POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Autonomous Navigation for Quadruped Robots: Development and Optimization on the Unitree Go1 Platform

Supervisor
Prof. Elisa Capello

Advisor at LINKS Foundation
Dott. Francesco Aglieco
Dott. Gianluca Prato

Candidate
Gabriele Caruso

307974

December 2024

Abstract

The growing adoption of service robotics in complex environments such as hospitals, industrial facilities, and public spaces highlights the need for versatile autonomous navigation. While some studies have addressed this challenge for specific robotic platforms, the application of such capabilities to different forms, like quadruped robots, remains relatively unexplored.

This thesis, developed in collaboration with the LINKS Foundation, aims to integrate an autonomous navigation system for Unitree Robotics' Go1 quadruped platform, using ROS 2 (Robot Operating System) and its Navigation 2 stack.

The project was structured in two main phases. The first phase focused on the general setup of the platform, with the aim of enabling communication between the different computers on board the robot and externally, allowing the remote execution of the software on external devices or in the cloud. Subsequently, the installation and configuration of the ROS 2 operating system was carried out on the robot, thus enabling the control of the robot itself and the readout from on-board sensors, including lidar, video camera and inertial sensors (IMU).
In the second phase, the installation and configuration of the Navigation 2 stack was performed to enable autonomous robot navigation. Therefore, the system was implemented on the real robot to empirically evaluate its performance and autonomous navigation capabilities under real operating conditions.

The results obtained confirm the achievement of the set objectives, laying the groundwork for future applications of quadruped robots in complex environments.

Three Laws of Robotics:

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey any orders given to it by human beings, except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Handbook of Robotics, 56th Edition, 2058 A.D.

Acknowledgements

Here we are. This thesis represents not only the culmination of two years of intense work and dedication, but also the conclusion of an important academic chapter at the Polytechnic University of Turin."

My first and most heartfelt thanks go to my wonderful fiancée, Margherita, who has supported and patiently endured me for years. To her I owe the strength and motivation that pushed me to achieve my goals. Thank you for always being by my side, sharing with me not only the joys but also the defeats.

Special thanks go to my family. To my parents, who never stopped encouraging me every day to give my best and continue my studies until the end, helping me with difficult and decisive choices. To my brothers who have always cheered for me all these years.

A heartfelt thank you to my fellow travellers, Jiahao and Iris. With you I have shared challenges, successes and countless moments of complicity. Your companionship, support and many laughs together have made this journey so much richer and more meaningful.

Special thanks to my academic supervisor, Prof. Elisa Capello, for the guidance, support, helpfulness and valuable advice that helped me grow during this project.

A heartfelt thanks also to Dr. Francesco Aglieco and Dr. Gianluca Prato, who have followed me with professionalism and availability, helping me to successfully face every challenge of this work.

Finally, a big thank you to Links Foundation, for offering me the opportunity to work on this thesis project and for providing me with the robotic platform that was the heart of it.

To everyone who stood by me, whether mentioned or not, goes my deepest and most sincere thanks: your support made all the difference.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this first chapter, the concept of robotics is introduced, with a detailed analysis of mobile robots. The key issues of mobile robotics are considered and the different categories of mobile robots are listed, with particular emphasis on quadrupedal robots. In addition, the technical specifications of the platform used in this project are outlined. Finally, the *Uncanny Valley* theory, an interesting hypothesis formulated by Japanese engineer Masahiro Mori, is discussed.

1.1 What is a robot?

Robotics aims to create machines that can replace humans in performing tasks, both physically and in decision-making.
The term '*robot*' was coined in 1920 by Czech writer *Karel Capek*, however, the concept of the anthropomorphic machine has much deeper and older cultural roots. Man has always sought to bring his creations to life, creating myths such as the legend of the *gigant Talo*, a living statue created by Hephaestus to defend the island of Crete, or the *Golem*, an imaginary figure from Jewish and medieval mythology. A clay giant with no intellectual faculties, the *Golem* was endowed with extraordinary stamina and strength, employed to perform heavy labour, carried out his creator's orders to the letter, but was incapable of thinking, speaking and feeling any kind of emotion because he had no soul.
In the modern Hebrew language *Golem* also means robot.

Moreover, in more recent times, a great number of science fiction literary and film narratives have conceived of humanoids or androids capable of autonomous interaction with humans.
The most important text is recognised in the narrative of *Isaac Asimov*, who conceived the robot as an automaton programmed to process information and organise certain rules of behaviour.
Asimov is acknowledged for the fundamental ethical principles for determining the behaviour of robots in their relationship with human beings aimed at ensuring their safety and well-being in human/machine interactions.

Famous is his enunciation of the **Three Laws of Robotics**, formulated as follows: [2]

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey any orders given to it by human beings, except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

These laws, taken as the cornerstone of science-fiction literature, have also helped to define the relationship between man and machine, inspiring ethical and philosophical postulates that go far beyond the boundaries of fiction, and have also become a fundamental reference for the development of artificial intelligence.

However, robots are not just a literary or fictional concept. The first practical applications date back to the 1950s and 1960s, when the first industrial robots were designed, such as the **Unimate**, a robotic arm used in the automotive industry's assembly lines. In the years that followed, technological progress enabled the creation of increasingly sophisticated machines capable of performing complex tasks and adapting to changing situations.

Two advanced humanoid robotics prototypes are now attracting great interest: **ASIMO** from *Honda* and **Atlas** from *Boston Dynamics*. They are designed to mimic the movements and movement capabilities of human beings. These robots are able to walk, run, jump and even perform complex actions such as climbing stairs.
The development of humanoid robots is not only of scientific value, it also serves to study and improve the understanding of human movement by designing machines that can assist people in domestic or care settings.

Moreover, with the advent of artificial intelligence, modern robots are able to make complex decisions autonomously, thanks to advanced neural learning and computer vision systems. This has led to the emergence of robots capable of interacting with humans in a more natural way, interpreting voice commands and responding to their emotions through facial recognition and other sophisticated technologies.

Nowadays, robots are not only a common topic for stories and tales, but also find applications in many civil and industrial sectors, improving efficiency, safety and quality.
In industry, for example, robots are used to perform repetitive and dangerous tasks, achieving accuracies that often exceed human precision and significantly improving safety and productivity levels; also used for logistics and warehouse management, they are able to speed up the internal transport of goods and reduce delivery times.
Agriculture also benefits, with the introduction of robots for sowing, weeding and harvesting, using drones to monitor crops and plantations from above, allowing less labour to be used and targeted and intelligent action to be taken where it is most needed.
There are applications in the healthcare sector, revolutionising surgery (such as the famous *da Vinci* system), transportation of medicines, rehabilitation and prostheses for patients.
Robots find applications in everyday services and assistance, in the transport sector, in exploration and research, in the energy sector and in construction, always improving, speeding up and helping humans even in extremely delicate operations.
The aim of robot automation is not to replace humans by taking away their work, but

to help and collaborate by reducing, if not eliminating, risks and accidents at work and allowing human qualities to be used in other areas.

As can be imagined, the variety of applications has led to the development of different types of robots, such as conventional manipulators or mobile robots.
Manipulators usually have a fixed base, so they work in a fixed, defined space. In contrast, mobile robots are able to move in their surroundings and thus have multiple applications.

1.2 Mobile robots

This type of robot today represents the category that is developing with the most speed and interest. In contrast to manipulators, which have less control complexity and which initially diffused more quickly, mobile robots are receiving more trust and attention; this is because, over time, new programming methods and navigation tools have been developed that have facilitated their use.
Nowadays, efforts are being made to give this category a greater ability to move autonomously and react to the changing environment.

Mobile robots have to answer three important questions [1]:

1. **where** am I?
2. how am I **supposed** to get to the goal?
3. how do I **actually** move?

1.2.1 Localization

In order to answer the first question, it must first be taken into account that the environment is dynamic and therefore the state - *the set of all aspects of the robot and environment that may have an impact on the future* - is uncertain. There is uncertainty because not all environmental information can be detected directly, sensors are imperfect and therefore have noise and the robot can influence the environment with its actions. Talk about *probabilistic robotics* where the idea is to estimate the state from sensor data.[23]

In order to be able to obtain a good approximation of the robot's current state, the concept of **belif** was introduced, which represents the robot's *estimation of its actual state*, based on the data at its disposal.

$$bel(x_t) = p(x_t|z_{1:t}, u_{1:t}) \tag{1.1}$$

Where x_t is the state of the robot at time t, and depends on past measurements $z_{1:t}$ and past actions $u_{1:t}$.

The robot's belif can be estimated using mathematical filters such as the Bayes filter, the Kalman filter, the EKF (extended Kalman filter) and the UKF (unscented Kalman filter).
The former is a general method for updating probabilistic state estimation based on

[1] Durrant-Whyte 1991; slightly revised

observations, but is computationally too heavy for continuous or complex systems, so it is rarely used. The Kalman Filter is an efficient approach for linear systems with Gaussian noise; it works well in simple contexts, but is not suitable for non-linear systems. The problem of non-linearity is solved with the EKF, which is an extension of the Kalman filter; it is more accurate in the presence of non-linearity, but does not handle highly non-linear dynamics well. For this we have the UKF, an advanced variant that uses sigma points to represent the state distribution without linearising, improving accuracy in non-linear systems compared to the EKF.

AMCL

AMCL (Adaptive Monte-Carlo Localiser) is a probabilistic localization system that uses a particle sampling and **resampling-based filter** to trace, against the map, the 2D pose of the robot. The algorithm works by representing the robot pose as a distribution of particles representing possible robot poses.

The robot starts with an approximate and sparse localisation, but this improves dramatically as it measures with its sensors at each time step and compares the results with a map of the environment, in order to calculate the probability that each particle represents the robot's actual location. Then, a new set of particles is created for the next time step by resampling those with high probability, while those with low probability are eliminated.

Although the robot's movement is unpredictable or the surroundings are only partially visible, the resampling procedure ensures that the particles remain dispersed around the robot's actual posture.

Position Tracking and SLAM

The robot localization can be done given an a prior known map, where the pose of the robot with respect to the map frame at time t has to be determined. The case where the initial pose of the robot is known (**Position Tracking**) can be had, so the uncertainty will only be confined to the local region around the true robot pose. However, if the initial pose of the robot is unknown (**Global localisation**), the situation will be more complex.

In the scenario where the map is not known a prior, the robot has to construct a representation of the environment while simultaneously estimating its own position within it. This process is known as **SLAM** (Simultaneous Localisation and Mapping), and consists of creating a map of the unknown area as the robot moves, continuously updating its own location in relation to the detected features. SLAM is essential for autonomous movement in unmapped or dynamic environments. It is a difficult approach because a map is needed to locate the robot and a good pose estimation is necessary for proper mapping.

1.2.2 Path planning

The second question is answered by Motion Planning. This involves finding the best trajectory to reach the target as quickly as possible without colliding with fixed or moving obstacles.
First of all, to proceed with path planning, the map must be discretized, and there are two general approaches: the **combinatorial planning** technique, which subdivides the

space into a finite set of cells, representing the planning problem as a search on a graph of these connected cells, allowing search algorithms to be applied to find optimal paths, and the **sampling-based planning** technique, which uses sampling techniques to explore the space of configurations, generating random nodes in the domain and constructing a graph of connections between these nodes to find valid trajectories.[14]

Once the map has been discretized and all possible paths, that can be executed by the robot, have been found, it is necessary to choose which path is the best. To achieve this, there are several methods and algorithms to choose from, which differ according to their effectiveness in finding the best path and the computing power required by the machine. The best path is defined as the path that allows the robot to reach the target in the shortest time possible while paying a low cost of movement. The movement cost depends on environmental factors such as slope and road conditions, which affect the robot's battery life, movement speed and platform integrity.
The **Dijkstra's algorithm** works well to find the fastest route, but consumes a lot of time exploring all directions, even the not promising ones, and thus consumes much more CPU power. Whereas the **Gready Best-First Search** explores heuristic-driven directions, which usually results in not finding the shortest path, but consumes less CPU power. The **A* algorithm** proposes a middle way between the two algorithms outlined above, combining the path cost (as in Dijkstra) with a heuristic function (as in Gready Best-First Search) to guide the path to the destination; this approach reduces the computational consumption by prioritising explorations in promising directions, while at the same time increasing the probability of finding the best path.

Figure 1.1. Comparison of the three algorithms. [14]

1.2.3 Motion control

To answer the last question, it is necessary to work in hardware and low-level programming, controlling the robot's actuators and motors. Motion control deals with the problem of determining a sequence of input controls (motor voltages or currents) in order to obtain a motion that allows the robot to reach desired positions and orientations.
These positions are typically provided by path planning and then the motion control goes to manage the forces and velocities required to reach the goal.

Motion control in a mobile robotics system is divided into three stages:

- **Speed and position control**: controls the activation of motors to achieve and maintain required speeds and positions.
- **Feedback loop**: uses sensors, such as LiDAR, ultrasonic or encoders, to constantly monitor the status and, if necessary, correct deviations from commands.
- **Dynamics management**: to improve the accuracy and stability of movement, consider the physical characteristics of the robot (mass, inertia) and how it interacts with the environment (friction, slopes).

1.2.4 Locomotion methods for mobile robots

Mobile robots can be classified according to their capabilities and modes of movement, including different types of means of locomotion that make them suitable for various environments and specific tasks. The main types of mobile robots include:

- **Wheeled robots**: are among the most common and easiest to control. They are used for ground movements, adapting well to smooth, well-defined surfaces such as industrial floors or roads. Examples can be found in robots for autonomous deliveries, robots for logistics and robots for domestic cleaning.
- **Tracked robots**: Robots similar in control to wheeled robots, but with the use of tracks that allow them to have a better grip on difficult and uneven terrain such as gravel, sand or slippery surfaces. They are robots often used for search and rescue missions.
- **Underwater robots**: designed for underwater navigation, they are able to move using propellers or other propulsion systems. They are used for environmental monitoring, underwater maintenance or scientific exploration.
- **Aerial robots**: known as drones or UAVs (Unmanned Aerial Vehicles), they fly and move by means of propellers or fixed wings. They find applications in a variety of situations, from agricultural monitoring to parcel deliveries.
- **Legged robots**: Designed to emulate animal and human movement, they are able to walk over rough and uneven terrain where wheels or tracks would fail. They are robots with significantly greater control complexity. They are in turn subdivided into bipedal, quadrupedal or multi-legged robots.

1.3 Quadruped robots

With regard to stability and mobility, four-legged robots are the best choice; robots with two or more legs are more difficult to control than four-legged robots. Since the early 1900s, many scientists have dedicated themselves to the development of four-legged locomotion. In 1870, Chebyshev developed the first mechanism capable of moving by converting rotational motion into translational motion at constant speed, as shown in Figure 1.2. This device was only able to move on level ground and the legs were not independent.[3]

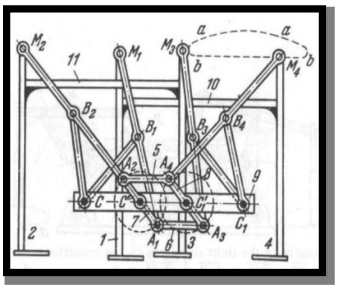

Figure 1.2. Chebyshev Mechanism. [3]

Nowadays, things have changed a lot, leading to much more complex robots, such as Spot from Boston Dynamics or Go1 from Unitree.

1.3.1 Unitree Go1 Edu

The Unitree Go1 Edu is an advanced mid-sized quadruped robot model presented in 2021 by the Chinese company Unitree. It is a model designed for educational, research and development applications that offers more advanced customisation than other models in the Unitree Go1 line.

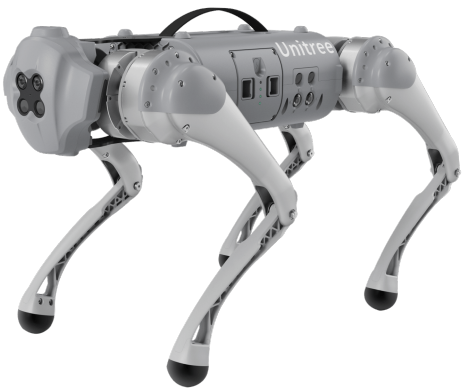

Figure 1.3. Profile view of Unitree Go1. [19]

Design and Mobility

The Unitree Go1 Edu was developed to mimic the stability and walk of a quadrupedal animal, enabling it to move with ease across different terrains. Its robust and compact mechanical structure allows the robot to easily perform complex movements such as walking, running and turning around. It is able to tackle moderate slopes and reach speeds of around 3.5 m/s thanks to its highly efficient brushless motors and advanced stabilisation.

Advanced sensor system

The Go1 Edu is equipped with many sensors that ensure safe navigation and accurate spatial perception. These include:

- **Wide-angle stereo cameras**: useful for providing a view of the surroundings. They are distributed on all sides of the robot (front, side and rear).
- **Motion and position sensors**: help keep the robot balanced and stabilise its movements on uneven surfaces, providing continuous information to the controller. They consist of proximity sensors and inertia measurement units (IMU).
- **Sonar sensors**: useful for identifying potential nearby obstacles and increasing safety when moving in enclosed spaces.

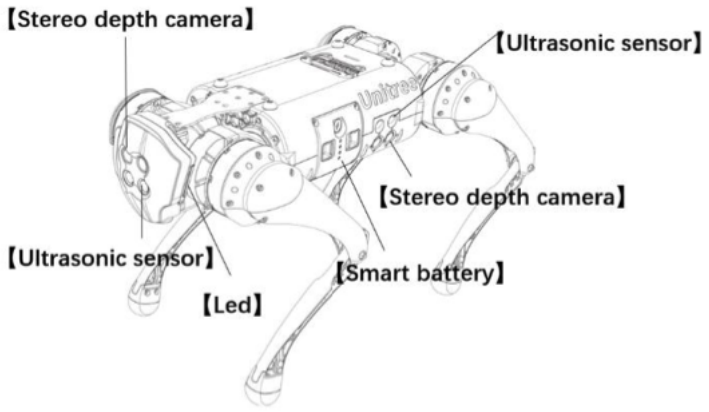

Figure 1.4. Robot sensor location. [19]

Computational power and load capacity

In contrast to the other versions of the Go1, which are more oriented towards the non-professional consumer, the Go1 Edu includes more powerful computers inside, capable of achieving considerable computing power, such as the **NVIDIA Jetson Nano** and **Jetson Xavier NX**, which are ideal for real-time artificial intelligence and computer vision applications. This allows the programmer to implement complex algorithms such as visual recognition, tracking and autonomous navigation.
The Go1 Edu is able to carry loads of up to 5 kg, allowing it to transport additional thinking sensors such as 3D LiDAR.

Programming skills and SDK

The Edu model allows low-level access to all hardware components and sensor data through Unitree's software development kit (SDK), making it fully programmable and customisable. This gives more freedom in being able to modify the motion control of the quadruped robot, abandoning the default walk used by Unitree. This SDK is optimised for programming environments such as ROS, allowing developers to integrate the robot platform into more complex development projects.

The SDK developed by Unitree Robotics for the Go1 can be found in the package `unitree_legged_sdk`.

1.4 Uncanny Valley

In 1970, Masahiro Mori, a Japanese robotics researcher, published an article in the journal *Energy* in which he explored the possible reactions of people to robots with near-human appearance and behaviour. Mori suggested that as a robot became more and more human-like, people's emotional response would quickly shift from empathy to discomfort. This phenomenon is known as the "*uncanny valley*". Although his work initially received little attention, it has gained increasing interest in recent years, both in the field of robotics and in popular culture, thanks to technological advances that allow the creation of increasingly realistic-looking robots.[13]

Uncanny Valley describes a negative emotional response that humans experience when they observe a robot that closely, but not completely, resembles a human being or an animal. The more similar the robot is to a living being, the more comfortable we feel with it - up to a certain point. A resemblance that is too close, but imperfect, triggers a feeling of unease or repulsion. This is the moment when one finds oneself in the "*uncanny valley*".
The Uncanny Valley is a two-axis graph:

- On the X-axis is the level of similarity to the human (or animal),
- On the Y-axis is the level of emotional affinity we feel for that object or robot.

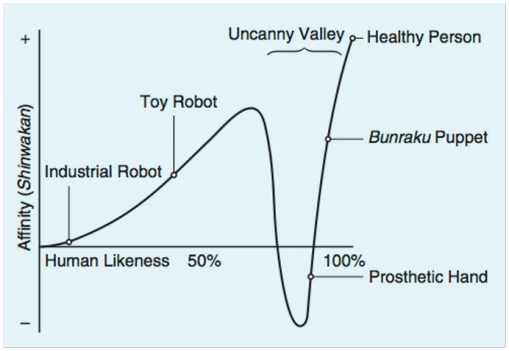

Figure 1.5. Uncanny Valley of a stationary robot. [13]

As can be seen from the Figure 1.5, at first, as a robot becomes more like a living being, our empathy increases. But when it reaches a point where it is almost perfect, but not quite, it collapses dramatically, causing a feeling of unease. This is the uncanny valley. If the resemblance continues to increase until it becomes indistinguishable from a living being, the emotional affinity rises again.

1.4.1 Importance of movement

Movement is a crucial element for animals, including humans, and is also of significant importance for robotics. Its presence alters the shape of the disturbance valley graph, as illustrated in the Figure 1.6, amplifying the peaks and valleys. In particular, an idle industrial robot is perceived as a featureless machine, while the implementation of movements

that mimic those of a human hand generates such an affinity that it elicits empathy from observers. For this positive effect to manifest itself, it is crucial that the robot's kinetic characteristics - such as speed, acceleration and deceleration - are designed to approximate human movement parameters.

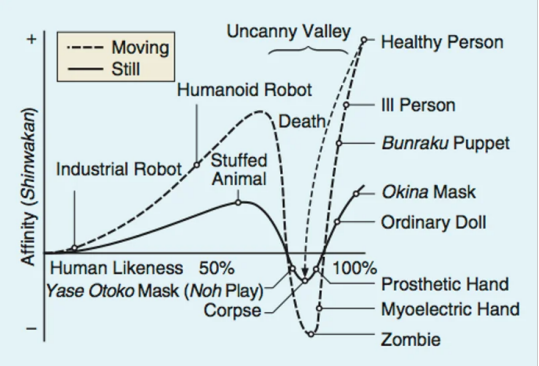

Figure 1.6. Uncanny Valley of a moving robot. [13]

1.4.2 Application of the Uncanny Valley to robot dogs

In the specific case of robot dogs, the Uncanny Valley concept can be applied in the same way. A robot dog that has clearly robotic features, such as the Unitree (GO1) robotic dogs, might be perceived as nice or interesting because we are not trying to compare it with a real dog: it is clearly a technological device. However, if a robot dog had very realistic features - synthetic fur, fluid movements, realistic eyes - but was not able to behave exactly like a real dog (for example, if its movements were slightly out of synchrony or its reactions were not natural), it might cause a feeling of unease.

Why is this happening?

- **Unfulfilled expectations**: If we see a robot dog with very realistic traits, we expect it to behave exactly like a real dog. When these behaviours are imperfect, our brain perceives an inconsistency that generates unease.
- **"Almost" natural signals**: When we see something similar to a real dog, our brain activates the same circuits that we use to interpret real animals. If these signals are inconsistent (perhaps an unnatural movement or facial expression), the result is a feeling of perturbation.
- **Emotional ambiguity**: We are naturally programmed to react empathetically to animals, especially dogs. A robot dog that seems "almost real" can trigger confused emotions: we do not know whether we should feel affection or detachment.

Unitree's Go1 robot has a clearly mechanical appearance, so it does not fit into the Uncanny Valley. It does not try to look like a real dog, but is designed to perform useful tasks and has a utilitarian design. It does not provoke an uncomfortable reaction because it does not try to imitate a real dog.

Chapter 2

Background Technologies

This chapter describes and explains the primary software tools that will be used in the remaining chapters of this project.
The following tools have been chosen based on specific needs of the project to make sure we end up with a stable, flexible and scalable environment for development and executions.

2.1 Docker

Docker is an open platform that helps to develop and deploy apps easily. It enables rapid software delivery through its ability to isolate applications from the infrastructure.[5]
As a result, developers can build software much more quickly. Infrastructure can be treated similarly to applications: it can be packaged, published, made available, controlled and distributed. This allows companies to minimize the interval between when code is written and when it is deployed.

2.1.1 Docker platform

Docker allows an application to be packaged and run in a container, a free and independent environment. These features provide security and isolation for multiple containers on a given host.
They are lightweight and contain everything you need to run the application contained within them (no dependency on what is packed on the host). You can share the container as you work and know that it always runs the same way, regardless of where you run the application.

2.1.2 Usage benefits

The use of Docker has been a definite advantage in that it has facilitated the use of ROS 2 on machines running older versions of Ubuntu, thanks to the isolation offered by containers. This allowed the creation of an environment compatible with the ROS 2 release, greatly limiting the complexity of processes and configuration time.

Docker offers the advantage of simplifying development by isolating environments, avoiding conflicts between dependencies and adapting to any system. The portability of containers allows the migration of applications between different platforms, the lightness of containers favours the execution of multiple operations on the same hardware, improving productivity.

Docker also makes the automation of continuous integration and deployment (CI/CD) flows more effective by optimising the release of applications and reducing error margins. The management and reliability of processes is ensured by the container's isolation, and its scalability allows applications to be quickly adapted to load requirements, making them effective for architectures and micro-services.
Systems such as Docker Compose and Kubernetes make it easier to manage complex applications. Docker in particular lends itself as a versatile platform that ensures efficient and reliable developments, which favour its use by developers.

2.1.3 Docker architectures

Docker uses a client-server architecture, where the client communicates with the Docker daemon using a communication via a REST API, using a UNIX socket. Both can be run on the same system, or by connecting the client to a remote daemon.

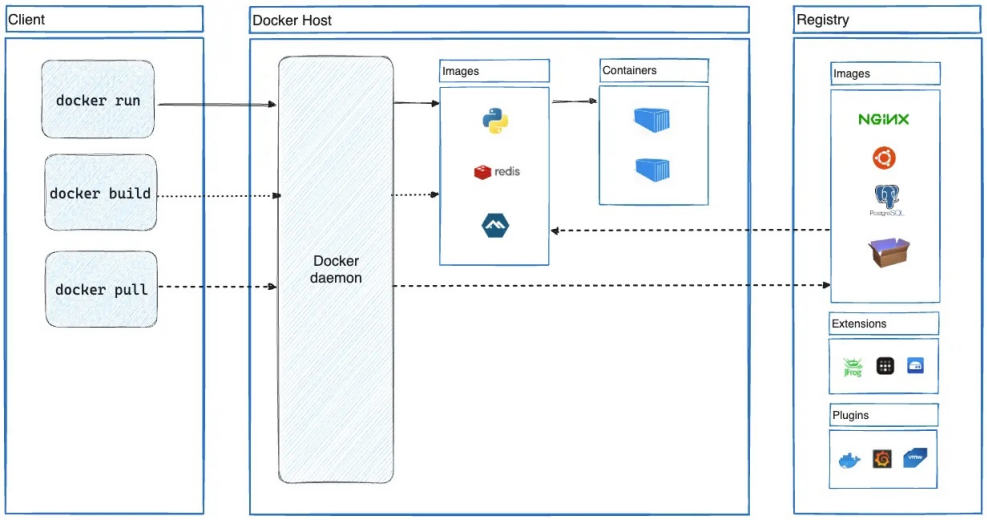

Figure 2.1. Docker architecture. [5]

Docker daemon

The Docker Daemon is the core of the architecture. It receives and processes requests from the client, managing the creation, execution and stopping of containers. It also handles the management of images, volumes and networks.

Docker client

The Docker Client is the main interface through which users can interact with Docker. When executing commands such as `docker run`, the client sends this command to the daemon which executes it. This separation of client and daemon allows Docker to be managed on remote machines, making the architecture flexible.

Registries

These are repositories containing Docker images. They can be public, like Docker Hub, or private, which companies can configure independently.

Docker objects

These are fundamental elements that Docker uses. Among the main ones are:

- **Images**: are read-only templates that contain what is needed to create containers. Images often originate from other images, but with additions. For example, a ROS 2 Humble image will be based on a Ubuntu 22.04 LTS image with Humble installed. Images can be downloaded from the Registries or created via Dockerfile, so as to have a more personal customisation.
- **Containers**: are active instances of images. They can be created, executed, stopped, transferred or deleted easily. Containers by default are isolated from other containers and the host machine, but this can change according to your needs, connecting the container to different networks, storage or allowing it access to hardware such as USB inputs to manage external devices.

2.2 ROS 2

Robot Operative System (ROS)[12], contrary to what the name might suggest, is not an operating system, but a set of software frameworks for robot software development.

Definition taken from the official ROS website:

> *"The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications. From drivers and state-of-the-art algorithms to powerful developer tools, ROS has the open source tools you need for your next robotics project."*

2.2.1 History of ROS

It was born in the early 2000s as a personal project of two Stanford University students, looking for a solution to the problem of having to '*reinvent the wheel*' that robotics was suffering from. Too much time was being wasted re-implementing the software infrastructure needed to build complex robotic algorithms (drivers for sensors and actuators, communications between different programmes within the robot) and too little time building intelligent robotic programmes based on that infrastructure.[22]

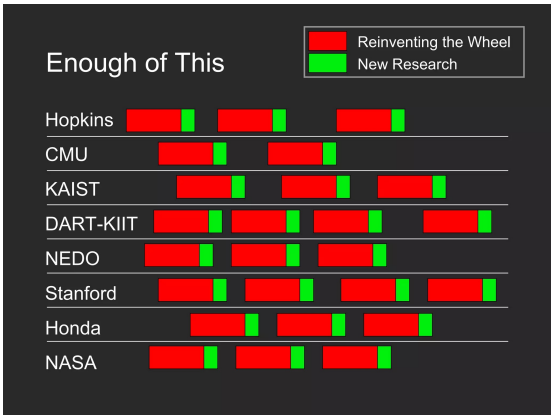

Figure 2.2. Time spent by robotics in reinventing the wheel (slide from Eric and Keenan pitch deck) [24]

In an attempt to remedy this situation, the two students set out to create a basic system that would provide an open source starting point on which others in academia could build.

Initially under the name Stanford Personal Robotics Program (SPRP), the system evolved year by year until it reached its first ROS distribution in 2009: ROS Mango Tango (ROS 0.4). With subsequent versions, ROS has been considered the standard in robotics since 2013.

But ROS carried with it important limitations, due in part to the strictly academic nature that had characterised its birth; limitations that became important when it wanted to proceed to industrial applications. For this reason, the first distribution of ROS 2.0 was released in 2017, with the aim of exploiting the potential of ROS 1 and implementing it.

Like its progenitor, ROS 2 continues to evolve year after year, always releasing new distros (distributions) with an end-of-life (EOL) date. This approach ensures continuous evolution in terms of security, resource management and compatibility with the original operating system. EOL implies that, after that date, the distro will no longer receive updates from the community on bugs, security and stability improvements, and new tools useful for robotics programming will no longer be released. A clear example is Navigation 2, a navigation system compatible only with ROS 2, or Zenoh, a middleware protocol compatible only from the Jazzy distro.

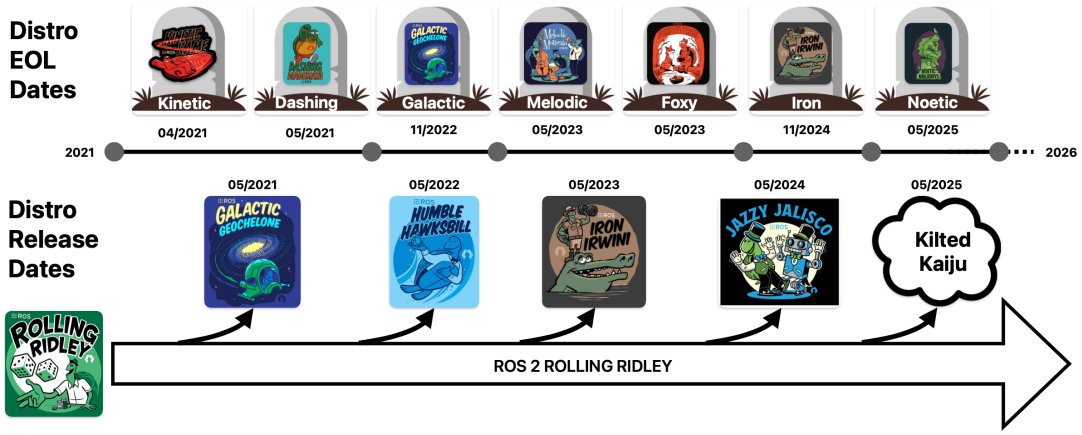

Figure 2.3. ROS Distros (REP-2000) [15]

For this project, the **Humble** distro was initially used. It then had to migrate to **Jazzy** for reasons that will be explained in more detail in Chapter 4.

2.2.2 Introduction to ROS 2

ROS 1 reached its last version with ROS Noetic (2020) and will reach its EOL (end of life) in 2025; this is one of the reasons why ROS 2 is the preferred choice for new long-term projects.

ROS 2 represents a significant advance in the field of software development for robotics. It meets modern requirements for security, scalability and interoperability. It is an open-source middleware designed to ensure a reliable and modular structure that meets the complex requirements of robotics engineering, from industrial automation to autonomous transport systems.

ROS 2 not only retains the ease of development and management of robotic systems of its predecessor, but at the same time overcomes many of the limitations previously encountered.

Main innovations

- **Multi-Threaded Execution**: ROS 2 supports multiple nodes running in parallel, allowing modern multi-core processors to be fully utilised.
- **ROS API**: In ROS 1, two specific independent libraries are used to provide an API to develop ROS nodes, roscpp is used for programming in Cpp, and rospy for Python. In ROS 2 you have multiple layers (see Figure 2.4). You have a single base library, called rcl and implemented in C, which contains all the main functions of ROS 2. In ROS 2, when writing a programme, you do not use the rcl library directly, but use another built on top of rcl. For programming in Cpp one uses rclcpp, for Python rclpy and so on. The advantage of ROS 2's layer structure lies in the fact that for each new feature you only have to implement it with rcl, and then only provide blinding for the client libraries.
- **Middleware Changes**: ROS 1 employs a Master-Slave architecture and XML-RPC middleware, which includes a customised serialisation format, a specific transport protocol and a centralised discovery mechanism. In contrast, ROS2 employs the Data Distribution Service (DDS), an abstracted middleware interface that manages serialisation, transport and discovery, designed for greater efficiency, reliability, low latency and scalability, with configurable Quality of Service (QoS) parameters. XML-RPC is suitable for simple remote procedure calls, while DDS better supports real-time systems and eliminates single points of failure in ROS2 communications.
- **Communication**: In ROS 1, in order for nodes to communicate with each other, they need the ROS master, which acts as a DNS server so that nodes can call each other. In ROS 2 you no longer have a centralised system, each node can discover the other nodes without a central DNS. With ROS 2 you have a distributed system where each node is independent.
- **QoS**: ROS 2 introduces quality of service, which allows the user to configure the way data is sent and received, affecting the flow of data. Among the available options are settings for reliability, expiry times and message priority. These configurations help ensure that crucial messages are delivered on time.
- **Security**: With the use of DDS, in addition to the above benefits, one has a protocol that provides security guarantees not present in ROS 1, which allows robots to communicate over unsecured networks (such as the Internet). You have authentication mechanisms that allow nodes to verify the identity of the communicating process so that unauthorised access and loss or tampering of data is not possible.

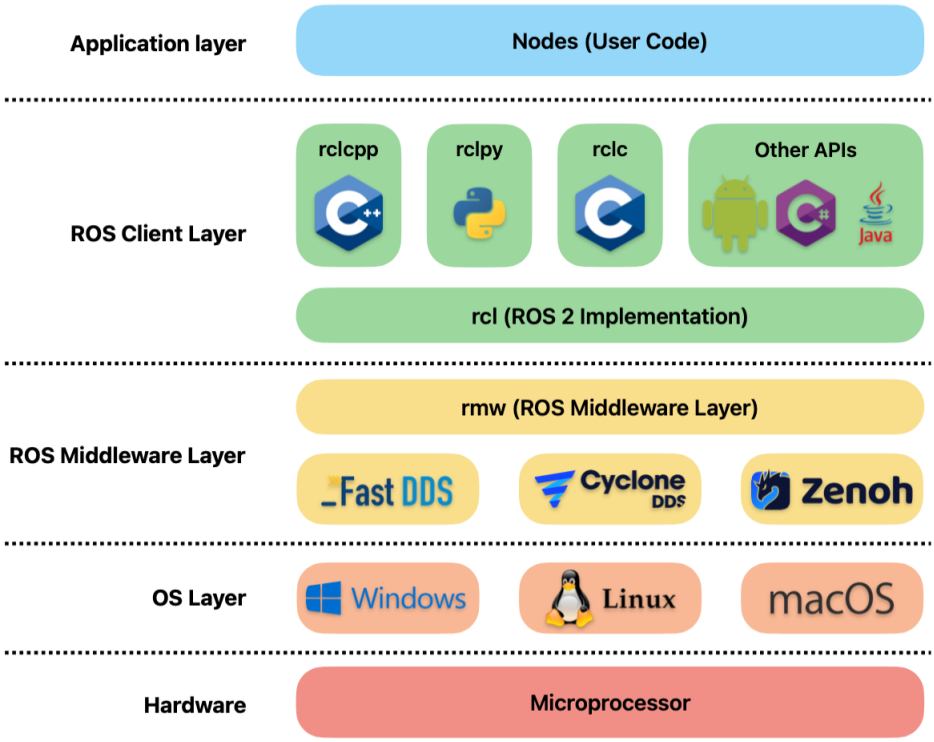

Figure 2.4. ROS 2 software layers

2.2.3 Computational Graph

The ROS 2 framework is a middleware architecture composed of a series of distributed nodes that communicate with each other by means of messages. The ROS 2 *Computational Graph* represents this network of interconnected nodes, which interact with each other using different communication paradigms to decompose complex problems into simpler ones.

Nodes

The *nodes* are the primary execution elements in ROS 2, within which code is developed following an object data structure. They perform several essential tasks in the ROS 2 system: they manage sensor drivers, receiving and analysing data; they implement high-level decision control algorithms, allowing the robot to plan and make decisions autonomously; they allow the control of external actuators and components, such as robotic arms, wheels and grippers.

The discovery of nodes occurs automatically: when a node is started, it signals its presence to the other nodes in the network that share the same ROS domain (configured via the environment variable `ROS_DOMAIN_ID`); the nodes respond to this signal by providing information about themselves, thus allowing connections to be established. Periodically,

the nodes continue to communicate their presence in the domain.

Nodes in ROS can be started and stopped individually, facilitating the addition, removal or replacement of nodes without affecting the rest of the system. This allows various nodes to be distributed across multiple machines, optimising performance and scalability.

Messages

I *messaggi* sono il mezzo con cui i nodi comunicano tra di loro. Sono dati strutturati che consentono di inviare più Messages are the means by which nodes communicate with each other. They are structured data that allow several pieces of information to be sent in the same 'packet'. The type of information that is sent can be described and defined within the `.msg` files. Messages are used to send information such as numbers, strings, images, sensor data.
ROS provides standard messages such as `geometry_msgs/Twist` to send angular and linear velocity information.

Topic

In ROS 2, *topic* topics are communication channels through which nodes can exchange messages. Each topic only allows one specific type of message. For example, the topic `cmd_vel`, used to send and receive commands related to linear and angular velocity, is configured to only allow messages containing data of type `geometry_msgs/Twist`.

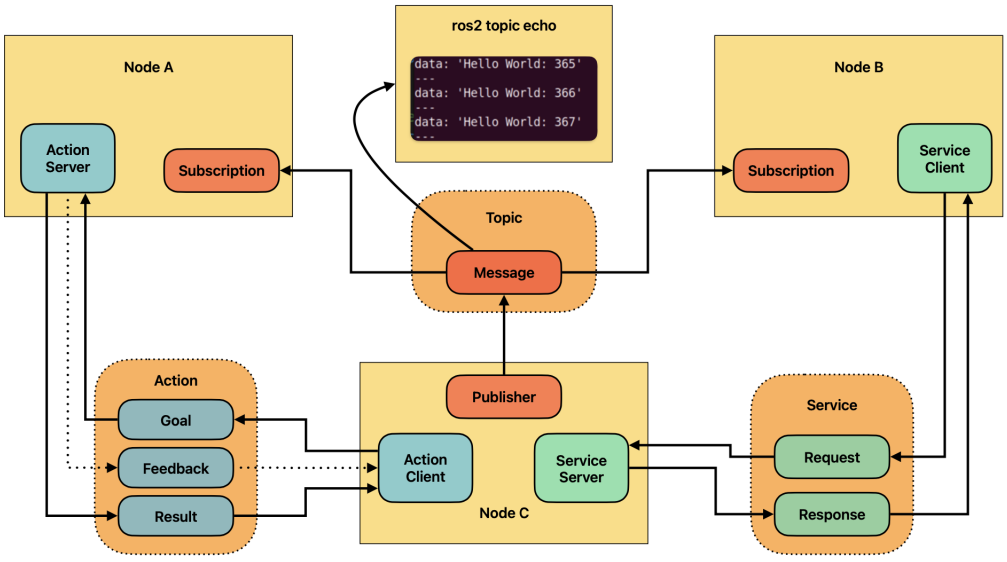

Figure 2.5. ROS 2 Graph

Nodes can communicate with each other using different communication paradigms:

- **Publisher / Subscriber**: this communication system consists of two types of nodes: a publisher and a subscriber. The first subscribes to a specific Topic and publishes a message periodically (asynchronous communication), which can be read by multiple subscriber nodes subscribing to the same Topic. A node may post in multiple Topics and simultaneously be a subscriber in multiple Topics.
- **Services**: is a communication system based on call-and-response (synchronous communication). There are two types of nodes, the client service and the server service. The first makes a request to the server, which immediately responds with a result.
- **Actions**: is a communication system built on topics and services. It functions similar to services, but unlike services, actions can be cancelled and provide immediate steady feedback to the node that called it. There are two types of nodes, the action client and the action server. The first sends a goal to the action server, which returns a feedback stream to it, using a Topic, until the requested result is achieved.

As shown in the figure 2.4, in ROS 2 is possible to choose which middleware to implement in the architecture. In more recent versions of ROS, the default middleware is FastDDS. However, the latter showed some limitations, which have been overcome by Zenoh in the last year.

2.3 FastDDS

FastDDS is an open-source implementation of DDS, developed by eProssima and optimised for ROS 2. It has been chosen as the default middleware for the latest versions of ROS 2.[6]
DDS (Data Distribution Service) is a communication protocol created for distributed systems, i.e. for applications consisting of several modules or services on different devices. This characteristic makes it the ideal middleware for communication in robots, where latency and precision in data exchange are essential requirements.

DDS is based on a model called Data-Centric Publish-Subscribe (DCPS).

2.3.1 DCPS

The DCPS model places the focus on the data itself. There are some applications (publishers) that transmit data via a certain topic and others (subscribers) that subscribe to the same topic and receive the published data. This allows DDS to have flexible communication in which you can have independent publishers and subscribers even on a large scale.

Four basic elements are defined in the DCPS model:

1. **Publisher**: are responsible for creating and configuring the *DataWriter*. DataWriters are entities that publish data and send it into the topic; they define the data that they publish and their publication properties, such as frequency or priority.

2. **Subscriber**: is the DCPS entity that receives the data of the topics it is subscribed to. There are one or more *DataReader* that read and use the data, defining their needs and the type of data they want to receive (i.e. only values exceeding a certain threshold).
3. **Topic**: is the channel that connects Publishers with Subscribers. Thanks to the *TopicDescription*, the topic can guarantee uniformity of data, ensuring that what is sent is compatible with what is requested.
4. **Domain**: is a concept that brings together all publishers and subscribers exchanging data in the various topics. An application participating in a domain is called *DomainParticipant*. Each domain is identified by its own ID. DomainParticipants are only able to see topics within their own Domain ID and therefore only communicate with other participants of the same ID. This separation is useful when you have several applications that must work separately without interference.

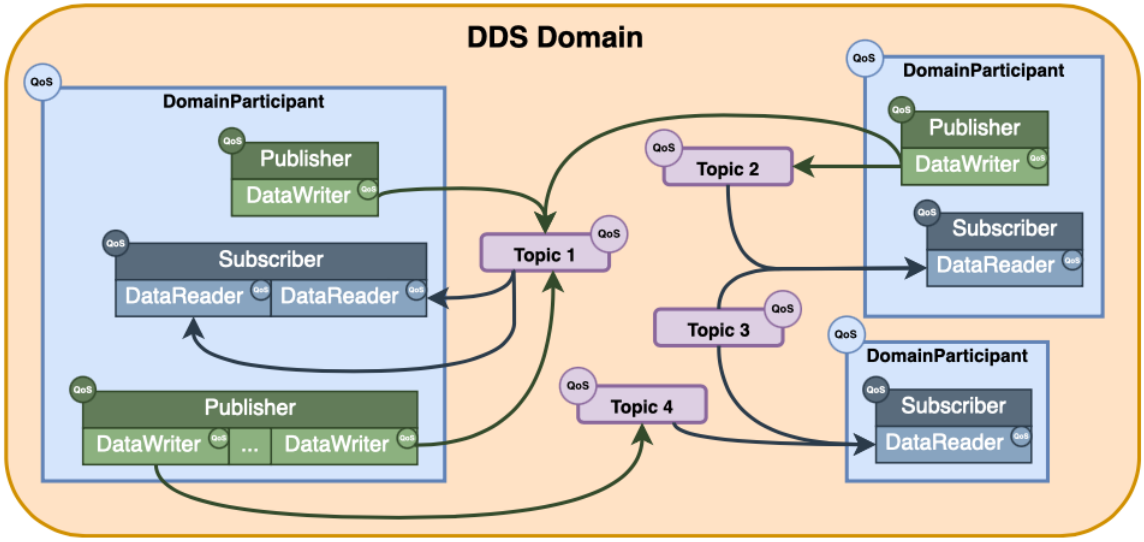

Figure 2.6. DCPS model entities in the DDS Domain.[6]

It is important to specify that communication via the FastDDS protocol is only allowed between DomainParticipants connected to the same subnetwork. Should we have one application contained in a machine connected to a home network and a second application contained in a machine connected to a different network (office network), even though they set the same Domain ID, they will not be able to communicate with each other.

2.3.2 Discovery

The discovery mechanisms provided by FastDDS enable the automatic search and matching of *DataWriter* and *DataReader* among domain participants.

Discovery occurs in two stages:

1. **Participant Discovery Phase (PDP)**: domain members identify themselves by periodically sending announcement messages that include the unicast addresses (IP and port) at which the domain participant listens to data traffic. Announcement messages are sent using multicast addresses and ports calculated from the Domain ID.
2. **Endpoint Discovery Phase (EDP)**: In this phase, the DataWriters and DataReaders identify each other, using the communication channels created during the PDP and sharing information about the topic and type of data.

The default discovery mechanism is **Simple Discovery** and is based on two independent protocols:

- **Simple Participant Discovery Protocol (SPDP)**: is concerned with identifying the *DomainParticipant* within a DDS domain.
- **Simple Endpoint Discovery Protocol (SEDP)**: once a *DomainParticipant* has been identified via SPDP, the SEDP protocol intervenes by initiating a data exchange to discover the relevant *DataWriter* and *DataReader*.

With this method, the amount of discovery data increases quadratically as the number of nodes increases. Specifically, in a system with n domain participants, each having r readers and w writers, the amount of discovery traffic increases by $n \times (n-1) \times (r+w)$.

2.4 Zenoh

Zenoh [4] was born with the aim of solving the problems of existing protocols, which were designed for specific use cases, creating 'islands of connectivity' that make interaction between cloud, edge devices and microcontrollers difficult, if not impossible. An example can be found in DDS, which was developed to offer an optimised pub/sub protocol for applications operating on hardware connected via multicast (UDP/IP) to the same local area network (LAN).

2.4.1 Protocol and abstractions

Zenoh is a Pub/Sub/Query protocol that allows working with moving data, static data and calculations, while maintaining efficiency even on limited hardware and complex networks. Zenoh supports peer-to-peer as well as routed and brokered communication. As a communication protocol, it is designed to operate flexibly on different layers of the ISO/OSI model. Specifically, it can work with:

- **Data Link Layer**: is the layer responsible for the creation of data packets and their direct transmission over physical connections such as Ethernet cables or Wi-Fi. When Zenoh operates on this layer it does not require more complex network

protocols, such as IP, reducing the data load.

- **Network Layer**: is responsible for routing, which is the choice of the best network path to use to reach the destination. It enables data transmission using protocols such as IP. Zenoh can exploit this layer to extend communication beyond the direct link, adapting to more complex networks.
- **Transport Layer**: In this layer, the goal is to ensure that packets arrive in the correct order without loss or error. Protocols such as TCP and UDP are used in this layer, protocols also taken up and used by Zenoh.

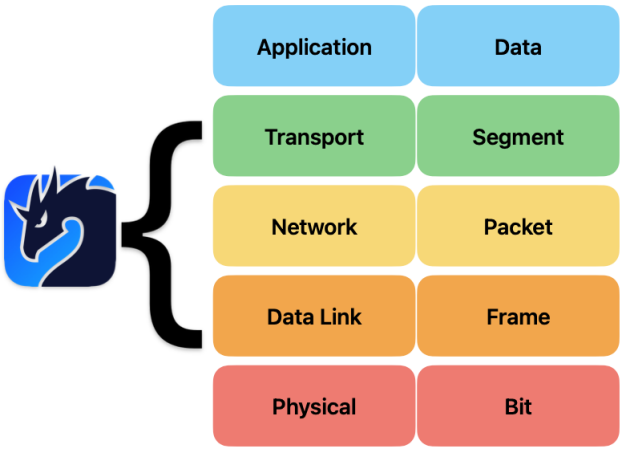

Figure 2.7. Zenoh protocol stack positioning

Zenoh uses some key abstractions:

- **Resources and Selectors**: Zenoh operates with resources. A resource is a key-value pair where the key is an array of feature arrays. A set of keys can be expressed by a key selector.
- **Publisher, Subscriber and Queryable**: As mentioned earlier, Zenoh defines three types of entities in the network:
  - **Publisher**: is the origin of the resources corresponding to the key expression.
  - **Subscriber**: is the destination of the resources corresponding to the key expression.
  - **Queryable**: is a passive store of resources corresponding to the key expression. It does not actively generate resources like the publisher, but stores them and makes them available.

2.4.2 Discovery

While DDS uses discovery protocols in which the number of active nodes significantly affects discovery traffic, since every time a publisher connects, it has to communicate its

presence to all readers.
A lighter approach is used with Zenoh: nodes only advertise resource interests and connect only with compatible readers, without wasting time with detailed notifications for each individual node or connection. In this way, good performance can be achieved even with a large number of active nodes. [7]

2.4.3 Deployment

As already mentioned in this section, Zenoh can handle different types of deployments.

- **Peer-to-peer**: default setting of Zenoh, in which , as with DDS, all nodes in the local network can exchange data directly with each other without the need for centralized configurations. Applications in this mode perform both multicast and gossip scouting:
  - **Multicast scouting**: to find nearby applications and routers, applications join the multicast group `224.0.0.224` - UDP port `7446` by broadcasting scouting messages at this address. When they find a router or application in peer mode, they immediately connect to it.
  - **Gossip scouting**: in case multicast communications are not available, zenoh forwards all discovered nodes and routers to the newly discovered applications.
- **Client**: Peer-to-peer communication may be undesirable for scalability reasons. So the node can be configured to work in client mode, going to create a single session with a router which will connect it with the rest of the system. Scouting is performed in multicast mode.
  Communication can be:
  - **Broked**: there is a central broker who acts as an intermediary between the nodes.
  - **Routed**: in which data are routed directly by routers without the need for a central broker.

  Routed, compared to Broked, is more scalable and distributed, since it does not depend on a single point of management.

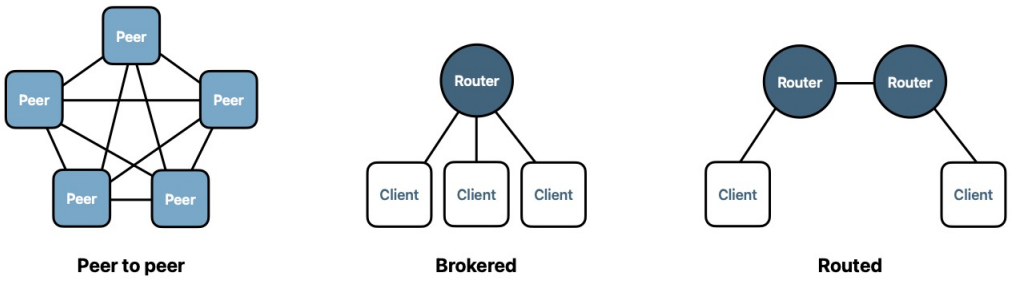

Figure 2.8. Discovery types

2.4.4 `rmw_zenoh`

It is an rmw (ROS 2 middleware) implementation based on Zenoh. It is compatible with ROS 2 Rolling, **Jazzy** and Iron distros.

Design

The package `rmw_zenoh_cpp` maps the RMW API of ROS 2 to the Zenoh API, so that users can use ROS 2 to send and receive data on Zenoh using APIs they already know.

- There is a Zenoh router running on the local system, which is used for discovery and communication with other routers. This router is not used for communication between clients, which is done through peer-to-peer connections.
- Each node, or group of nodes communicating with each other, is mapped to a single Zenoh session. In this way one can have many different clients sharing the same session.
- Data are sent and received via the zenoh API when the corresponding rmw API is called.

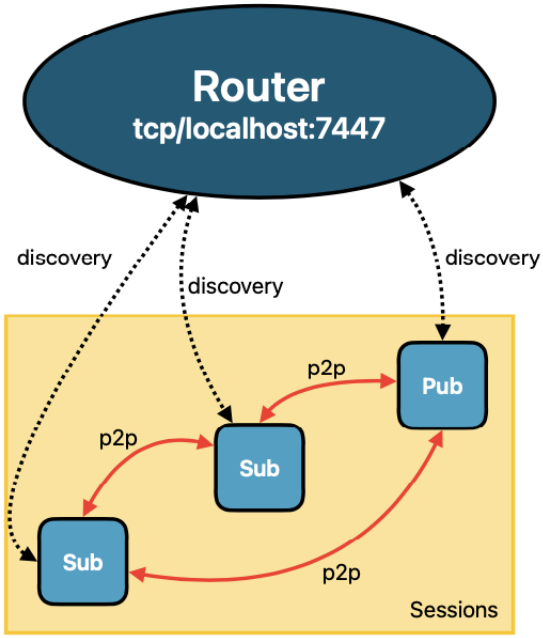

Figure 2.9. Design

For peer-to-peer discovery, `rmw_zenoh_cpp` needs a router to discover peers and forward discovery information to other peers (gossip scouting). So there is a need for the Zenoh router to be running.

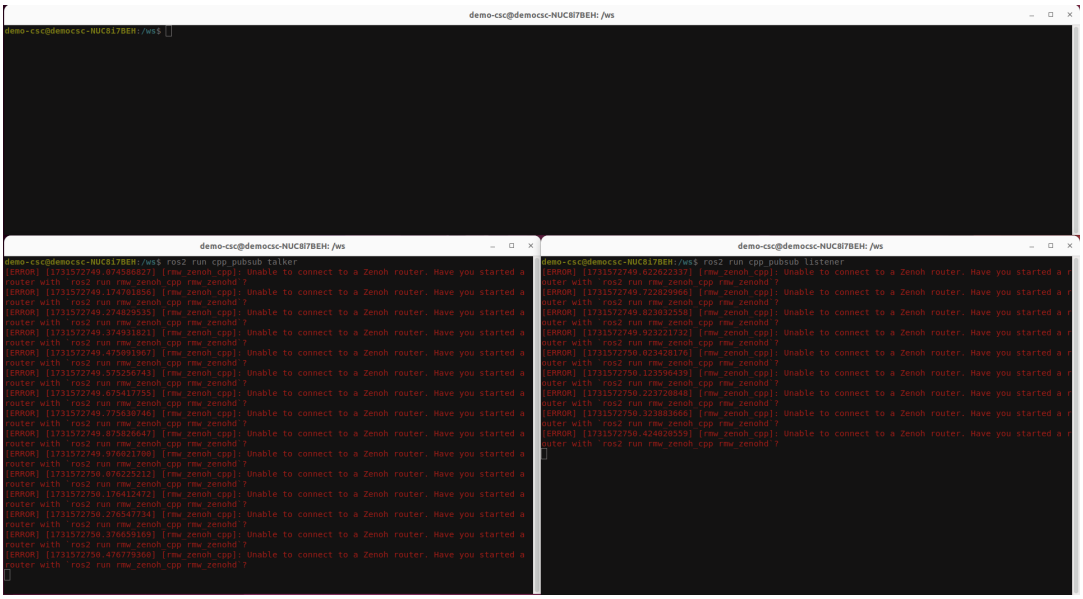

Figure 2.10. Running node without router on

Setup and test

In order to have `rmw_zenoh_cpp` executable on our machine, a workspace has to be created, the project downloaded and installed in ROS 2.

```
$ mkdir ~/ws_rmw_zenoh/src -p && cd ~/ws_rmw_zenoh/src
$ git clone https://github.com/ros2/rmw_zenoh.git
$ cd ~/ws_rmw_zenoh
$ rosdep install --from-paths src --ignore-src --rosdistro <
   DISTRO> -y # replace <DISTRO> with ROS 2 distro of choice
$ source /opt/ros/<DISTRO>/setup.bash # replace <DISTRO> with
    ROS 2 distro of choice
$ colcon build --cmake-args -DCMAKE_BUILD_TYPE=Release
```

With the goal of choosing `rmw_zenoh_cpp` as middleware, the file "`.bashrc`" is to be edited with the following command:

```
$ echo "export RMW_IMPLEMENTATION=rmw_zenoh_cpp" >> ~/.bashrc
```

At present, the router must be launched by hand by the user, using the command:

```
# terminal 1
$ source ~/ws_rmw_zenoh/install/setup.bash
```

```
$ ros2 run rmw_zenoh_cpp rmw_zenohd
```

Now, in two separate terminals on the same machine, the talker and listener can be run, so that Zenoh's operation can be verified:

```
# terminal 2
$ ros2 run demo_nodes_cpp talker
```

```
# terminal 3
$ ros2 run demo_nodes_cpp listener
```

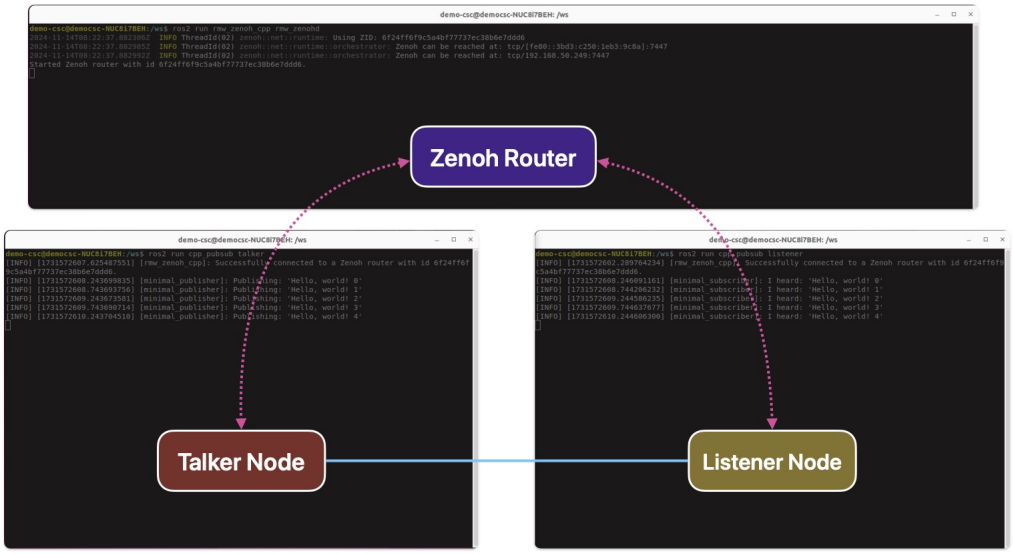

Figure 2.11. Testing the performance of talker and listener on the same machine

If router crashes, peer-to-peer communications remain.

Configuration

`rmw_zenoh` uses two different configuration files to configure the `Zenoh router` and `Zenoh session`. To set up a custom configuration file, the following command can be executed:

```
$ echo "export ZENOH_ROUTER_CONFIG_URI=$HOME/routerconfig.
    json5" >> ~/.bashrc
```

In order to establish a communication bridge between two hosts, it is possible to modify the `routerconfig.json5` of one of the hosts so that it connects to the other zenoh router at startup. This configuration will be done in Chapter 3.4.

2.5 VPN and Husarnet

A VPN (Virtual Private Network) is a network technology based on a secure, encrypted connection through the public network. It is a virtual contact point that offers users the advantage of sending and receiving data from other clients as if they were actually connecting directly to the same private network.

2.5.1 Husarnet

During the development of this project, use was made of Husarnet, a peer-to-peer VPN that aims to enable secure and direct communication between IoT devices or robots. With having Husarnet connected devices can talk to each other without need for centralized servers and complex network configurations like port forwarding.

Since robotic applications demand some superior level of security as part of it, then Husarnet is an ideal contender because the enlisted benefits using encryption protocols makes Husarnet ideal cloud-based VPN used to secure the data. It works on multiple OS and easy to configure with Docker, where an out of box image is available.

Setup and test

First of all, Husarnet is installed inside the machines that are to be connected.[10]

- In order to use this VPN you must create an account and log on to https://app.husarnet.com/.
- Once inside create a network and give it a name.
- Once the network is created, it is possible to connect the elements that will become part of the network.

As seen from Figure 2.12, communication via Husarnet occurs correctly between two devices connected to two different networks.

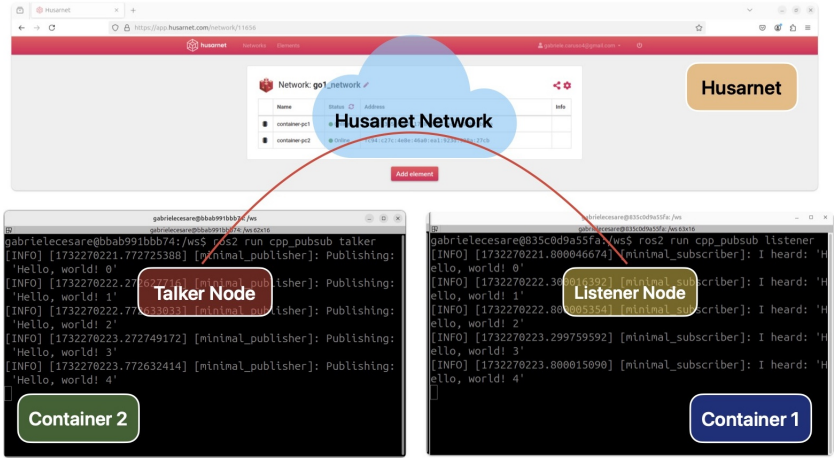

Figure 2.12. Testing the performance of talker and listener via Husarnet

2.6 Navigation 2

Nav2 represents a standard for autonomous navigation of mobile robots, and uses ROS 2 as the basic middleware. Conceptually, the Nav2 stack is simple; it guarantees the movement of a mobile robot from a starting position to a goal position. It consists of numerous packages, plugins, and libraries that provide the robot with the ability to move autonomously even uncertain and dynamic environments, using advanced algorithms for localization, planning, and motion control.[11]
Navigation tasks use a Behavior tree (BT) structure in which some nodes use ROS 2 action servers to perform tasks required by the tree condition.

2.6.1 Behavior Trees

Nowadays, decision trees are assuming a key role in the management of complex controls for robots, enabling them to organize and simplify decision-making processes, thanks to their ability to structure complex behaviors in a hierarchical and modular manner, making the structure more scalable and understandable for humans.

A clear application example is given in Figure 2.13, in which the diagram represents a BT for a robotic task in which one has to find the ball, catch it and place it.
As can be seen, there is a main structure, with a root node from which everything starts. This is a sequence node, which means that its children must be completed in the order specified by the arrow (in this case from left to right). **Find Ball** represents the first task, in which the robot searches for the ball, while **Place ball** the last task, in which it places the ball in a defined area.

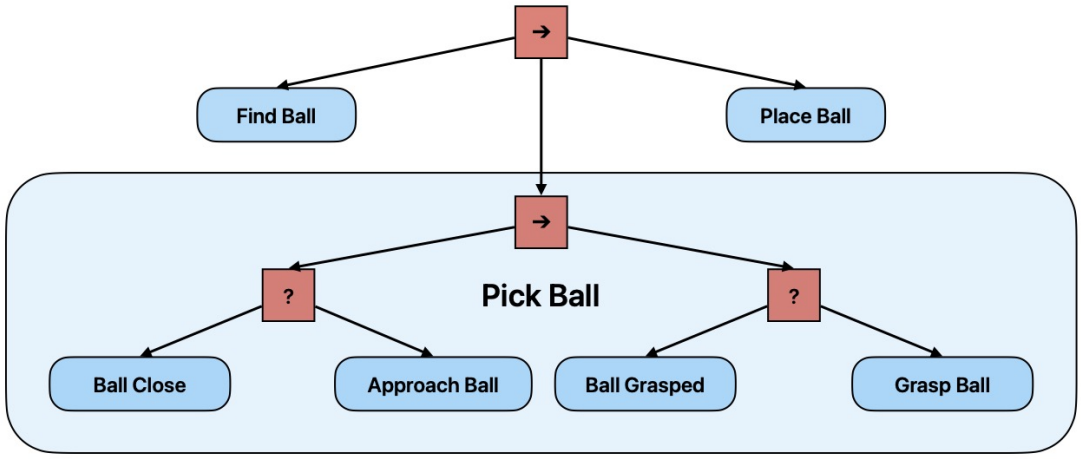

Figure 2.13. Behavior tree of a specific application

The node **Pick Ball** represents a complex sub-diagram, in which you have another sequence node under which its children are also executed this time from left to right. It can

be seen that in this sub-diagram there are two nodes represented by a question mark "**?**" which represent conditional checks that verify whether the previous action was successful or not; e.g. it checks whether the ball was caught otherwise it tries to catch it and checks again whether the computation was performed correctly; it repeats the process until the ball is successfully caught.

2.6.2 Navigation Servers

As mentioned before, Nav2 has a modular structure, consisting of a number of packages and algorithms that cooperate with the purpose of having autonomous navigation. The BT Navigation Server loads, executes and monitors predefined Behavior trees, which, as mentioned earlier, execute a series of Action Servers that allow the robot to move toward a destination, avoid obstacles, and interact with environment and objects.

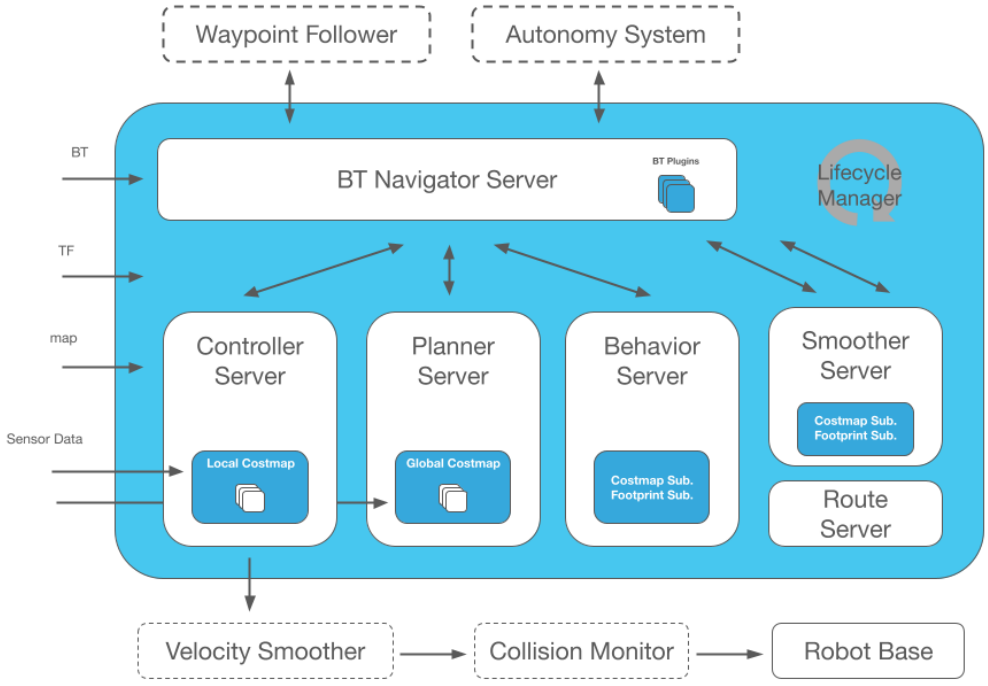

Figure 2.14. Nav2 architecture.

The main action servers used are: Planner Server, Controller Server, Behavior Server and Smoother Server. These servers execute actions that are called by plugins present in the Behavior Tree (BT) nodes. The action server callback invokes the selected algorithm, identifying it by its name, which is associated with a specific algorithm configured in the system.

Planner Server

The task of this server is to calculate a valid and optimal path to reach from the current pose to a target pose. Having access to the Global Costmap, it uses Path Planning algorithms such as the **Dijkstra's Algorithm** or the **A* algorithm** (previously explained in Chapter 1.2.2). The modularity of the system allows developers to choose the global planning algorithm best suited to their scenario, or to implement a custom one.

Controller Server

The Controller Server is tasked with ensuring that the robot follows the path calculated by the Planner Server, translating the global path into high-level commands that enable precise robot movement. With the help of sensors such as LiDAR, this Server can identify and avoid dynamic obstacles.
The main controller servers used by Nav2 are:

- **DWB Controller**: DWA (Dynamic Window Approach) successor. Based on the dynamic window, it optimizes speed and trajectory to ensure the robot avoids obstacles and follows the path. It takes into account the robot's dynamics (inertial masses and speed limits) to ensure safe commands.
- **Graceful Motion Controller**: is designed for smooth and natural robot movements, even in situations where sudden changes in direction or speed are required. Useful for service or social robot applications, as it makes movement more "pleasant."
- **MPPI (Model Predictive Path Integral) Controller**: is based on predictive model optimization. It simulates several possible trajectories and selects the optimal one considering robot dynamics, obstacles, and global path.
- **RPP (Regulated Pure Pursuit) Controller**: is based on the Pure Pursuit algorithm, in which it follows a landmark (which updates us continuously) on the planned path by adjusting speed and direction to have smooth movement. It is very simple to set up and effective in non-complex environments.
- **Rotation Shim Controller**: it is responsible for handling precise rotations of the robot to properly align it with the desired direction before moving. It is often used by combining it with other controllers so as to ensure that the robot always starts in the right direction.
- **Waypoint Follower Controller**: is ideal for following a sequence of predefined waypoints. Useful in applications where the robot must follow very specific or predetermined trajectories.

Behavior Server

The purpose of this server is to make the system fault tolerant so that the robot can use Recovery to recover from unknown faults/circumstances while navigating (dynamic objects, transient obstacles). Unlike the Server Planner and Controller which only help with local navigation within the known environment, the Recovery Server deals with any unforeseen events to allow the robot to return to the route plan as soon as the problem is

removed. This may mean going in another direction or moving away from the target to avoid blockage.

Smoother Server

This server aims to reduce trajectory inconsistency, so that for sudden turns, a smoother path is achieved, while maximizing trajectory distance from obstacles and high-cost areas.

2.6.3 Robot Footprints

In Nav2, a robot footprint basically the geometrical representation of your robot i.e. More precisely, the robot footprint is a geometric figure (typically circular or rectangular, though it could also be any other complicated polygon) that defines an area of space in its environment occupied by the physical geometry of the robot. It plays a crucial role in planning the trajectory and movement of the robot to avoid or predict any potential contact with obstacles.

2.6.4 State Estimation

Nav2 needs two main transformations to be provided in order to work: the **from map to odom** transformation, which is done by a positioning system (localization, mapping, SLAM). The second is the **from odom to base_link** transformation, which comes from the robot's odometry system.

Global Positioning: Localization and SLAM

This is the part of Navigation 2 that deals with locating the robot with respect to the map. It therefore provides the system with the transformation `map -> odom`.
Nav2 uses the Monte-Carlo algorithm (AMCL) seen in Chapter 1.2.1 for localization with a given map. In the case of not having an a priori map, this stack also provides a SLAM Toolbox, which is useful for mapping unknown environments.

Odometry

The transform `odom -> base_link` is obtained using odometry provided by the robot. Odometry is a technique used in robotics that estimates the robot's position based on data collected from its motion sensors (such as encoders or IMUs) or sensing sensors (such as LiDAR or Sonar). Using the speed, angle of motion, and position of objects over time, it calculates the change in the robot's position relative to the global reference system. However, while odometry helps provide a snapshot of the distance travelled, this information can accumulate errors over time due to sensor drift. To reduce these errors, filters, such as EKF (Chapter 1.2.1), are often integrated to improve the accuracy of position estimation.

2.6.5 Environmental Representation

Environmental representation describes how a robot perceives and interprets its surroundings, and is commonly implemented in the form of a cost map. A costmap is a

two-dimensional grid in which each cell represents a cost that can indicate an unknown, vacant, occupied or inflated state. This map is used to plan routes globally or to calculate local control strategies through sampling.

Several layers of the costmap are implemented as modules that can be used through the pluginlib framework, enabling the integration of data from sensors such as LiDAR. The costmap layers are designed to detect and track obstacles in the environment, facilitating collision avoidance through devices such as depth sensors or cameras.

In Navigation2 (Nav2), the costmap implementation is handled by a dedicated package called `nav2_costmap_2d`. Unlike traditional costmaps, which are often monolithic, Nav2's is modular and divided into several layers. This layer structure allows various types of obstacles in the environment to be represented.

The main layers of the costmap are three:

- **Static layer**: stores costs associated with fixed obstacles, such as walls or furniture, that do not change over time.
- **Obstacle layer**: dynamically updates map cells as vacant or occupied, based on data from sensors.
- **Inflation layer**e: creates a safety zone around obstacles, preventing the robot from getting too close. This layer generates a buffer area to reduce the risk of collisions.

This layered structure improves the flexibility and reliability of the system, allowing more effective management of obstacles and safe navigation zones.

2.7 TF Tree

Many ROS packages require defining a robot transformation tree using the **TF2 ROS** package. This tree establishes relationships between coordinate systems, considering translations, rotations, and relative motions. For example, as seen in Figure 2.15, a mobile robot with a laser sensor has two distinct frames: one at the center of the robot's base, called `base_link`, and one at the center of the mounted laser, called `base_laser`. This configuration allows for proper integration and localization of data between robot and sensors.

It is supposed to receive data from the laser, indicating the distance there is between the wall and the `base_laser` frame. However, as evidenced by Figure 2.15, this information does not correspond to the actual distance between the base_link frame and the wall, making it inssufficient for proper robot navigation.

To solve this problem, it is necessary to transform this data, moving from the `base_laser` frame to the `base_link` frame. This requires knowledge of the Cartesian distance between the two frames, allowing for correct translation of the distance values (as can be seen from Figure 2.16).

Therefore, it is essential to construct a correct structure of the TF tree, using information about the position of the `base_laser` frame, obtained from the `.xacro` file in which the robot model is defined.

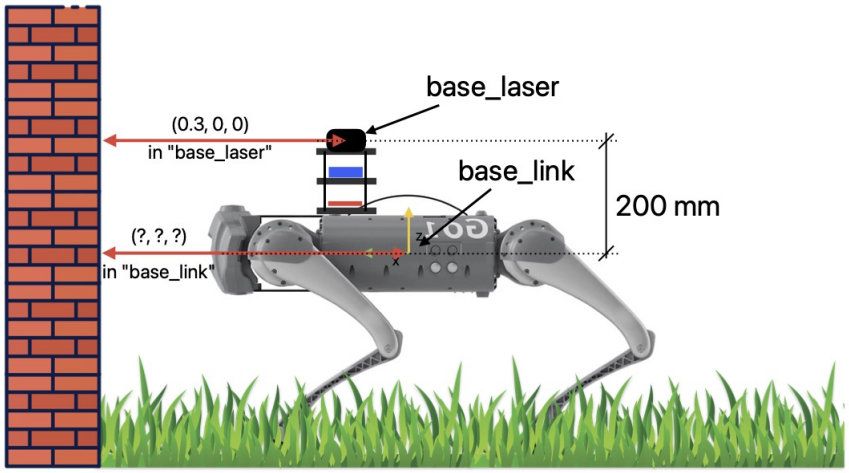

Figure 2.15. Frame reference scheme base_link and base_laser

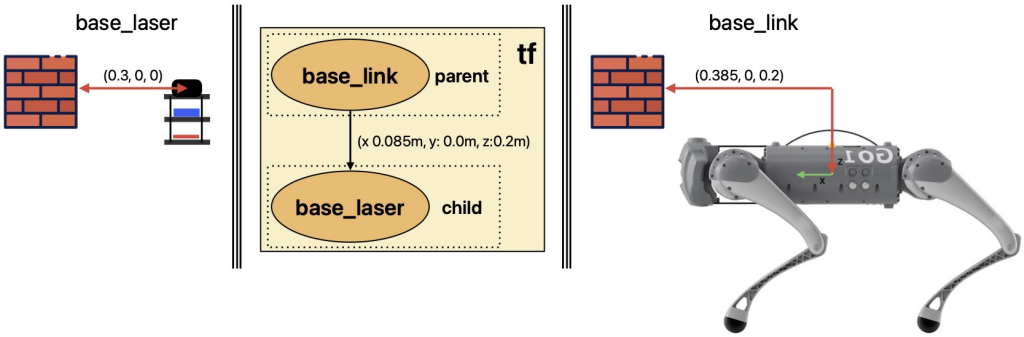

Figure 2.16. Transformation from base_link to base_laser

In ROS 2, the TF diagram can be displayed using the following command:

```
$ ros2 run tf2_tools view_frames
```

A PDF file containing the graph of TF transformations active in the system is generated. For the command to work properly, the tf2 node must be running.

2.8 Hardware

This section provides technical details about the hardware components used in the project, describing the fundamental devices for operating and controlling the robot, as well as the sensors used for data collection.

2.8.1 Robotic platform

For this project, use was made of the Unitree Go1 Edu robotic platform, which has already been described in Chapter 1.3.1.

The table with technical specifications is given below:

Feature	**Technical specifications**	
Dimensions	LxWxH (Stand)	0.645 * 0.28 * 0.4 m
	LxWxH (Folded)	0.54 * 0.29 * 0.13 m
Weight	12 kg (with battery)	
Playload	5 kg	
Speed	0-3,7 m/s	
Operating time	1-2 h	
Maximum angle	35 deg	
DOF	Total 12, one leg 3	
Power input	24 V, 4 A	
Knee joint	C1-8 x 1,5 ratio - 35,5 Nm	
Body/thigh joints	C1-8: 520 g, 23,70 Nm	
Stereo depth camera	5 sets	
Ultrasonic Sensor	4 sets	
Processor	1 Nano + (2 Nano o 2 Nx)	

Table 2.1. Unitree Go1 Edu technical specifications [19][16]

2.8.2 Intel NUC

The Intel NUC (Next Unit of Computing) is a compact and powerful mini-PC used as a central processing unit to manage robot control and data flow from sensors. In the course of this project, this mini-PC will be named "*Computer*" or "*PC*".

The following is the Table 2.2 with the main features:

Feature	**Details**
Model	Intel NUC8i7BEH
CPU	Intel Core i7-8559U
CPU Frequency	Base: 2.7 GHz, Turbo: 4.5 GHz
Core/Thread	4 core, 8 thread
Architecture	x86-64 (64-bit)
Cache	L1: 256 KiB, L2: 1 MiB, L3: 8 MiB
RAM	32 GB DDR4 (2 x 16 GB Kingston, 2400 MHz)
Graphics	Intel Iris Plus Graphics 655 (Integrated GPU)
Storage	SSD NVMe Samsung 970 PRO 512 GB
Network Ports	Gigabit Ethernet
Operating System	Ubuntu 22.04.5 LTS
Ports I/O	1 USB-C (Thunderbolt 3), 4 USB 3.1, 1 HDMI 2.0a, 1 Ethernet, 1 SDXC slot
Audio	Stereo output via audio jack 3.5mm
Dimension	117 x 112 x 51 mm
Power Consumption	Fino a 28W (TDP CPU)

Table 2.2. Intel NUC8i7BEH Datasheet

2.8.3 Raspberry Pi 4

The Raspberry Pi 4 is a low-cost, small microcomputer developed by the Raspberry Pi Foundation. Its small size makes it ideal for use in this project, as it can be conveniently placed on the back of the robot and used as a communication bridge between the robot platform and the office network. It also handles the acquisition and processing of data collected by the 2D LiDAR sensor.

Feature	**Details**
Model	Raspberry Pi 4 (4 GB RAM)
Processor	Broadcom BCM2711, Quad-core Cortex-A72 (ARMv8) 64-bit
CPU Frequency	1.5 GHz
Cores/Threads	4 cores, 4 threads
Architecture	ARMv8 (64-bit)
Cache	L1: 32 KiB (per core), L2: 512 MiB (shared)
RAM Memory	4 GB LPDDR4-3200 SDRAM
Graphics	Broadcom VideoCore VI (integrated GPU)
Storage	Kingston 128 GB SD card (SDXC)
Network Ports	Gigabit Ethernet, Wi-Fi 802.11ac (2.4 GHz and 5 GHz)
Operating System	Ubuntu Server 22.04 LTS
I/O Ports	2 USB 3.0, 2 USB 2.0, 2 micro-HDMI, 1 Gb Ethernet, 40 GPIO
Audio	Audio output via HDMI or 3.5mm audio jack

Table 2.3. Technical Specifications of the Raspberry Pi 4

2.8.4 NVIDIA Jetson

The NVIDIA Jetson series is one of the most powerful platforms for parallel computing and real-time processing, particularly suitable for artificial intelligence and robotics applications. Two models of the series were used in the project: the Jetson Nano and the Jetson Xavier.

Jetson Nano

The Jetson Nano is used in Chapter 3 to externally simulate the Jetson present inside the robot.

The following is the Table 2.4 with the main features:

Feature	**Description**
Model	NVIDIA Jetson Nano (P3450)
Processor	Quad-Core ARM Cortex-A57 MPCore
CPU Core/Thread	4 Cores / 4 Threads
CPU Frequency	1.43 GHz
GPU	128-core NVIDIA Maxwell
RAM	4 GB LPDDR4
Storage	64 GB Kingston MicroSD
Supported Operating Systems	Ubuntu 18.04 (Jetpack base)
I/O Interfaces	- 1x USB 3.0, 3x USB 2.0 - HDMI 2.0 and DisplayPort 1.2 - Gigabit Ethernet - GPIO, I2C, I2S, SPI, UART
Connectivity	Ethernet 10/100/1000 Mbps
Hardware Accelerators	- AI acceleration with TensorRT - Multimedia acceleration (H.264/H.265) - Support for deep neural networks
Dimensions	100 mm x 80 mm x 29 mm
Power Consumption	- Low-power mode: 5W - Maximum mode: 10W
Software Compatibility	- NVIDIA JetPack SDK - Support for TensorFlow, PyTorch, OpenCV

Table 2.4. Technical specifications of the NVIDIA Jetson Nano

Jetson Xavier

The Jetson Xavier is an advanced processing platform that offers higher performance than the Jetson Nano.

Feature	**Description**
Model	NVIDIA Jetson Xavier NX
Processor	Hexa-Core ARM v8.2 64-bit Carmel CPU
CPU Cores/Threads	6 Cores / 6 Threads
CPU Frequency	Up to 1.9 GHz
GPU	384-core NVIDIA Volta with 48 Tensor Cores
RAM Memory	8 GB LPDDR4x (128-bit bus)
Storage	64 GB Kingston MicroSD
Supported Operating Systems	Ubuntu 18.04 (Jetpack-based)
I/O Interfaces	- 2x USB 3.1, 4x USB 2.0 - HDMI 2.0 and DisplayPort 1.4 - Gigabit Ethernet - GPIO, I2C, I2S, SPI, UART
Connectivity	Ethernet 10/100/1000 Mbps
Hardware Accelerators	- AI acceleration with TensorRT - Multimedia acceleration (H.264/H.265) - Support for deep neural networks
Dimensions	70 mm x 45 mm
Power Consumption	- Low power mode: 10W - Maximum mode: 15W
Software Compatibility	- NVIDIA JetPack SDK - Support for TensorFlow, PyTorch, OpenCV

Table 2.5. Technical Specifications of the NVIDIA Jetson Xavier NX

2.8.5 LiDAR 2D

The LiDAR (Light Detection and Ranging) sensor is one of the most widely used sensors in robotic applications, useful for mapping and navigation.

Its operation is similar to sonar, except that instead of using sound waves, it uses laser pulses to calculate distances between the LiDAR and possible obstacles.
Distances are calculated based on the Time of Flight (ToF) principle, which is how long it takes the laser pulse to leave and return from the LiDAR.

Specifically, as seen in Figure 2.17, a laser pulse is emitted and projected into the environment. When the light beam hits a surface, some of its energy is reflected back (the amount of reflected light depends on the properties of the surface, such as color, material, and angle of incidence). Once the light beam returns to the sensor, the time it takes for the light to return is calculated, using the formula $d = \frac{c \times t}{2}$, where d is the distance to the object, c the speed of light and t the time of flight.

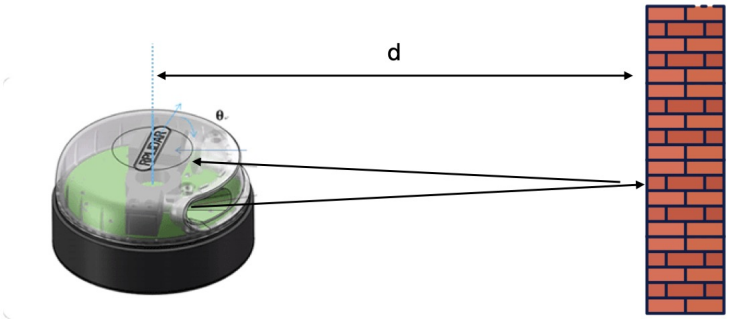

Figure 2.17. LiDAR operating diagram

RPLIDAR A3 is used in this project, Figure 2.18 .

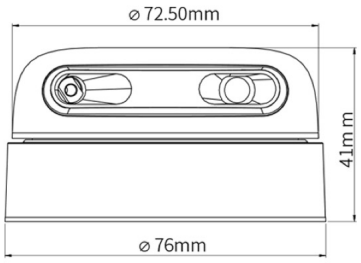

Figure 2.18. LiDAR dimensions

The following is the Table 2.6 with the main features:

Feature	**Details**
Model	RPLIDAR A3
Scanner Type	Laser Range Scanner
Communication Interface	TTL UART
Communication Speed	256000 bps
System Voltage	5V
System Current	450mA - 600mA
Power Consumption	2.25W - 3W
Angular Range	360°
Angular Resolution	0.225°
Sampling Rate	16000 samples/s (Advanced Mode)
Measurement Range (White Object)	25m (Advanced Mode), 20m (Outdoor Mode)
Measurement Range (Dark Object)	10m
Operating Temperature Range	0℃ - 40℃
Accuracy	1% of distance ($\leq$ 3m) 2% of distance (3 - 5m) 2.5% of distance (5-25m)
Scanning Rate	15 Hz (Adjustable between 10 Hz and 20 Hz)
Distance Resolution	$\leq$ 1% ($\leq$ 12m), $\leq$ 2% (12m $\sim$ 25m)
Output Interface	UART Serial (3.3V)
Operation Mode	Advanced Mode (Indoor Use) Outdoor Mode (For outdoor use)
Weight	190g
Dimensions (H x W)	41 mm x 76 mm

Table 2.6. Technical Specifications of the RPLIDAR A3

Chapter 3

Network Configuration

This chapter approached the problem of getting the computer to communicate with the robotic platform. It proceeded in five stages, a first one in which the problem is analysed and possible solutions are sought. In a second phase, the robotic platform is configured by making it accessible to the office network and the computer. In a third phase, the network configuration of ROS 2 is tested using FastDDS, a solution, however, that presented quite a few problems. In the fourth phase, better communication performance is achieved using Zenoh.
Then, in the fifth and final phase, the best solution is applied to the robotic platform and the proper functioning of the communication is finally tested.

3.1 Problem analysis

As seen in **Chapter 2**, in ROS 2 communication between nodes only takes place if the nodes are within the same `ROS_DOMAIN_ID` and in the same subnetwork. Go1 robot, as seen in Figure 3.1 provided by the manufacturer, has its own subnet `192.168.123.xx`.

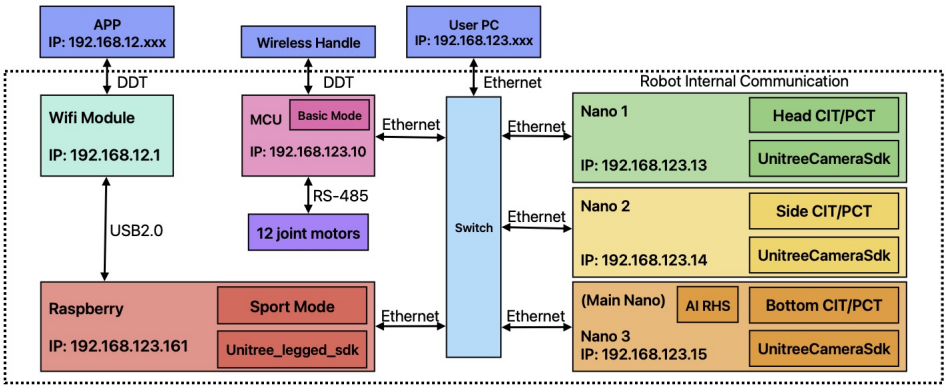

Figure 3.1. System diagram

Therefore, in order to communicate with the robot, the producer gives instructions to connect the computer via cable to the gigabit ethernet port on the back of the robot (port 7 in Figure 3.2) in order to proceed with programming. Uncomfortable and discarded solution.

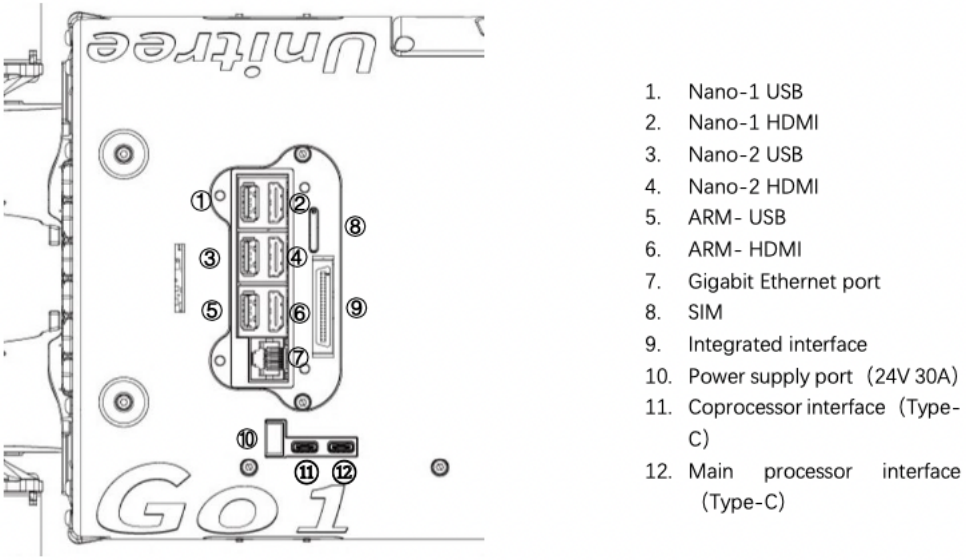

Figure 3.2. Schematic diagram of the back interface Go1 [18]

Thus, for more convenient programming and navigation, it is preferable not to have the PC directly connected to the robot; for this purpose, we chose to use a **Raspberry Pi 4**, connecting it via cable to the GO1's Ethernet port and via wi-fi to the laboratory network. This makes the robot's internal boards visible to the computer.

This problem was addressed and solved in section 3.2 of this chapter, where the connection was initially tested with a Jetson Nano instead of the robot, in order to ensure correct operation of the bridge without jeopardising the software integrity of the robot. The entire configuration is then applied to the robotic platform.

However, in this way, the Jetson and the computer are in two different subnets, so communication of ROS nodes is not possible unless appropriate measures are taken.

Initially, communication in ROS was tested using a **VPN** (Husarnet) and the **FastDDS** middleware - section 3.3 of this chapter.

Finally, ROS communication was chosen using the **Zenoh** middelware - section 3.4 - to overcome the problems present in FastDDS.

3.2 Network configuration on Raspberry Pi 4

The purpose of this configuration is to connect the Jetson with the office network and share the Internet connection that comes through Raspberry pi 4 from the wi-fi `wlan0` interface to the `eth0` Ethernet port.

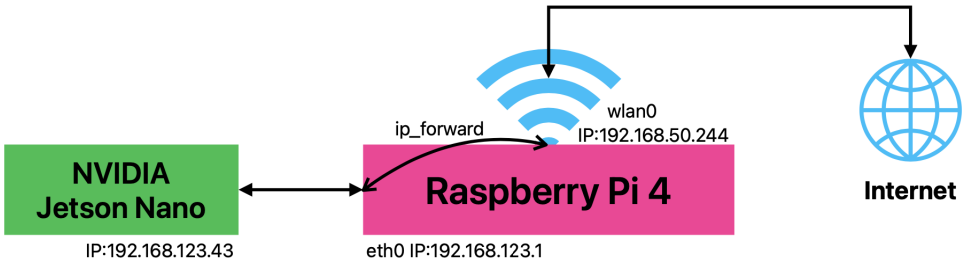

Figure 3.3. Raspberry bridge scheme

Tests were carried out to verify the functioning of the bridge, using a Jetson Nano instead of the robot. Once the configuration was found to be working properly, it was applied to the robot.

3.2.1 Netplan configuration

Netplan is a framework for declaratively configuring the network through `YAML` files. It acts as an intermediary between user configuration and network backends, such as Network Manager or systemd-networkd, applying these configurations to the underlying services. Netplan reads network configurations from "`/etc/netplan*.yaml`" and during boot-loading processes them by generating configuration files specific to the config backend and then passing control of the devices to a particular networking daemon.

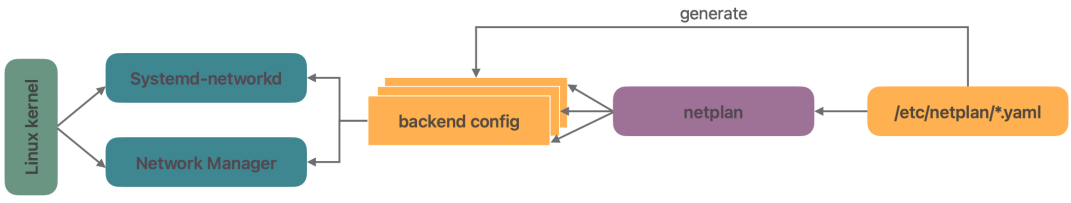

Figure 3.4. Netplan operation

/etc/netplan/01-LinksWifi.yaml

```
network:
  version: 2
  renderer: NetworkManager
  wifis:
    wlan0:
```

```
6       dhcp4: no
7       addresses: [192.168.50.244/24]
8       access-points:
9         "wifi_ssid":
10          password: "**********"
```

Listing 3.1. Wi-Fi Netplan configuration

In this first Netplan configuration (Listing 3.1), wi-fi credentials are entered to allow the Raspberry pi 4 internet access.

/etc/netplan/20-Ethernet.yaml

```
1 network:
2   version: 2
3   renderer: NetworkManager
4   ethernets:
5     eth0:
6       dhcp4: no
7       addresses: [192.168.123.1/24]
8       nameservers:
9         addresses: [8.8.8.8, 8.8.4.4]
```

Listing 3.2. Ethernet Netplan configuration

In this Netplan configuration (Listing 3.2), the router IP of the subnetwork that the raspberry is going to create is set in row 7.

3.2.2 Dnsmasq

To proceed with the creation of the subnetwork on Raspberry Pi 4, it is chosen to use Dsnmasq.

```
sudo apt-get install -y dnsmasq
```

Dnsmasq is an opensource software developed to provide network infrastructure: DNS, DHCP, router advertisement and network boot.

- **DNS**: can behave as a DNS caching server and forwarder. It means that it caches DNS responses, speeding up subsequent resolutions of the same domain names and reducing the load on upstream DNS servers.
- **DHCP**: Offers DHCP services, dynamically assigning IP addresses to devices within the local network.
- **Router Advertisement**: Dnsmasq supports router advertisement for IPv6 networks, helping devices automatically configure their IPv6 network settings.
- **Network Boot**: Supports booting devices on the local network via PXE (Preboot Execution Environment), useful for operating system installations or booting devices without local media.

Below is the configuration created for Dnsmasq (Listing 3.3)

/etc/dnsmasq.conf

```
interface=eth0
listen-address=192.168.123.1
bind-interfaces
server=8.8.8.8
domain-needed
bogus-priv
dhcp-range=192.168.123.24,192.168.123.49,12h
```

Listing 3.3. Subnet dnsmasq configuration

3.2.3 IP Forwarding and Firewall

In order to enable package forwarding between interfaces, the file "**/etc/sysctl.conf**" is modified by removing the "`#`" from the beginning of the line with `net.ipv4.ip_forward=1`, and the machine needs to be restarted.

The following terminal commands are executed:

```
$ sudo iptables -t nat -A POSTROUTING -o wlan0 -j MASQUERADE
$ sudo iptables -A FORWARD -i wlan0 -o eth0 -m state --state
   RELATED,ESTABLISHED -j ACCEPT
$ sudo iptables -A FORWARD -i eth0 -o wlan0 -j ACCEPT
```

1. The first command adds a rule to the **iptables NAT** table that applies to packets that are going to leave the `wlan0` interface. This rule changes the source IP address of the packets to match the IP address of the `wlan0` interface (using masquerading).
2. The second command adds a rule to the **FORWARD** chain of **iptables** that applies to packets entering from the `wlan0` interface and exiting from the `eth0` interface. This rule accepts packets that are part of already established connections or related to existing connections.
3. The last command adds a rule to the **FORWARD** chain of **iptables** that allows packets to pass through the system from `eth0` interface to `wlan0` interface. In other words, packets that arrive from the `eth0` interface and are destined for the `wlan0` interface will be accepted and then forwarded.

With the purpose of saving this configuration by making it always active on the next reboots of the machine, the following command is executed:

```
$ sudo apt-get install iptables-persistent
$ sudo iptables-save > /etc/iptables/rules.v4
$ sudo netfilter-persistent save
```

This command is executed whenever adding iptables rules that want to be always active on every reboot.

3.2.4 Testing

Once the configurations on the Raspberry are complete, the Jetson Nano is connected (as in the diagram in Figure 3.3) and it is verified that it can actually be read by the Raspberry using the command: `arp -n -i eth0`.
When the IP of the Jetson is identified, it is accessed via SSH and it is verified that the board is actually connected to the Internet by executing the command: `ping google.com`.

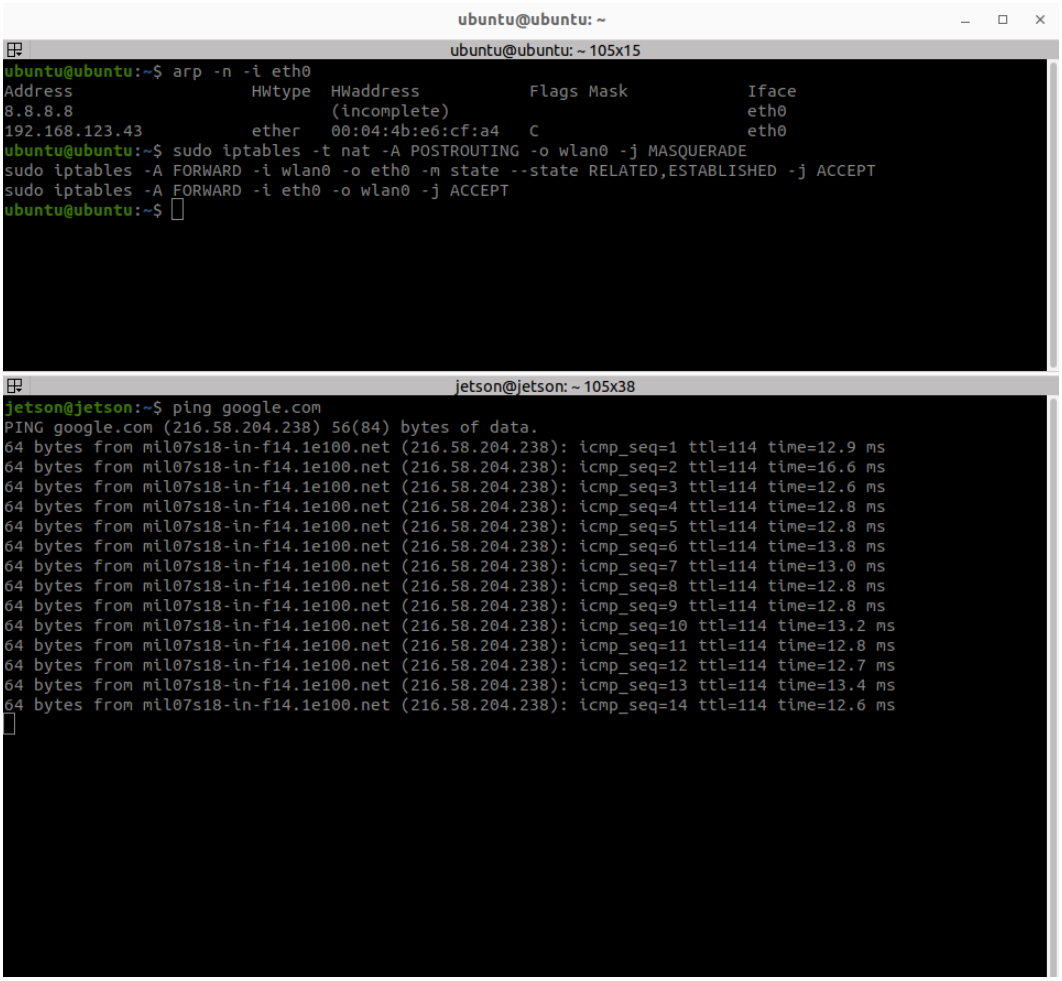

Figure 3.5. Jetson ping google.com

3.3 Implementation of communication via FastDDS middleware

As seen in Chapter 2, FastDDS is a middleware standard for **real-time communication** used by ROS2. Therefore, this solution was initially used.

3.3.1 Container Creation

A container was set up inside the Jetson Nano and on the computer. The reason for this is that the robotic platform has computers with old Ubuntu operating systems inside, so they cannot support the recent versions of ROS2. Using docker solves this problem and makes it possible to work more cleanly, which is the reason why docker is also used on computers.

A custom container was built using using the Dockerfile in the Appendix A, without making any project-specific changes, just to test how the communication between nodes works. The `cpp_pubsub` node from the ROS2 tutorials was built inside the container, which is an ideal tool for testing.

It is specified that the Dockerfile for AMD64 platforms is identical except for the first line, which specifies the source image as `FROM_IMAGE=osrf/ros:humble-desktop`.

3.3.2 First communication test

With the containers ready on Jetson and computer, the `ROS_DOMAIN_ID` in **channel 4** is set on both devices and the talker and listener nodes are launched respectively, as seen in the Figure 3.6.

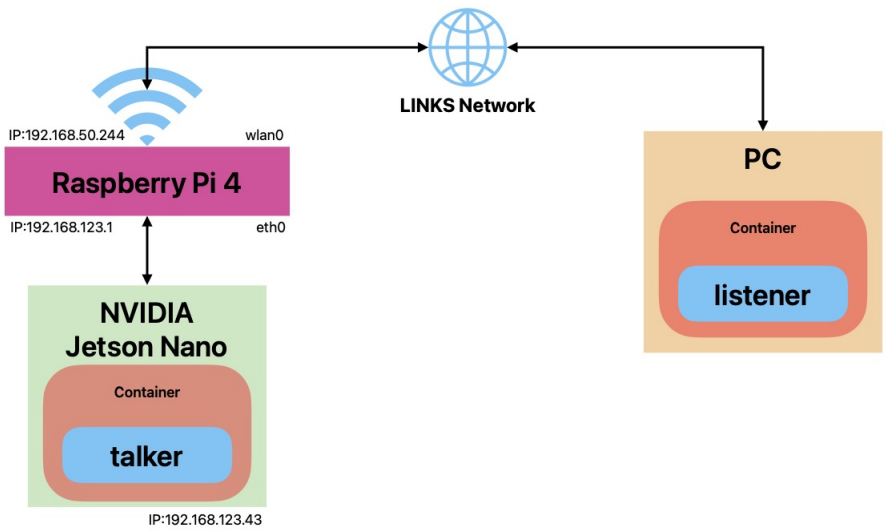

Figure 3.6. Communication scheme

As expected, the two nodes are unable to communicate with each other, as they are in two different subnets. In the first instance, an attempt was made to forward the `ROS_DOMAIN_ID` traffic via the specific port in which FastDDS communicates. After several failed attempts, communication could not be established.

However, it was observed that when launching a talker node in the bridging Raspberry Pi 4, its topic is visible from both devices (Jetson and PC). Various communication tests are carried out and the following results are obtained:

1. **Jetson talker**: Raspberry *reads* and *listens* the topic. PC *cannot read* and *listen* the topic.

2. **PC talker**: Raspberry *reads* and *listens* the topic. Jetson *cannot read* and *listen* the topic.

3. **Raspberry talker**: Jetson *reads* and *listens* the topic. PC *reads* and *listens* to the topic.

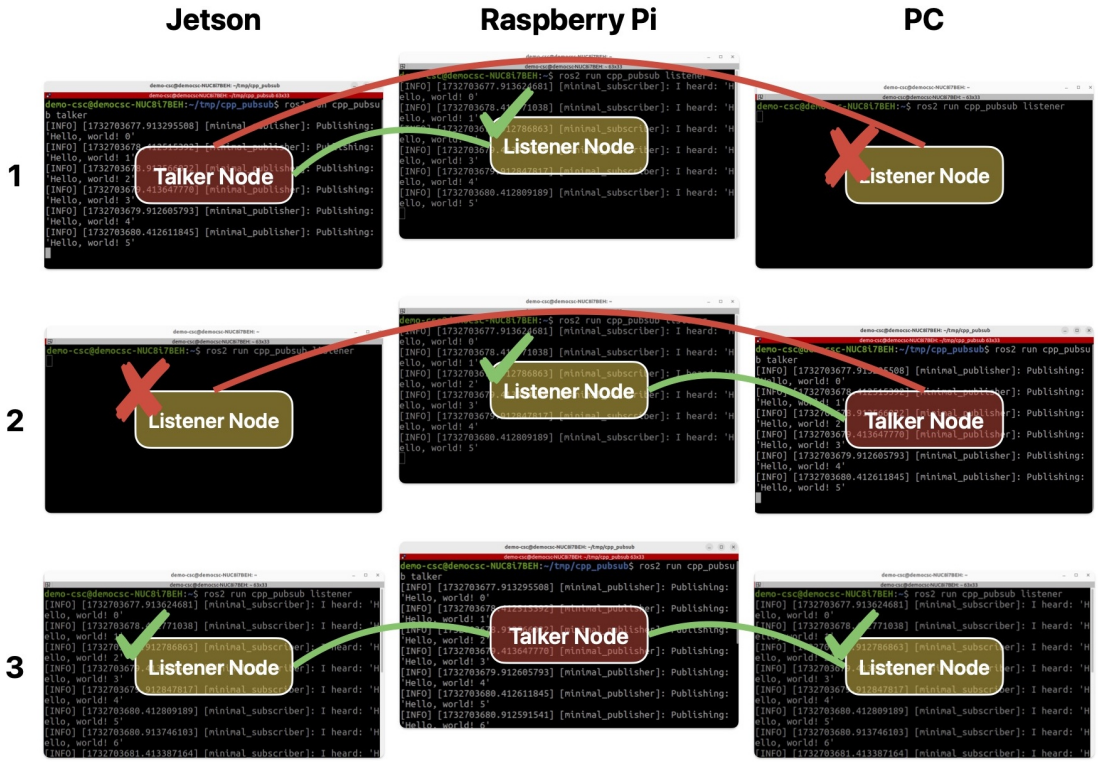

Figure 3.7. First communication test

This result shows that the Raspberry can listen to both subnets, but cannot forward and unify them.

3.3.3 Second communication test

Since a direct port-forward connection is not feasible, an attempt was made to proceed using the **Husarnet VPN**. Chapter 2.6 explains specifically what VPNs are.

In order to use Husarnet, configuration changes had to be made to the `Dockerfile` and `docker-compose`, so that a husarnet container could be pulled up in parallel, through which the main container takes the **service** for the network configuration.

The Dockerfile is the same in the Appendix A and docker-compose is in the Appendix D.

In addition to configuring the machines, Husarnet requires configuration on the site, going to create an account and creating a network - in this specific case called `go1_network`. On this page all devices that connect to the network are visible, showing their name, status and address.

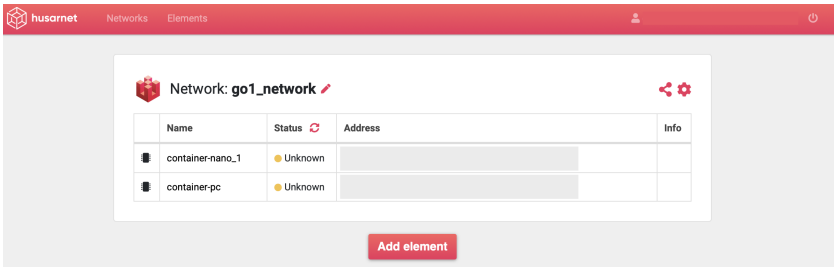

Figure 3.8. Graphical user interface of the Husarnet site.

With this new configuration, as can be seen from the Figure 3.9, the Jetson was able to communicate with the computer.

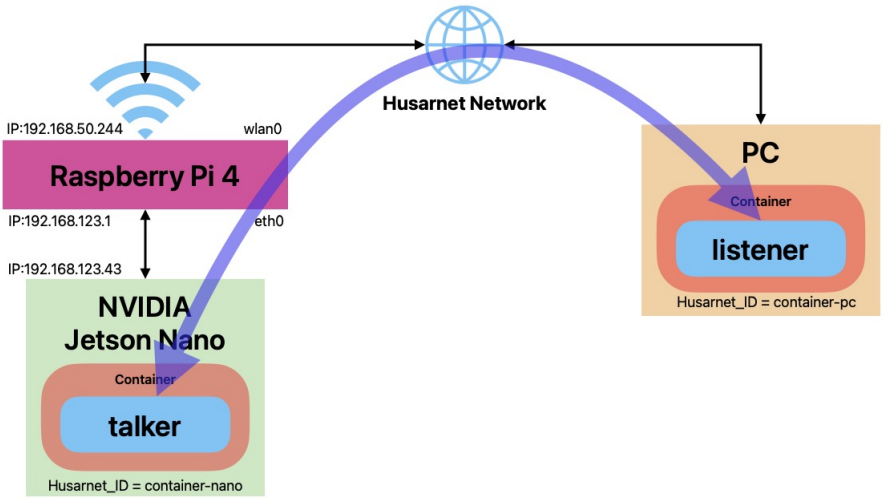

Figure 3.9. Communication scheme via Husarnet.

Communications problems began to emerge from the outset: the domain did not synchronise immediately, and it was necessary to change channels several times before proper communication was established.
Communication does not remain stable and, as the number of topics increases, it is not possible to see all the topics in the domain. Probably, FastDDS loses packets when passing through the VPN.

This solution was discarded due to the low reliability of the results.

3.4 Networking with Zenoh middleware

As mentioned in Chapter 2, FastDDS is the standard for ROS2, so Zenoh was not initially considered as a possible solution. But, on analysis of the **GitHub repository rmw_zenoh**, it was seen that the latter specifically addresses and solves the problem with this configuration.

3.4.1 Dockerfile changes

In order to apply this configuration, changes must be made to the Dockerfile to install the repository under consideration and to change the middleware protocol used by ROS2 for communication.

The `fastdds.xml` file has been replaced by the `routerconfig.json5` file, the latter of which is needed within the PC and Jestson in order to define a communication bridge between them and the Raspberry, by entering the raspberry's ip address and the relevant port used by zenoh on line 18.
The configuration used was taken from the **rmw_zenoh repository**, in the directory: `rmw_zenoh/rmw_zenoh_cpp/config/DEFAULT_RMW_ZENOH_ROUTER_CONFIG.json5` and renamed to `routerconfig.json5`.
Lines 14 to 20 of the `routerconfig.json5` of the PC and the Jetson are shown in listing 3.4 and listing 3.5.

```
16  connect: {
17    endpoints: [
18      "tcp/192.168.50.244:7447"
19    ],
20  },
```

Listing 3.4. PC's routerconfig.json5

```
16  connect: {
17    endpoints: [
18      "tcp/192.168.123.1:7447"
19    ],
20  },
```

Listing 3.5. Jetson's routerconfig.json5

3.4.2 Use of Zenoh

In contrast to previous network configurations, in this setup the Raspberry Pi assumes a more active role, functioning as a 'bridge' that that connects the client router inside the `container_pc` with the client router inside the `container_jetson`.

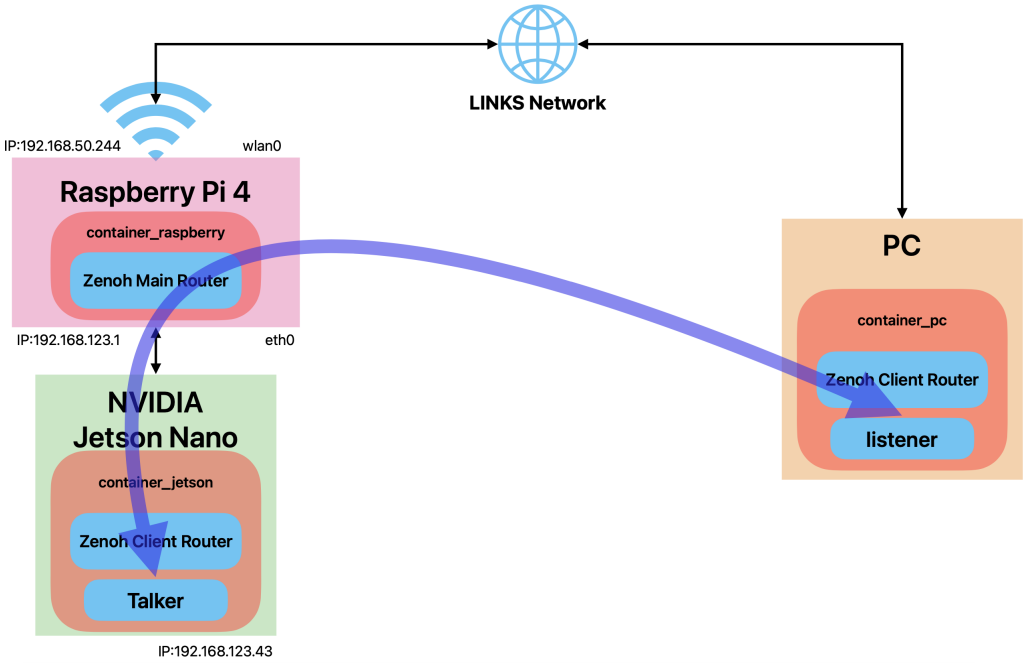

Figure 3.10. Communication scheme using Zenoh.

Below are the operating commands for ROS2 running through zenoh.

1. A terminal is opened on the `container_raspberry` and the following command is executed:

   ```
   $ ros2 run rmw_zenoh_cpp rmw_zenohd
   ```

   This node is used to run the client router zenoh, which is essential for discovering nodes, and which turns on the router - considered to be the main router - to connect client routers. As mentioned in Chapter 2.4.4, without this node active, any ROS2 command would produce the error shown in Figure 2.10.

2. Step 1 is also repeated in `container_pc` and `container_jetson`.

3. Next, the talker node is executed on a new terminal of the `container_jetson` and the listener node on a new terminal of the `container_pc`.

   As can be seen from Figure 3.11, the two nodes communicate correctly.

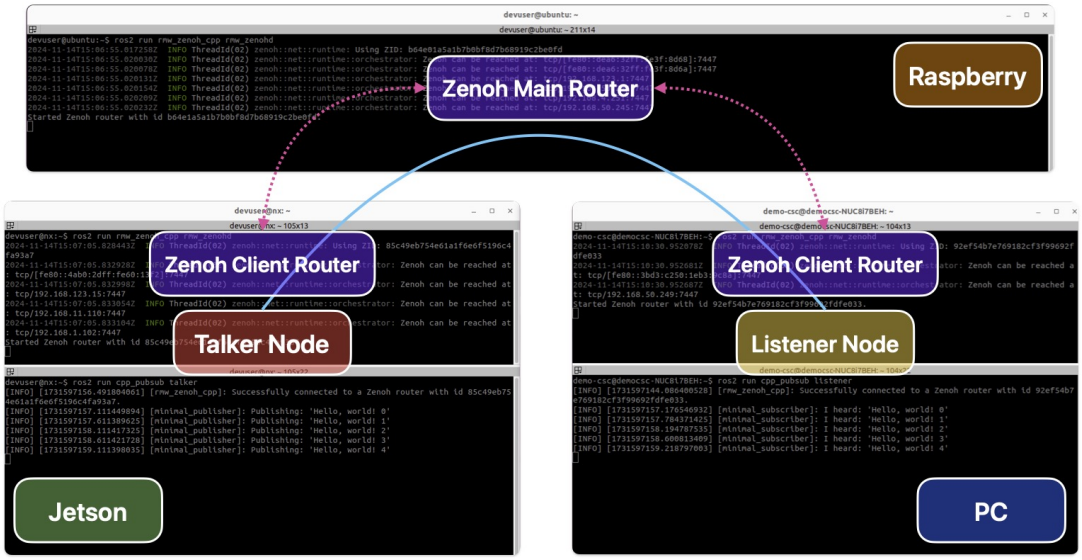

Figure 3.11. Communication between Jetson and PC.

3.4.3 Achievements

The results obtained with this middleware meet the requirements of the project, managing to overcome the limitations encountered with FastDDS, achieving stable and efficient communication between the devices involved.
Configuration through client routers, deployed in the various containers, consolidated the Raspberry Pi as a bridging node, facilitating a distributed network in which Jetson and PC devices communicate without errors. Therefore, Zenoh was confirmed as the best choice for rmw in this context.

Chapter 4

Robot implementation

In this chapter, the implementation of the project on the robot platform will be described and explained, up to the achievement of the project purpose.

In the first part, the networking configuration developed in the previous chapter is carried out, testing and verifying the effective communication between the PC and the robot.
Next, the packages required to control the robot with ROS 2 are installed and executed, allowing the platform to be controlled from the computer.
Finally, NAV 2 is implemented and configured, successfully testing autonomous navigation.

4.1 Robotic platform preparation

In order to apply the configuration tested in the previous chapter, the robot hardware must initially be prepared, proceeding with the connections and networking.

4.1.1 Assembly of the support structure and positioning of electronic components

In order to allow the robot's navigation to function correctly, it is necessary to place the Raspberry on the back of the robot, along with a 2D LiDAR and a battery to power both. Therefore, a structure built on three levels is set up: on the first level the Raspberry is placed, on the upper level the battery with the surplus of cables is placed, and on the last level the 2D LiDAR is placed.

Figure 4.1 shows the picture of the structure on the robot.

Figure 4.2 shows the configuration created with the position coordinates of the 2D LiDAR with respect to the robot's reference system. These coordinates will then be used in Chapter 4.3.

Figure 4.1. Photos of assembled structure

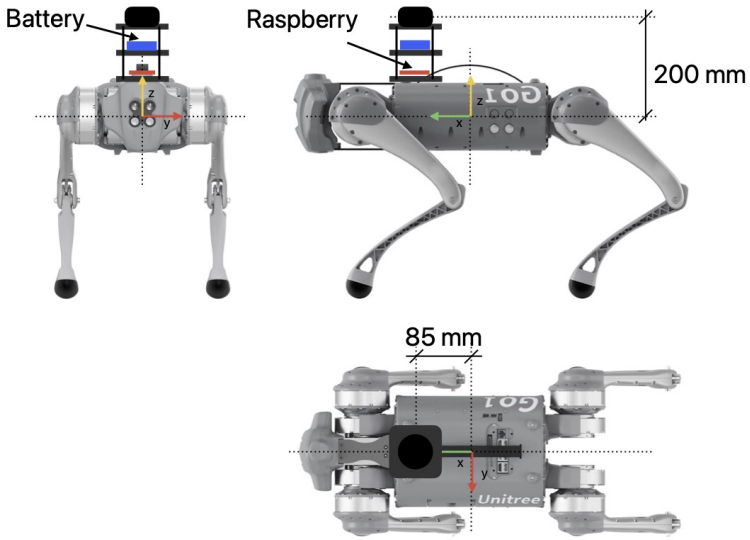

Figure 4.2. LiDAR position scheme

4.1.2 Creating robot networks

In order to connect the robot platform to the office network, the configuration developed in Chapter 3 is used, choosing Zenoh as the middleware.
Then, the Raspberry Pi 4 is connected to the robot's Ethernet port (shown in Figure 3.2) allowing access to the robot's internal subnet.

For the configuration of the robot, the containers drawn in Chapter 3 for the Jetson, the Raspberry and the PC will be reused.

Access to internal robot boards and docker activation

With all devices switched on, access is gained from the computer via SSH to the Raspberry Pi, previously configured in Chapter 3. The command `arp -n -i eth0` is executed, in order to see the visible boards connected to the eth0 interface.

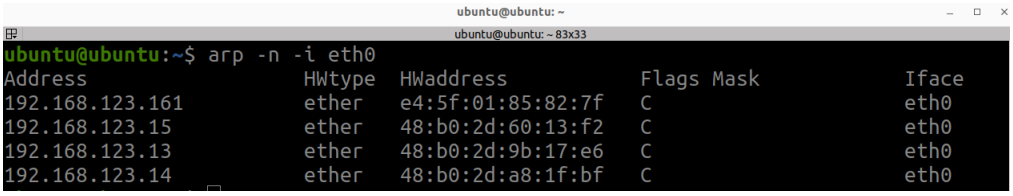

Figure 4.3. List of connected boards

As can be seen from Figure 4.3, all the boards of the Unitree Go1 were connected to the network created by the Raspberry. In order to create a connection between robot and PC, one of these 4 boards must be selected. The Jetson Xavier Series - `IP 192.168.123.15` - was chosen as it offers more computing power and storage space than the others.

To make this board accessible to the computer without having to go through the Raspberry's terminal each time, port forwarding is carried out by executing the following command on the Raspberry:

```
$ sudo iptables -t nat -A PREROUTING -p tcp --dport 4015 -j
   DNAT --to-destination 192.168.123.15:22
$ sudo iptables -t nat -A POSTROUTING -d 192.168.123.15 -j
   MASQUERADE
```

Then the board can be accessed directly by typing in the chosen port and the IP of the Raspberry:

```
$ ssh -p 4015 unitree@192.168.50.244
```

Inside the Jetson, to access the Internet, it is necessary to configure the Raspberry's IP as the default gateway, as it routes the network's data traffic.

```
$ sudo ip route add default via 192.168.123.1 dev eth0
```

Since this board, as mentioned in Chapter 2.8.4, provides an outdated operating system, it is not possible to install the latest version of ROS 2 Jazzy on it. Therefore, Docker is installed and a container is created, using the Dockerfile previously used in Chapter 3.4.1.

ROS communication test between PC and robot

Once all the required containers have been activated, the Chapter 3.4.2 test is performed. Figure 4.4 shows the results obtained.

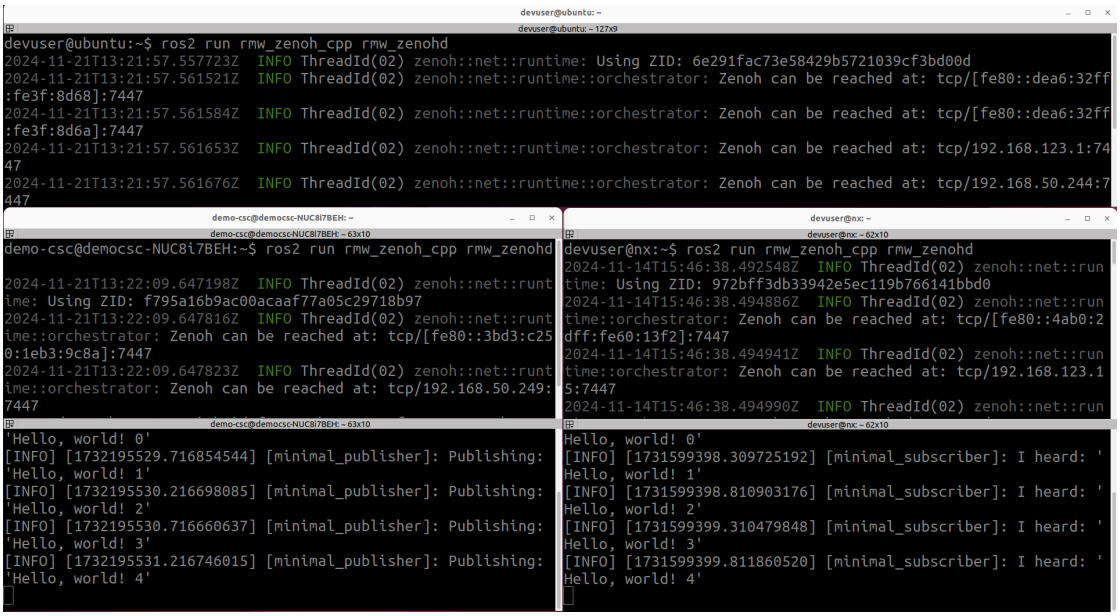

Figure 4.4. Pub/Sub communication test

Having established a communication connection between the PC and the robot, the next step can now be taken.

4.2 Moving the robot via PC

The establishment of communication between the PC and the robot allows the platform's movements to be controlled by accessing the `unitree_legged_sdk`[17].

The Unitree Go1 robot has a software architecture that includes native support for ROS 1. Subsequently, Unitree Robotics made the `unitree_ros2_to_real` repository available on GitHub [20], which allows the robot to be controlled via ROS 2 by sending UDP commands. This solution allows both low-level control, giving access to all the robot's joints, and high-level control, coordinating the robot's direction and speed of travel.
In this project, only high-level control will be used.

4.2.1 Installation and execution of the nodes required for communication with the robot

The GitHub repository of *katie-hughes* `unitree_ros2` [9], folk from Unitree's `unitree_ros2` package with some modifications that make it easier to add custom controls.

Below are the commands used to install the `unitree_ros2` package inside the robot's Jetson board container.

```
$ git clone https://github.com/GabrieleCesare/lcm-1.4.0.git
$ cd lcm-1.4.0
$ mkdir build
$ cd build
$ sudo cmake ..
$ sudo make
$ sudo make install # this will install lcm
$ sudo ldconfig -v # updates the shared library cache
$ cd ../..

$ mkdir -p unitree_ws/src
$ cd unitree_ws/src
$ git clone https://github.com/katie-hughes/unitree_ros2.git
$ cd ..
$ colcon build
```

Errors were encountered during the installation of lcm. Errors resolved by editing the file `lcm-python/module.c` replacing lines 46, 47 and 48 with:

```
46      pylcmeventlog_type.tp_base = &PyType_Type;
47      pylcm_type.tp_base = &PyType_Type;
48      pylcm_subscription_type.tp_base = &PyType_Type;
```

With this package, the `udp_high` node can be launched. This is a C++ node designed for passing high-level UDP commands and states. It subscribes to the `high_cmd` topic, converts the commands into a UDP message and sends it to Go1. It publishes the state (received via UDP from Go1) in the topic `high_state`.

```
$ ros2 launch unitree_legged_real high.launch.py
```

The diagram of the Figure 4.5 shows how it works.

The execution of this launch also allows the activation of the launch `load_go1.launch.py`, responsible for the publication of the robot's **TF Tree** through the topics `/tf` and `/tf_static`. It is therefore necessary to define, within the `robot.xacro` file, the structure of the parent-child relationships between the frames of the robot. As seen in Chapter 2.7, in the TF Tree it is essential to specify the Cartesian position of the LiDAR (`base_laser`) with respect to the robot's main reference system (`base_link`). Therefore, with reference to the LiDAR mounting position shown in Figure 4.1, we modify the `robot.xacro` in lines

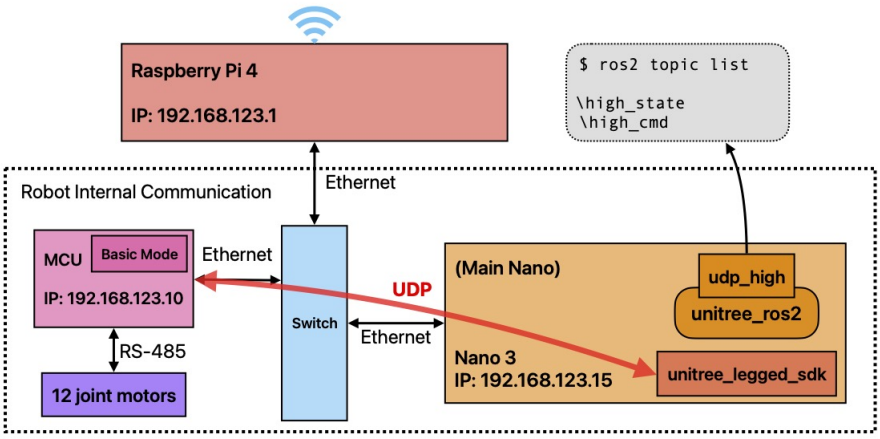

Figure 4.5. Schema esecuzione nodo `udp_high`

6, 7, 8 and 9 by activating the condition for Nav2 and defining the Cartesian coordinates of the LiDAR position:

```
6     <xacro:arg name="use_nav2_links" default="true"/>
7     <xacro:arg name="lidar_x_offset" default="0.085"/>
8     <xacro:arg name="lidar_y_offset" default="0.0"/>
9     <xacro:arg name="lidar_z_offset" default="0.20"/>
```

unitree_nav

In order to control the robot's movements, one needs to install a further package, again developed by *katie-hughes*, which allows navigation. The `unitree_nav` repository from GitHub [8] is installed in the same workspace that was previously created.

```
$ cd unitree_ws/src
$ git clone https://github.com/ngmor/unitree_nav.git
$ cd ..
$ sudo apt install python3-vcstool
$ vcs import < src/unitree_nav/nav.repos
$ cd src/rslidar_sdk_ros2
$ git submodule init
$ git submodule update
$ cd ../..
$ sudo apt-get install -y  libpcap-dev
$ sudo apt install ros-jazzy-nav2-msgs
$ sudo apt install python3-colcon-common-extensions
$ colcon build
```

This package launches the `cmd_processor` node, which is responsible for handling commands and converting them into `HighCmd` messages that can be read by the high-level UDP node `udp_high` and then sent to Go1 for motion control. This node processes the messages `geometry_msgs/Twist` sent in the topic `cmd_vel` to make the robot move. It also provides various services to activate functions built into Go1. The most important services include `lay_down` to command the robot to the rest position (lying down) and `stand_up` to raise the robot to a condition that allows it to move. This node is essential in order to work with the Nav 2 stack.

```
$ ros2 run unitree_nav cmd_processor
```

The diagram of the Figure 4.6 shows how it works.

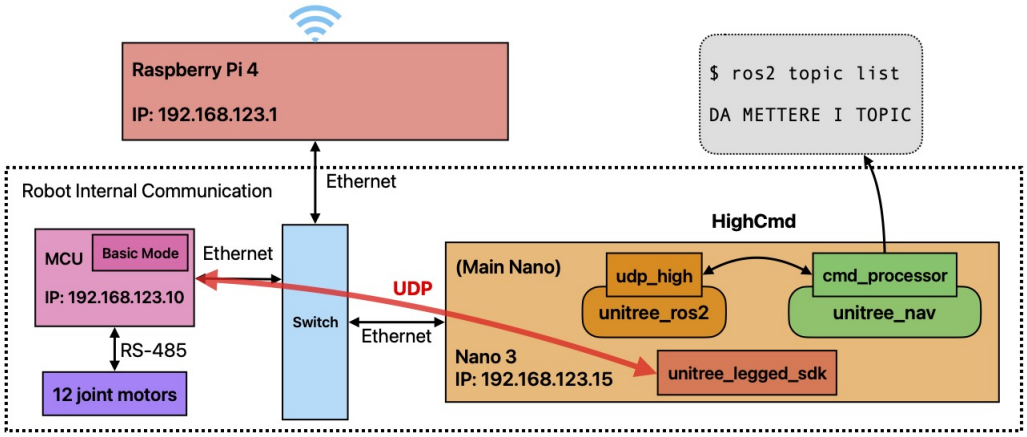

Figure 4.6. Node Execution Diagram `cmd_processor`

Both nodes can be activated with the launch file `control.launch.py` inside the project `unitree_nav`, specifying that Rviz is not needed.

```
$ ros2 launch unitree_nav control.launch.py use_rviz:=false
```

Important consideration: when the robot platform is switched on, it always has the date and time of its last switch-off. It is therefore necessary to align its time with the other devices by running a `sudo apt-get update` from the terminal each time the robot is powered up.
Without this, communication errors would occur when trying to run Nav2.

4.2.2 Test driving the robot with the computer keyboard

Once the robotics platform is ready and all nodes and topics are switched on and active, the control can be performed from the computer.
From the container on the PC it is firstly checked that all topics are active. The list of topics is shown in Figure 4.7.

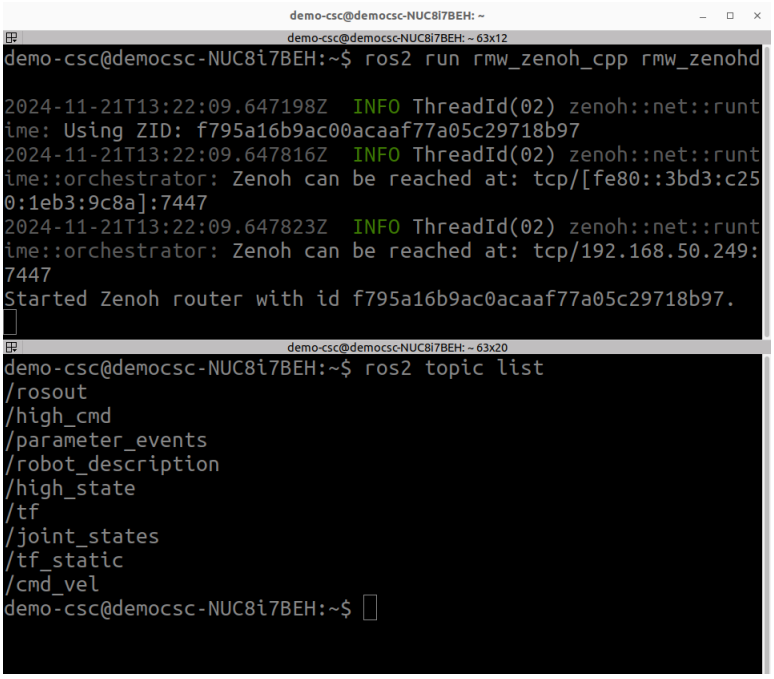

Figure 4.7. Topic list

From the computer, the `teleop_twist_keyboard` node is executed, allowing the user to drive the robot via the computer keyboard.

```
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

The objectives of this section were fully achieved, allowing the robot to be controlled via the keyboard.

4.3 Robot Control with Navigation 2

The final phase of the project, in which autonomous navigation using the Navigation 2 stack is configured and enabled, is carried out.

First of all, changes must be made to the Dockerfile used to build the container on the PC, since the version for the Jazzy distro of NAV 2 must be installed. The updated Dockerfile is listed in Appendix B.

The container is run and a first step is to create a `nav2_param.yaml` file in which to customise the values of the parameters that Nav2 needs to function. The corrected

`nav2_param.yaml` is listed in the Appendix F.
The main customised parameters are:

- all definitions of `use_simple_time` are set to `False`.
- **Row 29**: it is defined as `OmniMotionModel` as a type of robot model. This is because the Unitree Go1 robot is able to move in any direction in the plane (x, y) and rotate around its axis (yaw) simultaneously.
- **Row 111**: the `DWWBLocalPlanner` is set as the FollowPath plugin, this is because the robot is currently unable to move using the `MPPIController`.
- **row 164 and 213**: The shape and size of the robot footprint is defined: `"[ [0.410, 0.350], [0.410, -0.350], [-0.410, -0.350], [-0.410, 0.350] ]"`.
- **row 179 and 220**: the observation resource `scan` is defined. The alternative `scan3d` is chosen in case a 3D LiDAR is fitted.

4.3.1 2D LiDAR and Odometry node execution

As mentioned in Chapter 2.6.2, navigation needs a source of information about the external environment. This information comes from the 2D LiDAR previously presented in Chapter 2.8.5. This sensor was connected to the Raspberry Pi, so it was chosen to run the node in the container inside the Raspberry.

In order to make the USB port where the sensor is connected accessible to the container, it was necessary to modify the `docker-compose.yaml` by specifying which device to connect. This specification can be seen in row 53 of the Apendix E.
Having made this change, the container is rebuilt and the user is given permissions to access this interface with the following command:

```
$ sudo chmod 777 /dev/ttyUSB0
```

Now the node responsible for activating LiDAR must be installed and then executed. The package `sllidar_ros2` from the GitHub [21] repository is downloaded and installed.

The node is executed with the following command:

```
$ ros2 launch sllidar_ros2 sllidar_a3_launch.py frame_id:=
   base_laser
```

As can be seen from the command, it defined the `frame_id` by calling it `base_laser`. This is because within the BT located in the `robot.xacro` file, in lines between 67 and 72 below, the link name is defined as `base_laser`.

```
67      <link name="base_laser" />
68      <joint name="base_link_to_lidar" type="fixed">
69          <origin rpy="0 0 0" xyz="$(arg lidar_x_offset) $(arg
               lidar_y_offset) $(arg lidar_z_offset)"/>
70          <parent link="base_link"/>
71          <child link="base_laser"/>
72      </joint>
```

Odometry

However, the navigation needs information regarding odometry in order to work. The robotics platform does not give access to this information, so a repository from GitHub [1] that estimates of 2D odometry based on planar laser scans was chosen. It is decided to download and install this package inside the PC container, though it going to modify the file launch `rf2o_laser_odometry.launch.py` line 26 in the following way `'odom_topic' : '/odom'`. This will make sure that when the node is launched it will create a topic named `/odom`, a nomenclature recognized by Nav2.
The node is run with the following command:

```
$ ros2 launch rf2o_laser_odometry rf2o_laser_odometry.launch.
   py
```

4.3.2 Nav2 execution

All the necessary packages are installed and all the nodes have been launched. Now it is possible to proceed by going to run Nav2 with the following command:

```
$ ros2 launch nav2_bringup bringup_launch.py params_file:=/
   home/demo-csc/nav2_params.yaml map:=/home/demo-csc/
   csc_office.yaml use_sim_time:=False autostart:=False
```

It can be seen that along with the execution of `bringup_launch.py` is specified the directory of the file `nav2_params.yaml`, previously created and reported in Appendix F, and the directory where to find the map in which the robot is to navigate.

Running this launcher allows to apply the parameters set by going to call all the key nodes for autonomous robot navigation.

Now, in order to run Nav2, Rviz is used as the visualization and command tools. It is run with the command:

```
ros2 run rviz2 rviz2 -d install/nav2_bringup/share/
   nav2_bringup/rviz/nav2_default_view.rviz
```

Running this command, the option "`-d`" allows to specify an Rviz2 configuration file set to have an interface to Nav2. If no changes were made, the configuration would have errors, so lines 28 and 29 should be deleted.

So, as seen from Figure 4.8, the office map has opened and everything is ready to start autonomous navigation.

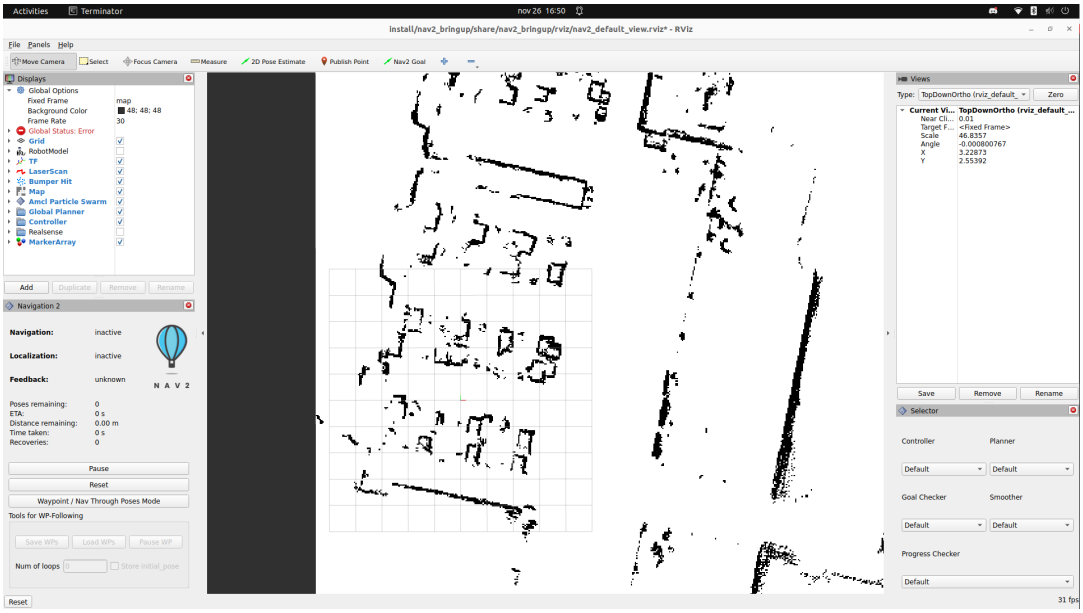

Figure 4.8. Nav2 Rviz view

Autonomous navigation test

After providing the initial **2D Pose Estimate** of the robot, the costmap layers are activated and the LiDAR scans become visible on Rviz. Then the **Navigation2 Goal** is defined and immediately the planned path appears on the map. The robot starts to move following that path.

However, it is noticed that the robot's movements are not smooth, but are rather uncertain and jerky. This occurs because Nav2's frequency of issuing motion commands is too low to ensure smooth movement. In the case of the TurtleBot, this does not happen because the robot executes the first movement command (e.g., in a straight line) until it receives a new command to stop or change direction. In the case of the Unitree Go1 control, the situation is different: the platform moves in the indicated direction until it receives a new motion command. The instant the command ceases, the robot stops instantly.

In order to minimize this inconvenience, we went to change the command frequency parameter in the :

```
navigation2/nav2_bt_navigator/behavior_trees/
   navigate_to_pose_w_replanning_and_recovery.xml
```

It goes to modify the `RateController` in line 13 by setting it to 30 Hz.

This modification improved the motion of the robot, but did not achieve an optimal result.

4.3.3 Achievements

As can be seen from the Figures 4.9, the robot reaches the goal; however, once it arrives at the destination point, the inaccuracy of the odometry node of the actual position calculation causes the robot to keep moving and rotating to align with the position and pose defined in the **Navigation2 Goal**.

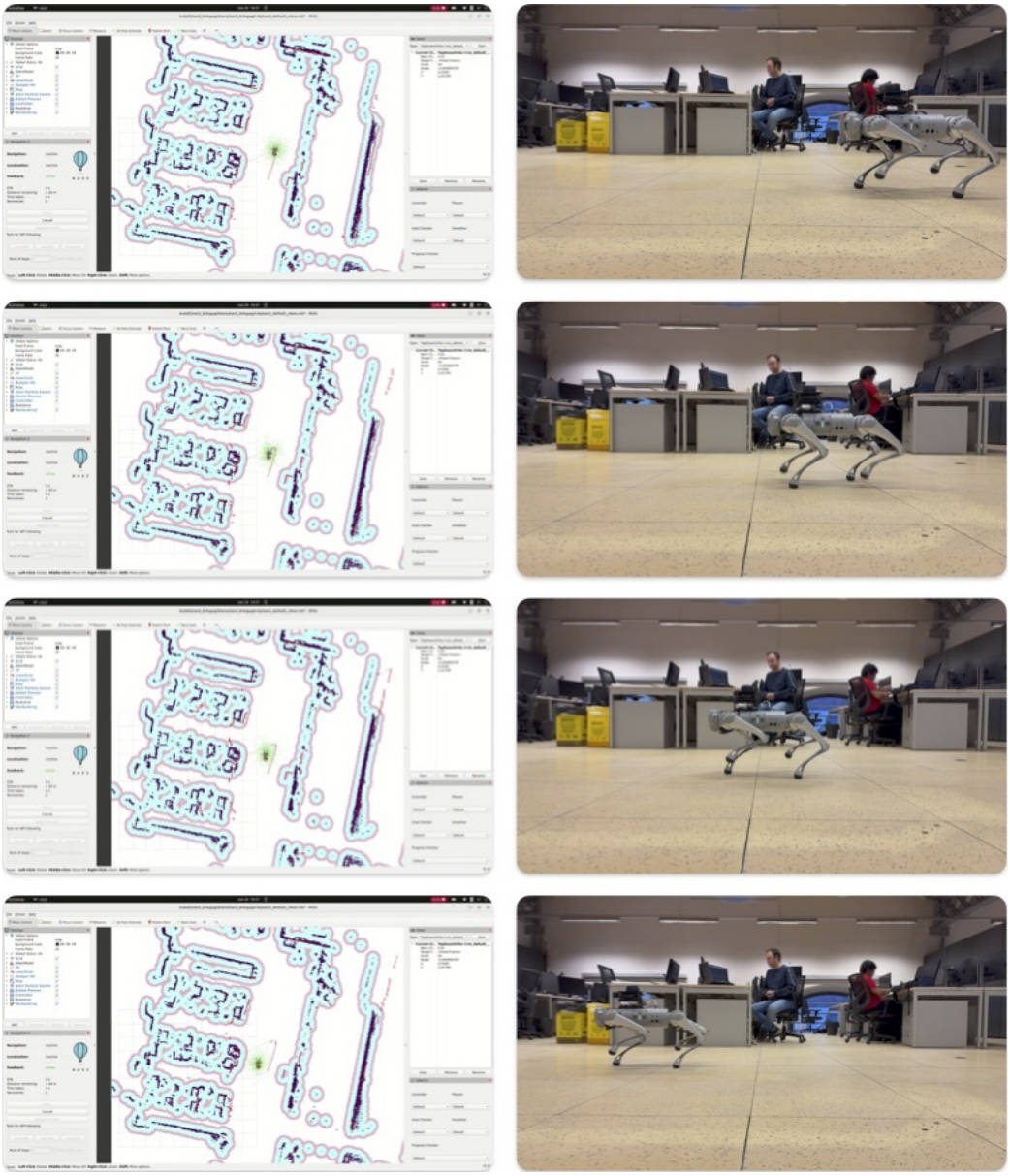

Figure 4.9. Set of photos of navigation to goal

Conclusions

The project reached its conclusion by hitting its intended goal of integrating an autonomous navigation system on the Unitree Go1 robotic platform through the use of the Navigation 2 stack.

Although the path was also characterized by difficulties and unsuccessful attempts, each obstacle faced helped to identify the optimal solution, allowing all the problems encountered to be successfully overcome.

This study consisted of several phases, going first to address and solve the communication problem between the robotic platform and the computer. At this stage, important limitations in the ROS 2 FastDDS communication protocol emerged, which required time and effort to find appropriate solutions. An optimal solution was found with the rmw Zenoh. In the second phase, the Nav2 stack was installed and configured, allowing the robot to have its own autonomous navigation. Even at this stage the work was not without obstacles, but in the end the goal was achieved.

Although the results obtained demonstrate the effectiveness of integrating the autonomous navigation system, some operational limitations have emerged that need further improvement. Currently, the robot does not move completely smoothly; having received irregular motion commands these resulted in jerky movement. In addition, inaccuracies about the robot's actual position calculated by the odometer node meant that even though the robot had reached the target point, it continued to move as the odometer values varied continuously, not reassuring it that it had reached the target.

To address the first problem, one possible solution that can be adopted is to create a dedicated node that sends in the topic `cmd_vel` constant motion commands. In this way, even if the frequency of directions received by Nav2 turns out to be relatively low, the robot would still receive regular and continuous movement instructions, improving its smoothness in movement. Regarding the behavior to be adopted when reaching the destination, a possible solution could be achieved by increasing the tolerance of the target point radius and pose angle, thus including possible corrections related to odometric inaccuracies. These interventions are prospectively effective solutions that could be implemented for the purpose.

This thesis leads the way for other possible developments and future applications, from multiplane autonomous navigation to the integration of advanced artificial intelligence systems. Now that remote control of the robotic platform is enabled, more freedom can

be acted upon in integrating IoT devices that will allow the robot to interact with the environment.

The robotic platform will practically be able to be used to operate in complex environments such as hospitals, industrial facilities or public spaces, increasing logistical efficiency and safety. In areas such as emergency and civil defense, four-legged robots could be vital in reaching dangerous or difficult-to-access areas, offering help during search and rescue operations.

In conclusion, the results of this study are an important starting point for exploring new challenges in autonomous robotics, with potential impacts on robot applications in real-world complex environments.

Appendix

A Dockerfile ARM architecture - Humble

```
ARG FROM_IMAGE=ros:humble-ros-core-jammy
ARG PKG_NAME=pkg_image_name
ARG PKG_WS=/ws

######### CLONING and INSTALLING #################
FROM $FROM_IMAGE AS base
SHELL ["/bin/bash", "-c"]

# Install the main dependences
ARG PKG_WS
WORKDIR $PKG_WS/src/$PKG_NAME
COPY ./ .
RUN apt-get update && \
    apt-get install -y build-essential git cmake libasio-dev
       && \
    apt-get install -y ros-humble-geometry2 && \
    apt-get install -y ros-humble-geometry-msgs && \
    apt-get install -y ros-humble-tf2* && \
    apt-get install -y ros-humble-robot-state-publisher && \
    apt-get install -y ros-humble-xacro && \
    rm -rf /var/lib/apt/lists/*

RUN apt-get update && apt-get install --no-install-recommends
    -y \
    build-essential \
    git \
    python3-colcon-common-extensions \
    python3-colcon-mixin \
    python3-rosdep \
    python3-vcstool \
    && rm -rf /var/lib/apt/lists/*


```

```
# bootstrap rosdep
RUN rosdep init && \
  rosdep update --rosdistro $ROS\_DISTRO

# setup colcon mixin and metadata
RUN colcon mixin add default \
      https://raw.githubusercontent.com/colcon/colcon-mixin-
         repository/master/index.yaml && \
    colcon mixin update && \
    colcon metadata add default \
      https://raw.githubusercontent.com/colcon/colcon-
         metadata-repository/master/index.yaml && \
    colcon metadata update

# install ros2 packages
RUN apt-get update && apt-get install -y --no-install-
   recommends \
    ros-humble-ros-base=0.10.0-1* \
    && rm -rf /var/lib/apt/lists/*

######### BUILDING #################
FROM base AS overlay

# Build all the packages
ARG PKG\_WS
WORKDIR $PKG_WS
COPY --from=base $PKG\_WS/src $PKG_WS/src
RUN source /opt/ros/humble/setup.bash && \
    colcon build --cmake-args -DCMAKE_CXX_FLAGS="-w"

# Setup the entry point and config file
ARG pkg_image_name
COPY docker/entrypoint.sh /
COPY docker/fastdds.xml /root/.ros

# Set the entrypoint
RUN chmod 777 /entrypoint.sh
ENTRYPOINT [ "/entrypoint.sh" ]

######### DEV ######################
FROM overlay as dev

# Set the dev arguments
ARG USERNAME=devuser
ARG UID=1000
ARG GID=${UID}

```

```
76 # Add dev dependences
77 RUN apt-get update && \
78     apt-get install -y --no-install-recommends && \
79     apt-get install -y --no-install-recommends gdb && \
80     apt-get install -y --no-install-recommends gdbserver && \
81     apt-get install -y --no-install-recommends nano && \
82     apt-get install -y --no-install-recommends bash-
          completion && \
83     # apt-get install -y --no-install-recommends terminator
          && \
84     rm -rf /var/lib/apt/lists/*
85
86 # Create new user and home directory
87 RUN groupadd --gid $GID $USERNAME && \
88     useradd --uid ${GID} --gid ${UID} --create-home ${
          USERNAME} && \
89     echo ${USERNAME} ALL=\(root\) NOPASSWD:ALL > /etc/sudoers
          .d/${USERNAME} && \
90     chmod 0440 /etc/sudoers.d/${USERNAME} && \
91     mkdir -p /home/${USERNAME} && \
92     chown -R ${UID}:${GID} /home/${USERNAME}
93
94 # Setup the access right
95 ARG PKG\_WS
96 RUN chown -R ${UID}:${GID} ${PKG_WS}
97
98 # Create the fastdds.xml configuration
99 ARG PKG_NAME
100 RUN mkdir /home/${USERNAME}/.ros
101 COPY docker/fastdds.xml /home/${USERNAME}/.ros
102 RUN chown -R ${UID}:${GID} /home/${USERNAME}/.ros
103
104 USER ${USERNAME}
105 RUN echo "source /entrypoint.sh && source /etc/
       bash_completion" >> /home/${USERNAME}/.bashrc
```

Listing 4.1. Dockerfile to build the container in arm64 device with Humble distro

B Dockerfile PC - Jazzy

```
ARG FROM_IMAGE=osrf/ros:jazzy-desktop
ARG PKG_NAME=pkg_image_name
ARG PKG_WS=/ws

######### CLONING and INSTALLING #################
FROM $FROM_IMAGE AS base
SHELL ["/bin/bash", "-c"]

# Install the main dependences
ARG PKG_WS
WORKDIR $PKG_WS/src/$PKG_NAME
COPY ./ .
RUN apt-get update && \
    apt-get install -y build-essential git cmake libasio-dev
       && \
    apt-get install -y ros-jazzy-geometry2 && \
    apt-get install -y ros-jazzy-geometry-msgs && \
    apt-get install -y ros-jazzy-tf2* && \
    apt-get install -y ros-jazzy-robot-state-publisher && \
    apt-get install -y ros-jazzy-xacro && \
    apt-get install -y wget unzip openjdk-8-jdk python3-
       vcstool libpcap-dev ros-jazzy-nav2-msgs python3-rosdep
        python3-colcon-common-extensions && \
    apt-get install -y ros-jazzy-bond && \
    apt-get install -y ros-jazzy-bondcpp && \
    apt-get install -y ros-jazzy-test-msgs && \
    apt-get install -y libgraphicsmagick++-dev && \
    apt-get install -y ros-jazzy-diagnostic-updater && \
    apt-get install -y ros-jazzy-behaviortree-cpp && \
    apt-get install -y libceres-dev && \
    apt-get install -y libxsimd-dev && \
    apt-get install -y libxtensor-dev && \
    apt-get install -y ros-jazzy-ament-cmake-core && \
    rm -rf /var/lib/apt/lists/*

######### BUILDING #################
FROM base AS overlay

# Build all the packages
ARG PKG_WS
WORKDIR $PKG_WS
COPY --from=base $PKG_WS/src $PKG_WS/src
COPY docker/entrypoint.sh /
COPY docker/routerconfig.json5 /root/.ros
RUN source /opt/ros/jazzy/setup.bash
```

```
43
44 WORKDIR $PKG_WS/src
45
46 # Clone the navigation2 repository
47 RUN mkdir -p nav2_ws/src && \
48     cd nav2_ws/src && \
49     git clone -b jazzy https://github.com/ros-navigation/
       navigation2.git
50
51 # Build the packages
52 WORKDIR $PKG_WS
53
54 # Source ROS environment and build dependencies
55 RUN source /opt/ros/jazzy/setup.bash && \
56     apt-get update && \
57     rosdep install --from-paths src --ignore-src -r -y && \
58     colcon build --parallel-workers 2
59
60 WORKDIR $PKG_WS
61 # # Setup the entry point and config file
62 ARG pkg_image_name
63
64 # Set the entrypoint
65 RUN chmod 777 /entrypoint.sh
66 ENTRYPOINT [ "/entrypoint.sh" ]
67
68 ######### DEV ######################
69 FROM overlay as dev
70
71 # Set the dev arguments
72 ARG USERNAME=devuser
73 ARG UID=1001
74 ARG GID=${UID}
75
76 # Add dev dependences
77 RUN apt-get update && \
78     apt-get install -y --no-install-recommends && \
79     apt-get install -y --no-install-recommends gdb && \
80     apt-get install -y --no-install-recommends gdbserver && \
81     apt-get install -y --no-install-recommends nano && \
82     apt-get install -y --no-install-recommends bash-
       completion && \
83     apt-get install -y --no-install-recommends terminator &&
       \
84     rm -rf /var/lib/apt/lists/*
85
86 # Create new user and home directory ?
```

```
87 RUN groupadd --gid $GID $USERNAME && \
88     useradd --uid ${GID} --gid ${UID} --create-home ${
       USERNAME} && \
89     echo ${USERNAME} ALL=\(root\) NOPASSWD:ALL > /etc/sudoers
       .d/${USERNAME} && \
90     chmod 0440 /etc/sudoers.d/${USERNAME} && \
91     mkdir -p /home/${USERNAME} && \
92     chown -R ${UID}:${GID} /home/${USERNAME}
93
94 # Setup the access right
95 ARG PKG_WS
96 RUN chown -R ${UID}:${GID} ${PKG_WS}
97
98 # Create the routerconfig.json5 configuration
99 ARG PKG_NAME
100 RUN mkdir /home/${USERNAME}/.ros
101 COPY docker/routerconfig.json5 /home/${USERNAME}/.ros
102 RUN chown -R ${UID}:${GID} /home/${USERNAME}/.ros
103
104 # Configuration for terminator
105 COPY tools/terminator_config /home/${USERNAME}/.config/
       terminator/config
106
107 WORKDIR /home/${USERNAME}
108 COPY docker/zenoh.sh /home/${USERNAME}/
109 RUN chmod +x /home/${USERNAME}/zenoh.sh
110 RUN /home/${USERNAME}/zenoh.sh
111 RUN chown -R ${USERNAME}:${USERNAME} /ws_rmw_zenoh
112
113 USER ${USERNAME}
114
115 RUN echo "source /entrypoint.sh && source /etc/
       bash_completion" >> /home/${USERNAME}/.bashrc
```

Listing 4.2. Dockerfile to build the container with Jazzy distro in the PC

C Dockerfile ARM architecture - Jazzy

```
ARG FROM_IMAGE=arm64v8/ros:jazzy-ros-core-noble
ARG PKG_NAME=pkg_image_name
ARG PKG_WS=/ws

######### CLONING and INSTALLING #################
FROM $FROM_IMAGE AS base
SHELL ["/bin/bash", "-c"]

# Install the main dependences
ARG PKG_WS
WORKDIR $PKG_WS/src/$PKG_NAME
COPY ./ .
RUN apt-get update && \
    apt-get install -y build-essential git cmake libasio-dev
       && \
    apt-get install -y ros-jazzy-geometry2 && \
    apt-get install -y ros-jazzy-geometry-msgs && \
    apt-get install -y ros-jazzy-tf2* && \
    apt-get install -y ros-jazzy-robot-state-publisher && \
    apt-get install -y ros-jazzy-xacro && \
    apt-get install -y wget unzip openjdk-8-jdk python3-
       vcstool libpcap-dev ros-jazzy-nav2-msgs python3-rosdep
        python3-colcon-common-extensions && \
    rm -rf /var/lib/apt/lists/*

######### BUILDING #################
FROM base AS overlay

# Build all the packages
ARG PKG_WS
WORKDIR $PKG_WS
COPY --from=base $PKG_WS/src $PKG_WS/src
RUN source /opt/ros/jazzy/setup.bash

ARG pkg_image_name
COPY docker/entrypoint.sh /
COPY docker/routerconfig.json5 /root/.ros

# Set the entrypoint
RUN chmod 777 /entrypoint.sh
ENTRYPOINT [ "/entrypoint.sh" ]

######### DEV ######################
FROM overlay as dev

```

```
43 # Set the dev arguments
44 ARG USERNAME=devuser
45 ARG UID=1001
46 ARG GID=${UID}
47
48 # Add dev dependences
49 RUN apt-get update && \
50     apt-get install -y --no-install-recommends && \
51     apt-get install -y --no-install-recommends gdb && \
52     apt-get install -y --no-install-recommends gdbserver && \
53     apt-get install -y --no-install-recommends nano && \
54     apt-get install -y --no-install-recommends bash-
       completion && \
55     apt-get install -y --no-install-recommends terminator &&
       \
56     rm -rf /var/lib/apt/lists/*
57
58 # Create new user and home directory ?
59 RUN groupadd --gid $GID $USERNAME && \
60     useradd --uid ${GID} --gid ${UID} --create-home ${
       USERNAME} && \
61     echo ${USERNAME} ALL=\(root\) NOPASSWD:ALL > /etc/sudoers
       .d/${USERNAME} && \
62     chmod 0440 /etc/sudoers.d/${USERNAME} && \
63     mkdir -p /home/${USERNAME} && \
64     chown -R ${UID}:${GID} /home/${USERNAME}
65
66 # Setup the access right
67 ARG PKG_WS
68 RUN chown -R ${UID}:${GID} ${PKG_WS}
69
70 # Create the routerconfig.json5 configuration
71 ARG PKG_NAME
72 RUN mkdir /home/${USERNAME}/.ros
73 COPY docker/routerconfig.json5 /home/${USERNAME}/.ros
74 RUN chown -R ${UID}:${GID} /home/${USERNAME}/.ros
75
76 # Configuration for terminator
77 COPY tools/terminator_config /home/${USERNAME}/.config/
     terminator/config
78
79 WORKDIR /home/${USERNAME}
80 COPY docker/zenoh.sh /home/${USERNAME}/
81 RUN chmod +x /home/${USERNAME}/zenoh.sh
82 RUN rosdep init
83 USER ${USERNAME}
84 RUN rosdep update
```

```
85 USER root
86 RUN /home/${USERNAME}/zenoh.sh
87
88 USER ${USERNAME}
89 RUN echo "source /entrypoint.sh && source /etc/
   bash_completion" >> /home/${USERNAME}/.bashrc
```

Listing 4.3. Dockerfile to build the container with Jazzy distro in arm64 device

D docker-compose.yaml - Husarnet

```yaml
services:
  # Base image containing dependencies.
  base:
    image: husarnet_image:base
    build:
      context: .
      dockerfile: docker/Dockerfile
      target: base
    # Interactive shell
    stdin_open: true
    tty: true
    # Networking and IPC for ROS 2
    network_mode: service:husarnet
    ipc: service:husarnet
    privileged: true
    # Allows graphical programs in the container.
    environment:
      - DISPLAY=${DISPLAY}
      - QT_X11_NO_MITSHM=1
      - ROS_DOMAIN_ID=${ROS_DOMAIN_ID:-12}
      - QT_X11_NO_MITSHM=1
      - NVIDIA_DRIVER_CAPABILITIES=all
    volumes:
      # Allows graphical programs in the container.
      - /tmp/.X11-unix:/tmp/.X11-unix:rw
      - /dev/dri:/dev/dri
      - ${XAUTHORITY:-$HOME/.Xauthority}:/root/.Xauthority

  overlay:
    extends: base
    network_mode: service:husarnet
    image: husarnet_image:overlay
    build:
      context: .
      dockerfile: docker/Dockerfile
      target: overlay

  dev:
    extends: overlay
    image: husarnet_image:dev
    build:
      context: .
      dockerfile: docker/Dockerfile
      target: dev
      args:
```

```
46          - UID=${UID:-1000}
47          - GID=${UID:-1000}
48          - USERNAME=${USERNAME:-devuser}
49      volumes:
50        - ./:/ws/src/
51      user: ${USERNAME:-devuser}
52      environment:
53          - SHELL=/bin/bash
54      command: sleep infinity
55
56    husarnet:
57      image: husarnet/husarnet:latest
58      volumes:
59        - /var/lib/husarnet
60      sysctls:
61        - net.ipv6.conf.all.disable_ipv6=0
62      cap_add:
63        - NET_ADMIN
64      devices:
65        - /dev/net/tun
66      environment:
67        - HOSTNAME=container-nano_1
68        - JOINCODE=fc94:b01d:1803:8dd8:b243:5c7d:9465:497b
             /**********************
69        - HUSARNET_DEBUG=1
```

Listing 4.4. docker-compose used for Husarnet Build

E docker-compose.yaml - Jazzy

```yaml
services:
  # Base image containing dependencies.
  base:
    image: jazzy:base
    build:
      context: .
      dockerfile: docker/Dockerfile
      target: base
    # Interactive shell
    stdin\_open: true
    tty: true
    # Networking and IPC for ROS 2
    network\_mode: "host"
    ipc: host
    privileged: true
    # Allows graphical programs in the container.
    environment:
      - DISPLAY=${DISPLAY}
      - QT\_X11\_NO\_MITSHM=1
      - ROS\_DOMAIN\_ID=${ROS\_DOMAIN\_ID:-12}
      - QT\_X11\_NO\_MITSHM=1
      - NVIDIA\_DRIVER\_CAPABILITIES=all
    volumes:
      # Allows graphical programs in the container.
      - /tmp/.X11-unix:/tmp/.X11-unix:rw
      - /dev/dri:/dev/dri
      - ${XAUTHORITY:-$HOME/.Xauthority}:/root/.Xauthority

  overlay:
    extends: base
    network\_mode: "host"
    image: jazzy:overlay
    build:
      context: .
      dockerfile: docker/Dockerfile
      target: overlay

  dev:
    extends: overlay
    image: jazzy:dev
    build:
      context: .
      dockerfile: docker/Dockerfile
      target: dev
      args:
```

```
46          - UID=${UID:-1001}
47          - GID=${UID:-1001}
48          - USERNAME=${USERNAME:-devuser}
49      privileged: true
50      volumes:
51        - ./:/ws/src/
52      devices:
53        - /dev/ttyUSB0:/dev/ttyUSB0
54      user: ${USERNAME:-devuser}
55      environment:
56          - SHELL=/bin/bash
57      command: sleep infinity
```

Listing 4.5. docker-compose used in Raspberry

F nav2_param.yaml

```yaml
amcl:
  ros__parameters:
    use_sim_time: False
    alpha1: 0.2
    alpha2: 0.2
    alpha3: 0.2
    alpha4: 0.2
    alpha5: 0.2
    base_frame_id: "base_footprint"
    beam_skip_distance: 0.5
    beam_skip_error_threshold: 0.9
    beam_skip_threshold: 0.3
    do_beamskip: false
    global_frame_id: "map"
    lambda_short: 0.1
    laser_likelihood_max_dist: 2.0
    laser_max_range: 100.0
    laser_min_range: -1.0
    laser_model_type: "likelihood_field"
    max_beams: 60
    max_particles: 2000
    min_particles: 500
    odom_frame_id: "odom"
    pf_err: 0.05
    pf_z: 0.99
    recovery_alpha_fast: 0.0
    recovery_alpha_slow: 0.0
    resample_interval: 1
    robot_model_type: "nav2_amcl::OmniMotionModel"
    save_pose_rate: 0.5
    sigma_hit: 0.2
    tf_broadcast: true
    transform_tolerance: 1.0
    update_min_a: 0.2
    update_min_d: 0.25
    z_hit: 0.5
    z_max: 0.05
    z_rand: 0.5
    z_short: 0.05
    scan_topic: scan

bt_navigator:
  ros__parameters:
    global_frame: map
    robot_base_frame: base_link
```

```yaml
    odom_topic: /odom
    bt_loop_duration: 10
    default_server_timeout: 20
    wait_for_service_timeout: 1000
    action_server_result_timeout: 900.0
    navigators: ["navigate_to_pose", "navigate_through_poses"]
    navigate_to_pose:
      plugin: "nav2_bt_navigator::NavigateToPoseNavigator"
    navigate_through_poses:
      plugin: "nav2_bt_navigator::NavigateThroughPosesNavigator"

    error_code_names:
      - compute_path_error_code
      - follow_path_error_code

bt_navigator_navigate_through_poses_rclcpp_node:
  ros__parameters:
    use_sim_time: False

bt_navigator_navigate_to_pose_rclcpp_node:
  ros__parameters:
    use_sim_time: False

controller_server:
  ros__parameters:
    use_sim_time: False
    controller_frequency: 20.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.001
    min_theta_velocity_threshold: 0.001
    failure_tolerance: 0.3
    progress_checker_plugin: "progress_checker"
    goal_checker_plugins: ["general_goal_checker"] # "precise_goal_checker"
    controller_plugins: ["FollowPath"]
    use_realtime_priority: false

    # Progress checker parameters
    progress_checker:
      plugin: "nav2_controller::SimpleProgressChecker"
      required_movement_radius: 0.5
      movement_time_allowance: 10.0
    general_goal_checker:
      stateful: True
      plugin: "nav2_controller::SimpleGoalChecker"
```

```
      xy_goal_tolerance: 0.1
      yaw_goal_tolerance: 0.25
    # DWB parameters
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      debug_trajectory_details: True
      min_vel_x: -0.15
      min_vel_y: -0.15
      max_vel_x: 0.15
      max_vel_y: 0.15
      max_vel_theta: 1.0
      min_speed_xy: 0.0
      max_speed_xy: 0.15
      min_speed_theta: 0.0
      acc_lim_x: 2.5
      acc_lim_y: 2.5
      acc_lim_theta: 3.2
      decel_lim_x: -2.5
      decel_lim_y: -2.5
      decel_lim_theta: -3.2
      vx_samples: 20
      vy_samples: 20
      vtheta_samples: 20
      sim_time: 1.7
      linear_granularity: 0.05
      angular_granularity: 0.025
      transform_tolerance: 0.2
      xy_goal_tolerance: 0.1
      trans_stopped_velocity: 0.25
      short_circuit_trajectory_evaluation: True
      stateful: True
      critics: ["RotateToGoal", "Oscillation", "BaseObstacle"
         , "GoalAlign", "PathAlign", "PathDist", "GoalDist"]
      BaseObstacle.scale: 0.02
      PathAlign.scale: 32.0
      PathAlign.forward_point_distance: 0.1
      GoalAlign.scale: 24.0
      GoalAlign.forward_point_distance: 0.1
      PathDist.scale: 32.0
      GoalDist.scale: 24.0
      RotateToGoal.scale: 32.0
      RotateToGoal.slowing_factor: 5.0
      RotateToGoal.lookahead_time: -1.0

local_costmap:
  local_costmap:
    ros__parameters:
```

```
136       update_frequency: 5.0
137       publish_frequency: 2.0
138       global_frame: odom
139       robot_base_frame: base_link
140       use_sim_time: False
141       rolling_window: true
142       width: 3
143       height: 3
144       resolution: 0.05
145       footprint: "[ [0.410, 0.350], [0.410, -0.350], [-0.410,
              -0.350], [-0.410, 0.350] ]"
146       plugins: ["voxel_layer", "inflation_layer"]
147       inflation_layer:
148         plugin: "nav2_costmap_2d::InflationLayer"
149         cost_scaling_factor: 3.0
150         inflation_radius: 0.55
151       voxel_layer:
152         plugin: "nav2_costmap_2d::VoxelLayer"
153         enabled: True
154         publish_voxel_map: True
155         origin_z: 0.0
156         z_resolution: 0.05
157         z_voxels: 16
158         max_obstacle_height: 2.0
159         mark_threshold: 0
160         observation_sources: scan
161         scan:
162           topic: /scan
163           max_obstacle_height: 2.0
164           clearing: True
165           marking: True
166           data_type: "LaserScan"
167           raytrace_max_range: 3.0
168           raytrace_min_range: 0.0
169           obstacle_max_range: 2.5
170           obstacle_min_range: 0.0
171         scan3d:
172           topic: /rslidar_points
173           max_obstacle_height: 2.0
174           clearing: True
175           marking: True
176           data_type: "PointCloud2"
177           raytrace_max_range: 3.0
178           raytrace_min_range: 0.0
179           obstacle_max_range: 2.5
180           obstacle_min_range: 0.0
181       static_layer:
```

```
182         plugin: "nav2_costmap_2d::StaticLayer"
183         map_subscribe_transient_local: True
184       always_send_full_costmap: True
185
186 global_costmap:
187   global_costmap:
188     ros__parameters:
189       update_frequency: 1.0
190       publish_frequency: 1.0
191       global_frame: map
192       robot_base_frame: base_link
193       use_sim_time: False
194       footprint: "[ [0.410, 0.350], [0.410, -0.350], [-0.410,
              -0.350], [-0.410, 0.350] ]"
195       resolution: 0.05
196       track_unknown_space: true
197       plugins: ["static_layer", "obstacle_layer", "
             inflation_layer"]
198       obstacle_layer:
199         plugin: "nav2_costmap_2d::ObstacleLayer"
200         enabled: True
201         observation_sources: scan
202         scan:
203           topic: /scan
204           max_obstacle_height: 2.0
205           clearing: True
206           marking: True
207           data_type: "LaserScan"
208           raytrace_max_range: 3.0
209           raytrace_min_range: 0.0
210           obstacle_max_range: 2.5
211           obstacle_min_range: 0.0
212         scan3d:
213           topic: /rslidar_points
214           max_obstacle_height: 2.0
215           clearing: True
216           marking: True
217           data_type: "PointCloud2"
218           raytrace_max_range: 3.0
219           raytrace_min_range: 0.0
220           obstacle_max_range: 2.5
221           obstacle_min_range: 0.0
222       static_layer:
223         plugin: "nav2_costmap_2d::StaticLayer"
224         map_subscribe_transient_local: True
225       inflation_layer:
226         plugin: "nav2_costmap_2d::InflationLayer"
```

```
227         cost_scaling_factor: 3.0
228         inflation_radius: 0.55
229       always_send_full_costmap: True
230
231 map_server:
232   ros__parameters:
233     use_sim_time: False
234     yaml_filename: ""
235
236 map_saver:
237   ros__parameters:
238     use_sim_time: False
239     save_map_timeout: 5.0
240     free_thresh_default: 0.25
241     occupied_thresh_default: 0.65
242     map_subscribe_transient_local: True
243
244 planner_server:
245   ros__parameters:
246     expected_planner_frequency: 20.0
247     use_sim_time: False
248     planner_plugins: ["GridBased"]
249     GridBased:
250       plugin: "nav2_navfn_planner::NavfnPlanner"
251       tolerance: 0.5
252       use_astar: false
253       allow_unknown: true
254
255 smoother_server:
256   ros__parameters:
257     use_sim_time: False
258     smoother_plugins: ["simple_smoother"]
259     simple_smoother:
260       plugin: "nav2_smoother::SimpleSmoother"
261       tolerance: 1.0e-10
262       max_its: 1000
263       do_refinement: True
264
265 behavior_server:
266   ros__parameters:
267     local_costmap_topic: local_costmap/costmap_raw
268     global_costmap_topic: global_costmap/costmap_raw
269     local_footprint_topic: local_costmap/published_footprint
270     global_footprint_topic: global_costmap/
        published_footprint
271     cycle_frequency: 10.0
```

```
    behavior_plugins: ["spin", "backup", "drive_on_heading",
       "assisted_teleop", "wait"]
    spin:
      plugin: "nav2_behaviors::Spin"
    backup:
      plugin: "nav2_behaviors::BackUp"
    drive_on_heading:
      plugin: "nav2_behaviors::DriveOnHeading"
    wait:
      plugin: "nav2_behaviors::Wait"
    assisted_teleop:
      plugin: "nav2_behaviors::AssistedTeleop"
    local_frame: odom
    global_frame: map
    robot_base_frame: base_link
    transform_tolerance: 0.1
    simulate_ahead_time: 2.0
    max_rotational_vel: 0.6
    min_rotational_vel: 0.4
    rotational_acc_lim: 3.2

waypoint_follower:
  ros__parameters:
    use_sim_time: False
    loop_rate: 20
    stop_on_failure: false
    waypoint_task_executor_plugin: "wait_at_waypoint"
    wait_at_waypoint:
      plugin: "nav2_waypoint_follower::WaitAtWaypoint"
      enabled: True
      waypoint_pause_duration: 200

velocity_smoother:
  ros__parameters:
    use_sim_time: False
    smoothing_frequency: 20.0
    scale_velocities: False
    feedback: "OPEN_LOOP"
    max_velocity: [0.15, 0.15, 0.6]
    min_velocity: [-0.15, -0.15, -0.6]
    max_accel: [2.5, 2.5, 3.2]
    max_decel: [-2.5, -2.5, -3.2]
    odom_topic: "odom"
    odom_duration: 0.1
    deadband_velocity: [0.0, 0.0, 0.0]
    velocity_timeout: 1.0

```

```yaml
collision_monitor:
  ros__parameters:
    base_frame_id: "base_footprint"
    odom_frame_id: "odom"
    cmd_vel_in_topic: "cmd_vel_smoothed"
    cmd_vel_out_topic: "cmd_vel"
    state_topic: "collision_monitor_state"
    transform_tolerance: 0.2
    source_timeout: 1.0
    base_shift_correction: True
    stop_pub_timeout: 2.0
    polygons: ["FootprintApproach"]
    FootprintApproach:
      type: "polygon"
      action_type: "approach"
      footprint_topic: "/local_costmap/published_footprint"
      time_before_collision: 1.2
      simulation_time_step: 0.1
      min_points: 6
      visualize: False
      enabled: True
    observation_sources: ["scan"]
    scan:
      type: "scan"
      topic: "scan"
      min_height: 0.15
      max_height: 2.0
      enabled: True

docking_server:
  ros__parameters:
    controller_frequency: 50.0
    initial_perception_timeout: 5.0
    wait_charge_timeout: 5.0
    dock_approach_timeout: 30.0
    undock_linear_tolerance: 0.05
    undock_angular_tolerance: 0.1
    max_retries: 3
    base_frame: "base_link"
    fixed_frame: "odom"
    dock_backwards: false
    dock_prestaging_tolerance: 0.5

    # Types of docks
    dock_plugins: ['simple_charging_dock']
    simple_charging_dock:
      plugin: 'opennav_docking::SimpleChargingDock'
```

```
365       docking_threshold: 0.05
366       staging_x_offset: -0.7
367       use_external_detection_pose: true
368       use_battery_status: false # true
369       use_stall_detection: false # true
370
371       external_detection_timeout: 1.0
372       external_detection_translation_x: -0.18
373       external_detection_translation_y: 0.0
374       external_detection_rotation_roll: -1.57
375       external_detection_rotation_pitch: -1.57
376       external_detection_rotation_yaw: 0.0
377       filter_coef: 0.1
378
379     controller:
380       k_phi: 3.0
381       k_delta: 2.0
382       v_linear_min: 0.15
383       v_linear_max: 0.15
384
385 loopback_simulator:
386   ros__parameters:
387     base_frame_id: "base_footprint"
388     odom_frame_id: "odom"
389     map_frame_id: "map"
390     scan_frame_id: "base_scan"  # tb4_loopback_simulator.
        launch.py remaps to 'rplidar_link'
391     update_duration: 0.02
```

Listing 4.6. Parameters for Nav2 execution

Bibliography

[1] Adlink-ROS. rf2o_laser_odometry. https://github.com/Adlink-ROS/rf2o_laser_odometry, 2024. Accessed: 2024-11-23.

[2] Isaac Asimov. *I, Robot*. Gnome Press, New York, 1950.

[3] Priyaranjan Biswal and Prases K. Mohanty. Development of quadruped walking robots: A review. *Ain Shams Engineering Journal*, 12(2):2017–2031, 2021. ISSN 2090-4479. doi: https://doi.org/10.1016/j.asej.2020.11.005. URL https://www.sciencedirect.com/science/article/pii/S2090447920302501.

[4] Angelo Corsaro, Luca Cominardi, Olivier Hecart, Gabriele Baldoni, Julien Enoch, Pierre Avital, Julien Loudet, Carlos Guimares, Michael Ilyin, and Dmitrii Bannov. Zenoh: Unifying communication, storage and computation from the cloud to the microcontroller. https://zettascale.tech, 2022.

[5] Docker, Inc. Docker Documentation: Overview, 2024. URL https://docs.docker.com/get-started/docker-overview/. Accessed: Nov. 5, 2024.

[6] eProsima. Definitions - fast dds. https://fast-dds.docs.eprosima.com/en/latest/fastdds/getting_started/definitions.html, 2024. Accessed: 2024-11-08.

[7] Eclipse Foundation. Minimizing discovery overhead in ros2, 2021. URL https://zenoh.io/blog/2021-03-23-discovery/. Accessed: 2024-11-13.

[8] Katie Hughes. unitree_nav. https://github.com/katie-hughes/unitree_nav, 2024. Accessed: 2024-11-15.

[9] Katie Hughes and Nick Morales. unitree_ros2. https://github.com/katie-hughes/unitree_ros2, 2023. Accessed: 2024-11-15.

[10] Husarnet. Husarnet documentation: Linux installation, 2024. URL https://husarnet.com/docs/platform-linux-install/. Accessed: 2024-11-22.

[11] Steven Macenski, Francisco Martin, Ruffin White, and Jonatan Ginés Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[12] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science*

Robotics, 7(66):eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. URL https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[13] Masahiro Mori. The uncanny valley: The original essay by masahiro mori, 2012. URL https://spectrum.ieee.org/the-uncanny-valley#_ftn2.

[14] Amit Patel. Introduction to a* pathfinding, 2023. URL https://www.redblobgames.com/pathfinding/a-star/introduction.html. Accessed: 2024-11-03.

[15] Open Robotics, editor. *Proceedings of ROSCon 2024*, October 2024. Open Robotics. Held from October 21 to 23, 2024.

[16] Quadruped Robotics. *Go1 Technical Datasheet*. Quadruped, 2021. URL https://www.mybotshop.de/Datasheet/Datasheet_QUADRUPED_Go1.pdf.

[17] Unitree Robotics. unitree_legged_sdk. https://github.com/unitreerobotics/unitree_legged_sdk, 2020. Accessed: 2024-11-15.

[18] Unitree Robotics. *Go1 Software Manual*. Unitree, 2021. URL https://docs.trossenrobotics.com/unitree_go1_docs/_downloads/23a15b8ec933927ef4ee1ba8f70abba6/go1_software_manual.pdf.

[19] Unitree Robotics. *Go1 Where You Will Go*. Unitree, 2021. URL https://www.generationrobots.com/media/unitree/Go1%20Datasheet_EN%20v3.0.pdf.

[20] Unitree Robotics. unitree_ros2_to_real. https://github.com/unitreerobotics/unitree_ros2_to_real, 2022. Accessed: 2024-11-15.

[21] Slamtec. sllidar_ros2: Ros 2 package for sllidar devices, 2024. URL https://github.com/Slamtec/sllidar_ros2. Accessed: 2024-11-21.

[22] Ricardo Tellez. A history of ros (robot operating system), 2023. URL https://www.theconstruct.ai/history-ros/. Accessed: 2024-11-06.

[23] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT Press, Cambridge, Mass., 2005. ISBN 0262201623 9780262201629. URL http://www.amazon.de/gp/product/0262201623/102-8479661-9831324?v=glance&n=283155&n=507846&s=books&v=glance.

[24] Keenan Wyrobek and Eric Berger. Personal robotics program fund fundraising deck, 2006. URL https://www.slideshare.net/KeenanWyrobek/personal-robotics-program-fund-fundraising-deck-from-2006.