_state_publisher_gui joint_state_publisher_gui

Again, if not installed,

sudo apt install ros-humble-joint-state-publisher

Note that we recovered the node graph that we had and we wanted.

rqt_graph

To visualize the result it can be run

Ros2 run rviz2 rviz2

By default there is no configuration in RViz; so *RobotModel* and *TF* has to be added form the *Add* button on the left bottom corner, and *Fixed Frame* has to be changed to the proper one. Finally, to enable the robot model, the *Description* tag in *RobotModel* has to be changed to */robot_description*. Furthermore, the RViz configuration can be save from *File* tag in the upper left corner of the RViz window. There can be found the *Save Config As* option.

4.4.3. – URDF launch package

Let us build a workspace to handle and launch all the files needed:

```
cd
mkdir ros2_ws
cd ros2_ws
mkdir src
sudo apt install python3-colcon-common-extension
colcon build      Every time the workspace has to be build has to be done from the ros2_ws folder
source install/setup.bash
gedit ~/bashrc                  # Add at the end of it: ~/ros2_ws/install/setup.bash
cd src
ros2 pkg create robot_name_description
cd robot_name_description
rm –rf include/ src/
mkdir urdf                      # To store the urdf files
mkdir launch                    # To store the display launch file
mkdir rviz                      # To store the RViz custom configurations
```

Some changes have to be done to the *CMakeLists.txt*. It has to result like this one: https://drive.google.com/drive/folders/1_lRfdi_nJjhcdVYCjF0wiVVldM8wyoPQ.

Launch files can be written in different languages; the most standard in the ROS2 community are XML and Python.

cd ros2_ws/src/launch/
touch display.launch.xml
touch display.launch.py

Recommendation: Despite the fact that Python launch files seem to be more popular inside the ROS2 community, XML syntax is much easier and compact in this particular case. It is recommended to write the launch files using XML and import a Python file whenever Python functionality is needed.

Either way, here can be found both display lunch files: https://drive.google.com/drive/folders/1XU9bWOLrPATd_wQQI-swRoP-81S-h3J5.

A launch file is launched from the terminal using the command line

ros2 launch <pkg_name> <launch_file_name>

4.5. – XACRO

Xacro is a ROS2 feature that allows URDF files to be cleaner, modular, scalable and more dynamic; this is by adding *properties* (variables) and *macros* (functions) to the URDF, and enabling the inclusion of URDF files into other URDF.

Firstly, to make a URDF compatible with Xacro, the extension has to be changed to *.xacro* and the *robot* tag has to be left as follows

*<robot name="robot_name" xmlns:xacro=*http://www.ros.org/wiki/xacro*>*

Once this is added to a URDF file, the Xacro functionalities will be available and

--symlink-install

as well. To use expressions that are meant to be recognised by Xacro, they have to be written inside

${ }

For example, Xacro includes constants like the number π. It is accessed including

${pi}

4.5.1. – Properties

Xacro enables the creation of constants variables, called *properties*. They are usually added at the beginning of the file, and it is done the following way:

<xacro:property name="var_name" value="var_value" />

To access it, again,

${var_name}

4.5.2. – Macros

Xacro includes as well a functionality to create reusable blocks of code known as *macros*. They are implemented as follows:

<xacro:macro name="macro_name" params="a b c">

</xacro:macro>

, and everything that lies inside of the tag will be called when written:

<xacro:macro_name a=*"value1" b="value2" c="value3" />*

4.5.3. – Xacro file inclusion

Xacro also allows splitting a programme into multiple files by enabling the inclusion of URDF into another URDF.

Note: In the chosen as principal file the robot name has to be specified in the *robot* tag. However, since the other files will be included is this one, it is not required to add a robot name as an argument in the robot tag.

The sintax to include a URDF file into another is the following:

<xacro:include filename="file_to_inlcude" />

Here lies the previous files optimized after applying Xacro functionalities: https://drive.google.com/drive/folders/1WnoUh5B9bqkQAODr_V79W1VWSThBeGLL.

4.6. – GAZEBO

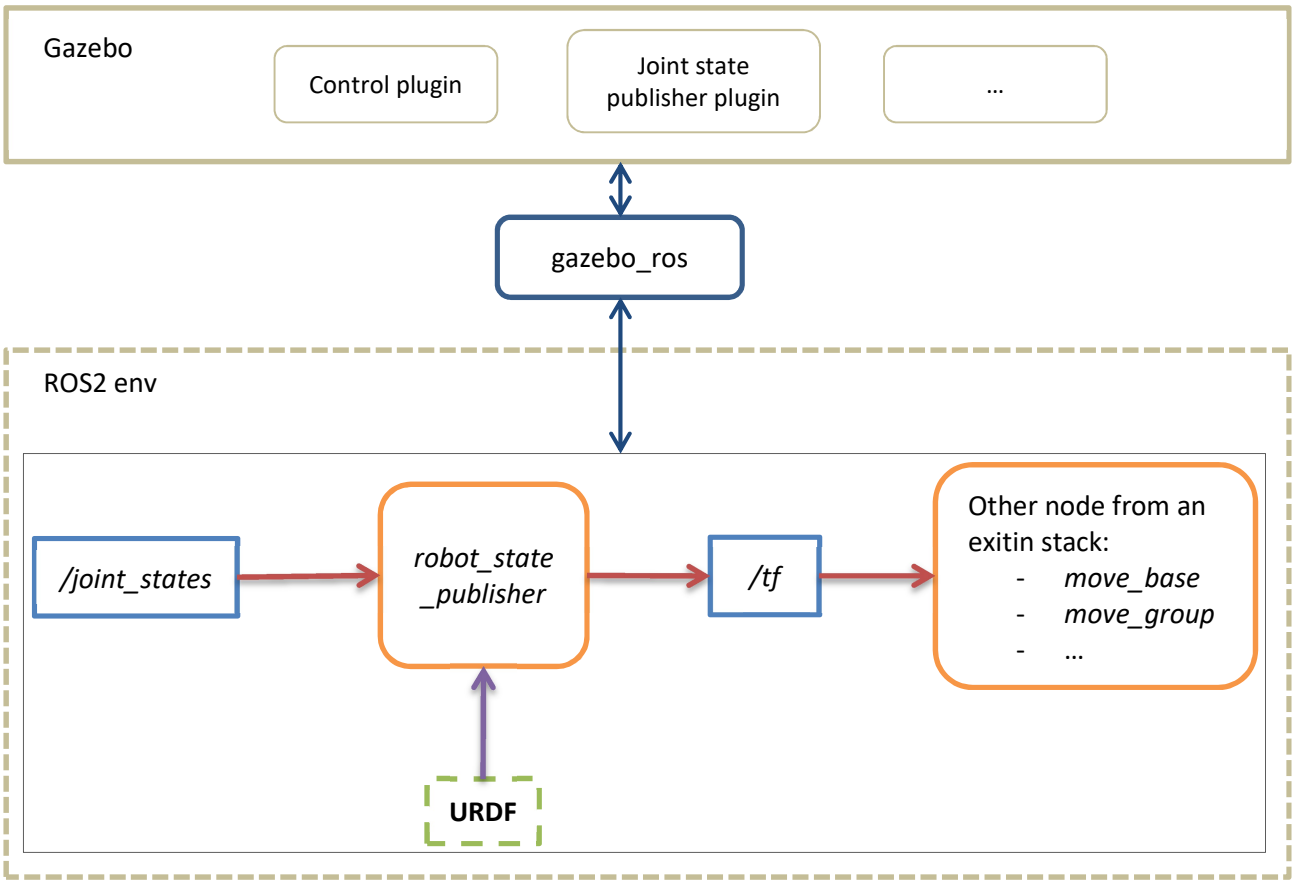
Simulating a robot in a realistic, fictional world—with gravity, friction, physical constraints, and more—is a best practice to prevent errors and ensure the proper functionality of the robot. Gazebo, in particular, is one of the most widely used simulation tools, not only within the ROS2 community but also in the TurtleBot4's.

Gazebo should have already been installed alongside the ROS2 environment, although it is, in fact, an independent tool. It can be opened by running in the terminal

gazebo

If opened, it will be shown a virtual empty world. Plugins to add objects and interact with them are placed in the left and upper side of the window's frame. Note that they behave as if they were in the real world; affected by real physical forces, like gravity.

Moreover, Gazebo and ROS2 are interconnected thanks to the *gazebo_ros* package, which allows Gazebo interact with all ROS2 functionalities.



Reciprocally, Gazebo adds plugins functionalities, which are susceptible to ROS2 communication tools. Plugins job is basically to simulate the hardware interaction with the robot.

4.6.1. – Inertial tags

The first necessary additions to adapt the URDF and enable a robot to spawn in Gazebo are *inertial* tags.

Each link has to be provided with inertia properties, so Gazebo can correctly simulate its behaviour. For their implementation, it will be needed the matrix of inertia of the solid object, that is of the link. In this link the list of 3D inertia tensors for basic solid forms can be found: https://en.wikipedia.org/wiki/List_of_moments_of_inertia#List_of_3D_inertia_tensors. If custom meshes are handled, the inertia matrix may be given by the used CAD software.

The *inertial* tags must be added in the *link* tags, and they are created as shown here: https://wiki.ros.org/urdf/Tutorials/Adding%20Physical%20and%20Collision%20Properties%20to%20a%20URDF%20Model (2.1 Inertia). Since it is not up to date, there is one tag missing. It has also to be added an *origin* tag

<origin xyz=" " rpy=" ">

Recommendation: Macros might be convenient to efficiently use *inertial* tags, since there might be more than one link using a certain type of geometry (Could be also place in a different folder, like one reserved for the robot common properties).

4.6.2 – Collision tags

Finally, in order to spawn a robot in Gazebo, it is necessary to add a *collision* tag to every link of it. In the *collision* tag has to be specified to Gazebo what is the shape of the object that is given the collision property; that is the hitbox which is given to the link. Thus, the collision shape is a simplified visual. If the visual is already a simple shape, it is common to copy the same figure into the *collision* tag.

The result once added the *collision* and *inertial* tags is the following: https://drive.google.com/drive/folders/1oSuxsyyyuN1Codc_mdzBouBJUNCHKKZY.

4.6.3 – Robot summoning in Gazebo

Once the URDF file of a robot is completely adapted, it can be launched in Gazebo running the following commands in the terminal:

- The *robot_state_publisher* is run with the corresponding URDF file:

ros2 run robot_state_publisher robot_state_publisher --ros-args -p robot_description:= "$(xacro absolute_path_of_the_urdf)"

- Gazebo is started through a launch file:

ros2 launch gazebo_ros gazebo.launch.py

- The *spawn_entity.py* node is activated with the proper arguments:

ros2 run gazebo_ros spawn_entity.py -topic robot_description -entity robot_name_in_the_urdf

Recommendation: In order not to do this whole process each time a robot wants to be launched in Gazebo, a launch file can be created. It is common to dedicate an alternative package to store the different launch files in a workspace, which is conventionally named *<robot_name>_bringup*.

Here is left the launch file to generate robots in Gazebo: https://drive.google.com/drive/folders/1QHPTCzgObL8acPOZ7ItKP7Bt_u-MtJsz.

Note: RViz might not load completely the robot model. This is because the *robot_description* is not publishing back into the joints (this could be seen using *rqt*). Later on, specifically once plugins are implemented, will not be a problem anymore.

4.6.4. – Fix of the colour and inertial values

Note that the colour of a robot is still not defined in Gazebo and, depending on the case, the robot might slightly move by itself.

The movement issue could occur due to different reason: maybe the contact surface is too small; the inertia is too small; the density of the shapes is not realistic; etc. However, it is often owing to week inertial values. It could be fixed, for example, rescaling the parameters passed, as shown here: https://drive.google.com/drive/folders/1oSuxsyyyuN1Codc_mdzBouBJUNCHKKZY.

To provide color to the robot in Gazebo, Gazebo specific tags referring the links to colour have to be added. A robot's Gazebo characteristics usually are gathered in a different file which then is included to the main one using Xacro functionalities. Here is shown how the colours are applied to the links in Gazebo: https://drive.google.com/drive/folders/1oSuxsyyyuN1Codc_mdzBouBJUNCHKKZY.

4.6.5 – Plugins

Plugins are used to control the robot; they allow sending commands to the robot that will eventually move the robot in Gazebo.

In the ROS2 documentation can be found the different available plugins: https://classic.gazebosim.org/tutorials?tut=ros_gzplugins.

Note: The plugins documentation is quite poor, and the previous is the only one despite being, in fact, outdated. Moreover, it is closer to explained example than to documentation. Ultimately, there is no defined recipe for plugins; the best way to deal with them is investigating and experimenting. As extra resource, here lie all the plugins' header files that can be used in Gazebo: https://github.com/ros-simulation/gazebo_ros_pkgs/tree/ros2/gazebo_plugins/include/gazebo_plugins. These files are generally updated and at the beginning of each one of them can be found, commented, an example of usage. From these can be extracted a decent plugin structure for quite a good amount of them. If something is missing and cannot be found, relaying on the community may be a good idea.

The *plugin* tag has to be added inside a *gazebo* tag. The plugin name is editable but the filename must not be changed. Inside the plugin is the required configuration, although may some fields may be lacking.

Here is an example of a plugin usage: https://drive.google.com/drive/folders/1oSuxsyyyuN1Codc_mdzBouBJUNCHKKZY.

The robot then can be moved running in the terminal:

ros2 topic pub /cmd_vel geometry_msgs/msg/Twist "{linear: {x: 0.5}, angular: {z: 0}}"

It has been added inside the *caster_wheel_link gazebo* tag these friction coefficients:

<mu1 value="0.1" />
<mu2 value="0.2" />

; in order for the movement of the robot to be smooth.

4.6.6. – Worlds

By default, the Gazebo generated virtual world is empty. Nevertheless, custom and reusable Gazebo worlds can be created.

Once Gazebo is opened, an *Insert* tag can be found on the left side of the window. After some time, some libraries will appear, and form there can be added in the environment. Moreover, in the *Edit* tag placed on the upper menu, between *File* and *Camera*, lies the *Building Editor* tag, which allows building walls and adding features, colour and texture to them. When the edition is finished, the world can be saved using the *.world* extension. Worlds could be gathered into a reserved for them folder inside the workspace (if done, it has to be added to the CMakeLists.txt).

The world can be used typing its name after the Gazebo command line. This is:

gazebo world_name.world

However, it can also be added to a launch file. Firstly, it will have to be ensured that the robot does not belong to the world, in order not to spawn it twice. Then, an argument will have to be added in the *include* tag of the launch file as it is shown here: https://drive.google.com/drive/folders/1QHPTCzgObL8acPOZ7ItKP7Bt_u-MtJsz.

4.6.7. – Sensors

Sensors can be also simulated in Gazebo. The way to implement sensors is, first, creating a link with its corresponding joint, which will work as the sensor physical part, and then, provide it with a certain Gazebo *sensor* tag and plugin, which will simulate its behaviour. Sensors are often developed in independent URDF files and included in the main one afterwards. Here is an example of a camera sensor: https://drive.google.com/drive/folders/1CKfhuPfVhKPKCyoFVUC1eLHnP8pSR5Cr.

4.7. – FINAL PROJECT

As a final project was built a complete mobile robot provided with a robotic arm on top, simulated in a custom world. Here can be found the complete final result of it: https://drive.google.com/drive/folders/1Lqpo8yxhmrnLDDe5yu6p5V5FWQNqcq0r.

Recommendation: When modeling more complex robots compound by independent parts, it is handier to build each one of the parts as an independent robot itself and, afterward, merge them to obtain the required result.

RESOURCES USED

- ROS2 User Manual: https://turtlebot.github.io/turtlebot4-user-manual/
- ROS2 Humble webpage: https://docs.ros.org/en/humble/
- ROS2 For Beginners (ROS Foxy, Humble - 2024) by Edouard Renard: https://www.udemy.com/course/ros2-for-beginners/
- ROS2 For Beginners Level 2 – TF | URDF | RViz | Gazebo by Edouard Renard: https://www.udemy.com/course/ros2-tf-urdf-rviz-gazebo/?couponCode=SKILLS4SALEA
- Drive folder with all the ROS2 programmes: https://drive.google.com/drive/folders/13GPDZOQidCi9oKH1mmPxQuI6n3jhK60K.
- Drive folder with all the simulation programmes: https://drive.google.com/drive/folders/1OxE6jL9rvpSaqq4c6FpLPMjgZWD18A8n.
- TurtlBot4 Github link: https://github.com/turtlebot/turtlebot4.