POLITECNICO DI TORINO

Master's Degree in Computer Networks and Cloud Computing

Master's Degree Thesis

Enabling an autonomous robot to transparently access local, edge and cloud services

Supervisors

Prof. Fulvio RISSO

Dr. Stefano GALANTINO

Candidate

Jonathan MARSIANO

October 2022

Abstract

Several companies and research groups are nowadays investigating the potential of autonomous driving and the improvements that it could bring to our lives. Not all of them, though, are focusing on large and fast vehicles like cars, as some have been concentrating their efforts in much slower and smaller ones, which is the case of *ALBA Robot* and their *SEDIA* project. Their focus has been shifted to wheelchairs, with much lower size and speed than other kinds of vehicles. Despite having less strict requirements, there is still the need of leveraging a lot of computational effort in these kinds of autonomous driving applications, which would be best suited for small edge data centers. Unloading the application's burdens to a close cloud server will bring to sensible improvements in the system's performance, but it also brings new problems. Indeed, all the data internally exchanged in the application have strict latency requirements: if the network infrastructure cannot sustain the workload at every moment, some malfunctioning surely occurs in the navigation process and that will no longer be considered safe.

The focus of this thesis is therefore on two main goals: the first one is to find a way of making the components of the *ALBA Robot* autonomous driving system work in a cloud server, by finding a way of allowing remote nodes to communicate. Indeed, the navigation architecture presented is based on ROS2, which has no spontaneous way of making its nodes talk over different L2 networks. The second goal will address the malfunctioning of the navigation process in case the network cannot sustain the internal messages exchange's requirements. One way of achieving resiliency is to guarantee the same service also locally, rather than only in cloud. The final architecture will eventually be composed by both local and cloud nodes, which are their exact copies; cloud nodes will be leveraged when the network connection lives up to the latency requirements, and the local ones otherwise. To reach this condition, a switching mechanism has been designed and implemented, namely an additional architecture thank to whom a switch from local to cloud and vice versa can be triggered, when considered necessary: this architecture is in fact only logical, as it is based on the same navigation mechanisms already existing. An important part of the switching is also the development of some logical criteria to establish when it must be triggered, regardless of how it actually works. A possible way of implementing these criteria was also explored in this work. It consists in monitoring the status of the network at every moment and taking a decision according to how the current connection will affect the navigation's performances: if using the cloud would mean a substantial loss of navigation performance, a switch to the local nodes is triggered, until network is up again and a switch back to local

can be performed. This mechanism is overall independent of the previous problem as well, because it assumes that all the ROS2 nodes present in the autonomous driving architecture can freely communicate, regardless of their physical location in space. Once built this system, the next step will be to execute the cloud nodes inside containers and add some orchestration technology as well: this thesis has chosen Kubernetes, as it is the most commonly used and practical one. The final autonomous driving system will be resilient and capable of avoiding the occurrence of safety critical issues.

Acknowledgements

"HI"
Goofy, Google by Google

Table of Contents

Chapter 1

Introduction

The world of robotics is experiencing a huge boost nowadays, thanks to the new technologies that brought a breath of fresh air into the implementation of Software solutions for robotic platforms. Professionals, big companies and even non-professionals are all participating in the challenges offered by the development of new solutions in this field, with a particular trend toward autonomous driving.

The development of a fully autonomous driving system must face several challenges, related to the humongous amount of data the intended vehicle has to deal with: every autonomous robot is indeed equipped with a large number of sensors (depth cameras, LIDAr, QRcode cameras etc. . .), which produce data useful for its autonomous navigation. Such data must be processed very fast to perform the main navigation tasks, above all a smooth obstacle avoidance process: a huge computational effort is consequently needed, and the majority of platforms available nowadays struggle with living up to the very strict time constraints the autonomous driving tasks bring with them. Indeed, "old" autonomous robots implemented their processing by means of an onboard computing system, which is responsible of collecting the data coming from the platform's Hardware, elaborating them and sending the proper controlling commands to the vehicle's driving system. To be capable of addressing the computational effort required in an autonomous driving application, huge computers were needed, basically leaving no space for a possible passenger: less cumbersome devices could be used, but their lack of performance meant an unoptimized and poor self-driving platform.

This is the reason why a new approach was soon needed: the introduction of new network technologies such as 5G was the starting point for a new conceptual view of Software development for autonomous robots. Indeed, the strong computational requirements of this kind of application are best suited if carried out by cloud servers, located in small datacenters at the edge of the network: using this approach, the data elaboration would be delegated to these servers, which have virtually

infinite computing capabilities, and data can be sent from the vehicle control system to the datacenter and back in time intervals comparable to the local (wired) ones. This could relieve the intelligence in the vehicle of most of its workload, being at the same time fast enough to guarantee a good performance of the autonomous system.

Of course, some latency will be inevitably added by the data transmission, as the message sending between the elaboration center and the driving system control is no longer wired (in old robots, the computational unit and the control system were connected through UART or serial): now, it has to trespass a physical network, reach the datacenter and come back. This must be done in a time interval comparable to the old wired ones, otherwise a good performance of the system cannot be guaranteed.

How to deal with network outages? Network is never 100% reliable, even though the new technologies try to bring some improvements: complete network failures or bad connection scenarios can still happen quite frequently. An autonomous driving application must be resilient to that, and in order to achieve a good degree of fault tolerance, **the vehicle's autonomous navigation tasks must be performed both locally and in the cloud**. In other words, a local instance of each navigation service must be available in the vehicle's intelligence, even if less powerful than the one hosted in a cloud server, as those instances will be used when the network cannot live up to the application's latency requirements. Consequently, we need a system to decide which of the two instances must be leveraged at a certain time, depending on the network status. However, the robot must be agnostic to the presence of such mechanism and continue to use its services normally: it should not care if the local or the cloud part of the system is being used, nor how to switch between the two.

In light of all of this, the goal of this thesis is to design a system capable of replicating the services of an autonomous robot in a cloud (edge) server and switching between the two instances of each service (or better, switching between the cloud and the local system) depending on network connection. Hence, the implementation of a proper switching mechanism will be explored, as well as an accurate way of deciding whether to trigger a switch or not.

1.1 The working environment

This thesis has been realized in collaboration with ***ALBA Robot*** s.p.a., a start-up company whose main focus is the development of a robotic platform with autonomous driving capabilities: specifically, their focus is on wheelchairs, which are therefore meant to transport people with limited motor skills or not, as well

as objects. As previously discussed, the onboard computing Hardware must be compact enough to let a person fit on the vehicle and a good obstacle avoidance process must be guaranteed. In this case, though, the latency constraints are not that strict, as this kind of vehicle goes at a maximum speed of 5m/s, hence it has more time to detect obstacles and let the navigation system intelligence process the corresponding data.

The work presented in the following chapters has been developed in the company's environment, with the possibility of accessing their Hardware components for the learning process and the eventual implementation of the Software solutions proposed. At the same time, a cloud server was made available by *TIM*, with a web interface developed by *Capgemini*, which allows the actual testing in the field.

1.2 Structure

This thesis is organized as follows:

- Chapter 2 gives a brief introduction to the basic concepts the reader needs to be introduced to the environment of this work and fully understand the solutions proposed: the robotic world of ROS2 and its frameworks.
- Chapter 3 presents the autonomous system as it was found at the time this thesis was developed. It is important to highlight that this work does not aim to implementing an autonomous driving system: it focuses only on the design of an architecture that can shift such system to the cloud.
- Chapter 4 describes such architecture.
- Chapter 5 extends a specific aspect of the proposed solution's logical behaviour, which is latency: it describes how the problem of the latency constraints is interpreted and faced in this thesis.
- Chapter 6 focuses on some considerations that must be made when a cloud platform is added to the system, in particular a Kubernetes cluster cloud-side.
- Chapter 7 presents the result of some empirical tests performed in the working environment described above.
- Chapter 8 projects the reader towards a future real implementation of the proposed architecture, presenting the main problems that could not be addressed in this work.

Chapter 2

Fundamental concepts

This chapter has been thought as an introduction for the reader to the fundamentals of the various technologies used in this work. As previously discussed, the goal of this thesis is to design and implement a cloud solution for a pre-existing autonomous driving architecture, hence the first thing to do is to give a hint on the technologies the ALBA Robot architecture is based on. After, we can describe the actual technology as it was found at the beginning of this work in chapter 3.

We'll see that ROS2 is at the very core of the development of the application, but other two technologies will be presented, which are respectively at the base of ROS2 and built on top on it.

Let's see things in order:

2.1 ROS2

The core technology of the ALBA Robot autonomous driving project is **ROS2**, which is the second release of the ROS project. ROS stands for *Robot Operating System*, but this name could result to be misleading, as it is in fact not an actual Operating System in its strict definition: it can be better described by a an **open-source robotics middleware suite**, or better a set of Software frameworks for robot applications development. It is has been called in such a way because it provides the services one would expect from an Operating System, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also adds tools and libraries for obtaining, building, writing, and running code across multiple computers.
Quoting the official ROS community's definition:

"The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source."

To understand the importance of this technology, it is sufficient to say that before it, building a complex and structured robot behaviour was not only a difficult and time-consuming task, but it was **strictly tied to the specific Hardware of the robot**: indeed, every effort made to develop Software for a robust robot platform was only useful for that specific platform as there was no portability. Every robotic application had to be written taking strongly in consideration the Hardware it was being built upon, and therefore it was nearly impossible to reuse an already existing piece of Software for a different kind of platform. This was a huge problem, specially for minor companies, which had to put an enormous effort on the development of a robotic solution from scratch. A new approach was strongly needed.

The solution came with the ROS community, which developed libraries, tools and architectural concepts (as we'll see) in order to let developers **share and reuse Software for robotic applications** throughout platforms different than the one it was initially though for. ROS is in fact not to be though as a framework with the most features: instead, its primary goal is to support code reuse in robotics research and development. it provides a base layer with fundamental services which developers can use as base for the construction of their Software, which can be specialized depending on the Hardware of the specific robotic platform developed.

This is the main difference between ROS and other robotic frameworks, and the most important thing to understand in order to enter into the robotic world of ROS: specialized solutions for the companies' robotic applications are not delivered by the ROS technology, as they are still tied to the specific platform's Hardware. ROS provides a **base layer** with Software libraries and a **conceptual architecture** which lets developers build their solutions on top of it.

Thanks to the ROS community efforts, **cooperation** was enhanced among developers in a company: different developers can have different expertise, for instance one could be specialized in robot's movements and another one in sensors' data elaboration. ROS provides a base framework that makes the process of sharing their effort in a much easier way than before, hence the company itself can benefit from a faster components integration.

In support of this primary goal of sharing and collaboration, there are several other goals of the ROS framework:

- It is designed to be as thin as possible so that code written for ROS can be used with other robot Software frameworks.

- It prefers a development model in which ROS-agnostic libraries with clean functional interfaces are written.
- The ROS framework is language-independent, meaning that it is easy to implement in any modern programming language (**C++** and **Python** are the main ones so far).
- ROS has a built-in unit/integration test framework called *rostest* that makes it easy to bring up and tear down test fixtures.
- It scales easily: ROS is appropriate for large runtime systems and development processes.

2.1.1 ROS History

The first ideas of what eventually would become the ROS framework were born at Stanford University at around 2007: two PhD students, Eric Berger and Keenan Wyrobek, were working in a robotics laboratory, leading the Personal Robotics Program. While working on robots to do manipulation tasks in human environments, they noticed that many of their colleagues were having problems with the diverse nature of robotics: an expert in Software developing might not be equally proficient in Hardware, or in computer vision, and so on. Hence, the two students faced this problem by designing a baseline system that would provide a starting place for others to build upon.

In order to have a base platform to develop their idea on, they built the PR1 robot as a Hardware prototype and began to work on it, taking advantage of granted early funding. While seeking further funding for future development, the two students met Scott Hassan, the founder of **Willow Garage**, a technology incubator: the project moved there in November 2007 and they started receiving fundings even from outside the incubator.

Willow Garage began developing the PR2 robot as a follow-up to the PR1, and ROS as the software to run it. Many institutions made contributions to ROS, especially from Standford's STAIR project, both to the core Software and the packages which form the ROS ecosystem. The program was focused on producing the PR2 as a research platform for academia and ROS as the open-source robotics stack that would underlie both academic research and tech startups.

An early version of ROS *(0.4 Mango Tango)* was released in 2008, followed by the first RVIZ documentation and the first paper on ROS. An important milestones was reached when two days of continuous navigation of the PR2 into an office were performed, in which the robot managed to open doors and plug itself into the power outlets. Then, the release of ROS 1.0 was reached in January 2010, together

with tons of documentation and tutorials for the enormous abilities that Willow Garage's engineers had developed over the preceding 3 years.

Following, Willow Garage achieved what had long been a goal of the founders, which is giving away 10 PR2 robots to worthy academic institutions, like the MIT. The PR2 represented a starting point for robotic academic research around the world and boosted the popularity of ROS. Then, ROS Box Turtle was released on 2 March 2010 and this was the first time ROS was officially distributed with a set of versioned packages. These developments led to the first drone and autonomous car running ROS. In 2012, Willow Garage created the *Open Source Robotics Foundation* (OSRF), which was immediately awarded a Software contract by the *Defense Advanced Research Projects Agency* (DARPA). The first ROSCon was held in St. Paul, Minnesota, the first book on ROS ("ROS By Example") was published and ROS began running on every continent on 3 December 2012, at its fifth year of life.

In February 2013, the OSRF acquired Willow Garage itself and became the primary Software maintainer for ROS. Since then, they started releasing a new version of ROS in a per-year basis, gaining more and more popularity: ROSCons have occurred every year since 2012, a number of books have been published and meetups of ROS developers have been organized in a variety of countries. Even giants like Microsoft and Amazon took interest in this Software and NASA announced the first robot running ROS in space in 2014.

The most recent step in ROS history and perhaps the most important one is the proposal of ROS2, a significant API change to ROS which is intended to support real time programming and a huge variety of computational modern environments. ROS 2 was announced at ROSCon 2014 and the alpha releases were made in August 2015. The first distribution release of ROS 2, *Ardent Apalone*, was made on 8 December 2017, leading to a new era of next-generation ROS development.

2.1.2 ROS2 working principles

This thesis, as previously discussed, has focused its attention on ROS2, as it is the latest and most modern version of ROS and the one used by *ALBA Robot* in their *SEDIA* project. There are some fundamental and big changes w.r.t. its predecessor, which we will not explain: for what concerns our goals, describing the main differences between the two releases would be useless. It is preferred to only consider the features and potentialities of the most recent version of ROS and guide the reader to think directly in a "ROS2 way". As the official documentation states, the goal of the ROS2 project is to adapt to the latest changes of the robotics and ROS Community, leveraging what is great about ROS1 and improving what is not.

ROS2 defines a whole set of new concepts in the robotic world which allow the

developer to build an entire Software architecture by using the base definitions provided. Indeed, it is not only a set of Software services anyone can use in their robotic application, but it also delineates a set of working principles which can be leveraged to build an entire application, such as the autonomous driving one used by *ALBA Robot*.

Over the next sections, a series of core ROS2 concepts will be introduced, together creating what is referred to as the *"ROS2 graph"*. Specifically, it is a network of ROS2 elements processing data together at one time, encompassing all executables and their connections if we were to map them all out and visualize them.

Nodes

The base concept, fundamental for understanding how a ROS2 application works, is the ROS2 **node**: ROS2 is in fact a **modular application**, composed by multiple not tightly coupled nodes, every one **performing one specific task** in the overall application. In other words, every node in ROS2 should be responsible for a single, module purpose (e.g. one node for controlling wheel motors, one node for controlling a laser range-finder, etc).

This definition, despite being very simple, represents the very strength of the ROS2 technology: every task in a robotic application is performed by a certain piece of code which is **independent and separated** from the others. It has in itself all it is required to perform that particular role it has been assigned to, however it is not enough to realize an entire robotic application by itself: such application will be composed of multiple nodes (an arbitrary number of them) each one cooperating with the others to reach its goals.

This basic feature leads to several benefits:

- Reduced code complexity: a monolithic application such as the ones used before the introduction of ROS is very hard to organize and understand for any developer. Its code is too large to be properly partitioned: distinguishing different parts of it for different tasks results to be a messy operation. A modular architecture like the one of ROS lets a new developer understand a previously written code in an easier way, as the application's tasks are divided in nodes (which are physically separated pieces of code) and can be easily distinguished.
- Easier debugging: nodes are independent even from the point of view of execution. If something in the application goes wrong, the debugging process

is much easier if simple tasks are performed by short pieces of code, rather than having a unique monolithic code in which spotting the error when it occurs is clearly more difficult.

- Fault tolerance: if a node goes down, its failure does not spread to the others. The only thing that could happen is that some nodes raise an error when other nodes, from which they were expecting some specific data for their execution, fail.

- Language portability: it is easier to use alternative implementations of the same node, even if written in a language different than the one it was originally developed in.

We will not go in the details on how to develop even a simple ROS2 node, as the community's official tutorials already provide a clear and complete introduction to the ROS2 basic coding principles. A ROS2 node can be written either in C++ or Python (they're the main languages supported so far, but support for further languages is being developed) and the developer can take advantage of the **ROS2 Client Library**.

Topics

Being ROS2 composed by single entities acting separately from each other, a form of communication between them is strongly needed, as one node is not sufficient to implement an entire robotic application. This is the reason why **topics** have been introduced: they are a vital element of the ROS2 technology that act as a bus for nodes to exchange messages. Together, nodes and topics form the simplest form of *ROS2 graph*, in which the nodes communicate with one another through messages published in topics.

We say "published" because of one specific reason, which is another key concept of ROS2: **communication between ROS2 nodes follows a publish/subscribe paradigm**, unlike other kinds of applications which commonly use a client/server approach. A single node may publish data to any number of topics, hence a single topic may have multiple publishers and multiple subscribers, but each specific one is strictly tied to a specific kind of message (see next section). If a node does not subscribe to a topic, it will not hear data published on it, of course. In few words, topics are named buses over which nodes can communicate, thought for unidirectional streams of data. The default data transport protocol for ROS2 communication between nodes is TCP, but UDP can be used.

How can this conceptual architecture be implemented? Which entity is the one in charge of actually connecting these nodes and make them exchange messages

through topics? We will have the answer in Section 2.3, in which we will talk about the DDS middleware: at this stage, we limit to consider this mechanism only from a high-level point of view, as a way to put our nodes in communication.

Messages

As previously said, a certain topic is strictly tied to a specific kind of message: **messages** are none other than data structures created to represent what kind of information is being exchanged between two or more nodes at a certain time. In other words, messages are a structured way of defining specific sequences of bits exchanged through topics.

Most primitive types, such as *Int32*, *Int64* and *String* are available, and the developer can define its own messages by means of a text file with the *.msg* extension. More complex kinds of messages can therefore be created, represented by arrays of primitive types, nested structures and so on. As we already discussed, ROS2 nodes can only exchange a message in a topic only if they both use the same message type. Some commonly used ROS2 messages, which are strongly leveraged by **ALBA Robot** in their navigation system, are defined in the `std_msgs`, `sensor_msgs` and `geometry_msgs` packages.

Services

Services are another method of communication for nodes in the *ROS2 graph*: they are based on a **request/response** model, in contrast with the topics' publish/-subscribe approach. In other words, while topics follow a paradigm in which the subscribed nodes can hear all the message independently published by another one on that topic, in the services working principle a node requests the activation of a certain "service" to a node holding it, and this node responds to the request with the data required by the asking node. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client.

There are some situations in which a publish/subscribe approach is not the best to adopt indeed, as some node may need some data at a certain moment in time, while they're not being published by any other node in the meantime. It is better to request those data directly to their holder in this case. This approach is very similar to the client/server paradigm, but under the hood it is realized with topics as well: indeed, ROS2 defines a "request" topic in which a certain kind of message is sent, representing the sending node's request to be delivered to the destination node, and this one will publish the reply in a "response" topic, which can be a message of a different kind. The user is agnostic to that, as it only needs to know that a client must request some data to a server through a ROS2 Service.

Indeed, a so-called "Service Client" must be defined in the requester node and a "Service Server" in the destination one, which holds the required data. Services can be defined in a *.srv* file, in which two kinds of messages are specified, one for the request and one for the response. Even in this case a specific Service request is tightly coupled with a specific type of message, and the same goes for the response.

Actions

It may happen, sometimes, that a certain service requested by a node intrinsically requires some time to be fulfilled. In this case, a "normal" ROS2 Service as described above is not the best way of implementing such data exchange, as it would assume that the answer is ready to go as soon as the request reaches the Server node. In this case, the Service Client does not know the state of its request (namely which stage of its completion it has reached) and cannot even cancel it.

Actions are another communication type in ROS2 specifically intended to address these kinds of situations, namely **providing requested data for long running tasks**. They consist of three parts:

1. **Goal**: it is the initial message sent by the node requesting the Action (Action Client), exactly as it does in a ROS2 Service. This message represents the final desired state the node wants the system to reach.

2. **Feedback**: it is a message periodically sent to the Action Client during the time required for the Action's completion. In other words, while the system is executing the requested Action, it informs the Client node of its current state, so that it knows how the Action's development is going. During this phase, an action can be interrupted by the Client.

3. **Result**: the Action's final message, sent by the Server node at its completion. It represents the fact that the Action has been performed to its end and the system has reached the desired final state.

These three phases are actually built on topics as well, and are tightly coupled with three respective types of messages. Indeed, a developer can define its own Action by specifying the message kinds for goal, feedback and result in a *.action* file. Their functionality is similar to Services, except Actions are preemptable (they can be interrupted during their execution) and also provide steady feedback. As already seen, Actions use a client/server model as well, in which an "Action Client" node sends a goal to an "Action Server" node that acknowledges such goal and returns a stream of feedback and a result.

2.2 Nav2

ROS2 is not the only Software framework used in the *ALBA Robot* navigation architecture: it is at the very core of the company's autonomous driving solution as the vehicle's navigation tasks are deployed inside ROS2 nodes. However, ROS2 alone is not sufficient to implement such a complex set of robot behaviours: an additional framework, built on top of it, is needed to fulfil the actual autonomous driving tasks: let's see in which way.

The **Nav2 project** (short for Navigation2) can be seen as a spiritual successor of the original ROS Navigation Stack, and it is mentioned by the ROS community as one of ROS sister organizations. Many groups originated in ROS have indeed grown to the point where they are large enough to be considered their own standalone organization. In particular, Nav2 is a project that **seeks to find a safe way to have a mobile robot move from point A to point B**: their prime goal is to implement a complete navigation system using ROS2 nodes, topics, services and actions. It may also be applied in other applications that involve robot navigation, like following dynamic points, dynamic path planning, compute velocities for motors, avoid obstacles, and structure recovery behaviors.

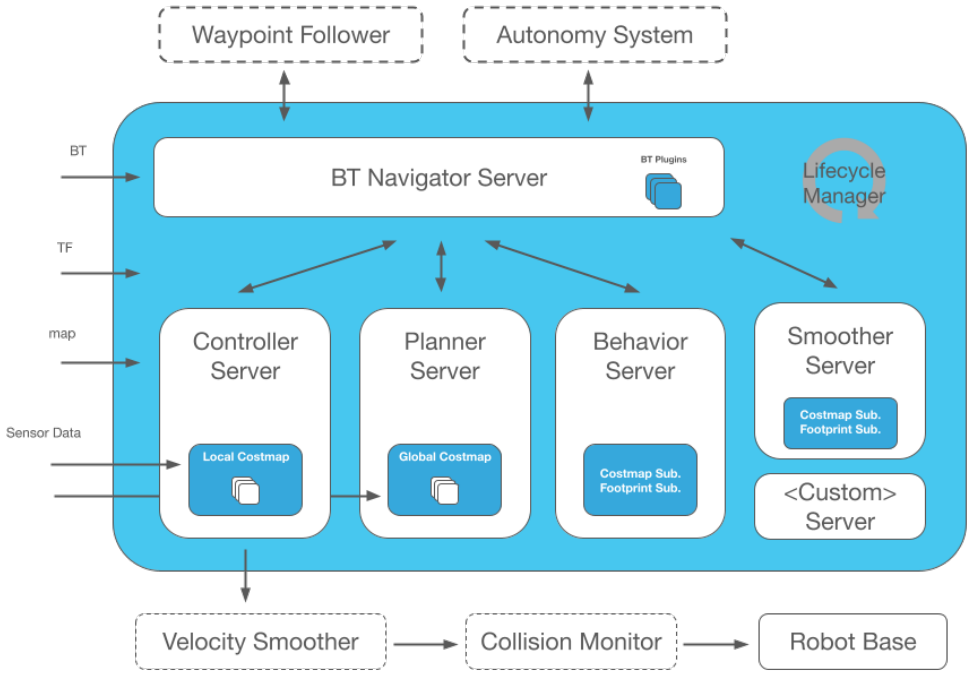

Figure 2.1: Nav2 architecture

Figure 2.1 shows a conceptual view of the so-called "Nav2 stack", highlighting the different and independent components the Nav2 project developed (which are actually represented by ROS2 nodes under the hood). As can be seen, the Nav2 stack uses the so-called behavior trees (BTs) to call **modular servers** which are in charge of completing a certain action, either compute a path, control effort, recovery, or any other navigation related action. These modular servers are each separate nodes that communicate with the BT through ROS2 actions. We will not enter in the detail of the working algorithm every single Nav2 component, like BTs: we just give a general description of the Nav2 architecture in the parts the reader needs to understand the work done in this thesis.

As we can see, the expected inputs to Nav2 are TF transformations, a map source if utilizing the Static Costmap Layer, a BT XML file, and any relevant sensor data sources: the stack will eventually provide valid velocity commands for the robot's motors to be followed, which are the output it produces. Currently, all of the major robot types are supported by the project (holonomic, differential-drive, legged, and ackermann (car-like) base types). Generally, we can say that the Nav2 stack has a set of tools with the following navigation capabilities:

- Load, serve, and store maps (Map Server)
- Localize the robot on the map (AMCL)
- Plan a path from A to B around obstacles (Nav2 Planner)
- Control the robot as it follows the path (Nav2 Controller)
- Smooth path plans to be more continuous and feasible (Nav2 Smoother)
- Convert sensor data into a Costmap representation of the world (Nav2 Costmap 2D)
- Build complicated robot behaviors using behavior trees (Nav2 Behavior Trees and BT Navigator)
- Compute recovery behaviors in case of failure (Nav2 Recoveries)
- Follow sequential waypoints (Nav2 Waypoint Follower)
- Manage the lifecycle and watchdog for the servers (Nav2 Lifecycle Manager)
- It has plugins to enable custom algorithms and behaviors (Nav2 Core)

We do not need, as previously mentioned, a detailed description of every single feature of Nav2, as this thesis is not focused on the development of an entire navigation system: we only need to shift an existing Nav2 architecture in a cloud server, hence we only focus on what's important in this sense.

2.2.1 Navigation concepts

As we know, Nav2 is based on ROS2, which basically means that the Nav2 project leveraged the libraries and services exposed by ROS2 for the implementation of their own version of navigation tasks for a generic robot. This does not mean that knowing ROS2 is enough to understand all it is to know about Nav2: quite the contrary, indeed. The Nav2 project introduces **new concepts** to the ROS2 model built on top of it, as well as making a massive use of the base ROS2 definitions. Let's have a look on the main features added by this project, as well as how they use the ones already defined. A more detailed description can be found in the official documentation of Nav2.

Navigation servers

This is not actually a new concept: they are exactly the actions servers defined by ROS, which we already talked about. Nav2 makes an extensive use of actions, as can be easily guessed, because they are the most appropriate way of controlling long running tasks such as the navigation of a robot from a point A to a point B. The Nav2 stack has exactly this goal in mind: consequently, actions servers are widely leveraged by this project.

Some specific action servers are defined in this stack, called **Navigation Servers**, to perform the main navigation tasks. Overall, they communicate with the BT navigator (see Figure 2.1) through a *NavigateToPose* action message. These messages are also used for the BT navigator to communicate with the smaller action servers to compute plans, control efforts, and recoveries. Let's see the main types of Navigation Servers defined by the Nav2 project and shown in Figure 2.1, in charge of implementing the key roles of the robot navigation:

- **Planner server**: the general task in Nav2 for the Planner is to **compute a valid and potentially optimal path** from the current pose to a goal pose. Many classes of supported plans and routes exist, and the path can also be known as "route", depending on the nomenclature and algorithm selected. Also, complete coverage can be performed, which basically means to compute paths that cover all free space. The planner will have access to a global environmental representation and sensor data buffered into it.
- **Controller server**: it is also known as "local planner" in ROS1, it is the action server in charge of **following the global path** computed by the Planner, or complete a local task. The Controller will have access to a local environment representation (the Local Costmap, as we'll see) to attempt to compute feasible control efforts for the base to follow. In other words, it produces commands for the robot's Hardware to make it follow a specific path, built after the footsteps of the global one.

- **Behaviours server**: recovery behaviours are a mainstay of fault-tolerant systems. The goal of recoveries is to **deal with failure conditions** of the system or conditions in which the robot does not know what to do, and autonomously handle them. An example may be sensors faults, which result in an environmental representation being full of fake obstacles: in this case, a recovery would be to clear the costmap to allow the robot to move. Another one would be the robot being stuck due to dynamic obstacles or poor control: backing up or spinning in place, if possible, allows the robot to move from a poor location into free space in which it may navigate successfully. Finally, in the case of a total failure, a recovery may be represented by calling an operator's attention for help via email or SMS, for instance.

- **Waypoint Following Server**: "waypoint following" is a basic feature of a navigation architecture. It tells the system how to use navigation to **get to multiple destinations**. The `nav2_waypoint_follower` package can also be useful if the robot must go to a given location in space and complete a specific task like pick up a box or wait for user input.

- **Map Server**: it is not an action server, but a simpler service server, which is in charge of **providing the base map** for the rest of the stack. Indeed, the Nav2 stack has a peculiar way of representing the navigation environment (see next subsection), by starting from a map saved in a static file, given to any requesting node as a ROS2 service. On top of this static map, all dynamic environmental information will be added as layers. This package also implements a Map Saver server which runs in the background and saves a map based on service requests, through a specific CLI.

Environmental representation

The Nav2 project defines specific ways in which the robot perceives the surrounding navigation environment. It also acts as the central localization for various algorithms (which we'll not see) and data sources to combine their information into a single space. This space is then used by the previously mentioned Navigation Servers to compute their tasks safely and efficiently.

As we said earlier, the representation of the environment in which the robot moves is based on a **static map** provided by the Map Server: then, a so-called **costmap** is added, which is a regular 2D grid of cells each containing a cost taken from some specific values: "*unknown*", "*free*", "*occupied*", or "*inflated*". The costmap is implemented as an additional layer which comes in two versions: the first one is called "Global Costmap", inspected to compute a global plan, while the other is known as "Local Costmap", sampled to compute local control efforts. Various layers are implemented as plugins which buffer information into the costmap,

including info from sensors, cameras, etc. It may be wise to process sensor data before inputting it into this layer, but that is up to the developer.

In other words, this costmap layer provides a structure that maintains information about where the robot should navigate, in the form of an occupancy grid. It uses the static map and sensor data to store and update information about obstacles in the world. Costmaps can be created to detect and track obstacles in the scene, for collision avoidance, using camera or depth sensors. Also, they may be used to buffer live data into the 2D or 3D world for binary obstacle marking.

The Costmap automatically subscribes to sensors topics over ROS2 and updates itself accordingly: each sensor is used to either mark the Costmap (insert obstacle information), clear it (remove obstacle information), or both. Finally, if a 3D structure is used to store obstacle detections, it gets projected down into two dimensions when put into the Costmap.

2.2.2 Lifecycle nodes

The concept of **lifecycle nodes** surely is one of the most important for a deep comprehension of this thesis' solution for a ROS2 autonomous driving system in cloud. Lifecycle (or *managed*, more correctly) nodes are unique to ROS2 and they're strongly leveraged by the Nav2 project for their model of robotic applications. In a nutshell, they are **nodes with a lifecyle**, as the name itself states, which follows the behaviour of a *Finite State Machine* (FSM): therefore, they contain state machine transitions for their bringup and teardown.

Figure 2.2 below shows a complete view over the lifecycle states any managed node must follow, which can be divided in two groups (highlighted with different colours):

- *Primary* states, which are:

    1. **Unconfigured**: a managed node gets in this state immediately after being instantiated. Also, it may return to it after an error happens, transitioning from a previous state. When in this condition, the node is not even configured with the primary components needed to perform its tasks.
    2. **Inactive**: during this state, a node that is not currently performing any processing. Its main purpose is to allow a node to be (re-)configured, which means changing its configuration parameters and/or adding topic publications and subscriptions. It is always a better choice to configure a node when it is not up and running, to avoid altering its working algorithm. In the Inactive state, managed (lifecycle) topics are deactivated and any data that arrives on them will not be read and or processed. Similarly, any managed service requests to a node will not be answered.

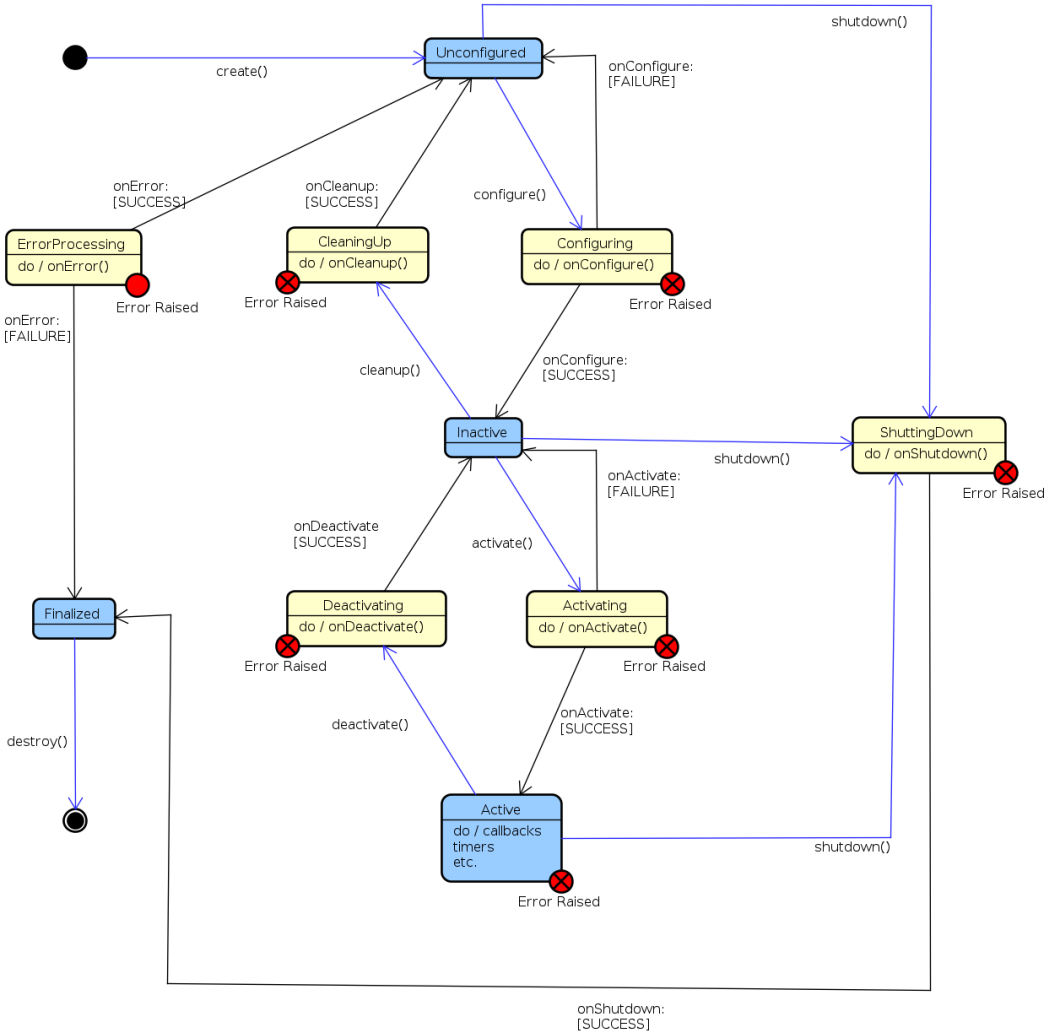

Figure 2.2: Lifecycle states transitions

3. **Active**: the main state of a managed node's lifecycle. While in this state, the node performs its main executions, responds to service requests, reads and processes data, produces outputs, etc... If an error that cannot be handled by the node or the system occurs in this state, the node will trigger a transition to ErrorProcessing.
4. **Finalized**: a node ends up in this state immediately before being destroyed. It is always terminal, as the only transition from here is elimination. This state exists to support debugging and introspection, because a failed node remains visible to any system introspection and may be potentially inspected by debugging tools rather than directly being destructed.

- *Transition* states, which are:

    1. **Configuring**: the node's `on_configure()` callback is called to allow loading its configuration (as said, creation of publishers/subscribers and parameters configuration) and conduct all its required setup. Indeed, the tasks that shall be performed in this phase are the ones to be made only once during the entire node's lifetime.
    2. **CleaningUp**: in this transition state, the node's callback `on_cleanup()` is called, which is expected to clear the node's state and make it return to a functionally equivalent status as when first created. If the cleanup cannot be successfully achieved, it will transition to ErrorProcessing.
    3. **ShuttingDown**: in this state, the callback `on_shutdown()` will be executed to do any cleanup necessary before destruction. It can only be entered from any Primary State except Finalized.
    4. **Activating**: the callback `on_activate()` performs final preparations for execution: this may include acquiring resources to be reserved exclusively while the node is actually active, such as Hardware access. No preparation that requires significant time should be performed in this callback, as the process of activating a managed node should, theoretically, be fast.
    5. **Deactivating**: in this transition state, the callback `on_deactivate()` is called, and it performs all the cleanup of the components needed to start executing. It should therefore reverse the changes made by the `on_activate()` function.
    6. **ErrorProcessing**: this is where any error can be cleaned up. It is possible to enter this state from any other state where user code is being executed. If errors have successfully been handled, the node can return to Unconfigured, while if a full cleanup is not possible it must fail and transition to Finalized, in preparation for destruction.
    7. **Destroy**: this transition will simply cause the deallocation of the node. In an object-oriented environment, it just involves the invocation of the destructor. This transition should always succeed.

There are several ways in which a managed node can trigger a transition between lifecycle states. However, most of them are expected to be **coordinated** by an external management component, which should configure the node, start it, monitor it and execute recovery behaviors in case of failures. *ALBA Robot* have developed their own "management tool", which is the `alba_v2_guardian` node: we will extensively talk about this particular node, as it represents the key for our local/cloud switching mechanism.

The fact that our ROS2 nodes follow a precise lifecycle allows greater control on the overall state of the robotic application we're developing. It is a way of knowing that all nodes have been instantiated correctly before allowing them to be executed and perform their tasks. It will also allow nodes to be restarted or replaced on-the-fly. In few words, a managed node presents a known interface, executes according to a known life cycle state machine, and otherwise can be considered a black box. This helps in deterministic behaviour of ROS2 systems in startup and shutdown and allows freedom on how to provide the lifecycle functionalities, while also ensuring that any tools created for managing nodes can work with any compliant node. It also helps users structure programs in reasonable ways for commercial uses and debugging.

Nav2 uses a wrapper of the ROS2 Client Libraries' LifecycleNodes, which is `nav2_util` *LifecycleNode*. It wraps most of the original ROS2 *LifecycleNode*'s complexity and includes a bond connection for the lifecycle manager to ensure that after a transition a node is up, it also remains active. If a server crashes, it lets the lifecycle manager know and transition down the system to prevent a critical failure.

2.3 DDS

So far, we have described the basic features offered by ROS2 and the more complex concepts and behaviours developed by Nav2 and built on top of that. Now, let's make a step back for a moment: how is the **communication** between ROS2 nodes actually implemented? What lies under the hood of the ROS2 technology that makes all the message exchange among nodes possible? Where does the primitive concept of topic come from?

All the answers to these questions can be found in the **network middleware** laying underneath the ROS2 technology, namely **DDS**. The so-called *Data Distribution Service* (DDS), defined by the Object Management Group (OMG), is a *"data-centric communication protocol used for distributed software application communications"* [1]. In simpler words, DDS is not a technology in itself, but a set of **specifications** that must be followed in any possible actual implementation of the middleware, in order to be compliant to the base concepts defined by the DDS community. It describes the APIs and semantics that enable communication between data providers and data consumers.

Since it is a *Data-Centric Publish Subscribe* (DCPS) model, it defines four basic elements, which have a 1:1 match with the already discussed ROS2 principles:

[1] Definition from the Fast DDS official documentation

- **Publisher**: publishers are publication entities, which define the information-generating objects and their properties. They're the DCPS component in charge of the creation and configuration of the **DataWriters** they implement, which are the entities in charge of the actual publication of the messages. Each DataWriter will have an assigned topic under which the messages are published. It is easy to understand that **ROS2 publishers are none other than DDS publishers** under the hood.

- **Subscriber**: the same goes for subscribers, which are subscription entities in charge of defining the information-consuming objects and their properties. They receive the data published under the topics which they're subscribed to and serve one or more **DataReader** objects, which are responsible for communicating the availability of new data to the application. As well as publishers, ROS2 subscribers can be mapped to DDS subscribers.

- **Topic**: it is the entity that binds publications and subscriptions, namely DataWriters and DataReaders. It is unique within a DDS domain and allows the uniformity of data types in publications and subscriptions.

- **Domain**: this is the basic concept that binds all publishers and subscribers exchanging data: indeed, a DDS Domain is uniquely identified by a domain ID and every one of the DDS (or ROS2) entities executed in a certain Local Area Network must operate under a certain DDS Domain ID in order to communicate. The DDS domain Id which directly maps into the concept of `ROS_DOMAIN_ID`. Every individual element participating in a Domain is called **DomainParticipant**: two participants with different IDs are not aware of each other's presence in the network, therefore the DDS specifications allow for great possibilities of **isolation** between processes.

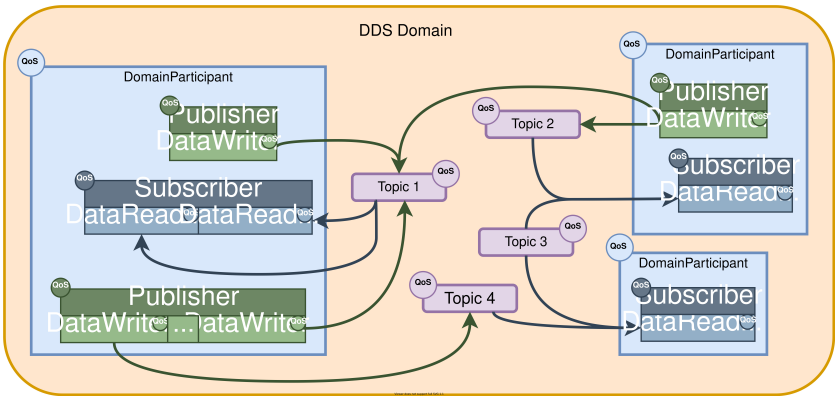

Figure 2.3: DDS domain

To go into further detail, the DDS specification is actually built on top of a proper **network protocol**, which is the **RTSP** *(Real Time Publish Subscribe)* protocol. It is a publication-subscription communication middleware over best-effort transports such as UDP/IP, designed to support both unicast and multicast communications. This is the original layer that defines the specifications for all the basic entities, such as publishers and subscribers, which are then wrapped and implemented by DDS and mapped to ROS2. However, this thesis has no interest of investigating the network middleware and the underlying network protocol in their slightest detail, as our focus is on understanding the main concepts behind the ROS2 technology and how we can leverage them in order to design a cloud solution for our autonomous driving system. For further investigation, the Fast DDS official documentation can be consulted.

The peculiar thing about DDS, which can be guessed in Figure 2.3, is that despite being a publish/subscribe protocol exactly like MQTT, for instance, it is **not tied to the presence of a central server**, acting as a broker. Indeed, in most publish/subscribe protocols the broker is the one in charge of matching publishers and subscribers, in the sense that it is the only entity aware of who is subscribed to the topics in which a certain publisher is sending data into: this broker is the one delivering messages to all interested subscribers. In DDS, the situation is slightly different: the paradigm adopted is exactly the same, but there is a complete absence of any central broker and the DDS middleware itself can be seen as the one in charge of the publishers/subscribers matching.

How can this be achieved? As a data-centric model, DDS is built on the concept of a *"global data space"* accessible to all interested entities. Whoever wants to contribute with information in the global data space just declares its intent to become a publisher, whereas applications that want to access portions of the data space declare their intent to become subscribers. Each time a publisher posts new data, DDS propagates the information to all interested subscribers through this space. The middleware itself can be seen as a distributed broker, not only composed of a single machine (which is a SPoF), but of a distributed "cloud" that embraces all DomainParticipants in a certain DDS Domain and puts them in communication.

This may be the greatest strength of DDS, namely the fact that it is a real distributed system, without the necessity for the developer to instantiate a central server to enable communication. And this is what a ROS2 user sees every day: when ROS2 nodes are executed in the same LAN, with the same Domain ID and the same Transport Protocol, they can immediately communicate. These three variables are the ones that define a **DDS Network**, as we'll see in the next sections.

2.3.1 eProsima Fast DDS

One of the most important implementations of DDS is **Fast DDS**, which is an open-source C++ implementation developed by eProsima. Their library provides both an API and a communication protocol that deploy a *Data-Centric Publish Subscribe* (DCPS) model as implementation of the DDS specifications, with the purpose of establishing efficient and reliable information distribution among real-time Systems.

There's not that much of a difference in terms of concepts if we compare different implementations of the same DDS middleware, as all of them are based on the same exact fundamental guidelines. In the next sections, we're going to consider some key features of the DDS middleware we need to understand for our purposes, and we're going to take Fast DDS as reference. Why this specific product and not another? The answer is straightforward: the ROS2 community has chosen Fast DDS as the **default implementation** used by ROS2 nodes, in particular for the *Foxy Fitzroy* release, which is the one leveraged by this work. Other implementations can be used when launching ROS2 nodes, but they must be explicitly installed and set in the process of executing a node, otherwise Fast DDS will always be preferred. Actually, at the time of this thesis a new version of ROS2 has being released, namely the *Galactic Geochelone*, which uses Cyclone DDS as base middleware, but we're not going to consider that.

As we already said, the base DDS working mechanisms are not of our interest and a curious reader can directly consult the , which goes deeply in details on the functioning and implementation of the Fast DDS library, as well as the official GitHub repository, in which the whole source code is available for use and study. In these sections, the only thing we're going to mention is a basic DDS feature, very important for us: Discovery.

2.3.2 Discovery

This is the most important concept we must retrieve out of this section: Fast DDS, as a DDS implementation, provides a **Discovery mechanisms** that allow for automatically finding and matching DataWriters and DataReaders across Domain-Participants, so they can start sharing data. In other words, the main DDS entities such as Publishers and Subscribers **must discover each other** in the network they're being executed in, in order to be aware of the presence of all the others and communicate with them. If a certain entity is not discovered by the other ones, that entity will be isolated: it will be like they are in separated and unreachable networks, even if they subscribe to the same topics.

How is Discovery achieved in Fast DDS, then? First, we can say that the Discovery process is performed in two stages:

1. Participant Discovery Phase (PDP): during this phase, the DomainParticipants acknowledge each other's existence. To do that, each DomainParticipant sends **periodic announcements** specifying, among other things, unicast addresses (IP and port) where that participant is listening for incoming Discovery metatraffic. Two given participants will match when they exist in the same DDS Domain: by default, the announcement messages are sent using well-known **multicast addresses and ports**, calculated using the DDS DomainID. Furthermore, it is possible to specify a list of addresses to send announcements using **unicast** (we will see it with the Initial Peers List) and the periodicity of such announcements is configurable.

2. Endpoint Discovery Phase (EDP): the DataWriters and DataReaders discover each other in this phase. To do that, the DomainParticipants share their information using the communication channels established during the PDP. This information contains, among other things, the **topics** and their **message data type**: for two endpoints to match, their topic and data type must coincide. Once DataWriter and DataReader have matched, they are ready for sending/receiving user data traffic.

We do not need to refer to these stages in DDS terms: if we bring these concepts into a ROS2 high-level model, a DomainParticipant is none other than a ROS2 node, while DataReaders and DataWriters are Publishers and Subscribers. In the PDP, nodes discover each other in order to be aware of the presence of the others in the DDS Network (the DDS Network itself is actually defined by that). Once done that, they start matching their Publishers and Subscribers basing on the name of the topic and the message data type, so that every message published in a certain topic will be visible by any node subscribed to that topic: this is the EDP.

With this little introspection on DDS, it is now clear how the base middleware implements the more high-level concepts of ROS2 developers use every day, even without knowing what lies underneath. There is not, though, only a single way in which DDS entities such as ROS2 nodes can discover each other, as Fast DDS provides **four main Discovery mechanisms** that can be alternatively used, depending on the explored use-case:

- **SIMPLE Discovery protocol**: it is the base mechanism resolving the establishment of the end-to-end connection between various DDS Entities. The eProsima Fast DDS implementation of this protocol provides compatibility with the RTPS standard, as it is the basic Discovery setting defined by it. It upholds this standard for both PDP and EDP and therefore provides compatibility with any other DDS and RTPS implementations.

  In ROS2 terms, this SIMPLE protocol is the *default* one, used when we execute any ROS2 node in a certain network: all other nodes already running in the same LAN, specifying the same `ROS_DOMAIN_ID` (DDS DomainID), will automatically be discovered: the user does not have to put any effort in making nodes inside a LAN communicate if this protocol is used.

  It is important to highlight that the Discovery traffic used in this phase by a DDS entity to acknowledge the presence of possible other entities is **multicast** traffic. As well as other IP broadcast mechanisms, multicast cannot exceed the LAN boundaries, and it is not supported in most environments outside it. We will return on this problem, as it will be the base for our considerations on how to make ROS2 nodes communicate between different LANs, over WAN networks.

- **STATIC Discovery protocol**: Fast DDS allows for the usage of a specific protocol for the EDP phase, different than the one used in the SIMPLE Discovery mechanism, which is a static version that completely eliminates EDP meta traffic. This is very useful when we're dealing with limited network bandwidth and a well-known schema of DataWriters and DataReaders.

  In other words, by leveraging this protocol DDS entities are discovered "manually", meaning that if a well-known set of entities (DataWriters, DataReaders, their topics and respective data types) is already deployed in the network, Discovery meta-traffic can be avoided by replacing it with a **static configuration** of peers, called Initial Peers List. It is important to highlight that by doing this, no EDP discovery meta traffic will be generated, and only those peers defined in the configuration will be able to communicate.

  The same concept matches to ROS2: if the user knows about the presence of other ROS2 nodes running in the network, they can be specified in the new node's configuration, so that it will automatically assume their presence without trying to discover them with Discovery metatraffic.

- **Discovery Server Discovery protocol**: This mechanism uses a **centralized discovery architecture**, where a DomainParticipant referred as Server, acts as a hub for meta traffic discovery. Hence, this protocol is based on a client-server paradigm, in which the Discovery metatraffic is managed by one or several central DomainParticipants servers, as opposed to simple discovery,

where metatraffic is exchanged using a message broadcast mechanism like an IP multicast protocol. We will return on this type of Discovery when we'll talk about cloud environments.

- Manual Discovery: This mechanism disables the PDP phase, letting the user manually match and unmatch RTPSParticipants, RTPSReaders, and RTPSWriters using whatever external meta-information channel of its choice, not even using DDS. We will not consider this Discovery protocol at all.

Discovery is actually the only thing we really need to know about DDS in this thesis, as different Discovery mechanisms are leveraged in the definition of a cloud solution for the *ALBA Robot* autonomous driving system.

At this point, we have all the fundamental building blocks: ROS2, Nav2, Lifecycle nodes and the Discovery processes defined by the DDS specification. Of course, the company uses all these concepts as base for their autonomous driving application, but their specific field of interest requires specialization, adapting these technologies to the precise solution they want to acquire. For instance, specific custom ROS2 node are being developed.

Hence, the next step in the definition of the work done in the company's environment is to dig into the actual project they're currently working at, describing its main features and components. The cloud autonomous driving solution developed by this thesis is built on top of their architecture, hence we need to introduce it in order to deeply understand such solution.

Chapter 3

The *SEDIA* project

SEDIA is the main project developed within *Alba Robot srl*, consisting in an innovative mobility platform, based on ROS2 and Nav2. Basically, it is a self-driving wheelchair, at this time developing into a proper vehicle, intended for people with reduced mobility who can take advantage of the its capabilities in various facilities such as airports, hospitals and museums.

At the time of this thesis, *SEDIA* is not yet a product: it is in fact a **prototype** being developed by the Engineers inside the company, with the support of all the other members for complementary work, such as testing.

3.1 *SEDIA* state-of-the-art

The *SEDIA* project can be divided in two main parts:

- **Low Level**: it concerns the Hardware components such as cameras, sensors and all the electronic and mechanical parts that compose the vehicle and perform its physical navigation. It is the "body" of *SEDIA*, without any computational intelligence such as the one offered by a CPU, meaning that every processing performed by the low level is done in Hardware. The goal of the Low Level is to gather information about the surrounding environment for the High Level to process, and to take navigation inputs from the High Level, intended to perform some navigation tasks, and move the wheelchair accordingly.

- **High level**: the "brain" of *SEDIA*. Visibly (at the time of the thesis), it is solely composed by a *Jetson Xavier Nvidia NX* single-board computer *(SBC)*, hence a very tiny part of the overall Hardware, and constitutes the intelligence in which the ROS2+Nav2 system runs. The role of the High Level is to take input data about the environment, take decisions to enable the intended

navigation task (making the robot move from a point A to a point B, for example) and generate commands for the Low Level to make it perform that navigation.

We're focusing our attention on the **High Level**, being it the part of the system we can shift to the Cloud, but we'll see a brief overview of both parts of *SEDIA*.

3.1.1 Low Level

As said, the **Low Level** is the name by which *Alba Robot* identifies the set of electronic components, such as cameras, sensors, boards, and controllers which are applied to the structure of the vehicle, being it a wheelchair or a scooter, to enable its mechanical movements to perform navigation and to gather information about the environment, in order to let the High Level take decisions about the navigation itself.

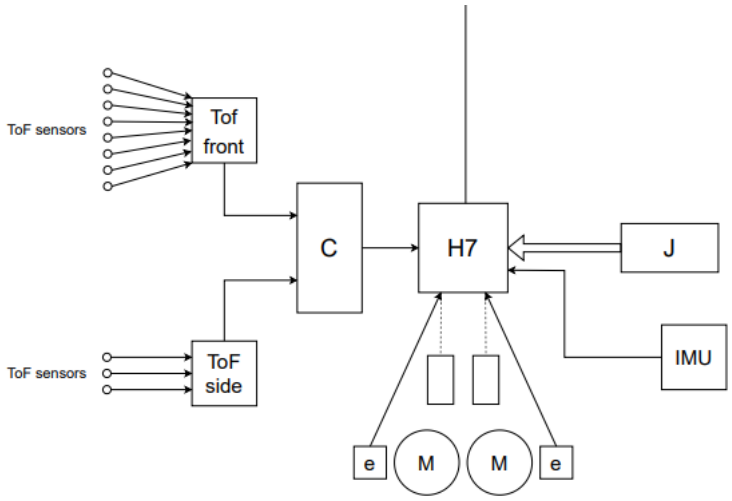

Figure 3.1: *SEDIA* Low Level schema

Figure 3.1 represents a simple explanatory picture of the Low Level, with its main components, which we're not going to focus on in detail, being the Low Level strictly tied to the physical structure and mechanics of *SEDIA* and therefore not prone to be moved elsewhere (like to the Cloud).

All the communications between the Low Level components, shown in Figure 3.1, are *UART* communications at the time of this thesis (effort is being done at this time to switch to *CAN* communication), except for the one between the Joystick (J) and the Controller (H7), which is a *serial* communication.

Starting from the left, we have:

ToF sensors

The *Time of Flight (ToF)* sensors belong to the family of perception sensors, providing with 360-degree close-range and fast-rate obstacle coverage. The Navigation system uses the *ToF*-retrieved information to avoid close objects and for emergency stop, in other words they're mainly used for obstacle avoidance and their info can be used by the Nav2 system to force the vehicle to stop if an obstacle cannot be avoided.

The *ToF* sensors' system on *SEDIA* is divided in two clusters, each one composed of a pair of 3D-printed scaffolds in which the single sensors are inserted:

- The **front clusters** are located in the frontal corners of the wheelchair, with up to 8 sensors inserted in each 3D-printed support, resulting in about 200 degrees horizontal *FoV* (Field of View) per corner and about 27 degrees in vertical *FoV*. The rate at which each sensor can work depends on configurable parameters. In order to achieve better performance and measure reliability, the sensors operate in a "short distance" mode, meaning that the maximum sensed distance is up to 1,3 meters. The resulting field of view of each group has the following pattern:

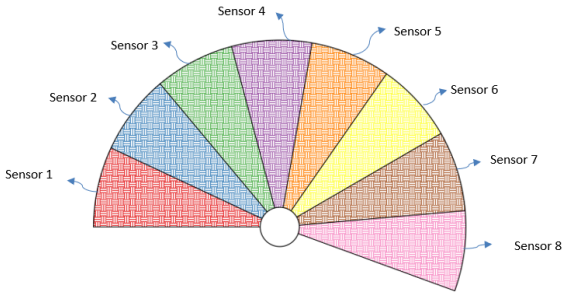

Figure 3.2: *ToF* front cluster

- The **side clusters** are placed in the rear corners, also on the left and on the right of the wheelchair, in order to achieve lateral view, crucial for rotations in tight spaces. They consist in 3 *ToF* sensors and cover around 75 degrees in an arc of 125 degrees:

In addition to the previous sensors, for each side board an ultrasound sensor has been connected in order to get obstacles data from the rear part of the system.

The *ToF* Sensor System is characterized by the *Embedded Firmwares*: they manage the different *ToF* sensors clusters and communication with the Collector.

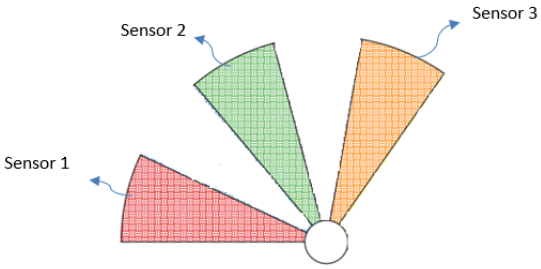

Figure 3.3: *ToF* side cluster

Collector

The **Collector** is simply a module in charge of gathering the *ToF* data and sending them to the Controller.

Actually, it is a bit more complex than this and performs some additional tasks than the one mentioned: the collector firmware has three main functions:

1. Get the data from all four *ToF* clusters through the different *UART* interfaces

2. Handle and filtering measures with algorithms

3. Send the data to the Controller (H743ZI2) via *UART* interface

The Collector is characterized by the *Firmware Bridge*, which collects *ToF* clusters data and communicates with the Controller.

Controller

The NUCLEO-H743ZI2 module, **H7** in Figure 3.1. It is the most complex module, because handles the majority of system signals, elaborates and redirects them to the other boards of the system. It is also able to compute odometric data of the wheelchair thanks two sensors: the **optical incremental encoders**, that give position (velocity) data of the wheelchairt and the **IMU** board, that gives heading and angular velocity data. The Controller combines the velocity and heading data to retrieve continuously the odometric information.

Being the component that makes all the other elements of the Low Level speak to each other, the Controller is the one in charge of collecting all the system's data in order to send them to the High Level, as well as retrieve data from the High Level to give them to the Low Level, such as navigation commands to make the wheels move.

Encoders

There are two optical encoders (**e** in 3.1) EH17-30MH, and each of them is fixed on one of the two motors (**M** in 3.1).

Each encoder is composed of two main parts:

- The optical unit
- The codewheel

The optical unit, coupled with the custom codewheel, is able to provide relative position. They are fixed to the wheel thanks to gears.

IMU

An *Inertial Measurement Unit (IMU)* measures and reports a body's specific force, angular rate, and the orientation of the body, using a combination of accelerometers and gyroscopes.

It is connected to an STM32 L432KCU microcontroller which retrieves the data from the sensor, combines the information and provides the result to the Controller.

Joystick

J in 3.1, it is the component used for manual controlling of the wheelchair. The joystick has priority over the autonomous driving system, meaning that if a navigation command comes from it all the other commands generated by the ROS2 system (High Level) are ignored.

This feature can be useful either for controlling the wheelchair manually when someone wants to quickly move it without providing autonomous navigation commands, or in an emergency situation, when the vehicle is needed to be stopped, or moved towards another direction, than the one decided by the autonomous driving system.

3.1.2 High Level

The part of the *SEDIA* project known as **High Level** is the one we're focusing on in this thesis. It contains the intelligence that makes the wheelchair drive autonomously, namely a ROS2+Nav2 architecture, capable of implementing a complete autonomous navigation system.

At the time of this thesis, the Hardware module in charge of executing such system is a *Nvidia Jetson Xavier NX Developer Kit*, whilst effort is being made in order to switch to a more powerful SoC, based on a x86 architecture. This happens because the *NX* was found not capable of sustaining large amount of data, such as the ones exchanged between ROS2 nodes in very large navigation environments.

Nvidia Jetson Xavier NX

Jetson Xavier NX is the World's Smallest AI Supercomputer for Embedded and Edge Systems with cloud-native support, which allows to use the same technologies and workflows that revolutionized cloud platforms to build, deploy and manage applications on Jetson edge devices.

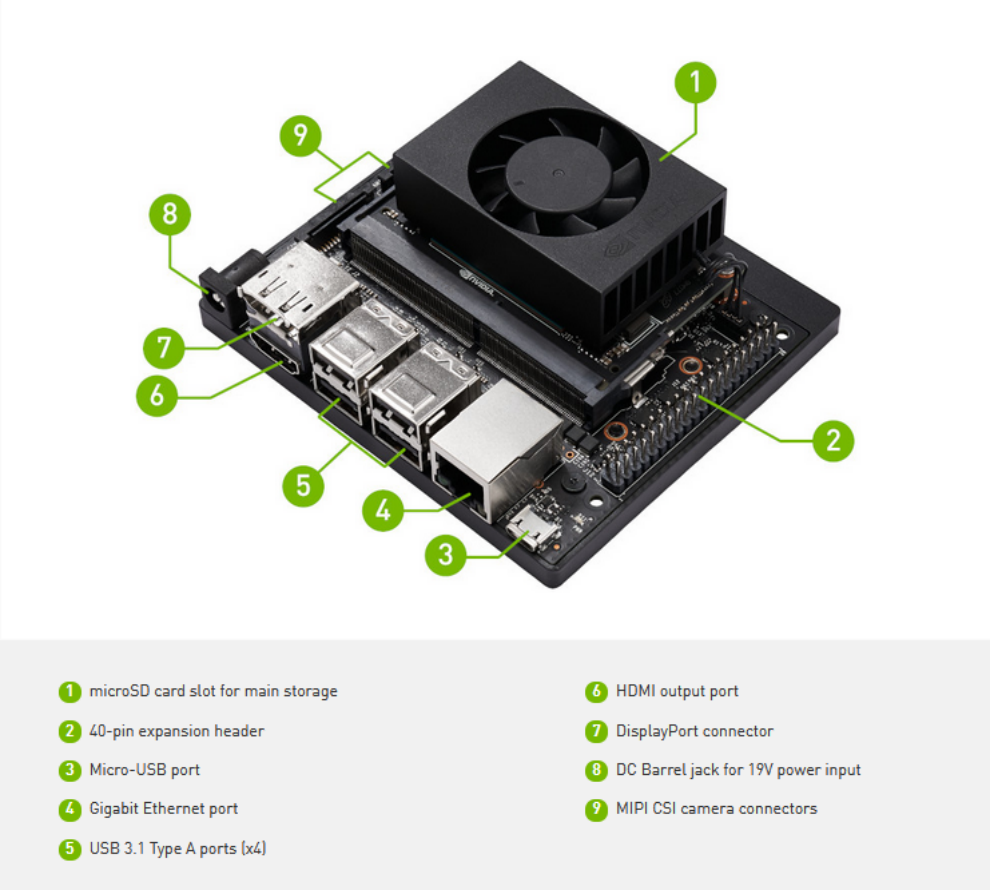

Figure 3.4: Nvidia Jetson Xavier NX Developer Kit

The **Jetson Xavier NX Developer Kit** module is the only one, among the electronic components of *SEDIA*, in which the High Level part of the system runs, while all the others constitute the Low Level, as previously said, and it is connected to the Controller through a *UART* interface.

GPU	NVIDIA Volta architecture with 384 NVIDIA CUDA® cores and 48 Tensor cores
CPU	6-core NVIDIA Carmel ARM®v8.2 64-bit CPU 6 MB L2 + 4 MB L3
DL Accelerator	2x NVDLA Engines
Vision Accelerator	7-Way VLIW Vision Processor
Memory	8 GB 128-bit LPDDR4x @ 51.2GB/s
Storage	microSD (not included)
Video Encode	2x 4K @ 30 | 6x 1080p @ 60 | 14x 1080p @ 30 (H.265/H.264)
Video Decode	2x 4K @ 60 | 4x 4K @ 30 | 12x 1080p @ 60 | 32x 1080p @ 30 (H.265) 2x 4K @ 30 | 6x 1080p @ 60 | 16x 1080p @ 30 (H.264)
Camera	2x MIPI CSI-2 DPHY lanes
Connectivity	Gigabit Ethernet, M.2 Key E (WiFi/BT included), M.2 Key M (NVMe)
Display	HDMI and display port
USB	4x USB 3.1, USB 2.0 Micro-B
Others	GPIO, I^2C, I^2S, SPI, UART
Mechanical	103 mm x 90.5 mm x 34.66 mm

Figure 3.5: Nvidia Jetson NX specifications

NX usage

As stated in the specifications, the storage module of this board is a **microSD**, which must be inserted in a specific slot located under the *Aluminium Heatsink* cooling module. The microSD must be priorly flashed with the *Jetson Xavier NX Developer Kit SD Card Image*, which in our case contains an **Ubuntu 20.04** Operating System, compatible with the *Foxy Fitzroy* release of ROS2, the one used by the company at the time of this thesis.

Once the microSD is properly flashed and inserted in the *NX*, we give power supply through the Barrel jack (8 in 3.4) and the system boots with the OS and the File System contained in the microSD. We can have access to the *NX* in two ways:

- Through **SSH**: we connect a laptop to the *Jetson NX* through the Gigabit Ethernet port (4 in 3.4), give the laptop's interface an IP address in the same subnet of the *NX*'s interface and start an SSH session in order to have a prompt in the File System of the board. Of course, the IP address of the Gigabit Ethernet interface, called *eth0*, must be previously statically set in

the */etc/network/interfaces.d/eth0* file (in Ubuntu 20.04).

- **Directly** using the HDMI (6 in 3.4) or DisplayPort (7 in 3.4) insteraces: in this case, only an external monitor is needed, along with either a HDMI or DisplayPort cable, connected to the relative connector. A terminal shows up right away, requesting to login, and a graphical interface can also be activated with the command *sudo isolate graphics*.

Repo installation and run

Once inside the *NX*, if not already done, ROS2 must be installed (the desktop version), following the public documentation, in order to be able to eventually execute the Navigation system of *Alba Robot*, contained in the company's private repository. The distro of ROS2 used at the time of this thesis, as previously mentioned, is ROS2 *Foxy Fitzroy*, but the future shifting of the company's system towards the new SoC based on x86 architecture as the hosting module for the ROS2+Nav2 system will bring to the adoption of the ROS2 *Galactic Geochelone*. We're focusing on the former, as it is the one this thesis has been working on.

Once ROS2 has been installed, we need to open a terminal and source the environment with the following command:

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

This command will execute a setup.bash script in the current terminal. That script, automatically installed with ROS2, contains all the environmental variables and definitions needed by ROS2 to run, namely the *ros2* command will work in that terminal. A test can be done, as suggested by the official ROS2 documentation, by launching some example nodes, taken from the `demo_nodes_cpp` ROS2 package, publicly available on Github. If the test is successful, it means we can proceed.

In order to execute the navigation ROS2-based system of the company in the *NX* we're using, we need to follow the instructions contained in the private repository of *Alba Robot*, which explain how to clone the repository and install all the required dependencies. Of course, we must refer to a certain release of the company's code. This thesis refers to the release v1.4, which is the one it's been working on. Once everything is installed, we also need to build all the ROS2 packages, and this can be done by means of the **colcon** tool, as explained in the repository (it is the very same tool used in the official ROS2 documentation).

If it is the first time we try to make the Alba Robot repository work in our computer, some dependencies will most likely be missing. Hence, instead of trying

to build the repo, get an error and install the missing dependency every time, wasting a lot of time, we can take advantage of the **rosdep** tool, which will search for all the dependencies required by every package contained in a user-given path and automatically installs them.

Once successfully built all the packages of the repo, we need to source the environment (terminal) in which we want to execute the system and run the nodes that compose the company's navigation system. Each node is meant to be executed in a different terminal, therefore in order to avoid typing the commands to launch the single nodes every time in a lot of different terminals, sourcing each one of them priorly, the entire architecture can be launched by means of the *screen* command: "screening" a certain file contained in the Alba Robot repository will automatically run a set of terminals following specific instructions. In this way, all our nodes will be executed at once.

3.2 Current ROS2 architecture on *SEDIA*

The next and last step needed to fully understand the *SEDIA* project, which is also the most important one, is to get into the architecture of ROS2 nodes running when the whole system is executed. As said, when we screen the *Alba Robot* launch file, one terminal is opened for each node, and all nodes run at the same time. If they are launched in the same `ROS_DOMAIN_ID` and in the same LAN (which is the case here because we suppose to execute them all inside the *Nvidia Jetson Xavier NX* board with the same `ROS_DOMAIN_ID`), the DDS implementation we use (in this case is *eProsima Fast DDS*) will recognize their presence and their relationships, hence they'll be able to communicate through topics, services and actions, implementing the *Alba Robot* navigation system.

The navigation system *Alba Robot* uses is based on Nav2, as we said, meaning that it is composed in its majority by nodes already defined by the **Nav2 project**, for generic navigation. They are however placed side by side with other **custom nodes** created by the company itself, intended to handle data from their specific components, namely the cameras and sensors we spoke about in Section 3.1.1. Some Nav2 nodes have been modified themselves for the same purpose as well, so that the overall system can interact with the custom Hardware beneath (Low Level) in the most optimized way.

3.2.1 `rqt_graph`

The best way of having an empiric overview of the ROS2 architecture running on *SEDIA* is through the usage of the `rqt_graph` tool.

RQt is a graphical user interface framework that implements various tools and interfaces in the form of **plugins**. The tools can still run in a traditional standalone method, but RQt makes it easier to manage all the various windows in a single screen layout.

Among the possible plugins, the `rqt_graph` tool provides a GUI for visualizing the **ROS2 computation graph**. Namely, the ROS2 graph represents all the ROS2 nodes that are running in a certain LAN and under a certain `ROS_DOMAIN_ID`, along with the topic they're communicating through. The `rqt_graph`, composed only by nodes and topics (no services are shown in it), is visualized in a graphical interface, so that the ROS2 architecture running in a certain LAN becomes visible and straightforward.

This plugin can get extremely useful in various situations: with it, you'll get a global overview of your system and that can really come in handy to take better decisions for the new future parts of your application. Also, when you have a bug somewhere due to communication between nodes, you will be able to easily spot the problem. A node may be not correctly connected to another, or there could be two nodes publishing on a given topic instead of just one as it was supposed to be, which is why you get some weird values on the subscriber side. All these problems can be easily spotted with the ROS2 system overview offered by the `rqt_graph` plugin.

We are not taking advantage of the `rqt_graph` for either of these two situations, but in our case, it becomes very useful to study the architecture of the *Alba Robot* navigation system, namely the ROS2 nodes it is composed of and their relationships.

Since the tool can turn to be a little bit messy when showing an architecture as complex as the one of a whole navigation system, and the output will likely be differently shown each time, the complete graph has been represented in a sorted and clearer way through the usage of the *Diagrams.net* tool. In Figure 3.6, the v1.4 release of the *Alba Robot* architecture is shown, with all the topics involved. Also, it was considered appropriate to show the actions which tie some nodes to each other (not normally represented by the `rqt_graph` tool as said), because they constitute a fundamental part of the navigation process.

3.2.2 *SEDIA* ROS2 architecture

Following, a brief description of each node and its role in the overall system is presented.

The central part in Figure 3.6, enclosed in a rectangular area for simplicity, represents the core of navigation, as it includes the Nav2 nodes also known as ***the***

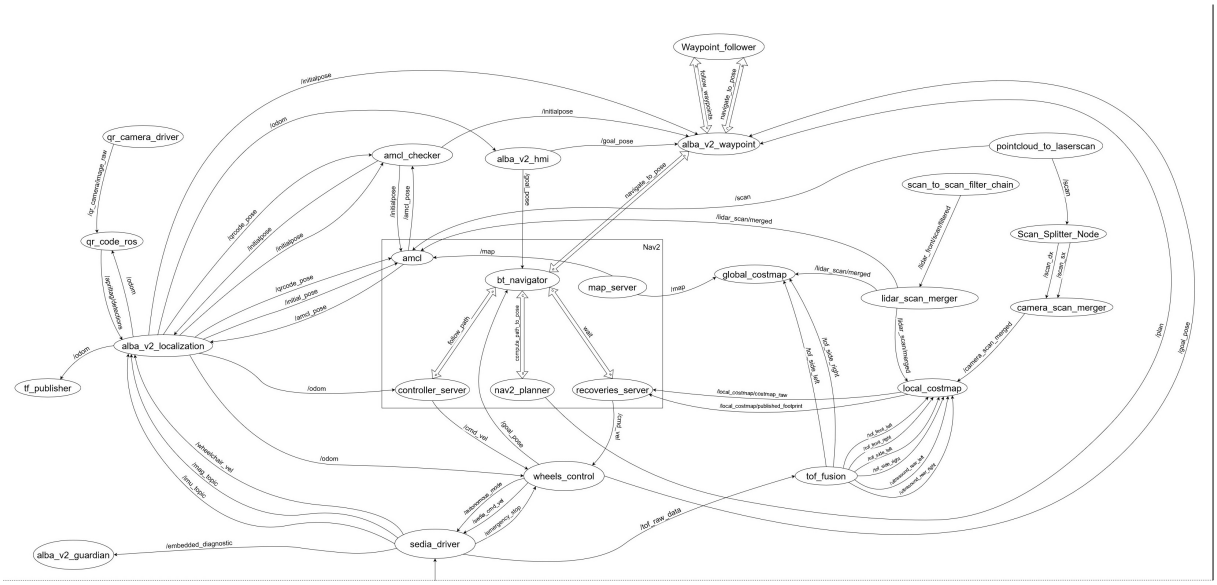

Figure 3.6: *SEDIA* High Level v1.4

Nav2 stack:

- `controller_server`: the Nav2 Controller Server. it is the module responsible for generating **velocity commands** for the robot, starting from the path computed by the Planner Server. Such commands, given to the `wheels_control` node, will result in the robot following the planned path, basing on the Local Costmap in this case, which is the layer of the map constantly updated with the detection of the various sensors. Hence, the path previously computed by the Planner can vary depending on possible detected obstacles.

- `nav2_planner`: it is the Nav2 Planner Server, in charge of **generating a feasible path** given start and end robot poses (position and orientation). In other words, the node takes the user-given info about the goal of the navigation, which is a pose of the vehicle (point in space to reach, with a certain orientation) and, knowing the current pose of the vehicle, computes a global path which is saved in the Global Costmap. It loads a map provided by the map server and generates the path in different user-defined situations.

- `recoveries_server`: its goal is to give **recovery behaviours** in situations in which the robot gets stuck as the normal planned navigation does not work, due to limited room for manoeuvre or the sudden presence of a too close obstacle, for example. It fuses three different types of behaviour implemented by means of ROS2 action servers:

  - **Spin** performs an in-place rotation by a given angle.
  - **Back Up** performs a linear translation by a given distance
  - **Stop** brings the robot to a stationary state. Actually, in the current implementation it is used only to wait a certain amount of time after the clearing of Costmaps.

- `bt_navigator`: it is a Behavior Tree-based implementation of navigation which is intended for flexibility in the navigation task and allows to easily specify complex robot behaviors. It receives a goal pose and navigates the robot to the specified destination, therefore it acts as an **interface** between the user-defined goal and the Controller Server, which physically executes the navigation task by computing the intended path. To do so, the `bt_navigator` module reads an XML specification of the BT (Behavior Tree) from a file and executes it.

- `map_server`: it **provides a map** to the interested nodes, having read it from a file, using topics or services interfaces. The map, in a navigation context, represents the base navigation environment in which the vehicle moves and, in the current implementation, it is a static map given to the Nav2 stack in the form of a *pgm* file, plus a *yaml* file for some metadata. The package is actually divided in three parts:

  1. The `map_server` is responsible for loading the map from a file, as mentioned, through command-line interface or by using service requests.
  2. The `map_saver` saves the map into a file, hence it is used in the mapping phase, previously discussed, to generate the map file from the info gathered by means of the `slam_toolbox`, for instance. Like the `map_server`, it has an ability to save the map from command-line or by calling a service.
  3. `map_io` is a map input-output library.

- `amcl`: it is a module implementing the AMCL (Adaptive Monte Carlo Localization) algorithm, which is a probabilistic localization module to **estimate the absolute pose** (position and orientation) of a robot in a given known map, using a particle filter approach. It is an alternative way of estimating the position w.r.t. the AprilTag detections (discussed below), but it has some limitations: the absolute position can be computed only if there is a previous (imprecise) estimation, hence it just provides adjustment for a previously estimated pose, while AprilTags can localize the robot even without knowing anything about its previous location.

- `global_costmap`: the Costmap2D is a 2D representation of the environment, consisting of several "layers" of data about it, built on top of a base map. The

latter is provided by the `map_server` (see above), while the layers are updated by the sensors' observations. This node **handles the Global Costmap**, hence it stores all data known by the robot from previous visits and generates the Global Costmap layer, which is used by the Planner Server to compute overall paths throughout the map.

- `local_costmap`: it is the other flavour of the 2D Costmap, in charge of storing everything that can be known through the sensors from the current position at every moment, mainly the data about the near obstacles detected. Consequently, this node is in charge of the **generation of the Local Costmap** layer, used by the Controller. This one, basing on the overall paths previously computed by the Planner Server, uses the data of the Local Costmap to try to follow the path.

The remaining nodes are not predefined by the Nav2 project, but customly written by *Alba Robot*, defined accordingly to the Hardware specifications of *SEDIA*, i.e. handling sensors and cameras on board among other things:

- `sedia_driver`: it represents the **interface node between the Low Level and the High Level**: it takes all data from the H7 (Controller, see Section 3.1.1) and distributes them to the rest of the nodes. This is the only node that communicates with the Low Level: in particular, it retrieves topic information on angle, position, *ToF*, ultrasound, battery status, wheelchair status and emergency stop through serial communication and then sends them on the ROS2 domain. It also sends the speed commands of the Planner to the Controller via serial.

- `wheels_control`: it is the node that **handles velocity commands** in two modes: *autonomous* and *manual*. Also, it handles **emergency stops**.

  - In *manual mode*, even if the node is in its *ACTIVE* lifecycle state, it can inhibit the autonomous driving, meaning that the velocity commands coming from the ROS2 system are ignored and the only commands actually sent to the Low Level Controller (H7) are the ones coming from human input. In other words, any user-given input will be followed priorly.
  - When not inhibited, it is instead in *autonomous mode*, and in this state it acts when velocity commands are received via the autonomous driving topic (***`speed_sub`***). In any case, it sends navigation commands to the wheelchair for the wheels to move, which pass through the `sedia_driver` node before actually reaching the Low Level.

- `tf_publisher`: together with `wheels_control`, it belongs to the `alba_v2_wheelchair` module: it takes care of **periodically send transforms** that represent rototranslations between different reference systems (*frames*). Most of the transforms are static and don't change over time because they represent systems which are integral to each other, like the ones between different components of the robot. The only other thing the node takes care of is the creation of the dynamic transform between the odom and `base_link` frames which, to avoid delays, is performed directly when the `odom` topic is received.

- `alba_v2_localization`: the `alba_v2_localization` module has the purpose of **estimating the position of the wheelchair** instant by instant in order to allow navigation. Localization can be expressed and notified to the system both in relative (related to the `odom` frame, hence the robot reference system) and absolute (related to the `map` frame, hence the map reference system) terms. The relative position evolves based on data from encoders and IMU, while the absolute position is instead corrected only by AprilTag detection or corrections from the `amcl` node.

  This node gets input data from various sources:

  - Apriltag detections: a detected apriltag with known absolute position (with respect to map) and relative position (with respect to robot/camera).
  - IMU: ROS2 message. The type is the one of the IMU but it contains odometry position information instead of linear accellerations.
  - Amcl pose estimate: an estimation of the pose computed by Adapative Monte Carlo Localizaiton node.
  - Initial position: pose provided via ROS2 topic. Currently this should be only provided by human input (e.g. via Rviz2).

  The provided outputs are the two fundamental transforms of NAV2, namely *"map to odom"* and *"odom to base link"*. To transmit the position in absolute coordinates to the rest of the system, the node periodically publishes a transformation in the topic called *tf* in broadcast, in particular the transform between frames `map` and `odom` as mentioned. The node also takes care of making the request for changing the map when necessary.

- `qr_camera_driver`: as previously discussed, the QR camera is the one in charge of **looking for AprilTags** spread in the navigation environment, placed as flat sheets on the floor, **to obtain the absolute position of the wheelchair**. The previous architecture had only the `qr_code_ros` node (see below), but this node was eventually added in order to obtain the QR camera image using its driver and publish it on a topic, for the `qr_code_ros` node to

use. Therefore, it acts as an interface between the camera driver, which looks for the AprilTag image, and the `qr_code_ros` node.

- `qr_code_ros`: it gets the image taken by the qr code camera from the `qr_camera_driver` node and localization data from the `alba_v2_localization`, detects the AprilTag and participates in **computing the precise 3D position and orientation of the vehicle** using the AprilTag library. It produces the transform between the camera frame and the AprilTag frame, and an array containing info on the AprilTag detection. All the info generated by this node are used by the localization module to obtain the absolute position and orientation in the map frame reference.

- `tof_fusion`: this node **fuses the raw data coming from the ToF sensors** (see Section 3.1.1) and creates a message of type *Laserscan* to be sent to the Local Costmap, in order to visualize possible obstacles in it. The complete flow of information starts from serial messages, coming from the sensors. They go to the Collector and from the Collector to the Controller (H7), and it sends them to the `sedia_driver` node. Then it provides them to the `tof_fusion` node, which generates its output for the Local Costmap. These newly added info will be used by the Controller Server to avoid such newly detected obstacles, as previously discussed.

- `pointcloud_to_laserscan`: it **gets 3D pointcloud data and translates them in 2D laser scan data**. This is useful to make 3D data appear as 2D detections and give them to 2D-based algorithms.

- `scan_splitter_node`: it **takes data from the stereo cameras' driver node** through the *scan* topic and **splits them into left and right scans**, for the following algorithms to work.

- `camera_scan_merger`: the package `alba_ros_scan_merger` easily and dynamically **merges multiple, single scanning plane, laser scans into a single one**. This is useful to work with multiple laser scanners and provide a unified input to some nodes (like `amcl`), which require a single scan. The output will result like it was generated from a single scanner, but it will contain all the information generated by the ones actually used, as if we used a single scanner with occlusions. Belonging to this package are the nodes `camera_scan_merger` and `lidar_scan_merger`. They share the same exact source code, and the only difference between them is their configuration: the former takes data from the camera and the latter from the lidar sensor.

- `lidar_scan_merger`: see `camera_scan_merger`.

- `Waypoint_follower`: this node **gets a set of poses** (position in space + orientation), called *waypoints*, and **navigates to all of them in order**. Custom behaviours can be set in order to make the robot perform actions when reaching certain waypoints, but at the moment the only operation it performs when reaching one is to continue to the following. If a waypoint fails, it can be configured to continue to the others or stop the navigation.

- `alba_v2_waypoint`: it is a module that aims at **modifying the planned route so that it passes through certain coordinates in the map**, namely waypoints, which in this case are points of interest close to the planned route. For instance, a waypoint can be in correspondence of an AprilTag in order to force the robot to pass over it and help the system correct the localization. The main reason of the existence of this node is to **adjust the planned route**: the planned trajectory indeed adopts the strategy of the minimum path and, in environments with large spaces, this can lead to the wheelchair passing close to walls or far from the AprilTags spread in the map. With this mechanism, when the planned route passes close to one of the waypoints, the path is adjusted to pass through it. The node achieves its goals by taking a list of waypoints as its parameters and sending them to the `Waypoint_follower` module, so that it makes the vehicle pass through one of these waypoints as soon as the planned route is at a certain distance from one of them. In this way, the path is split into more sub-paths and a route is calculated as soon as a waypoint is reached.

- `alba_v2_hmi`: it is the **human-to-machine interface** (Human-Machine Interface) node, the only one capable of communicating with the HMI application. It retrieves information about, for instance, the estimated pose of the wheelchair or the desired navigation goal through serial communication and sends them on ROS2 topics. It also sends info on the pose and wheelchair status to the HMI application via serial.

Although being aside in Figure 3.6, a very important role in the ROS2 architecture of *SEDIA* is covered by the `alba_v2_guardian` node, which will also be fundamental in our final Cloud solution, hence it deserves some considerations in a section of its own.

3.2.3 The `alba_v2_guardian` node

This node performs two main tasks in the overall ROS2 navigation solution of SEDIA:

- It takes care of the nodes' lifecycle management, i.e. it configures, activates and deactivates all the other nodes. Also, it monitors and shows the lifecycle status

of the nodes with statistical data. Depending on the lifecycle nodes' status, it can perform a system "stop" and inhibit the driver node (`wheels_control` in our case).

- It handles the nodes' diagnostic (purely logging at the moment), listening to the messages sent by them.

In other words, the Guardian is the orchestrator of the whole ROS2 architecture: when screening the Alba Robot script for the system launch, as previously discussed, each ROS2 node is executed in a terminal of its own, but its lifecycle status stops in Unconfigured state, meaning that the node is not yet configured and active, namely not ready to execute its tasks. The only node whose lifecycle is automatically handled (hardcoded) is the Guardian who, let's say, configures and activates itself.

Once active, the Guardian will start handling the lifecycle of all the other nodes, triggering their state changes (called transitions). The names of all the nodes in the ROS2 domain and their metadata must be properly stored in a configuration file, to be read by the Guardian, otherwise it won't know which nodes are present in the system, hence whose lifecycle it has to manage.

Following, a summarized description of the Guardian working algorithm is presented, which does not describe each single detail of the actions performed by the node, but gives a pretty accurate idea of the main functioning steps, considering the node as it was found at the time of this thesis:

Chapter 4

Proposed solution for the Cloud system

The very goal of this thesis is to find a solution to make an autonomous driving navigation system, such as the one running in *SEDIA*, work in the **Cloud**. In order to fully comprehend what the Cloud solution proposed here stands for, first we need to have clarity on what goes by the name of **Cloud Computing**, and which are the main **Cloud paradigms** used nowadays. Then, we can decide which of the presented paradigms is the best suited for our Cloud solution and we can proceed describing it in detail.

4.1 Overview on Cloud Computing services and paradigms

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Definition from the National Institute of Standards and Technology (NIST).

Simply put, Cloud Computing is the **delivery of computing services** — including servers, storage, databases, networking, software, analytics, and intelligence — **over the Internet** ("the Cloud") to offer faster innovation, flexible resources, and economies of scale. You typically pay only for cloud services you use, helping you lower your operating costs, run your infrastructure more efficiently, and scale as your business needs change. When we generally talk about Cloud Computing, we refer indeed to servers, hence very powerful computers located in a specific

geographical area.

Cloud Computing started in 2006 with *AWS* (Amazon Web Services), mainly to generate new incomes from unused resources, namely a large number of servers Amazon had at its disposal. At first, servers have been installed across the company, but their number kept increasing to such an extent that it became necessary to move them to **Datacenters**, in order to preserve data and make management easier. At that point, companies discovered that there were too many servers around and most of them were idle, not used to their full potential. This was mainly caused by the "one application per server" rule, according to which a single server was bound to one single application, as it was tied to one specific Operating System and computational environment (e.g. shared libraries and patches). This led to a huge amount of servers being massively underutilized, and consuming a lot of electrical power, in which a lot of CPUs were unutilized.

This is the reason why Cloud Computing started making a large use of **virtualization**, as VMs (Virtual Machines) enable a much more flexible use of the resources: with virtualization, the "one application per server" rule was overcome, as many environments could coexist in the same physical machine, each one with its installed libraries and patches, thank to the leverage of VMs. They logically separate one physical machine in many virtual ones which are strongly isolated to each other, as only a part of the physical Hardware is assigned to a single VM: in other words, a VM is not aware of the entire Hardware of the server, but only of the portion assigned to it by the *Hypervisor* (the enabling technology for virtualization). In a nutshell, Computing Virtualization is a flexible way to share hardware resources between different (un-modified) Operating Systems, so that critical applications can run in different and easily isolated OS and different services could run on the same hosts with an improved degree of isolation.

There are in fact limitations of the VMs technology, mainly an additional overhead in running the same application, as each application has its own OS running, hence more requirements in terms of disks, memory and CPU. This also leads to a very long booting time for each VM we want to run. These limitations will be overcome through the usage of Containerization, which we'll talk about in Section 6.1.

Cloud computing can be separated into four general service delivery categories:

- **Hardware as a Service** *(HaaS)*: The operator provides physical Hardware, space (often strongly protected) and facilities (buildings, power, conditioning, Internet connectivity). A company who buys a *HaaS* service physically rents the Hardware, not very differently from having servers at home. This is often used in very specific cases, e.g. in case the company has strong security requirements (e.g., necessity to control the physical access to the servers).

- **Infrastructure as a Service** *(IaaS)*: In this case, the company rents the infrastructure (network, computing, storage) from the provider, in which it can create virtual machines, attach to virtual disks, and create virtual networks, install their own operating system and traditional (legacy) applications. Users have an allocated storage capacity and can start, stop, access and configure the VM and storage as desired. *IaaS* providers can offer almost unlimited instances of servers to customers and make cost-effective use of the hosting Hardware.

- **Platform as a Service** *(PaaS)*: In the *PaaS* model, Cloud Providers host development tools on their infrastructures and users access these tools over the internet using *API*s, web portals or gateway software. The platform takes care of deploying the service on the actual servers (or on many servers at the same time), hiding "details" such as operating system, network connectivity, automatic scaling etc. In other words, the user develops and installs its apps on the provided infrastructure without taking care of the hardware, security model and OS.

- **Software as a Service** *(SaaS)*: Complete applications are directly provided to users, who have access to them without writing a single line of code, which alleviates the burden of software maintenance and support. Users can access *SaaS* applications and services from any location using a computer or mobile device that has internet access.

As previously discussed, Cloud Service Providers (CSPs) have many servers spread all over the world, which can be used by companies to build their own businesses, without having to take care of any physical issue coming with the management of such servers (buying the Hardware and renting or buying the physical location for the Datacenters to be built, paying for electricity, maintenance, security etc..). It is becoming increasingly common for the companies to choose not to handle all these problems by themselves: they rely instead on Cloud providers to rent some computational power to be used for their business. Generally, Cloud providers make available a set of servers located in different geographical areas spread all over the world, which companies can choose from depending on their needs, e.g. the location of their intended users in the world. This is known as "**Public Cloud**", as anyone can rent some computational power, only paying for what is being used. In other words, in the Public Cloud model, a third-party CSP delivers the cloud services over the internet and such services are sold on demand, typically by the minute or hour, though long-term commitments are available for many services.

In the past, before the Public Cloud paradigm spread, companies had to invest their money in their own servers' infrastructure: it was generally located in the

company's facilities themselves and they had to fully manage it, dealing with all the issues related to the Hardware maintenance. This paradigm is known as "**Private Cloud**" or "**Cloud on Premise**". Of course, there are advantages in choosing this paradigm, i.e. having full control over the Cloud infrastructure: if the companies' data are managed by a Public Cloud provider, there is no guarantee that such data will not be leaked, as whoever has the hold on the physical infrastructure can easily have access to the information stored in it. With Cloud on Premise, the company can rest assure that all its data are under its control (given that good security is implemented). For this and other reasons, not all companies rely on Public Cloud for their businesses: they prefer to handle their data on their own. This paradigm is very similar to buying a *HaaS* Cloud service, as it implies physical access over the Hardware infrastructure. This model offers the versatility and convenience of the Cloud, while preserving the management, control and security common to local Datacenters.

Another paradigm, which fuses the previous two, is the "**Hybrid Cloud**" paradigm, a combination of Public Cloud services and an on Premises Private Cloud, with orchestration and automation between the two. Companies can run mission-critical workloads or sensitive applications on the Private Cloud and use the Public Cloud to handle workload bursts or spikes in demand. The goal of a Hybrid Cloud is to create a unified, automated, scalable environment that takes advantage of all that a Public Cloud infrastructure can provide, while still maintaining control over mission-critical data.

There is a fourth paradigm, which is spreading more and more, called "**Edge Computing**". Edge Computing is a distributed cloud architecture in which client data is processed at the periphery of the network, as close to the originating source as possible. The traditional computing paradigm (Public Cloud) built on a centralized Datacenter isn't well suited to moving endlessly growing rivers of real-world data. Bandwidth limitations, latency issues and unpredictable network disruptions can all conspire to impair such efforts. Hence, businesses are responding to these data challenges through the use of Edge Computing architecture: in simplest terms, Edge Computing moves some portion of storage and compute resources out of the central Datacenter and closer to the source of the data itself. Rather than transmitting raw data to a central Datacenter for processing and analysis, that work is instead performed where the data is actually generated. Only the result of that computing work performed at the edge, such as real-time business insights, equipment maintenance predictions or other actionable answers, is sent back to the main Datacenter for review and other human interactions.

A fifth and last paradigm can be named, known as "**Fog Computing**": in some cases, the edge deployment might simply be too resource-limited, or physically scattered or distributed, to be practical for real-world data. On the other hand, a Public Cloud Datacenter, although being very powerful, might be too far away to be

practical either, in this case in terms of latency constraints. A middle layer between Edge Computing and Pubic Cloud is represented by Fog Computing, which typically takes a step back and puts compute and storage resources *within* the data, but not necessarily *at* the data. With this paradigm, huge amounts of data, e.g. generated by sensors and IoT infrastructures, can be processed: latency constraints are still satisfied, through the usage of Datacenters with enough computational power (more than Edge Computing) to handle the data, being not too far away from their sources.

Now that we have more clarity about what we refer to as Cloud Computing, we need to choose one of the discussed paradigms for our autonomous driving Cloud solution. The choice must be guided by the needs of the application, exactly as an application intended for users in a certain part of the world, with loose latency constraints, will choose a Public Cloud service using servers in that specific location, for instance. To understand the Cloud Computing service chosen in this thesis, we need to have a deep introspection into the nature of the proposed Cloud solution: once we describe and that, the choice will be straightforward.

4.2 The Cloud solution

As previously stated, the very aim of this thesis is to propose a Cloud solution for an existing autonomous driving system. This solution has two main goals:

- **Shift the intelligence of the navigation architecture** (High Level), currently running locally in the wheelchair (in the *Nvidia Jetson Xavier NX* as discussed in Section 3.1.1), **to a Cloud server**, partly or completely.
- **Create a switching mechanism between local and remote instances** of the ROS2 nodes running in the system to decide, basing on the network connection status, whether to use the Cloud system or to switch to the local one. In other words, this solution implies that the navigation architecture will not only be shifted to the Cloud server, but also duplicated locally for backup, in case the network does not respond.

These two goals are not isolated of course, as **the second one can be seen as a step forward from the first one**. Clearly there will be several problems to solve in order to reach both goals, and we'll discuss them in the following sections. Section 4.3 will be dedicated to the first problem, while Section 4.4 will face the second. In the current section, we just make some general considerations about the discussed problems in order to have a better view over them and be ready to overcome the obstacles they present.

First, considering the first problem: can the entire architecture be moved to the Cloud, or do we need to move just part of it? In the latter case, which nodes are suited to be moved?

The first question can be answered right away, as there is not that much of a choice. At first sight, the final goal might seem to be having the entire ROS2+Nav2 navigation system in the Cloud server, implementing all the navigation intelligence from remote, but there is one reason that makes this **impossible**. We have seen in Figure 3.6 (Section 3.1.2) that a lot of the ROS2 nodes which compose the *Alba Robot* navigation architecture are strictly tied to the Hardware of *SEDIA*: indeed, those nodes communicate with the Hardware components of the wheelchair through UART or serial, making it impossible to move them to other locations such as a Cloud server: they need to be left where they currently are, otherwise the communication with the physical Hardware cannot be achieved. Some examples can be found in the `qr_camera_driver` node, which talks to the QR camera driver in order to take the information generated by it, or `sedia_driver`, which is the clearest example of a ROS2 node strictly bond to the Hardware, as it is the one that makes the High Level and the Low Level communicate, through UART. How can a ROS2 node, which puts in communication a ROS2 Software architecture and a physical (Hardware) one, be moved to a different location than the one of the physical Hardware?

Considering these two nodes as example, it is clear that we cannot move the entire ROS2+Nav2 architecture to a cloud server, but only a part of it.

The second question is instead not of simple address, and this thesis will not answer it thoroughly: why? Despite it is very easy to spot which nodes are not suited to be moved to the cloud, this does not mean that all the remaining nodes are. They can all potentially be shifted, of course, because they're not tied to the Hardware, but the process of moving them is not straightforward, and that is because of the second goal of this thesis, namely the switching solution.

Indeed, even if we overcame all the technical problems encountered when trying to execute some nodes in the cloud server and make them talk with the local nodes, once we add the switching system and duplicate the nodes, new problems will arise, related to the nature of each single node and the way it functions. Each node of the ROS2+Nav2 architecture of *Alba Robot* has its own working algorithm, of course, and having it duplicated both locally and in the cloud is not straightforward: we will explore this in Section 4.4.1, but we can anticipate that implementing a local/cloud switching solution requires a lot more design effort than this thesis aims to do. Therefore, we will not explore which nodes are suited for being shifted to be executed in the cloud server and duplicated locally at the same time, because that would mean additional effort which has not very much to do with the switching

solution itself: it would just comprehend a series of additional mechanisms to make the overall system work, built on top of the solution proposed here.

We have generally presented the overall aim of this work, which inevitably arises a series of problems: first of all, which one of the existing cloud services must we use? How is the physical cloud setup going to be?

This first question has a simple answer: **on-premise**. As said, we are developing a solution for a ROS2+Nav2 autonomous navigation system, hence it is pretty easy to understand that we are tied to one specific constraint, and that constraint is latency. Our solution will eventually comprehend some ROS2 nodes being executed locally and some in the cloud, as previously discussed, but we want the nodes to be able to communicate between each other very fast, as if they were located in the same spot (all local nodes for instance, as they are by the time of this thesis).
Of course, a little bit of latency is inevitable (we'll discuss how this thesis addresses it in Chapter 5), but we still need to have the minimum added latency we can for the system to work. The natural choice, given this strict requirement, results to be the on-premise cloud solution, because we need the server to be located as close as possible to the vehicle, in order to have as less latency as possible. We do not need a complex cloud infrastructure either at this stage: one simple server, even not too powerful, is more than enough to handle some ROS2 nodes, which do not generate a huge amount of data, and this answers the second question. The important thing, as already said, is that those data reach the wheelchair nearly instantly, in order not to have malfunctioning of the autonomous driving system, such as sensors' data not processed in time, resulting in an obstacle not avoided.

Hence, the resulting scenario will, at a first glance, be composed by the wheelchair navigating in a certain environment, with some ROS2 nodes (at least the ones tied to the Hardware of the vehicle) running in the *Nvidia Jetson Xavier NX*, and an on-premise cloud server, located as near as possible to the navigation environment, in which all the other nodes composing the Alba Robot navigation system are executed. Eventually, these nodes will be duplicated local-side and a switching mechanism will be added to the system, in order to use the cloud when the network conditions make it possible, and switch locally otherwise.

Once stated which cloud service is best suited for our solution, we can dive into its technical features and explain how we're going to make the *Alba Robot* system for autonomous driving work in the cloud. According to the two conditions we want achieve, described above, we need to solve two main problems:

1. **How to enable communication between ROS2 nodes located in remote LANs?**

2. **How to implement the switching between local and Cloud nodes?**

In other words, the first and maybe most important step to do is to **enable ROS2 remote communication between different LANs**: once done that, supposing that the ROS2 nodes of *SEDIA* are able to communicate between the local setup and the Cloud server, we need to figure out how to design the solution to **implement the switching between local and Cloud replicas of such nodes**.

These two problems will be addressed in the next two sections, in this order (the natural one, considering that the switching solution will be built on top of the assurance that local and remote nodes can communicate as if they were in the same network).

4.3 Problem1: ROS2 remote communication

The first problem to solve is to make sure that the communication between remote nodes is enabled. As explored in Chapter 2 and in Section 2.3.2 specifically, ROS2 nodes can automatically talk to each other if they are located in the same LAN (and in the same `ROS_DOMAIN_ID`, of course), as by means of the "SIMPLE" Discovery mechanism the DDS middleware acknowledges the presence of all the nodes existing in the LAN, using multicast Discovery traffic. Those nodes will be able to talk to each other through topics, as DDS makes sure that all subscribers of a certain topic will receive all the information published in that topic. ROS2 uses this mechanism **automatically** and no configuration of the middleware has to be done.

It is not the same, though, if two ROS2 nodes are located in different LANs: remote nodes do not communicate to each other automatically, even if they are launched in the same `ROS_DOMAIN_ID`, because the multicast Discovery traffic mentioned above is limited to the LAN's boundaries. No external node will be discovered automatically by DDS, which will consequently not forward the information exchanged by the different nodes outside of the LAN.

Surely some effort must be done in order to make a ROS2 system work in the Cloud, as this means having ROS2 local nodes communicating with other nodes located in a Cloud server, which will most likely be in another LAN. If the on-premise server is in the same LAN, this whole section is to be skipped, as no effort must be done in order to have the Cloud nodes talk to the local ones, for the reasons presented. In most cases, though, the situation is the former, bringing the need for some consideration on this matter.

Three solutions have been explored, each one with its pros and cons, but only one of them has been considered feasible of implementing a final product, for reasons that will be properly discussed in the next paragraphs.

4.3.1 Initial Peers List

The first solution is based on the fact that the middleware which ROS2 is based on, namely DDS, has a lot more potential than the one exploited by default in ROS2. DDS is a very powerful creation thought for **distributed applications**, such as IoT ones, in which a lot of devices such as sensors and drones are scattered in different parts of the world, talking to each other and creating a huge distributed DDS network. Like other implementations, eProsima Fast DDS has as well the possibility of unlocking more potential than the one normally used by ROS2: this allows us to take advantage of the middleware itself to reach our goal, which is achieving remote ROS2 communication.

By potential, we mean that Fast DDS can be easily **configured** to act differently than the way ROS2 users usually think, so that some different mechanisms can be leveraged. In particular, Fast DDS, as stated in the official documentation, can be configured by means of **XML configuration files**. They are the way we can bend the DDS middleware's behaviour for our own goals, defining the so-called *profiles*. The main reason we usually want to do that is to unlock different Quality of Service *(QoS)* behaviours than the default ones, in order to control how messages are exchanged in a DDS Domain.

In our case, though, we do not want to configure some *QoS* settings, but slightly modify the SIMPLE Discovery protocol behaviour, as for this first solution we're indeed taking advantage of the so-called **Initial Peers List**. According to the RTPS standard, each *RTPSParticipant* (in our case a ROS2 node) must listen for incoming Participant Discovery Protocol (PDP) discovery metatraffic in two different ports, one linked with a multicast address, and another one linked to a unicast address. Fast DDS allows for the configuration of an Initial Peers List which contains one or more such IP-port address pairs corresponding to remote *DomainParticipants* PDP discovery listening resources, so that the local *DomainParticipant* will not only send its PDP traffic to the default multicast address-port specified by its domain, but also to all the IP-port address pairs specified in the Initial Peers List.

Simply put, Fast DDS allows a node to specify this list of Initial Peers, so that such node will send discovery traffic not only to all the other nodes in the LAN, as the SIMPLE Discovery protocol already does on its own, but also to all the Peers declared in that list: they will be contacted one by one in unicast.

A *DomainParticipant*'s Initial Peers List contains the list of IP-port address pairs

belonging to all the other *DomainParticipants* which it will send its Discovery traffic to. It is a list of addresses that a *DomainParticipant* will use in the unicast Discovery mechanism, together or as an alternative to multicast Discovery. Therefore, this approach also applies to those scenarios in which multicast functionality is not available.

The following constitutes an example XML configuration file which sets an Initial Peers list with one peer on host 192.168.10.13 with *DomainParticipant* ID 1 in domain 0:

```
<!--
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
-->
    <participant profile_name="initial_peers_example_profile" is_default_profile="true">
        <rtps>
            <builtin>
                <initialPeersList>
                    <locator>
                        <udpv4>
                            <address>192.168.10.13</address>
                            <port>7412</port>
                        </udpv4>
                    </locator>
                </initialPeersList>
            </builtin>
        </rtps>
    </participant>
```

Figure 4.1: XML configuration file for an Initial Peers List

With this mechanism, remote ROS2 nodes can be declared to be part of the working scenario, following some simple steps.

For the local nodes:

- We declare a set of **<locator>** fields, one for each node running cloud-side.
- We specify, in every **<address>** field, the IP of the cloud server in which all the other nodes, which we wish to discover, are being executed.
- For each of them, a different value must be set in the **<port>** field, which must correspond to the port which the node is listening at. Ports are not randomly chosen, but each node listens at a specific one, which can be computed with the following equation:

$$port = 7400 + 250 * domainID + 10 + 2 * participantID$$

- We save these tags in an XML file, which will become the configuration file of every node we launch. In order to make each node accept the XML profile, first we must export an environment variable, which is `RMW_FASTRTPS_USE_QOS_FROM_XML`. By default, its value is set to 0, and by changing it to 1 we force the node we're launching to look for an XML configuration file.
- Now that we have a proper XML configuration file, we need to force the node to look for it, and we have two options:
  1. Declare the `FASTRTPS_DEFAULT_PROFILES_FILE` environmental variable, with the path to the location of the XML file as its value. This will make the node look for the file in the specified path.
  2. Name this file *`DEFAULT_FASTRTPS_PROFILES.xml`* and save it in the working directory in which we execute the node, so that it will be automatically recognized as an XML configuration file.

For the Cloud nodes, it is mirrored:

- We declare a set of **<locator>** fields, one for each node running local-side.
- We specify, in every **<address>** field, the IP of the device (in our case the *Nvidia Jetson Xavier NX*) in which all the other nodes, which we wish to discover, are being executed.
- For each of them, a different value must be set in the **<port>** field, which must correspond to the port which the node is listening at. Ports are not randomly chosen, but each node listens at a specific one, found with the following equation:

$$7400 + 250 * domainID + 10 + 2 * participantID$$

- We save these tags in an XML file, which will become the configuration file of every node we launch. In order to make each node accept the XML profile, first we must export an environment variable, which is `RMW_FASTRTPS_USE_QOS_FROM_XML`. By default, its value is set to 0, and by changing it to 1 we force the node we're launching to look for an XML configuration file.
- Now that we have a proper XML configuration file, we need to force the node to look for it, and we have two options:
  1. Declare the `FASTRTPS_DEFAULT_PROFILES_FILE` environmental variable, with the path to the location of the XML file as its value. This will make the node look for the file in the specified path.

2. Name this file `DEFAULT_FASTRTPS_PROFILES.xml` and save it in the working directory in which we execute the node, so that it will be automatically recognizes as XML configuration file.

Note: There is also the possibility of not defining the Initial Peer listening port. In this case, the Discovery information would be sent to every port ranging from *participantID=0* to a maximum number set by the Fast DDS specifications (see TransportDescriptorInterface).

Of course, we must make sure that the `RMW_IMPLEMENTATION` environment variable is set equal to `rmw_fastrtps_cpp`, otherwise we're not using Fast DDS in the first place.

Note: An important thing to highlight is that the RTPS protocol under DDS is not aware of ROS2 types, so the *MicroRTPS* agent registers a default generic type in FastRTPS. This generic type has a maximum size of 1028 and by default FastRTPS entities uses the memory policy `PREALLOCATED_MEMORY_MODE`. We must solve this by explicitly defining a **<historyMemoryPolicy>** tag inside the **<publisher>** and/or **<subscriber>** tags, setting the value `DYNAMIC` or `PREALLOCATED_WITH_REALLOC`, so no errors on the payload size will be generated by RTPS:

```
<publisher profile_name="/turtle1/cmd_vel" is_default_profile="true">
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
</publisher>
```

Pros

This solution has the advantage of being very simple, as it does not imply any modification on the ROS2 architecture of the navigation system: only the generation of a proper XML configuration file for DDS is required, which constitutes a task of minimal effort for the developer. Once done and exported the right environment variables, every node will be executed normally and will reach the remote peers on its own.

Cons

Despite the discussed clear advantage, this solution has not been considered feasible for an actual deployment.

The main problem resides in ROS2 itself and in simple words is **mutual reachability**: ROS2 assumes that every machine in which its demon is running

can reach all the others where the other ROS2 nodes are being executed, which they have to talk to, and vice versa. Indeed, it is not a client-server paradigm, but a publish-subscribe one: in any mechanism based on a request and a response, a request would be delivered and the response entity, although not being able to reach the requester independently, will use the information on its address contained in the request to reply, which does not imply the need for mutual reachability. In our case, this is not true, as any node must be able to contact all the other ones independently: consequently, the machine at one side, in our case the *Nvidia Jetson Xavier NX*, must be able to reach (physically, by means of `ping`), the machine at the other side, in our case the cloud server.

In a real case scenario, mutual reachability **cannot be easily assumed**, as it requires additional network configurations than the ones mostly found: a server is in fact usually reachable from clients, but clients (*SEDIA* in this case) are not always reachable by the server, which means that the ROS2 nodes in the server publishing on topics cannot independently contact the ones running in *SEDIA*.

There is another disadvantage: this method consists in each node sending Discovery metatraffic to each other node in unicast (as well as the normal multicast one), to the IP address and the port specified in the XML configuration file. Assuming that the two machines are mutually reachable in terms of network, what if either of them is **behind a NAT**?

In the majority of scenarios, a Cloud server is publicly reachable, with a public IP address, but a device such as our *Nvidia Jetson Xavier NX* usually lives inside a private network, the one of the facility in which *SEDIA* is navigating. This means that the network module used for the communication with the cloud server has a private IP address, and a NAT is used at the edge of the network to put it in communication with the outside. In this case, while SEDIA is able to contact the server directly through its public address, the same does not apply to the opposite direction, as **the server is not able to independently contact *SEDIA***.

The way in which this is usually solved consists in the creation of some port forwarding rules is the NAT, implementing a so-called "NAT traversal," which allows in our case the server to contact *SEDIA* regardless of the fact that it has been contacted earlier by it. But right here resides the second problem: we know that each node listens to unicast Discovery traffic at a certain port, which is the one specified in the XML file, but this means that **a different port forwarding rule must be set for each different node**, resulting in a very long list of rules. If the amount of nodes is too large, the router could also not be able to support such huge list.

Also, even if we suppose to actually create that set of rules in the edge device, there is no guarantee that the *participantID*s of the remote nodes are not going to change, or a different *domainID* is not decided by however is launching the *SEDIA*

application. Changing either of the two parameters will change the listening ports of the launched nodes, which calls for a manual modification of the XML configuration file, in order to match the newly allocated ports. It is not the most comfortable and plug-and-play solution to apply for a proper ROS2 remote communication.

In conclusion, even if very simple and straightforward, the solution of the Initial Peers configuration, through an XML configuration file, results to be not practical, due to the fact that the location of the remote peers is manually set, and this brings to a lot of problems at the end of the day. For these reasons, other solutions have been explored.

4.3.2 VPN

The second solution this thesis has explored is the establishment of a **VPN between the local and the Cloud side**. First of all, let's have a look in what the VPN technology is about (not exploring it in details, but for what concerns this thesis), then we'll describe the second proposed solution for Problem 1.

A definition of a VPN, which stands for "Virtual Private Network", could be the following: *connectivity realized on a shared infrastructure such that policies can be enforced as in a private network.*

The shared infrastructure can be a private or a public network, usually the internet itself. The mentioned policies could be security, QoS, reliability, addressing, etc...

In other words, a VPN is an **encrypted connection over the Internet** from a device to a network (although other types of VPN deployment can be created, as we'll see). The encryption helps ensure that sensitive data is safely transmitted: it prevents unauthorized people from eavesdropping on the traffic and allows the user to conduct work remotely. The VPN technology is widely used in corporate environments.

For the aims of our solution, we're not focusing our attention on the security side, as our goal is very different. Security can surely be an issue in a final deployment, which will definitely bring to considering a secure communication channel between the two peers, but let's keep in mind that, at this stage, we're only trying to overcome the technical problem of communication between ROS2 nodes in different networks: that's why no security is considered.

A VPN extends a corporate network through connections made over the Internet. For instance, an employee can work outside the office and still (securely) connect to the corporate network. In other words, through the usage of a VPN, a device

can remotely connect to a private network, exactly like it was located inside such network: that device will be part of the private LAN, with a proper private IP address, without being physically connected to it. The private traffic will pass through the internet and will be delivered to the mentioned LAN.

The key elements, useful to understand a VPN are:

- **Tunnel**: it is a (secure) encapsulation of the VPN traffic while in transit on the shared network.
- **VPN Gateway**: Termination device on the corporate network, tunnel endpoint.

Indeed, the enabling technology for VPNs is **tunneling**: with it, a packet (or frame) between private sites is carried through a public network within a packet handled by public nodes. The original frame or packet is meant to live inside the private network of the VPN: then it gets encapsulated in another frame, depending on the VPN flavour used. In a Layer 3 VPN, for instance, the packet/frame is carried through an IP network within an IP packet, meaning that the original packet (Header + Payload) is encapsulated in an IP header, which the public network will use to deliver it to its destination. In this case, it can be an IP packet within an IP packet (IP-in-IP) or a layer 2 frame, within an IP packet. There are other flavours of VPN, but the basic concept is that the original packet or frame is transmitted over the public network, which in some way knows how to reach the private corporate network: inside that network, the packet will act as if it was generated by a device physically connected to it.

There are actually three dimensions to consider when describing all the possible VPN solutions, and those are:

1. **Deployment model**: how the public network transports the encapsulated packet or frame.
2. **Provisioning model**: how the VPN technology is enabled, in terms of who owns the specific devices for VPNs.
3. **Protocol Layer**: as previously discussed, a VPN can be implemented in any layer of the *ISO/OSI* stack between 2 and 4, and this determines how the packet/frame will be encapsulated.

Regarding point 1, there are two deployment models:

- The **Overlay Model**, in which the public network does not participate in realizing the VPN: it does not know where VPN destinations are and just

implements connectivity among VPN gateways. Routing is performed by the gateways themselves, which are the only ones knowing how to reach their peers. They properly encapsulate the packet/frame with the right destination addresses and the network will forward them as if they were normal packets, meant for the public network.

- The **Peer Model**, in which each VPN gateway interacts with a public router (its peer), exchanging actual routing information. The service provider network disseminates such information among the peers in order to make them aware of the presence of a VPN, hence the public network participates and routes traffic between gateways of the same VPN.

About point 2, there are two provisioning models:

- **Customer Provisioned**: the customer implements the VPN solution, hence it owns, configures, manages devices implementing the VPN functionalities, called the Customer Equipment (CE). Network provider is not aware that the traffic generated by customer is part of a VPN, as all its features are implemented in the CE, which in this case terminates the tunnels.
- **Provider Provisioned**: the provider implements the VPN solution, hence it owns, configures, manages devices implementing VPN functionalities, called the Provider Equipment (PE). The state of the VPN is maintained by the PE, which in this case terminates the tunnels. Traffic belonging to different VPNs is separated by the provider devices, while CEs may behave as if they were connected to a private network, not knowing anything about the presence of a VPN in the first place.

There is actually another key distinction to determine which type of VPN is being used, which completes the picture. This distinction is about between which devices the tunnel is set, as the instauration of the tunnel between VPN Gateways is not the only choice:

- **Site-to-site** *(s2s)* VPN: In this case, the tunnel is created **between two VPN Gateways**, which are its endpoints. A site-to-site VPN is often used to connect a corporate office to its respective branch offices over the Internet, as it puts in communication entire private Layer 2 networks: it is used when distance makes it impractical to have direct network connections between these offices. Dedicated equipment, such as the mentioned VPN Gateway edge devices, is used to establish and maintain a connection, which can be thought of as a `network-to-network` connection.

- **End-to-end** *(e2e)* VPN: the tunnel endpoints are not VPN Gateways (which are edge devices), but the end systems themselves. In this case, a VPN connection is set **between two devices**, and not two networks, as the tunnel endpoints are the devices themselves. They may be laptops, tablets, or smartphones and this type of VPN can be thought of as a `device-to-device` connection.

- **Remote access** VPN: a hybrid solution, which consists in the tunnel being set **between a VPN Gateway** (edge device) **and an end system**. This is usually used to (securely) connect an outside device to the corporate office, and can be seen as a `device-to-network` connection.

How can a VPN be a possible solution for a remote ROS2 communication? The answer is quite simple and resides in the SIMPLE DDS Discovery process: we know that DDS automatically discovers ROS2 nodes if they're being executed in devices belonging to the same LAN. It is straightforward to understand that the same goes for a Virtual LAN, hence **all the nodes belonging to the virtual private network created by means of a VPN will automatically discover each other**.

A VPN consists indeed in a Virtual Private Network which connects remote networks (or devices) as if they were in the same LAN: it is just a mechanism built on top of the normal functioning of networks which allows to consider remote and distinct networks as a unique private LAN. The same goes for devices, in the case of a Remote Access or an *e2e* VPN, meaning that those devices will act as if they belonged to the same Local Area Network, like being physically connected to the network device (access point + switch) of that LAN.

In light of all this, the second solution for ROS2 remote communication eventually consists in **linking the cloud server and *SEDIA*** (precisely the *Nvidia Jetson Xavier NX*) **together with a VPN**. Being a simple scenario, in which we have only those two devices with ROS2 nodes running, the preferred solution is an *e2e* (end-to-end) VPN between such devices.

As a matter of actual implementation, there are simple open-source solutions to create a VPN with nearly no effort, but we're not exploring it in this section: our aim here is to provide a general solution which makes the ROS2 nodes running at the two sides talk to each other.

Pros

This second solution definitely seems comely, due to the total absence of effort in the developer side. Indeed, differently than the previous one, no configuration of the DDS middleware is needed, and no change in the architecture either: only an

additional layer, constituted by the VPN establishment, is necessary, which has nothing to do with the nature of the application itself. Hence, the advantage of this solution is **simplicity**, assuming that we have a simple way of setting up a VPN connection between the server and *SEDIA*, regardless of the applications running on them.

Cons

Despite the clear advantage of simplicity, a VPN has not been considered a feasible solution to our problem as well. Why?

The main problem is actually not technical: despite there are plenty of simple and straightforward open-source solutions for establishing a VPN, on the other hand **there are scarce possibilities that the customers of your product will let you create a VPN between your device and a Datacenter network**, for security reasons. In our simple scenario, we only have one cloud server, which has to communicate with the single board built on *SEDIA*: in complex scenarios, instead, considering a fleet of wheelchairs and an entire Datacenter to manage their navigation processes remotely, a much larger VPN would most likely be the solution needed, connecting the two networks. In this case, it would be very difficult to have complete freedom in its establishment, as more than one party can be involved, and not all of them will probably agree with this solution.

Another problem arises when containerizing our ROS2 application (which we will do in the next chapters): when dealing with containers in a cloud server, the natural choice is to manage them using a cloud container orchestrator, and the most widely spread is Kubernetes. Although being very efficient and easy to use, Kubernetes is not meant to work having at the other side a network connected with a VPN. **Establishing a VPN between *SEDIA* and a K8s Ingress** (for instance) **will bring to a lot of technical problems**, hence developing effort which cancels the advantage of simplicity previously discussed.

For these reasons, a third solution, which is the one adopted by this thesis, is presented in the next section.

4.3.3 eProsima DDS Router

The third and final solution explored by this work has been considered the only one feasible for implementing an actual deployment of a ROS2 system with remote communication between nodes. This solution is based on an open-source software created by eProsima themselves, the very same company that implemented the Fast DDS middleware technology: **the DDS Router**.

First, a deep introspective of what goes by the name of eProsima DDS Router is necessary, with a presentation of its feaures and capabilities. Once understood them, we'll explain how we intends to use this technology at first glance (meaning without containers).

The DDS Router technology

As stated in the official documentation, the eProsima DDS Router is *"an end-user software application that enables the connection of distributed DDS networks. That is, DDS entities such as publishers and subscribers deployed in one geographic location and using a dedicated local network will be able to communicate with other DDS entities deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of eProsima DDS Router. This is achieved by deploying a DDS Router on an edge device of each local network so that the DDS Router routes DDS traffic from one network to the other through WAN communication".*

In other words: *"the DDS Router is a cross-platform non-graphical application developed by eProsima and powered by Fast DDS that allows to create a communication bridge that connects two DDS networks that otherwise would be isolated. The main use case of the DDS Router is to communicate two DDS networks that are physically or virtually separated and belong to different LANs, allowing the entities of each network to publish and subscribe to local and remote topics indistinctly".*

The definition alone is enough to understand that this technology has been created for the very same purposes of this thesis, namely **making DDS entities** (in our case ROS2 nodes) **talk to each other from different DDS networks, as if they were on the same network**. As we know, one DDS network corresponds to one single LAN, plus one `ROS_DOMAIN_ID`: inside this space, all DDS entities like ROS2 nodes will automatically discover each other, thanks to the SIMPLE DDS Discovery process. Two different DDS networks can be isolated by an L3 network if they physically belong to different LANs, virtually if different `ROS_DOMAIN_ID`s are defined.

With the DDS Router, two different DDS network will actually remain distinct, but they will no longer be isolated. This technology will manage all the complexity

of taking the application data from the whole DDS network at one side (or part of it) and delivering it to the DDS network at the other side, hiding such complexity under the hood. It will be like there is only one large DDS network in which DDS entities can freely communicate.

This solution is very similar to the second one previously presented (VPN), and an actual VPN cloud be used by the DDS Router under the hood. But, even so, this solution is clearly more appropriate, as it is aware of the presence of actual DDS entities in the first place and manages the complexity of such entities' communication, which is something we need to do on our own if using a naked VPN.

Furthermore, *"DDS Router is a software designed for various forms of distributed networks, such as mesh networks in which nodes are deployed in different private local networks that are auto-discovered without any centralized network node, or cloud-based networks where there is a data processing cloud and multiple geographically distributed edge devices".* This second part of the definition stresses the concept of **geographically distributed networks**, which is the strength of the DDS Router technology: thanks to its mechanisms, it can create a globally spread DDS network, composed by different private LANs scattered all over the world, without relying on a centralized node as IoT networks normally do nowadays. There can be distributed nodes retrieving data from applications of various nature, and a data processing system composed by cloud servers, which is distributed as well.

We do not need to unlock the full potential of this technology, because it is far greater than our goals: we do not aim to a globally distributed cloud system with thousands of nodes acquiring data around the world, for instance, but only one single cloud server with a certain number of nodes, talking to an SBC installed on our vehicle where the rest of the ROS2 nodes are executed. A possible final deployment of a fully functional navigation system with a fleet of wheelchairs will require some extensions, but they are by far covered by the capabilities of the eProsima DDS Router.

Project overview

First, some nomenclature is necessary:

- **Participant**: the DDS Router communication interface, an abstraction over the DDS DomainParticipant. This entity manages the dynamic discovery of DDS entities on a specific network or interface.

- **Participant Name**: It is an alphanumeric string that uniquely identifies a Participant in a DDS Router execution.

- **Participant Kind**: It specifies the kind of the Participant. There are several of them already defined, which will specify in general terms how the Participant behaves. Some of the predefined Participant Kinds will be used in our solution.
- **DataReader**: DDS element that subscribes to a specific Topic. It belongs to one and only one Participant, and it is uniquely identified by a Guid.
- **DataWriter**: DDS entity that publish data in a specific Topic. It belongs to one and only one Participant, and it is uniquely identified by a Guid.
- **Discovery Server**: the Discovery Server Discovery Protocol (already discussed in Section 2.3.2) is a Fast DDS feature that enables a new Discovery mechanism based on a server that filters and distributes the discovery information. This is highly recommended in networks where multicast is not available (e.g. WAN).
- **Domain ID**: The Domain ID is a virtual partition for DDS networks. Only DomainParticipants with the same Domain ID would be able to communicate to each other, hence DomainParticipants in different Domains will not even discover each other. This concept has a 1:1 match with the `ROS_DOMAIN_ID`.
- **DomainParticipant**: a DomainParticipant is the entry point of the application to a DDS Domain. Every one of them is linked to a single domain from its creation and cannot change such domain. It also acts as a factory (creator) for Publisher, Subscriber and Topic.
- **Guid**: *Global Unique Identifier*. It uniquely identifies a DDS entity (DomainParticipant, DataWriter or DataReader) and contains a GuidPrefix and an EntityId. The EntityId uniquely identifies sub-entities inside a Participant.
- **GuidPrefix**: Global Unique Identifier shared by a Participant and all its sub-entities. Identifies uniquely a DDS Participant.
- **Payload**: Raw data (no format specified) that is received and sent forward from the DDS Router.

All these terms will be useful to understand this technology and its components.

As said, the DDS Router is an application that internally runs Participants. Each one of these entities is a communication interface, a "door" to a specific DDS network configuration. A closer look at the existing (predefined) Participant Kinds is shown in Table 4.1 below:

Participant Kind	**Aliases**	**Description**
Echo Participant	echo	Print in *stdout* every data received
Simple Participant	simple/local	Simple DDS DomainParticipant
Local Discovery Server Participant	discovery-server/local-ds/ds	Discovery Server DDS DomainParticipant for local communication
WAN Participant	wan/router	Discovery Server DDS DomainParticipant for WAN communication

Table 4.1: Participant Kinds

The Participants are the main components for a DDS Router application, as they're the key for connecting to different DDS networks together. Every time one of these Participants receives a message from their DDS network, they will forward the data and the source of this message through the other Participants. In order to have a running DDS Router application with some Participants, the only thing we need to do is to **configure** it, and we'll see how to do it in the next paragraphs.

The following schema represents a DDS Router local use case, namely in the same LAN. This scenario presents different DDS networks that are isolated one to each other due to the Transport Protocol (UDP, TCP, etc.), the Discovery Protocol (Simple, Discovery Server, etc.) or the DDS Domain ID used by each DDS entity. Being the same LAN, these are the only factors we can use to separate DDS networks. The DDS Router is configured in this case with 4 different Participants, each of them in one different DDS network. And all the data that arrive to one of the Participants will be forwarded through the others, allowing all the machines to connect to each other independently of their different configurations. One important thing to highlight is that this data transmission will be accomplished with a *zero-copy* communication mechanism, as all participants will share the pointer to the allocated data.

As shown in Figure 4.2, the DDS Router takes care of all the complexity of the communication between DDS entities: they can have a different nature, in terms of their application's goal (one could be a ROS2 application, but there can be IoT ones etc...), they can run in very different machines in terms of their technology, and they can have whatever combination of the mentioned variables (Transport Protocol, Discovery Protocol and Domain ID). We just have to run a

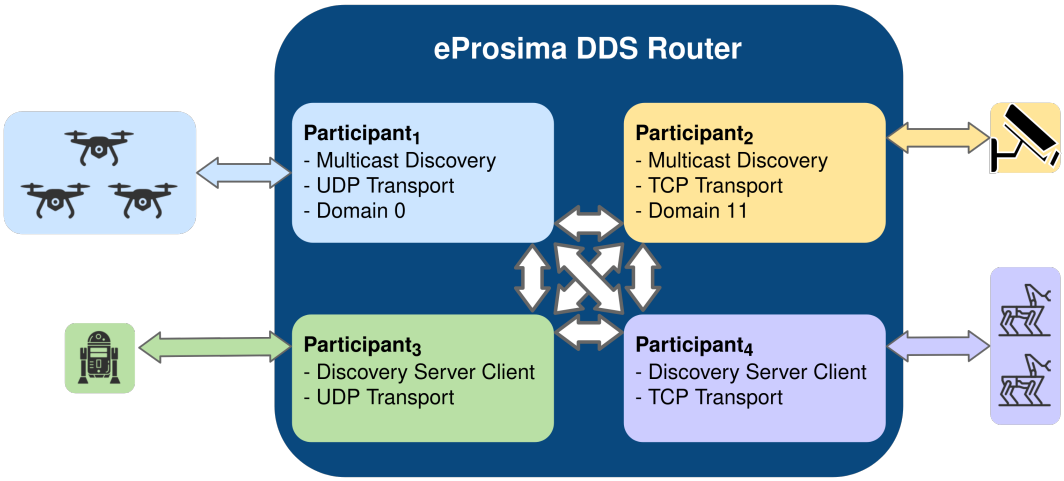

Figure 4.2: eProsima DDS Router local use case

DDS application in the same LAN, in whatever device connected to that LAN, and we configure a DDS Router with the proper Participants: doing so, we'll make all the DDS entities in the LAN talk to each other, creating a greater DDS network that, in this case, corresponds to the LAN itself.

Is it the use case we are looking for? Does this solve remote communication over L3 networks, connecting different LANs? No, it doesn't, because this example is about creating different DDS networks over the same LAN and put them in communication. In order to solve our problem, an additional DDS Router Participant must be presented, which is the key to our problem.

WAN communication

The DDS Router technology was actually meant, among other things, to address the very same problem of this thesis in some way, which is communication of DDS entities between remote DDS networks, separated by a WAN. This will work with ROS2 nodes as well, as they are DDS entities under the hood. How can we use this technology to do so?

First of all, to achieve a WAN communication of two DDS networks (or even more, but let's focus on two) that work in different LANs, we need a running DDS Router application on each LAN. The DDS Router deployed in one side will communicate to the other Router in the other side, which is its peer, using DDS over WAN. Once this receives the data, it'll transmit them to the local DDS networks to which it is connected, reaching the DDS entities eventually. This way, both DDS networks will behave as if they would belong to the same LAN.

Figure 4.3 shows the overall mechanism we need to leverage in order to make

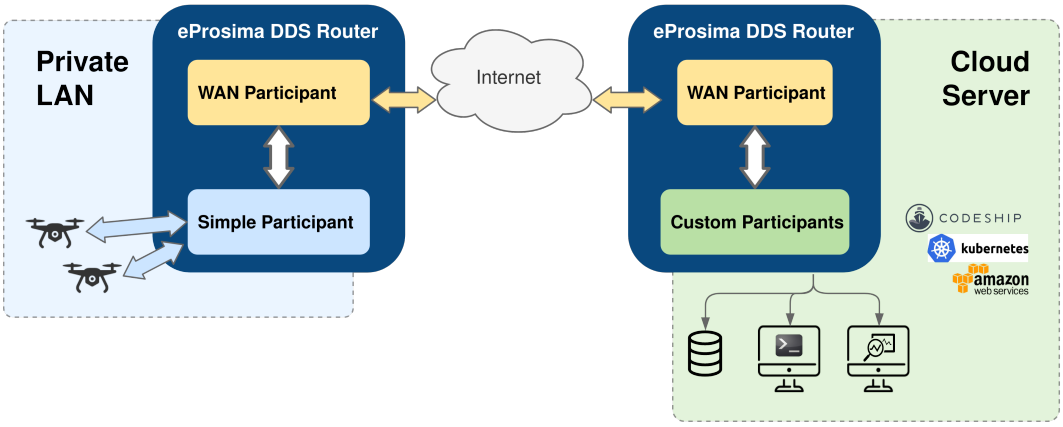

Figure 4.3: WAN communication with the DDS Router

two remote DDS networks exchange traffic: we need to configure two DDS Router instances in the two LANs, respectively, in a precise way. In the situation shown above, a "local" DDS network, with DDS entities running on it, wants to connect to a "cloud" DDS network, hosted by a cloud server. For now, we do not consider any containerization technology running on the server, differently than what the image shows, as we'll focus on container orchestrations in another chapter. Also, we'll talk about ROS2 nodes instead of generic DDS entities, to be more specific for our working environment, but everything can be extended to any kind of DDS application.

The local DDS Router configuration needs to have:

1. A **Simple Participant**: this kind of Participant, as shown in Table 4.1, is a Simple DDS DomainParticipant: this component acts as an **interface between the local DDS Router instance and all the ROS2 nodes running in the DDS network it is deployed in**. Indeed, a Simple Participant acknowledges the presence of all the ROS2 nodes running in the that LAN through the SIMPLE (multicast) discovery mechanism, but only the ones with a `ROS_DOMAIN_ID` that corresponds to the DDS Domain ID it is configured with. In fact, the Simple Participant is configured to listen at a certain domain, as we'll discuss in the next paragraphs. Once done this, it'll take all the data published by the discovered nodes and forward them to the WAN Participant (with a *zero-copy* mechanism). Actually, not all data are going to be forwarded to it, because the DDS Router configuration allows to filter some of them and control which data are allowed to pass.

2. A **WAN Participant**: this Participant does not communicate directly with the laying DDS network, but receives the (filtered) messages published in it,

through the previously mentioned Simple Participant. Its role is to **connect to the remote WAN Participant running in the other LAN and send it all the data it receives from the Simple Participant**. Hence, it acts as a bridge that connects two DDS networks over WAN, hiding the complexity of such communication under the hood. As can be easily guessed, this Participant is the key for our ROS2 remote communication problem, and in order to use it, the developer only needs to give it a proper configuration so that it can reach its remote peer: everything else is handled by the DDS Router technology itself.

The cloud DDS Router configuration is the following:

1. A **WAN Participant**: it is the remote peer of the previous one, which it has to connect to. It receives the data coming from the local WAN Participant and, with a *zero-copy* mechanism, forwards them to the other Participant in the cloud DDS Router.

2. The nomenclature *"Custom Participant"* shown in Figure **??** actually means that there could be different Participant Kinds cloud-side, depending on the situation. There are basically two cases we can identify:

   - If the ROS2 nodes running in the cloud server are being executed in a "normal" Linux environment, the situation is symmetrical to the local one (even a "local" and "cloud" distinction is meaningless at this point): a **Simple Participant** is deployed in this case, which takes the data received by the WAN Participant and forwards them to the ROS2 nodes running in any machine in the server's LAN, given they have the same `ROS_DOMAIN_ID` of this Simple Participant.
   - There is another situation, which is the most common in cloud environments, and will be the one we focus on in this thesis: a container orchestrator such as Kubernetes is running in the server, given that the entire ROS2 architecture has previously been containerized. In other words, ROS2 nodes are running in the cloud server inside containers (Docker containers are the most common ones) and they are being executed inside Kubernetes PODs, hence orchestrated. In this case, another kind of Participant is necessary, which is the **Local Discovery Server Participant**: the reason we must use this Participant and not a Simple one is the fact that most cloud environments do not supported multicast, hence a SIMPLE Discovery mechanism would not work. This Participant has the same role as the Simple one, but instead of discovering the ROS2 nodes in the LAN of the server through multicast, it uses **unicast metatraffic**, so that all nodes running in the LAN (which in this case is the virtual

LAN created by K8s) will be discovered. The same goes for other Cloud technologies, also shown as examples in Figure **??**: a Simple or local Discovery Server Participant will be deployed, depending on the multicast support.

Put in these terms, if we do not consider any containerization technology, the DDS Router configuration seems symmetrical in the two sides, like there are two exactly equal WAN Participants and Simple Participants, respectively, but in fact it is not: the implementation of the DDS Router actually expects a Client and a Server instance, which must be properly configured. Hence, to understand this concept in detail, a presentation on how to configure a DDS Router instance to be executed is necessary.

DDS Router configuration

Configuration is the only thing a developer must take care of in order to create the required DDS Router application. Any complexity regarding how the actual DDS communication is fulfilled by the DDS Router technology is hidden, as previously said. Let's see how to configure a DDS Router instance and which configuration is best suited for us.

A DDS Router can be configured by means of a *.yaml* configuration file: this file contains all the information regarding the components we wish to execute inside the DDS Router instance, such as topics filtering and Participants. In order to make the DDS Router accept this YAML configuration file, it must be passed as an argument when executed: if no file is provided, the application will attempt to load a file named `DDS_ROUTER_CONFIGURATION.yaml` that must be in the same working directory where it is executed. If no file is provided and such default file does not exist in the current directory, the application will fail. Configuration files may be easily validated by using the *YAML Validator* tool.

Figure 4.4 shows an example DDS Router configuration, which is explanatory of the structure of the file. Of course, there are many more possibilities than the ones presented here, but we only introduce the ones that will be useful for our problem: for any further investigation, the official documentation can be inspected.

1. *Configuration version*: The YAML configuration supports a version value to uniquely identify the format by which the file has to be parsed, so that new formats can be created without interfering with old ones.

2. *Topic filtering*: the DDS Router application allows for the filtering of DDS (ROS2) topics, which means defining the specific topics that are going to be relayed by the application. In this way, it is possible to define a set of rules by

```yaml
# local-ddsrouter.yaml

version: v3.0

allowlist:
  - name: rt/chatter
    type: std_msgs::msg::dds_::String_

participants:

  - name: SimpleParticipant
    kind: local
    domain: 0

  - name: LocalWAN
    kind: wan
    discovery-server-guid:
      id: 3
    listening-addresses:      # Needed for UDP communication
      - ip: 3.3.3.3           # LAN public IP
        port: 30003
        transport: udp
    connection-addresses:
      - discovery-server-guid:
          id: 2
        addresses:
          - ip: 2.2.2.2       # Public IP exposed by the k8s cluster to reach the cloud DDS-Router
            port: 30002
            transport: udp
```

Figure 4.4: DDS Router example configuration file

which the DDS Router will filter those data samples the user does not wish to forward. To define such data filtering rules, three kinds of list are available:

- *allowlist*: the list of allowed topics, hence the list of topics that the DDS Router will forward.
- *blocklist*: the list of blocked (not allowed) Topics, that is, all data published under the topics matching the filters specified in the blocklist will be discarded and therefore not relayed.
- *builtin-topics*: if a list of such topics is specified, the Discovery phase is accelerated, as the DDS router will create the DataWriters and DataReaders for these topics without waiting for them to be discovered. It just assumes these topics are present in the network right away.

It is not mandatory to define them in the configuration file: if not specified, a DDS Router will forward all the data published under the topics that it automatically discovers within the DDS network which it is conneted to. In our case, we're only allowing messages of type `std_msgs::msg::dds_::String_`, published in the topic `rt/chatter`.

3. *Participants*: this field specifies which Participants must be created inside a DDS Router instance. Each of them is identified by a unique Participant Name and a Kind: each Kind has its own configuration, with its own specific fields. There could be any number of Participants, and Participant Kinds could be repeated. In this case, two Participants are defined:

   (a) A Simple Participant, with a Name, a Kind of type local (alias for *simple*, as shown in Table 4.1) and a domain field, which represents the DDS Domain ID in which it is listening. A domain field with value 0, as in this case, means that the Simple Participant will be listening to ROS2 nodes publishing in `ROS_DOMAIN_ID`=0.

   (b) A WAN Participant (kind *wan*), whose configuration is much more complex:

   - Every WAN Participant needs to internally define a **Discovery Server**, which allows its remote peers to discover it over WAN (multicast is in most cases not supported in public networks). A Discovery Server requires a DDS GuidPrefix, which will be used by other Participants to connect to it, with a discovery-server-guid tag. There are several possibilities we can leverage for configuring a GuidPrefix: the simplest is the one shown in Figure 4.4, which consists in defining an **id**: by using the tag *id*, the GuidPrefix will be computed arbitrarily using a default DDS Router GuidPrefix, which is `01.0f.<id>.00.00.00.00.00.00.00.ca.fe`. There are other ways of defining it, explained in detail in the official documentation.
   - The tag *listening-addresses* configures the Network Addresses this Participant is going to listen at, in order to receive data from remote Participants. It requires an array of Network Addresses (only one is shown in Figure 4.4): every one of them must be specified in terms of the *ip*, *port* and *transport* tags and they will be used by a remote DDS Router to connect to this instance as a Client.
   - The tag *connection-addresses* allows us to configure a connection with one or multiple remote WAN Participants, and this is the key for achieving ROS2 remote communication. It requires an array composed of a Network Address and a GuidPrefix: the former specifies the *ip*, *port* and *protocol* tags which the remote peer is listening at, and they have to match the very same values specified in its own configuration file. The latter references the Discovery Server to connect with, which is the same Discovery Server specified in the same file (*id* tag). This allows the DDS Router we're configuring to connect to another instance executed in a remote (reachable) LAN, as a Client.

Once we make sure that the remote DDS Router instance we want to connect to is up and running, we need to take note of its listening Network Address, namely an *ip*, a *port* and a *transport protocol* (TCP/UDP). Then, we can use that information to configure our own DDS Router instance, that we want to run in our LAN, through a configuration file such as the one shown in Figure 4.4, which must have the corresponding values in its tags. Then, we source the Fast DDS library and execute the DDS Router by means of the command:

```
$ ./<install-path>/ddsrouter_tool/bin/ddsrouter
```

Or alternatively,

```
$ ddsrouter -c config-file.yaml
```

where config-file.yaml is the path of the configuration file: we can omit this argument by giving the file the name `DDS_ROUTER_CONFIGURATION.yaml` and saving it in the same directory the DDS Router is being executed.

Of course, the DDS Router must be properly installed and built as shown in the official documentation.

The solution to Problem 1

Using what we've been discussing, we can finally give a solution to our Problem 1, which is remote ROS2 communication.The setup is simple:

- Local-side, which means in the DDS network with the ROS2 architecture of *ALBA Robot* running, we add a DDS Router instance with the same configuration discussed above, the one shown in Figure 4.4 (with proper addresses, ports and Discovery Server Guid). The Simple Participant must be configured to listen at the DDS domain (*domain* tag) that matches the `ROS_DOMAIN_ID` of the nodes of *SEDIA*: this allows to hear all the data exchanged locally by the ROS2 nodes and forward them to the WAN Participant. If we configure it with proper values, this Participant will in turn automatically connect its remote peer as a Client and send it all the data received from the Simple Participant.

- Cloud-side, we shall execute all the ROS2 nodes of the *SEDIA* architecture that we consider feasible to run inside the cloud server. Then, another instance of the DDS Router must be launched, configured in a similar way than the local one. In this case, though, no *connection-addresses* tag must be specified, as the Server instance only requires a listening Network Address (*listening-addresses*

tag), plus the Discovery Server Guid (*id* tag). The WAN Participant, acting as a Server, will connect to the Client requesting the pairing and will receive all the data sent by it. Then, it will forward thee data to the Simple/Local Discovery Server Participant, which publishes all data into the cloud DDS network, at the `ROS_DOMAIN_ID` configured.

Note: the two *domain* tags in the Client and Server WAN Participant configurations can be different, meaning that two DDS networks with different DDS Domain IDs (corresponding to the `ROS_DOMAIN_ID`s) are connected through WAN.

In this way, we have achieved **DDS communication over WAN**, allowing ROS2 nodes to listen to each other's data from remote LANs, separated by an L3 network. Every ROS2 node executed in one of the two LANs with a proper `ROS_DOMAIN_ID` will discover nodes physically located in a remote DDS network as if they were in the same one, thanks to the DDS Router technology. Of course, some delay will be added in the data exchange, which will lead to some problems, but we're going to discuss this in a separate chapter.

Also, at this stage we do not consider any containerization technology such as Docker, nor any orchestrator like Kubernetes. When these technologies will be added, the deployment of our ROS2 architecture and the DDS Router configuration will be slightly more complex. We're going to discuss this in a separate chapter as well.

4.4 Problem 2: cloud/local switching

This section will explore in detail the solution developed and adopted by this work to overcome the problem of **creating a switching mechanism between local and cloud replicas of the ROS2 nodes** belonging to the *ALBA Robot* navigation system. This can be seen as the real goal of this thesis.

The problem of deciding at runtime whether to leverage components running in the cloud or fallback to local resources is discussed quite a bit in today's research: if we manage to find a way of doing that, we can take advantage of the enormous computational power of cloud servers for our autonomous driving applications, shifting there the weight that small Hardware components would have carried on their shoulders otherwise. This seems like an optimal condition, given that we can find an optimized solution for the implementation of such switching mechanism. The real question is whether this solution is convenient or not: is today's network infrastructure capable of living up to the tiny little latency constraints of a system like this? In other words, are we actually capable of implementing an autonomous driving system working (completely or partly) in the cloud?

There are no easy answers, and moreover there are not unique answers to

these questions: everything depends on the nature of the application (even among different autonomous driving ones) and the actual values of latency constraints. There are a lot of variables implied in these considerations, hence we're not able to provide easy answers at this stage. As a matter of fact, we're not actually addressing these questions in this section in the first place, as the only thing we're focusing on here is describing a technical way of achieving some sort of switching system. A possible way of understanding whether such system is feasible or not is to put it to work in a real environment and collect some data regarding the actual latency the system can sustain. We'll focus on this in the next chapters.

It is very important to highlight that, as discussed in previous considerations, this switching system is not an alternative to the remote communication solution presented in the previous section, nor it requires its presence to be implemented. The two issues are definitely independent and can coexist in the same environment.

The mechanism we're introducing aims to achieving a way of having one or more replicas of the same ROS2 node running at the same time, but only one at a time must be actually performing its tasks. This must be decided according to the current network connection status or any other sort of logic. Furthermore, these nodes must be part of the same DDS network for this mechanism to work, so that each node will acknowledge the presence of all the others regardless of their physical location in space. Among the ones a node discovers there will be its replicas as well. They can be in the same machine, in different machines in the same LAN, or in different (remote) LANs: in the first two cases, nothing must be done in order to make these nodes belong to the same DDS network (as we know from previous sections), whilst in the latter case an additional logical mechanism is required, and that is the one described in Section 4.3.

In this section, we assume that ROS2 remote communication has already been implemented using one of the possible existing solutions, either one of the three discussed in this thesis (Section 4.3) or any other possible solution. Hence, we're focusing our attention on the switching mechanism only, which we'll describe in all its components.

The solution that has been developed by this work was guided by one simple principle: **having the minimum impact possible in the existing configuration**. In other words, the implementation of the switching mechanism must be based on the system currently running on *SEDIA* and must not represent a huge change in its architecture. It should be only an addition built on top of it, created to support the local/cloud switching. For instance, one may consider using some external framework, which has nothing to do with ROS2 or Nav2, to implement the logic which enables the switching, but a solution like this would be too pricy for the company. It would mean a complete upheaval of the system they're used to

work with and the installation of the framework itself on the company's Hardware, not to mention the huge effort to make it work in symbiosis with the ROS2 + Nav2 architecture. Hence, the right approach will be to base on the currently working setup and add minimal changes in order to support the switching mechanism.

4.4.1 The overall solution

The solution for cloud/local switching is based on the Nav2 concept of **Lifecycle nodes**. As explained more in detail in Section 2.2.2, the Nav2 project defines a precise lifecycle for each ROS2 node, whose behaviour is very similar to a state machine. The state transitions of every node are managed by the `guardian` (see Section 3.2.3), which is the only one with a self-managed lifecycle and in charge of handling the one of all the other managed nodes present in the same DDS network. Thanks to the `guardian`, when launching a ROS2 navigation system like the one of *ALBA Robot*, every node experiences a series of lifecycle transitions which bring it from the Unconfigured state to the Active one, in which it is ready to perform its tasks.

As we know, the condition we want to achieve is to have one replica for each node (only the ones able to run in the cloud), so that every local node has its cloud "alter ego". The two nodes are exactly equal and completely agnostic of the fact that another remote replica of itself exist. This goes very well with the principle we explained earlier, which states that the new mechanism must not have any strong impact on the existing architecture and only some additional logic should be implemented to orchestrate the switching "from above". Also, we must remember that all nodes, both local and cloud, must be part of the same DDS network, so that every node of the High Level system discovers all the other ones, regardless of the way this condition is achieved.

How are we going to exploit the concept of lifecycle to create the switching mechanism?

The answer is actually very simple: we have at our disposal a very powerful node, which is the `guardian` mentioned above. We can see it as a "control tower" which lets us manage all the nodes of the architecture from a superior perspective, hence we can use it to orchestrate the life of the nodes in every way we wish. Therefore, considering each local node and its respective cloud replica, there is a very easy way to control which one of the two nodes must be performing its tasks at a certain time: **only one of those two nodes must be Active, whilst the other one must be Inactive**.

In other words, we exploit the power of the `guardian` node on the nodes' lifecycle to activate all the cloud ones when a certain condition is satisfied, and the local ones are deactivated in the meantime. When that condition is not yet true, the

switching is performed the other way around, by just activating the local nodes while deactivating the cloud ones. At every moment, only one of the two nodes of a local/cloud couple must be in Active state, performing its tasks and publishing data in its topics, while the other one must be Inactive, idle, not participating in the navigation process.

This is the basic principle that will help us build a proper solution for our switching problem: of course, it is not complete in itself, as more considerations have to be done in order to implement it in a proper way.

Problems

- How can we manage a **sudden loss of network connectivity**? If the guardian is only present locally, it won't be able to send the data useful to deactivate the cloud nodes, ergo it cannot switch them off. At the same time, it'll try to activate the local ones, hence there could be some moments in which no ROS2 node is up and running. Of Navigation cannot take place in that period of time, even if pretty short, and this could call for a sudden stop of the wheelchair until the local setup is all up and running.

- Then, **how will the switching logic be implemented**? Saying that the switching will take place if the network connection with the server is lost is not enough: of course, it must be done in correspondence of sudden network malfunctioning, but what if the connection is gradually getting worse? Shouldn't we implement some mechanism which decides to fallback to the local nodes even if connection is not yet completely gone? Indeed, there cloud be some case in which the local setup becomes more convenient than the cloud one, even if data can be physically exchanged and the connection is not completely gone. This suggests that a more complex logic, based on the nature of the ROS2 application itself, should be developed, with a very solid way of computing some sort of threshold.

These issues must be addressed, and we'll do it in this thesis, in order to implement a proper and robust switching mechanism. This will actually be useful to solve the problems that come with implementation, as the main concept remains the one explained above, which is ideally the only thing we need to implement the switching mechanism: managing the lifecycle of nodes in a smart way.

However, these are not all the matters we will face if we wish to realize an actual final product: solving those matters will indeed help in creating a mechanism capable of switching between local and cloud instances of **generic** ROS2 nodes. What if we consider the actual Nav2 nodes composing the *ALBA Robot* High Level architecture? This will add a whole new series of problems. However, we're only

giving a hint and some possible ideas on how to solve them, but they're not going to be the focus o this work. They will be considered additional issues, independent of the proposed switching solution, and they will be considered by whoever wants to implement an actual product out of this solution:

1. **Synchronization**: the activation of a local node and deactivation of its remote replica (and the opposite process of course) must be synchronized. Each node needs some processing time to be activated/deactivated by the `guardian`, hence the two operations must be orchestrated in order to avoid two bad situations: both nodes are deactivated at a certain moment (we'll actually address this problem in this thesis, as said), or two exactly equal nodes publish the same information, which will be heard twice by the target(s) node(s). The fact itself that this can represent a problem or not is defined by the nature of the application itself: most likely, the majority of nodes will not have any problem in receiving the same information twice, but it could be an issue for some of them.

   A solution for this could be to define, for each couple of local/cloud nodes, a third node which acts as an intermediate collector. The local node and its cloud replica will be forced to publish their messages in different topics than the ones currently adopted: this intermediate node will take the two messages and publish only one of them in the topic the receiving node is actually listening to. In this way, duplicated messages can be avoided for all the nodes affected by them.

2. **Actions**: if the remote node starts an action while a switching is triggered at the same time, that action will be interrupted. This may be one of the most likely scenarios happening if this switching mechanism is actually implemented, as every navigation goal that *SEDIA* performs is no less than an action: there will be multiple occasions in which, during a navigation (*SEDIA* moving from a point A to a point B) the switching is activated. In this case, the nodes performing such action are going to be deactivated to be replaced by their remote replicas. How can the newly activated version know about the action that was being performed?

   An idea could be to have some sort of mechanism to prevent this situation, by saving elsewhere a copy of the information about the interrupted action. For instance, a local and a cloud node could be added, in charge of storing the info about every action performed by the system: they will be exactly equal and store the same info. When an action is interrupted by a switch, the nodes in charge of continuing it can ask the newly added storing ones for its state as it was at the time it was interrupted. In this way, they can start a new action exactly from that state, acting as the previous one was never interrupted.

3. **Stateful nodes**: some nodes, in their working algorithm, need some past data, namely data they computed previously. When a switching is triggered, the newly activated replica needs the same data, otherwise it cannot perform the same tasks.

   A possible solution is actually very similar to the one of the previous problem: in order to provide the missing data, each time the information needed to perform one node's next tasks are computed, they will be stored in a "database" node, which will provide them to the instance activated by the switching.

These issues will be seen as **future work** for whoever wishes to implement a final product out of this solution, whilst our aim is to just explain the core of our solution for the switching system.

Figure 4.5 shows the architecture that has been developed to support the cloud/local switching system between ROS2 nodes.

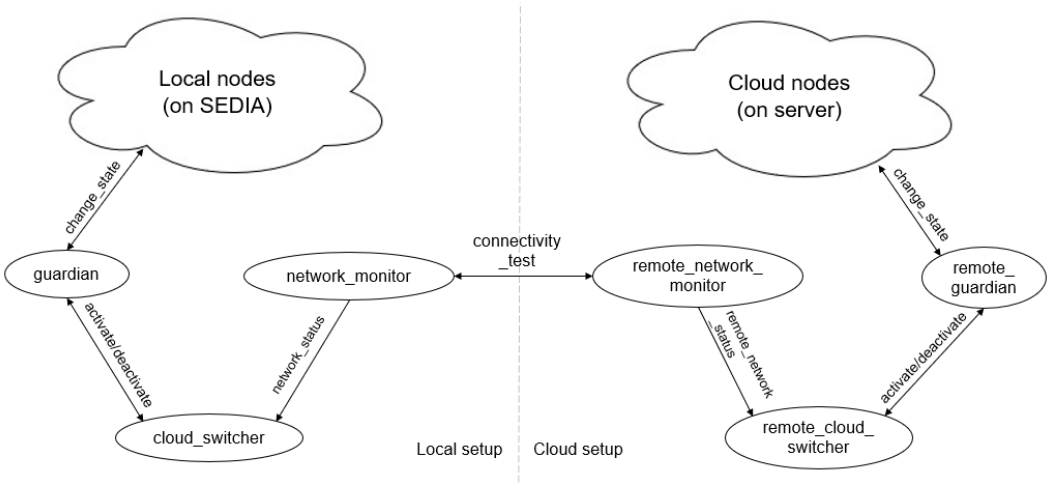

Figure 4.5: Switching mechanism architecture

We'll explore all the represented components' roles in the system, but we can understand at first sight that it is a very simple architecture, composed of six main nodes in charge of implementing the switching, while all the others remain unchanged. In fact, the two little cloud shapes represent the whole set of nodes composing SEDIA High Level architecture as it was before this thesis, divided into local (original) nodes and their cloud subset, namely the cloud replicas of only the duplicable part of the local nodes. The architecture to realize the switching is logically built on top of them, in a way that allows them to be agnostic to its very existence.

Note that the two cloud shapes in Figure 4.5 seem independent and separated in this representation, and the only contact point between the local and cloud nodes seems to be the `network_monitor`. The figure can in fact be misleading from this point of view, as **all the nodes in the *ALBA Robot* navigation system, both local and cloud, will run in the same DDS network**. This means that they can freely discover each other and everything published by the local nodes in certain topics will be heard by all the cloud ones listening to those specific topics. It will be concern of the switching mechanism to orchestrate all the components' behaviour, such that only one of the nodes belonging to a local/cloud couple works at a certain time.

The six nodes we're presenting can also be grouped into local and cloud versions, as shown in Figure 4.5: each of the three nodes in the left has its respective "alter ego" running in the cloud server, but in this case the two nodes of a local/cloud couple slightly differ: they have basically the same role in the architecture, but their behaviour can be considered mirrored, as we'll see in the next sections. Indeed, these six nodes are the very core of our solution to Problem 2 and we'll introduce every one of them in terms of their characteristics and their role in the fulfilling of the switching mechanism.

4.4.2 The `network_monitor` and `remote_network_monitor`

The first couple of nodes we're presenting is crucial for our application, as it represents **the core of the logic which decides whether to trigger the switching or not**. In other words, we know that the switching process consists of the decision to avoid the usage of cloud nodes and fallback to the local ones if the network does not allow it. When the network is up again, the mechanism returns to the cloud nodes. In order to do so, we need some ROS2 nodes capable of understanding the state of the network at every moment and deciding for the entire system if it is time for it to perform switch. The `network_monitor` and `remote_network_monitor` nodes are the ones in charge of this role, hence they listen to the network (as its name suggests) and tell the other nodes to trigger a switch if necessary. In this section, we're having a close look at the features of these nodes separately, to understand how they cooperate in performing their role.

`network_monitor`

It is a managed node, meaning that it follows the same lifecycle of most of the nodes in the architecture (only a bunch of them are not managed), handled by the `guardian`. This is the node in charge of **telling the local part of the navigation system that a switching must be triggered**, as previously said. Actually, it does not directly tell the `guardian` to activate or deactivate the local nodes,

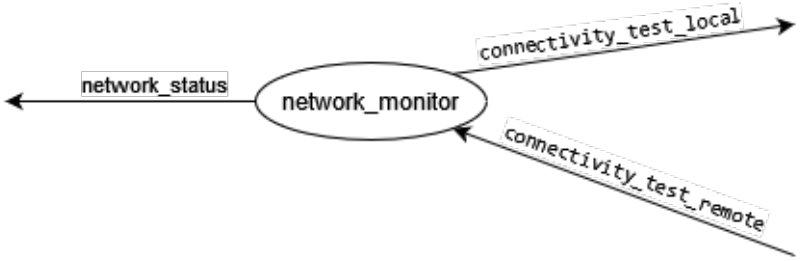

Figure 4.6: The `network_monitor` node

but tells the `cloud_switcher` node if the network is capable of sustaining the navigation proces by using the cloud or not. Let's see its main features:

- It subscribes to the `/connectivity_test_remote` topic and publishes in the `/connectivity_test_local`. These topics are represented in Figure 4.5 as one single double-headed arrow for simplicity, but they're actually separated and independent: the `network_monitor` node and its cloud version use them to exchange information about the state of the network. Indeed, the way that this thesis has used to implement the network monitoring is to make these two nodes exchange a specific type of ROS2 message, called *Header*, defined in the ROS2 `std_msgs`.

  A ROS2 message of type *Header* contains a field of type Time, a structure generally used to communicate timestamped data, which is a two-integer timestamp expressed as seconds and nanoseconds. Using this field, a node can communicate the exact time in which a certain message has been sent, and the receiving node can use that information for its goals. In our case, the `network_monitor` node initializes a timer (with a configurable value) and publishes in loop a *Header* message with the current time in nanoseconds in the `connectivity_test_local` topic, which will be used by the cloud version of the node. Meanwhile, the latter sends same kind of message and the `network_monitor` compares the received information with the current time to compute the period that the message has spent to go from the cloud to the local node, namely the latency. The computed latency will be in turn compared to a predefined threshold to state if the system can use the cloud nodes, or not. How this threshold is defined depends on the logic we want to base our switching system on and will be discussed in a chapter of its own. Regardless of such logic, if the computed value is above the threshold, the system must be informed.

- Another topic this node publishes in is the `/network_status` one, which is used to communicate the outcome of the connectivity test to the `cloud_switcher`.

In other words, once the connectivity test has been fulfilled as discussed above, the `network_monitor` has the means to say whether the network can sustain a navigation system in cloud or not. This node is actually not the one in charge of deciding if the actual switching must be activated, as this role will be covered by the `cloud_switcher`. The `network_monitor`'s role is only limited to publish, in the `network_status` topic, a message of type Int32 (also defined in the `std_msgs` of ROS2) with one of these two values:

- **0** if the latency is above the threshold, meaning that "the network cannot sustain the navigation system in cloud".
- **1** otherwise, meaning that "the network can sustain the navigation system in cloud".

When being activated by the `guardian`, the `network_monitor` node initializes the timer at the end of which a message in the `/connectivity_test_local` is published. Then, an additional timer is defined, to address another problem: what if connection is completely lost, meaning that the network cannot deliver messages from the cloud version? The node would wait until it receives a message and we don't want that, because in that case we need to trigger the switch to the local system. Hence, this timer represents the maximum waiting time of the node at the end of which the network is considered down and a message with value 0 is consequently published in the `/network_status` topic. Of course, this timer must be restarted every time a message is received in the `/connectivity_test_remote` topic, because receiving a message from the cloud replica clearly implies that the network is not down.

Then, also during the `on_activate` transition, the two publishers mentioned above are activated: lifecycle nodes can indeed take advantage of a more complex version of publishers, which are managed entities themselves, with a lifecycle. When the `on_activate()` function is called on one of these publishers, messages start being published in its respective topic as soon as a `publish()` function is called. If a lifecycle publisher is instead deactivated (by means of the `on_deactivate()` function), even if such `publish()` function is called the node will try publishing the message, but without success. A warning will be displayed in the node's logs and no message will actually be heard by whoever listens on that topic: this must be done in the `on_deactivate()` state transition. In this way, the `guardian` can finely control the behaviour of the managed nodes, as they will not be able to exchange any message if not explicitly put in Active state by it.

`remote_network_monitor`

This is the cloud version of the `network_monitor` node and it has the same role and same characteristics of its local version, except for the fact that their behaviours

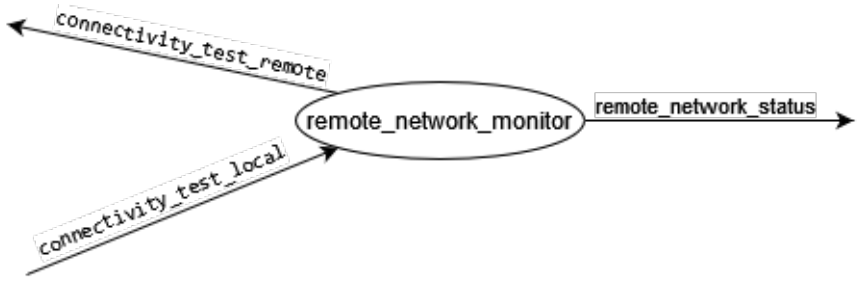

Figure 4.7: The `remote_network_monitor` node

are mirrored. Let's have an overview on this node as well:

- It is a managed node just like its local version, whose lifecycle is controlled by the `guardian` like all the other managed nodes in the architecture.
- It subscribes to the `/connectivity_test_local` topic, in which it receives the timestamped messages of type *Header* from the `network_monitor`, as described above.
- It publishes timestamped messages of type *Header* in the `/connectivity_test_remote` topic for its local version to hear and process in the way we've described. The time of publishing is recommended to match the one of the `network_monitor` node, so that the two behaviours are equally reactive (not necessarily synchronized due to little differences in their activation process).
- Every time it receives a message in the `/connectivity_test_local` topic, it compares the timestamp contained in that message with the current time and computes the latency spent by the network to deliver it. Then it compares the newly computed value with a threshold, which must be the same than the one used by its local version, and publishes an Int32 message in the topic `/remote_network_status`, with one of these two values:
  - **0** if the latency is above the threshold.
  - **1** otherwise.

  Even in this case 0 means that the network cannot sustain the usage of the cloud for the navigation tasks, while 1 means that it can. Here, the difference is that this message will be received by the `remote_cloud_switcher` node (as shown in Figure 4.5) and that will react in a slightly differently than its local replica, as we'll see in the next paragraphs.
- Another timer is defined, also in this case used by the node to state that the network is not responding. When it expires, a 0 value is published in the

`/remote_network_status` topic, in order to inform the `remote_cloud_switcher` that the system is not able to use the cloud.

- In the `on_activate()` transition, the two timers are initialized and the two publishers are activated, while in the `on_deactivate()` the two publishers are deactivated, preventing them from publishing data that must not be sent at that time (local nodes are being used).

Together, these two nodes represent the **interface between the local and cloud part of the switching mechanism**, as they are the only ones directly communicating to take decisions about the state of the network.

4.4.3 The `cloud_switcher` and `remote_cloud_switcher` nodes

These two nodes represent an **interface between the network monitoring node and the `guardian`**, each one in their respective side (local and cloud). Let's focus first on the local version and describe its characteristics: then, the remote one will have a mirrored behaviour as usual.

`cloud_switcher`

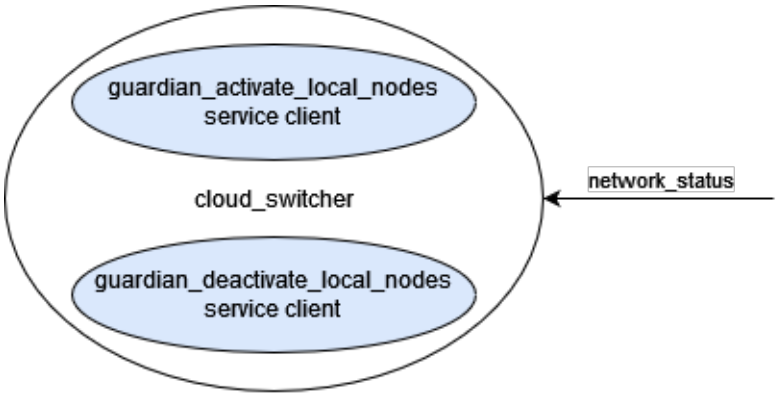

Figure 4.8: The `cloud_switcher` node

The `cloud_switcher` node is the one in charge of deciding whether to trigger the switch or not. It listens to the network status measurement in the `network_status` topic and **takes the decision of whether to activate or deactivate the local nodes** accordingly. In fact, since the `guardian` is the one and only node capable of controlling all nodes' lifecycle, the `cloud_switcher` will not directly implement the switching mechanism by triggering the state changes, but it will ask the `guardian` to do so, through a specific couple of services we'll talk about in

some lines. According to the network monitoring info, one of the two services is called, which will ask the guardian to activate or deactivate the local nodes, given the network is respectively down or up.

Let's have a look into the fundamental features of this node:

- The node has a flag, called `network_is_up_`, used to store the information about the previous status of the network. In other words, this flag tells the `cloud_switcher` whether connection was up the last time it received a message in the `/network_status` topic: if this flag is set to true, it means that the cloud part of the architecture was being used before the last information about the network connection was received. The local setup was being used otherwise. This flag will be fundamental in the working algorithm of the `cloud_switcher`, as we'll see, and it is initialized as true in the `on_configure()` state transition.

- In the same transition, the node also initializes two service clients, respectively `guardian_activate_local_nodes` and `guardian_deactivate_local_nodes`. We're going to talk in detail about these two ROS2 services in another section: for now, it is enough to say that the `cloud_switcher` can call these two services when needed, as they have been defined in the local version of the `guardian` node.

- It subscribes to the `/network_status` topic, as previously mentioned, in which it receives the information about the status of the network, in the way presented: it gets the information of whether the current connection status can sustain the usage of the cloud system, by publishing 1, and 0 otherwise. The main working algorithm of this node is based on the `monitoring_available()` callback, which is automatically invoked every time a message is received in that specific topic, as that is the function tied to it. Here, the main task of the `cloud_switcher` node is fulfilled, namely deciding whether to ask the guardian to trigger the switching or not: to do this, one ROS2 service request for each of the two services defined in the `guardian` is initialized, and the node verifies if such services are available at that moment. Then, it follows the following logic:

  The two variables used to define the conditions expressed above are:

  - The **`network_is_up_` flag** mentioned above, used to state if the network was up or down before receiving the message in the `/network_status` topic.
  - The **actual message** currently received in such topic, which states if the network is up or down in that specific moment.

Algorithm 1 The `cloud_switcher` node's algorithm

```
if the network was up and it stays up then
    do nothing other than printing it on screen.
else
    if the network was down and now it's up then
        ask the local guardian to deactivate the local nodes.
    else
        if the network was up and now it's down then
            ask the local guardian to activate the local nodes.
        else
            if the network was down and it stays down then
                do nothing other than printing it on screen.
            end if
        end if
    end if
end if
```

Note that being up or down for the network connection means that it is able to sustain the navigation system using the cloud nodes or not, which is something up to the `network_monitor` node to decide. Let's see an example.

Let's assume that the `network_is_up_` flag is equal to 1, which means we can say that the navigation system of *SEDIA* is currently using the cloud nodes: if the `cloud_switcher` node receives a value equal to 0 in the `/network_status` topic, it understands that the network can no longer sustain the workload of communication between remote nodes, hence it tells the `guardian` to activate the local nodes, which are the ones the system needs to use. This corresponds to the third condition expressed above. If 1 would be received instead, it means that the network is still able to use the cloud, hence nothing has to be done: this equals the first condition.

Of course, the `network_is_up_` flag must be updated with the current status of the network when this changes: namely, in condition 2 and 3 the value of the flag has to change from false to true and from true to false respectively. In each case, a logging message is displayed in order to inform the user whether a switching is being triggered or not.

`remote_cloud_switcher`

The `remote_cloud_switcher` is the cloud version of the previous one, hence its characteristics and behaviour are pretty much the same, but mirrored.

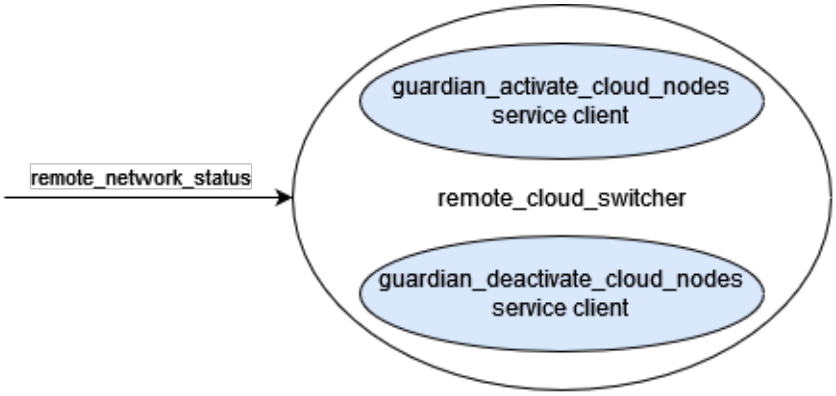

Figure 4.9: The `remote_cloud_switcher` node

- It has the same `network_is_up_` flag of its local version, which has its exact role. The two flags must of course be synchronized, as they state if the overall system is using the cloud nodes or not, an information that both the local and the cloud sides must have. This is useful, among other reasons, when sudden connection losses occur, as in this case the two peers will independently figure out that the cloud cannot be used and will change the value of this flag to false at the same time. Consequently, this flag is initialized to true as well.
- It initializes two ROS2 service clients, namely `guardian_activate_cloud_nodes` and `guardian_deactivate_cloud_nodes`, which in this case are the services defined in the cloud version of the `guardian`.
- It subscribes to the `/remote_network_status` topic, in which information about the status of the network are publishes, as seen. Such information is exactly equal as the one exchanged through in the local version of this topic, namely `/network_status`. Even in this case a callback function is invoked as soon as some information is available, which in this case follows a mirrored behaviour than the local `cloud_switcher`:

  Indeed, the `remote_cloud_switcher` must decide whether to switch to the cloud or local system, or remain in the current one, in symbiosis with its local peer: the two nodes must come to the same exact conclusion, which is one of the 4 conditions expressed above, but the actions performed by them will be symmetrical: if nothing changes (conditions 1 and 4) nothing is done; then, if the navigation system must switch to the cloud, the `remote_cloud_switcher` tells the `remote_guardian` to activate the cloud nodes, while the `cloud_switcher` tells the local `guardian` to deactivate the local ones (condition 3). The behaviour will finally be the opposite if a switch

Algorithm 2 The `remote_cloud_switcher` node's algorithm

```
if the network was up and it stays up then
    do nothing other than printing it on screen.
else
    if the network was down and now it's up then
        ask the remote guardian to activate the cloud nodes.
    else
        if the network was up and now it's down then
            ask the remote guardian to deactivate the cloud nodes.
        else
            if the network was down and it stays down then
                do nothing other than printing it on screen.
            end if
        end if
    end if
end if
```

to the local system is needed (condition 2).

4.4.4 The modified `guardian` and the `remote_guardian` node

The core of our solution is represented by the `guardian` node, or better by a local/cloud couple of guardians. The one already existing in the original *ALBA Robot* architecture has been used, with all its existing features, but some key modifications have been made in order to support the implementation of the switching: then, the cloud version of this node has been introduced, being a mirrored "alter ego" of the local one.

The `guardian` is the node handling the lifecycle of all the others, as we know, and we already mentioned that the switching mechanism is based on the fact that only one node of each couple composed by a local one and its cloud replica must be active at a certain moment. Hence, **the guardians have been modified in order to support the activation and deactivation of the set of nodes belonging to each respective side**. Indeed, the local `guardian` will trigger the state transitions of the local side of the navigation architecture, and the cloud one will do the same with the cloud nodes. Actually, they do not know anything about the logic according to which they must trigger a switch: they just receive a request to do so, through the previously mentioned ROS2 services, and they'll just perform it. The role of defining such logic is assigned, as we know, to the `cloud_switcher` and `remote_cloud_switcher` nodes.

Some important changes had to be made to the original `guardian` in order to support the switching solution: at the time of this thesis, its official version is of course not aware of the fact that some nodes might have a replica in cloud, so additional logic in its working algorithm (described in Section 4.4.4) had to be introduced. Its cloud replica has been added as well, with a mirrored behaviour as usual.

We'll go through these key modifications one by one, considering in turn the local and cloud versions.

`guardian`

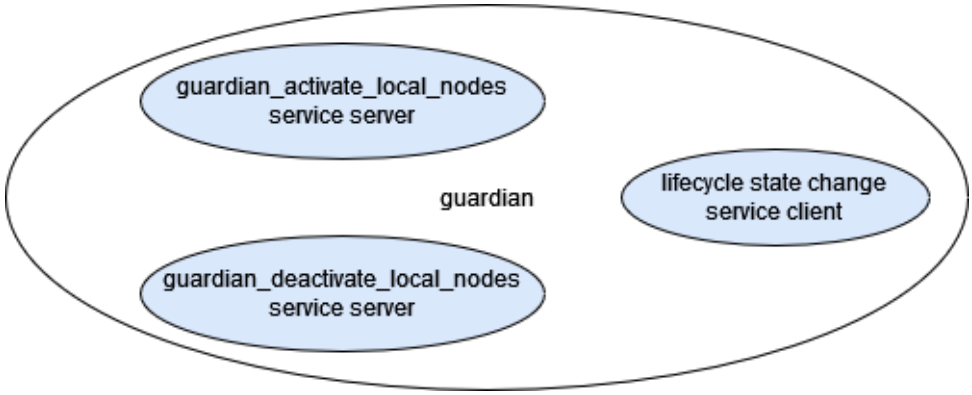

Figure 4.10: The `guardian` node

The modifications on the local version of the guardian node are about both **new additions in its data structures and components**, and **new logical behaviours**. Let's consider them in order:

- First of all, we need to make the guardian aware of the fact that some of the nodes it has to manage could have a cloud replica. The local `guardian` will only be in charge of the lifecycle of the local nodes, but its behaviour will be different according to the fact that those nodes are the only ones existing in the navigation system or not. For this reason, an **additional flag has been introduced** in the `guardian`'s configuration file, which states if a specific node has a cloud replica or not: if that flag is true, it assumes that a cloud version of the node has been defined (although there is not any guarantee of that, the developer must be careful of actually defining it), and will treat that node accordingly by following a different logic than the other ones. For the nodes with that same flag equal to false, the normal behaviour of the `guardian` is maintained, as in that case the local one is the only working

version of it: this is, for instance, the case of those nodes which cannot work in the cloud, being tied to the Hardware of *SEDIA* (as discussed in Section 4.2).

- The mentioned flag will be used by the `guardian` to categorize the nodes, in order to have clearance on which of the nodes it has to manage has a cloud replica or not. Hence, **a new list of nodes has been added**, called `duplicated_nodes_list`, together with the original list which collects all the managed nodes' names. this new list is contains all the names of the nodes duplicated in cloud, hence it is a subset of the other one and can be empty if no cloud system is implemented. The nodes contained in that list will be treated differently, and we'll see how.

- **Another flag**, with the name `cloud_is_enabled_`, **has been introduced**. This one is meant to state if the cloud system is currently being used or not, which can be said just by looking at the duplicated nodes (if any), from the local `guardian`'s point of view: if such nodes are in Active state, it means someone previously requested their activation (and that can only be the `cloud_switcher` through one of the ROS2 services discussed below), because the cloud part of the system cannot be used. This information will be very useful in the newly added logical mechanisms of the `guardian`.

- **Two services**, called `guardian_activate_local_nodes` and `guardian_deactivate_local_nodes`, **have been added as well**. We spoke about them in the previous sections, hence we already know that these services are the mean used by the `cloud_switcher` to ask the `guardian` to trigger a switch. When one of the two services receives a request, a function is called, either `activate_local_nodes` or `deactivate_local_nodes`, respectively. Let's consider the first mentioned function's very simple algorithm:

Algorithm 3 The `guardian_activate_local_nodes()` algorithm

1. First, check if there is any duplicated node in the `duplicated_nodes_list`. If that list is empty, the navigation system's cloud part does not exist at all: hence, return.

2. Then, for each node in that list, trigger its `on_activate()` transition, setting it in Active state. Let's keep in mind that we're talking about the local `guardian`, hence these nodes are local ones, the subset of them which have a cloud replica.

3. Once activated all the duplicated local nodes, set the `cloud_is_enabled_` flag to false: indeed, if the `cloud_switcher` requested the activation of the local subsystem system, it means that the cloud cannot currently be used.

The `guardian_deactivate_local_nodes()` function has the same exact behaviour, but instead of activating the duplicated local nodes, it deactivates them, setting the `cloud_is_enabled_` flag to true.

After considering the newly added components in terms of data structures, let's talk about the few key modifications in the working algorithm of the `guardian` node made to support the switching:

- The first logical behaviour to modify is about the **inhibition of the driver node**, which in our case is "`wheels_control`". Normally, the `guardian` would send a request to this node to ask for a stop of the wheelchair, through a specific service, as soon as a CORE node is found in a state different than Active. In our case, there are some nodes, likely CORE ones, which are duplicated in cloud, and the normal logic of the switching solution contemplates some local nodes being not active when their cloud versions are being used in their place. It is pretty clear, then, that this logic does not cooperate with the switching mechanism thought by this thesis and must be extended in order to support it. How?

  It is sufficient to change the condition by which the guardian decides whether to call for the stopping of the navigation or not: considering each single CORE node in the list of lifecycle nodes, the condition was changed from *if that node is not Active* to *if that node is not Active) AND ((it is not duplicated) OR (it is duplicated AND cloud is not enabled))*. In simpler words, if a CORE node is found not to be in Active state, there are two conditions that make the `guardian` call for a wheelchair stop:

1. If that node is not a duplicated one, meaning it is the only version of itself running in the system, it is the same as the original system, without any switching. Namely, that node must be always Active and if it is not, the driver must stop the vehicle.
2. If that node is duplicated, it could mean that the cloud system is being used, hence its cloud version is the one currently working. But if at the same time the `cloud_is_enabled_` flag is false, it means that the part of the system currently meant to be used is the local one, ergo that node should be Active. Also in this case, the `guardian` requests a stop of the wheelchair.

In all the other cases, the `guardian` must keep the system as it is, with the local duplicated nodes in Inactive state, as it is part of the switching solution's working algorithm to have inactive nodes in the system. Note that in order to state if the considered node is duplicated or not it is sufficient to see if its name is contained in the list of duplicated nodes.

- The same logical behaviour has been introduced in the part of the `guardian`'s algorithm in which it **checks the lifecycle state of all the nodes** in the list of managed ones (not only CORE), this time **in order to trigger their transitions from the Unconfigured to Active state**, which is the very role of the guardian (see Section 3.2.3). Namely, when a node is found Inactive, if not duplicated or if duplicated but cloud is not enabled, the `guardian` must trigger its state transition to Active. In all other cases, this node is meant to stay Inactive, as its cloud version is currently being used. It is the same exact logic than the one discussed above, but this time applied in the main part of the `guardian`'s algorithm, in which all nodes are checked in order to act on their lifecycle state.

These are the main changes applied to the local version of the `guardian` node (the only one currently existing in the *ALBA Robot* architecture), necessary to support the implementation of the switching mechanism developed by this thesis.

Note that these modifications do not upset the original working mechanisms of the `guardian`, as they just represent some additions built on top of them and compatible with the already existing base behaviours. In other words, some changes are necessary to support a local/cloud switching system, but they were thought not to represent a radical move from the architecture as it was before.

Also, the `guardian` is the only node that was truly modified in its hardcode, whilst any other one in the architecture was not touched. Also, two new nodes were added (together with their cloud versions). As previously stated, this solution aims to have the minimum impact possible on the existing architecture.

`remote_guardian`

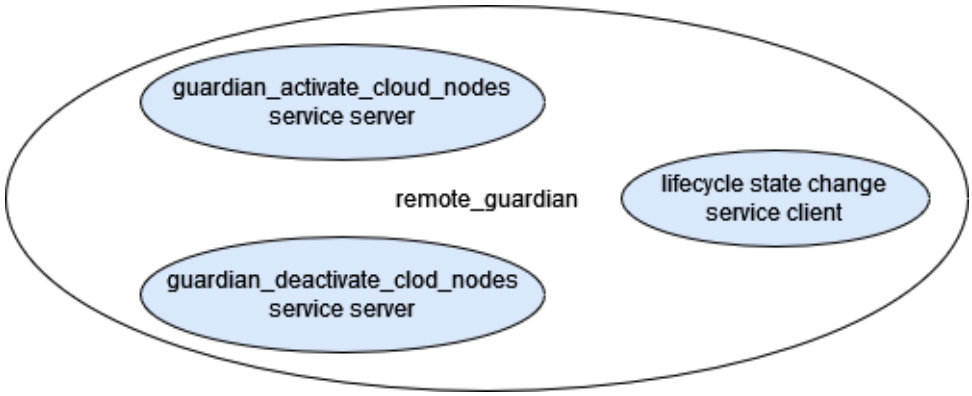

Figure 4.11: The `remote_guardian` node

Similar considerations can be done for the cloud version of the guardian, but not exactly equal. Even in this case indeed the working algorithm of the node is mirrored w.r.t. the local version, but some additions made in the local `guardian` would be useless here, and for these reason they've been avoided. Let's see in detail:

- In this case, no flag has been introduced to state if a managed node is duplicated or not: in fact, the `remote_guardian` handles the lifecycle of only the cloud versions of the High Level nodes, and they are all duplicated nodes by definition. In simpler words, the cloud nodes are a duplicated version of a subset of all the local ones, hence every node executed in the cloud server is for sure a replica of a node existing locally. Hence, **no flag is needed**.

  Of course, there is no guarantee that there actually is a local replica running in *SEDIA* for each node executed in the cloud server. Theoretically, there should be one and the switching will not work if something is wrongly configured, but no checking mechanism has been added by this work to warn the user that some local node is missing. The same can be said the other way around, as the system has no assurance that, if one node is declared to be duplicated, in the local `guardian`'s configuration file, its respective cloud version is actually there, as previously discussed. It'll be the developer's burden to manually configure the system in a proper way or creating some additional logic to check whether proper configuration has been applied or not.

- Also, the `duplicated_nodes_list` introduced in the local version of the guardian is useless here, for the same reasons. Indeed, **only one list is needed cloud-side**, as every lifecycle node the `remote_guardian` has to manage is a duplicated node by default (with the same considerations as before).

- **The `cloud_is_enabled_` flag is instead present**, because it will be useful in the `remote_guardian` working algorithm, in the same way as it was in the local one. The two guardians must also agree on the fact that the navigation system is using either the cloud or the local architecture, hence the two flags must be synchronized all the time.
- The **two services** declared here are called `guardian_activate_cloud_nodes` and `guardian_deactivate_cloud_nodes`, which respectively trigger the callback functions `activate_cloud_nodes()` and `deactivate_cloud_nodes()`. They have a very similar behaviour if compared to the ones of the local guardian; let's see the first one:

Algorithm 4 The `guardian_activate_cloud_nodes()` algorithm

1. If the list of managed nodes is empty, return. In this case, it means someone activated the `remote_guardian`, but without any other node in the cloud server, which is an odd but possible situation.

2. Then, for each node in the list of lifecycle managed nodes, trigger its `on_activate()` transition and set it in Active state. In this case, though, there is an additional consideration we must do: the `remote_cloud_switcher` and the `remote_network_monitor` must be exluded from this behaviour.

In fact, since there is only one list of nodes cloud-side, the ones composing the switching mechanism itself will belong to that list as well. They must not however be part of the nodes that are activated or deactivated when a switch is triggered: hence, an additional check is needed in order not to include these two nodes in the two functions.

In the local case, this problem is not present as these nodes will not be part of the list of duplicated ones (also conceptually they're not duplicated as copies, but they have a local and cloud version which are pretty different, even if similar in behaviour). That list must contain only the nodes of the actual base navigation system, not the switching mechanism.

3. Once activated all the cloud nodes, set the `cloud_is_enabled_` flag to true, as the cloud nodes are the ones currently used by the system.

The `guardian_deactivate_cloud_nodes()` function has again the opposite behaviour, namely it deactivates all the cloud nodes and sets the flag to false.

Also the changes applied to the logical behaviour of the `remote_guardian` have some sensible differences if compared to the ones of the local `guardian`:

- In the part of the `remote_guardian`'s algorithm in which the CORE nodes are checked to be Active and request a stop of the wheelchair if not, the only check to be done is that **the cloud must be active**. Indeed, if the cloud system is currently being used (`cloud_is_enabled_=true`), all the CORE cloud nodes must be in Active state: if even one is not, a stop of the vehicle is requested.

  Differently than in the local `guardian`, there is of course no check of the fact that the considered CORE node is duplicated or not, as in the cloud case all nodes are duplicated locally for sure, as previously said.

- The same condition is checked when the `remote_guardian` verifies the current status of each node in order to trigger its lifecycle state transitions: when a node is found to be Inactive, **if the cloud is enabled**, it must be activated. Indeed, that is the version expected to be used by the system in that moment, while if the cloud is not enabled the node is expected to be Inactive and it is left as it was found.

The solution to Problem 2

All the components described so far will cooperate in order to implement the switching between local and cloud nodes, in the way we talked about. This whole system, composed by the presented six nodes, constitutes the solution to Problem 2 adopted by this thesis.

To sum up the whole system, the final setup will be the following (as shown in Figure 4.5):

- The `network_monitor` and `remote_network_monitor` will exchange ROS2 *Header* messages in order to measure the network's status at each moment. Every time a message is received, they will publish either 0 or 1 in their respective topic.

- The `cloud_switcher` and `remote_cloud_switcher` nodes will listen to the network monitoring node in each respective side to take a decision about whether to trigger a switching in the system or not. This decision will be based on some logic, simple or complex, independent of the switching system itself. When a switching must be triggered, they will both contact their side's guardian node, asking to activate one side's nodes and deactivate the other side's ones, which is the way of physically implementing the switching.

- The guardian and `remote_guardian` nodes will physically activate or deactivate the nodes, according to the requests received by their respective side's

cloud switcher. Of course, each guardian node is in charge of only the lifecycle of the nodes belonging to its side of the system.

As previously discussed, the navigation system beneath will be **agnostic** to the presence of such switching mechanism, hence every node will continue publishing and listening to the same topics as before its presence.

Note: there is actually one modification, which is the only one needed for the nodes which compose the base architecture of *SEDIA*: when duplicating them, we need to **give every cloud node a different name than its local version**, in order to let the two guardians distinguish them. If we don't do that, the local `guardian` will try to manage the lifecycle of the cloud nodes and vice versa. This can be easily solved by launching the cloud nodes with a "`remote_`" prefix added to their local version's name: for example, the `map_server`'s cloud version will be called "`remote_map_server`".

An actual implementation of the overall architecture described has been realized in this work, but since it cannot work thoroughly with all the complexity brought by the *SEDIA* mechanisms, it was deployed in a simpler scenario, and every node was executed locally in the NX: instead of a `network_monitor` node, a `dummy_network_monitor` has been leveraged, which simulates a network failure for 4 seconds every 20 seconds, in loop, to be heard by the `cloud_switcher`. Its cloud version, the `remote_dummy_network_monitor`, basically does the same thing, but for the `remote_cloud_switcher` to hear. In this case, the *ALBA Robot* ROS2 system has been replaced by three simple nodes from the `demo_nodes_cpp` package: a local `talker`, a `remote_talker` and a `listener`, which hears every message published by the previous two. The system works smoothly, as a switch from cloud to local was triggered after 16 seconds from start, the system remained on cloud for 4 seconds, and a switch back was registered when network was simulated to be up again. Then, the whole process starts all over again in loop.

Chapter 5

Latency requirements

We have already defined, in the previous chapter (see Section 4.4.2) the way this thesis addresses the problem of knowing the status of the network at every moment: the network monitoring task is assigned to the `cloud_switcher` node and its remote peer, the `remote_cloud_switcher`. They continuously talk to each other in order to see if the network connection is still able to sustain the communication between remote ROS2 nodes (given that a way of making such nodes talk over remote networks has been defined and implemented).

It is not enough, though, to say that a switch from the cloud to the local system must be triggered when the network does not respond at all: of course, in that case a fallback to the local nodes is strictly necessary as the cloud ones are not even reachable, but it is a very loose logic, not capable of addressing all the navigation system's problems that can occur even if connection is not completely gone. In other words, **the moment in which connection is lost is not the only one in which a switch from cloud to local is necessary**. Similarly, having the network connection up again does not mean that a switching back to cloud must be triggered, as the navigation system could still behave better locally than in cloud.

This is why the nodes mentioned above exchange ROS2 *Header* messages, which internally have the information about the time at which they were sent: thanks to these messages, the two nodes can compute the latency spent to cover the (physical or virtual) distance that separates them and knowing such latency values is the first step towards a better switching logic. Of course, it is not enough, as we need to develop that logic, in order to have a resilient navigation system capable of finely addressing network malfunctioning: for instance, how high must that latency be to confidently say that the cloud nodes cannot be used? It would be very useful to have some **criteria to compute a threshold** which the `cloud_switcher` and `remote_cloud_switcher` can use to agree that a switching must be triggered. That logic will be implemented inside them.

5.1 The Nav2 stack's latency contraints

In order to understand the logical behaviour by which the `cloud_switcher` and `remote_cloud_switcher` nodes decide to trigger a switch from local to cloud or vice versa, we need to deeply investigate the nature of the navigation application running in *SEDIA*, which as we know is based on the Nav2 project. In fact, we do not need a lot of detail on how the Nav2 system actually works, but some information will be fundamental to address our problem of finding a switching logic.

As we know, we want to implement a distributed navigation system with a switching mechanism capable of providing the same exact service in cloud and locally, using the former when the network can sustain it and the latter when it cannot. Also, we said earlier that the complete network loss implies that no navigation can be fulfilled in cloud as the nodes running inside the cloud server will not be reachable at all: this situation surely calls for a switch to local. However, another possible situation is that **the network connection is up, but not strong enough** to sustain an efficient wheelchair navigation: what does this exactly mean? What must an efficient navigation system provide exactly?

One way of defining a navigation system efficient is that it must **provide a good-quality obstacle avoidance**. Indeed, among all the processes implied in an autonomous driving application, which could be paths precomputation, obstacle detection and many others, obstacle avoidance is the most safety critical one and it requires low latencies (relatively speaking) to be executed smoothly and safely. Recognizing objects or people along the path and forcing the vehicle to avoid them requires a lot of computational power, but above all very fast data processing, meaning that the data about the obstacle(s) acquired in the detection phase must be processed very fast and the system must respond to the presence of such obstacle(s) as rapidly as possible.

This brings us to an important consideration: **the latency of data transmission between the local and the cloud nodes must be such that Nav2 guarantees a good obstacle avoidance process**. Hence, in order to define a latency threshold to reach this level of safety when using the cloud, we need to understand how the Nav2 obstacle avoidance system works.

5.1.1 The Controller Server and the Local Costmap

The first component we must consider, in order to have a closer look at Nav2's obstacle avoidance process, is the **Controller Server**. It is one of the main components of the Nav2 stack and its role is to compute the local path used to reach the user-defined destination, following the steps of a global one precomputed

by the Planner Server. Once done this, the Controller gives the wheelchair the proper velocity commands through the `/cmd_vel` topic: they will be transferred to the Low Level by the `sedia_driver` node (see Section 3.2.2) and physically be executed, resulting in the wheelchair following the local path. Along this path, obstacles might of course be seen by the sensors and cameras installed in *SEDIA*, and the Controller surely must consider their presence in its processing and modify the current path to avoid them. This is the very core of the obstacle avoidance process: we must take in consideration the working algorithm of the Controller Server in order to develop a logic for our switching system, which is our final goal.

The Controller Server, as well as most components of the Nav2 stack, works following an **infinite loop**: it is agnostic to the sensors and cameras' detections as it does not directly contact them to obtain the information about the obstacles: it acts on its own, periodically producing its output and basing its computation exclusively on the information stored in the Local Costmap. Indeed, in order to compute the local path and generate the velocity commands as mentioned, it only reads the data already present in such map, coming from previous environment detections. This is done at the beginning of every iteration, which is performed following a timer: the Controller sleeps until this timer expires, and when it does it executes its algorithm using only the data found in the Local Costmap at that specific moment. This timer can be configured by the developer, in terms of frequency: currently, the Controller Server's frequency normally used by *ALBA Robot* is 20Hz, meaning that an execution of its working algorithm is performed every 50ms, but it can be changed according to user needs.

The **Local Costmap** is another fundamental component of the Nav2 stack: it does not represent the base map of the environment, as discussed in Section 2.2.1, but an additional layer built on top of it. In the Nav2 project, the dynamic information about the base map's environment is indeed defined as a set of layers added to such map, and the Local Costmap one is one of them, defined to make the system aware of the presence of dynamic obstacles: it is indeed a cost map, as its name suggests, meaning that it divides the base map in a grid of points and a value is assigned to every one of them. Each value represents the cost of passing over that specific point: the higher the cost, the more the vehicle should avoid it in the paths it follows.

This information is used by the Controller Server to compute the local paths, such as when the user wants the wheelchair to move from a point A to a point B the Controller computes the local path to reach that point with the minimum cost possible. This path can however change in the process, as the Local Costmap is constantly updated with the sensors and cameras' continue detections: every time they sense an object, or a person, or anything which is not part of the base map's components such as walls, the cost values in the Local Costmap are changed

accordingly, meaning that the points corresponding to the newly detected obstacle increases its cost. This change will consequently have an impact on the local path's computation, which has to go around the new obstacle now: this is the implementation of the obstacle avoidance process as it is defined by the Nav2 project.

It is important to repeat that **the two processes are independent**: the sensors and cameras installed in the vehicle do not know anything about the presence of an entire navigation system which uses their detections to make a vehicle move in a certain environment: they just perform their task, which is acquiring as much information as possible about that environment and publishing them in some topics. Only we know that those data will update the Local Costmap and will be used by the Controller Server to avoid obstacles. Similarly, the Controller is agnostic, as said, to the presence of such sensors: its task is performed using the data found in the Local Costmap, regardless of how such data were computed. This is the very strength of a modular architecture such as ROS2 (which Nav2 is based on), as the roles of the specific navigation system's components can remain separated and independent.

5.2 The switching logic

Now that we have a hint how the obstacle avoidance process has been designed by the Nav2 project(we do not need a deep knowledge on the details of such mechanism, as an overall view is sufficient to leverage its features for our goals) we can focus our attention on how we can actually use it to develop our switching logic. How are we going to define the latency threshold to tell the system to trigger a switch?

In this Section we assume that the entire Nav2 stack, or at least the Controller Server, is duplicated in cloud and the switching mechanism designed by this thesis has been implemented. In this way, the core Nav2 processes will be part of the system components executed by means of either cloud or local nodes, and the definition of the latency threshold will be based on this assumption. There are of course other ways of designing how to computing it by choosing a different logic, but this is the one adopted by this thesis.

The first question we should ask in this topic is: what is the minimum distance at which an obstacle can be avoided by the wheelchair? In other words, how far an obstacle must be to let the wheelchair not collide with it?

This question can be misleading, as one can think that a good latency threshold could be defined depending on the detected obstacles after they've actually been detected. In other words, we might think to consider the time the vehicle spends

to reach an obstacle it has already seen, and that time must be greater or equal than the one required to consider the obstacle in the navigation, namely the time from the obstacle's detection to its insertion in the Local Costmap (time at which it can be avoided by the navigation process).

This is definitely not the way to go, because the idea itself of switching to the local system after actually seeing an obstacle is meaningless: the switching mechanism will probably require more time that the actual process executed to avoid the obstacle. In simpler words, switching local if the cloud is not able to avoid an already detected obstacle means that the wheelchair will most likely not avoid the obstacle anyway.

The right thing to do would be to **prevent the detection of obstacles** instead and switch local if the cloud latency would definitely not allow to avoid them: in this case, we switch to the local system before an actual obstacle is detected, so that it'll be avoided for sure.

Hence, to establish a latency threshold we must consider another thing, namely the time required for the detected obstacle to be actually considered in the paths computed by the Controller. This time is the sum of three factors in the worst-case scenario:

- The time between two detections of one sensor.
- The latency required by the sensor's data to reach the cloud server.
- The time between two iterations of the Controller Server.

The first and the last factors are constant, given a fixed frequency priorly configured in the sensors and the Controller Server. Indeed, the sensors' publishing rate is always the same, as there is only one version of each sensor, while the Controller's frequency will differ between its local and cloud version: we'll configure it on purpose to be higher in the cloud server in order to leverage its virtually limitless computational power (it is the very goal of an autonomous driving system in cloud in the first place).

The gap between the local and the cloud system is therefore based on the differences in performance and latency, both higher in the cloud than locally. The key to define a latency threshold resides in what is described above: **the time required for obstacle avoidance in cloud must be smaller or equal than the time to do the same thing locally**. In order to do this, by considering the time intervals discussed, we can extrapolate the following inequality, which represents our logical switch triggering condition:

$$\boxed{t_s + t_l + {t_c}^{\text{cloud}} \leq t_s + {t_c}^{\text{local}} + tol} \tag{5.1}$$

Where:

- t_s is the time between two scans of one sensor. In fact, the contribution is the same in the two members, hence it can be simplified.
- t_l is the latency introduced by the cloud.
- ${t_c}^{\text{cloud}}$ is the time between two iterations of the cloud Controller Server.
- ${t_c}^{\text{local}}$ is the time between two iterations of the local Controller Server.
- tol is a tolerance value.

As expressed by this formula, the time for obstacle avoidance in cloud must be smaller or equal than the same time locally, plus a certain tolerance. The member at the right of the inequality is constant as soon as we set a certain value for *tol* and we configure the Controller's frequency. The first member is instead variable due to the value of t_l: this is the factor that determines the inequality during the navigation, as previously expressed. By satisfying this requirement, the cloud solution will always represent an improvement in terms of navigation if compared to the local one, as it will allow for a good obstacle avoidance process and consequently it will live up to any other navigation task, which surely has less strict latency requirements.

The value of tolerance *tol* has been introduced to address those situations in which the performance of obstacle avoidance in cloud is slightly worse than the local one: in this case, we shall choose not to switch to local, because it would represent a very little improvement for the system and the connection will most likely be up again to support the cloud. The switching process itself requires some time to be performed, as it consists in activating an entire set of nodes and deactivating the others in the meantime, hence as far as the cloud does not represent a substantial worsening than the local system, triggering a switching is not convenient.

How much should this tolerance be? In order to define a reasonable value for this parameter, we should consider the **minimum distance** that separates an obstacle from the wheelchair in order for the Nav2 system to be able to avoid it. Indeed, an object can be detected too late to be avoided, meaning that it is sensed by the sensors when it is too close and therefore will not be considered by the Controller in time.

This distance can be computed by looking at the same thing we considered so far: the time itself required for the wheelchair to detect and avoid an obstacle, which is a fixed value in the local system and a variable one in the cloud (it depends

on the latency, as mentioned). If the wheelchair takes less than this time to reach the obstacle, it won't avoid it. Such time can be converted to a distance assuming that the vehicle navigates at most at 0.5m/s. The resulting distance will be the minimum possible for the system to avoid that obstacle, and it will have different values in local and in cloud: the cloud requires indeed shorter times to process the obstacle, assuming that the network connection is strong enough and the latency remains consequently limited.

This distance can be used to define the value of *tol*: if the minimum distance results to be very small, we can have a greater tolerance, because in this case *SEDIA* has a sensitive obstacle avoidance process. If that distance is pretty big, we will need to have less tolerance. Of course, it is still an **arbitrary value**, not precisely defined: the developer can manually configure it depending on the specific situation, in terms of navigation environment and network connection.

Chapter 6

ROS2 and Kubernetes

The next step for the construction of our cloud autonomous driving architecture will be to add another technology that can be built on top of the ROS2 nodes composing the system, only cloud-side: that technology is **containerization**. Indeed, any application which has to be executed in a cloud server can be managed in a more optimized and cloud-native way by running it inside containers: ROS2 is very much prone to this technology as it is by itself a modular application, composed of single and independent units of computation. Each node acts as an independent entity, separated from the others, performing a specific limited task and cooperating with the others.

Once introduced such technology for the ROS2 nodes running in the cloud side of our system, it would be very useful to add a **container orchestrator** as well. The best way of managing the lifecycle and the operations of containers in a certain environment is indeed to assign this role to an orchestration technology: this thesis has chosen Kubernetes, as it is the most popular, efficient and optimized one, but any other can be used.

Hence, in order to understand how *SEDIA*'s ROS2 nodes can be run inside containers and how we're going to manage all of them with Kubernetes, first we need a general explanation of these technologies and how they work.

6.1 Linux Containers

The idea of developing a new technology such as Linux containers came with the need for**lightweight virtualization**: as we know, companies started using virtualization for their servers, such that they could divide the physical Hardware in separated and isolated Virtual Machines. Each VM will act as if it was an independent physical machine having just the part of the physical Hardware assigned to it by the Hypervisor: there is actually more Hardware available, but

the Hypervisor is in charge of hiding it.

This virtualization technology has the advantage of a strong isolation between the different VMs created in a physical machine's, as this isolation is backed by Hardware components. Also, each application can be executed in its own OS, which is a vanilla one, as the kernel of a host machine is exactly equal to the one of a guest.

Then, why lightweight virtualization? The need for a more agile and lean virtualization modality was driven by some key faults of the VMs technology:

- The overhead introduced by the necessity to execute an entire guest OS, in terms of memory and CPU used.
- The necessity to configure and keep up-to-date each instance of the guest OS.
- Booting time: a Virtual Machine requires a lot of time to be activated, just like a physical one, as an entire Operating System needs to be executed and it takes time to enable all its services.

Hence, the cloud world needed a much faster way of executing applications inside servers that could guarantee the nice properties of computer virtualization (scalability, elasticity, isolation) but that would consume less resources. Indeed, there are cases in which the overhead introduced by VMs in terms of resource usage and booting time is not acceptable. The answer to these necessities is lightweight virtualization, represented by the containers technology:

"A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings." (definition from the Docker official website)

In more technical terms, a container is a process executed inside an isolated space, but all containers share the same Operating System, which is the very same one of the host. From the host OS's point of view, the are just like normal processes executed in the machine, but if we have a look inside, they behave just like VMs: they have inside them all the components required to perform their tasks. Indeed, all the dependencies and libraries required for a containerized application to run are present inside the container itself, which looks like a complete system on its own.

This technology is made possible by process isolation and virtualization capabilities built into the Linux kernel, namely control groups *(Cgroups)* for allocating

resources among processes, and *namespaces* for restricting the access and/or visibility of processes into other resources or areas of the system. They enable multiple application components to share the resources of a single host OS in much the same way that a Hypervisor enables multiple VMs to share the CPU, memory and other resources of the same server's Hardware.

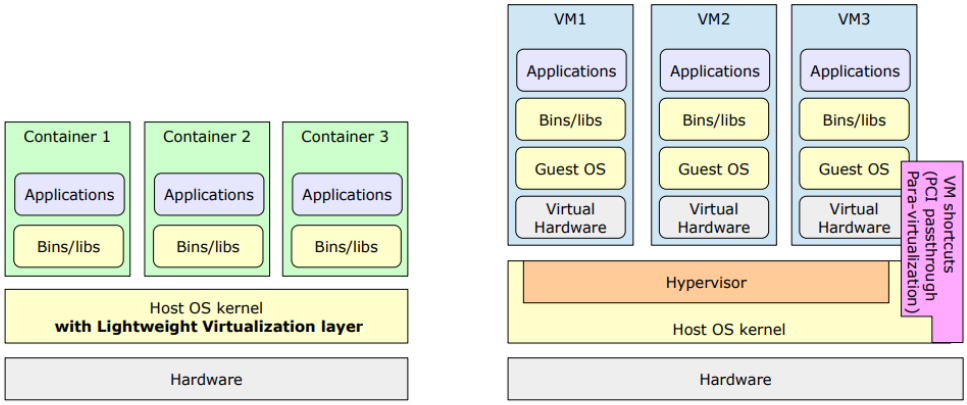

Figure 6.1: Containers vs VMs

Containers do not, however, emulate Hardware, as they can see the whole physical machine's available resources, only they cannot use them all. Due to the fact that they share the same kernel, there can be no Hypervisor hiding such resources to them: they can only be limited in terms of usage.

In short, containers allow a developer to **package up an application with all the components it needs to be executed**, such as libraries and other dependencies, and ship it all out as one package: using containers gives a significant performance boost and reduces the size of the application. One very important strength of containers is that they are 100% portable: they just need a containerization technology such as Docker (see next) to be executed, so that nothing has to be installed in the host machine.

6.1.1 Docker

When we want to deploy an application in the form of a container, the most commonly used and popular technology is **Docker**. It is an open-source platform for building, deploying, and managing containerized applications.

"Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share containers while you work and be sure that everyone you share with gets the same container that works in the same way."

In other words, it is a tool we can leverage when we want to containerize an application: it can use the isolation Linux capabilities to create containers in which such application will be executed, but it makes this process faster, safer and much more user-friendly. We do not need details on how the Docker technology works under the hood as we only need to use it for our aim, namely containerize ROS2 nodes and run such containers in our edge server. Hence, let's have a quick introspection on some of the main concepts introduced by this technology, which we will leverage for our goals:

- **Docker image**: An image is a read-only template with instructions for creating a Docker container. It is not a running instance of a container, but a static file containing all the indications Docker needs to create such instance and execute it.

- **Docker registry**: it is a repository in which Docker images are stored, in order to be available for the creation of a container. There is public registry that anyone can use, which is called Docker Hub and Docker is configured to look for images there by default. One can alternatively have his Docker images in his own private registry.

- **Dockerfile** : a *Dockerfile* is the way developers can create and build their own images: it is a file with with a simple syntax for defining the steps needed to create the image and run it.

- **Docker deamon**: a service that creates and manages Docker images, using the commands from the client. Essentially Docker daemon serves as the control center of your Docker implementation.

The fact that creating an image becomes very easy and user-friendly is represented by the modality defined to interact with it: the so-called Docker Client (a fundamental part of the Docker architecture, which we did not see in detail), defines a command line interface (CLI) which the developer can use to request a build, run, and a stop of the application to the Docker daemon. Some key commands are defined, which allow the users to build their own containers out of the applications they

developed, starting from a well-written Dockerfile, save that image in a registry (public or private), pull the image and have it running on Docker.

Specifically, let's see the main steps required to containerize a pre-existing application:

1. We start from the application itself, which could be of any nature, created for our specific use case. The first thing we need to do is to write a Dockerfile with the proper instructions. Further information about the syntax and key features of this file can be found in the official documentation. The Dockerfile tells the Docker deamon which application we want to save inside the new image we're creating, and to install all the required dependencies.

2. Once done that, we can build the image by means of the *docker build* command. This will create the image locally.

3. Then, we can push that image, with the*docker push* command, into the registry we wish to store the image into. If we do not have access to the repository, we can login by means of the *docker login* command: also, the image must be tagged in a proper way, namely with its name being preceded by the one of the registry, in order to be accepted.

4. The image is now ready to be executed (we stress the fact that it is not a running application, but a static file with only the instructions needed to actually execute it). Hence, we can pull it from the registry with the *docker pull* command to have it locally : of course, if already present, there is no need to pull it from a Docker registry.

5. Finally, the application can be executed with the *docker run* command, which creates an actual instance of a container with our application inside. From that moment on, that container will be treated as a normal process by the host machine's OS.

Of course, we did not see each single step of the process in the slightest detail: users can find online everything needed to overcome the problems found at every described step.

The following question to ask ourselves is: what if that application we want to run inside a container is a ROS2 node?

6.2 Containerizing ROS2 nodes

So far, we've seen the general way of using a technology such as Docker to containerize an application of any nature, but in our specific case we need to execute ROS2 nodes. Indeed, each node of the *ALBA Robot* architecture can be seen as a microservice on its own and a certain number of nodes cooperate to build a more complex modular application. We want each one of *SEDIA*'s cloud nodes which compose to be executed inside a container, hence we need to follow the same steps described above, with specific indications for our problem.

First of all, we need to start from a predefined image: which is the one best suited for us?

At a first glance, we could think that a feasible choice would be to start from an Ubuntu 20.04 image (the release of the Linux distribution which supports ROS2) and to install ROS2 and its dependencies in the resulting image. However, a much better choice is to start from an image which **already contains ROS2**: there are a lot of such images available in Docker Hub, but the one chosen by this thesis is the *ros* official image from Docker Hub.

The documentation gives a lot of information about this image: it is built on top of the official Ubuntu 20.04 image and ROS's official Debian packages, and it includes recent supported releases for quick access and download. It explains how to create a proper Dockerfile as well and run ROS2 nodes inside the resulting container.

As we know, the first step towards containerization of a ROS2 node is to create an image which contains everything needed by such node to be executed: to do this, we need a properly written Dockerfile. This thesis has developed a **Dockerfile template** for containerizing a generic ROS2 node which users can consult and adapt to their specific case, shown in Figure 6.2.

Let's have a look on the instructions presented in this file: the first thing we clearly notice is that it is divided in two parts and each one of them starts with a FROM statement. The way this Dockerfile is written follows a specific pattern, which is called ***multistage build***:

- The first block is meant to define a container which we only use for the process of **building the application** from its source code. In this case, we base on the *ros* official image taken from Docker Hub and add the node's source code in a specific workspace inside the container: such code is downloaded from the remote repository it is saved in, but it could also be taken from the machine's local storage and copied in the container's working directory by means of the COPY command. The <URL> statement at line 6 must be replaced by the actual repository URL, and the <PATH> represents the relative path of the specific package which contains the ROS2 node we want to containerize.

```
FROM ros:foxy AS builder
WORKDIR /home/<WORKSPACE>/src
RUN mkdir alba_high_level && \
    cd alba_high_level && \
    git init && \
    git remote add -f origin <URL> && \
    git config core.sparseCheckout true && \
    echo "<PATH>" >> .git/info/sparse-checkout && \
    git pull origin <BRANCH> && \
    cd ../.. && \
    . /opt/ros/$ROS_DISTRO/setup.sh && \
    colcon build --symlink-install --packages-select <PACKAGE_NAME>

FROM ros:foxy
WORKDIR /home/<WORKSPACE>
ENV RMW_IMPLEMENTATION=rmw_fastrtps_cpp \
    ROS_DOMAIN_ID=<DOMAIN_ID>
COPY --from=builder /home/<WORKSPACE>/build /home/<WORKSPACE>/build
COPY --from=builder /home/<WORKSPACE>/install /home/<WORKSPACE>/install
RUN sudo apt update && \
    apt-get install -y psmisc && \
    . /opt/ros/$ROS_DISTRO/setup.sh && \
    rosdep install -y -r -q --from-paths build --ignore-src --rosdistro $ROS_DISTRO && \
    sed --in-place --expression \
      '$isource "/home/<WORKSPACE>/install/setup.bash"' \
      /ros_entrypoint.sh
CMD ["ros2", "run", "<PACKAGE>", "<EXECUTABLE>"]
```

Figure 6.2: Template for containerizing ROS2 nodes

Indeed, in Figure 6.2 a specific paradigm is used, which consists of writing the node's repo path in *.git/info/sparse-checkout*: this allows to avoid cloning the entire remote repository in the container, possibly with a lot of other nodes. Once done that, we source the workspace with the `ROS_DISTRO` we installed (*foxy* in this case) and build the specific package which contains our node with the *colcon* tool. At the end of this stage, we have both the source code and the `/build`, `/install` and `/log` directories in our container, but the only things actually needed to run it are `/build` and `/install`: all the rest was used for the sole purpose of producing them.

- Then, we create another container from the same base image. First, we export the environment variables we need, namely the ones that define the middleware that will be used by this node and in which `ROS_DOMAIN_ID` we want to execute it. Then, we can take the `/build` and `/install` directories from the previous container (called *builder*) and put them in the WORKDIR of the newly created one, with the COPY statement. Now we're free to proceed by installing all the dependencies needed by the node to be executed by means of the *rosdep* tool, which is a very useful application (already installed in the base image) that automatically detects all the dependencies declared in the node's package and

installs them one by one. We source the workspace with the *sed* command (suggested by the official Docker documentation) and declare the entrypoint of the node, which will be the command executed inside the container when we launch it with *docker run*. In our case, this command is *ros2 run*, followed by the names of the ROS2 package and the node's executable, which must be properly specified in this Dockerfile.

We stress the concept that these lines are not actually executed until we call the *docker build* command. When we do that, Docker will create these two containers in order, following the specified instructions. Only the second container's definition will be actually saved into the image, and this represents the very strength of the *multistage build* pattern: instead of storing all the source code, the dependencies and the actual built package(s) inside the container, we only limit to what is needed for its actual execution, namely the result of the *colcon build* command and the dependencies. In this way, a more lightweight image will be created, removing all it is useless for the actual execution of the container.

Note: this represents only a template that can be consulted when a user wants to containerize a ROS2 node for its own application, but some things can be freely added and/or changed in order to adapt it to the specific use case. For instance, if some dependency is not detected by *rosdep*, it must be added manually in the second stage's RUN statement, or if the users wish to execute the node through a launch file they must add such file in the final container and change the CMD entrypoint to *ros2 launch*.

Once created the proper Dockerfile for our use case, the steps to follow in order to actually execute the ROS2 node as a container are exactly the ones described in Section 6.1.1.

6.3 Kubernetes

Once we understood how to execute our ROS2 nodes inside containers, we can have a step forward and add a **container orchestrator**. Let us suppose that all the nodes we want to run in the cloud server have been stored inside properly built images, as explained so far: this means we're able to execute the cloud part of our autonomous driving architecture inside separate containers, which can talk to each other if deployed in the same LAN and configured with the same `ROS_DOMAIN_ID`. They could be made capable of communicating across remote LANs through the usage of a DDS Router instance (see Section 4.3.3), but it won't be necessary in our case, as all cloud nodes will be executed in the same edge server, hence in the same LAN.

However, managing an entire architecture composed by a high number of nodes requires some additional effort: containers must be handled in a proper and structured way, as on one hand we need to make them cooperate to reach our application's goals, while on the other hand we need resiliency and fault tolerance. Executing independent and separate containers is not enough to have these features at our disposal, hence the best and most structured way of managing an entire set of containers is to use an orchestrator, as said. Basically any tool with these capabilities can be used: Docker Swarm is an example, developed in this case by the Docker community itself, or Apache Mesos, but this thesis has chosen **Kubernetes**, which is the most efficient and commonly used one, with a lot of optimized features and an user-friendly interface.

As the official documentation states, Kubernetes (pronounced *coo-ber-net-ees* and abbreviated to K8s) is *"a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available."* In a production environment, the user needs to manage containers ensuring that there is no downtime, hence a new container is started when one goes down. Kubernetes is none other than a framework to run distributed container systems resiliently and it takes care of scaling and application failover, provides deployment patterns, and more.

Of course, this technology has much more potential than the one presented in this work. We will not describe every possible feature of Kubernetes in detail, and if the reader needs some deeper introspective, the official documentation can be consulted. In this section, we limit to present the concepts which are useful for our autonomous driving goals:

- **Kubernetes cluster**: it is a set of worker machines, called **nodes**, that run containerized applications. The Kubernetes architecture defines different roles for different nodes, following a master-slave pattern in which the master is the node executing the core of the orchestration logic and holding the interface that lets the user interact with the cluster: the slaves are the ones actually executing the containerized applications blindly following the master's instructions.
- **Kubernets API**: The core of Kubernetes' control plane (the master mentioned above) is the API server: it exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another. It lets you query and manipulate the objects‘ state in Kubernetes, and defines some ways through which we can interact with the cluster: one of them is the kubectl CLI.
- **Kubernetes objects**: they are persistent entities in a K8s cluster. Specifically,

they can describe what containerized applications are running and on which nodes, the resources available to those applications and the policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance. We interact with K8s by telling it to create specific objects basing on the container architecture we want to manage, and we can do this through the API server. The Kubernetes control plane continually and actively manages every object's actual state to match the desired state the user supplied.

- **Pod**: Pods are the smallest deployable units of computing that you can create and manage in Kubernetes, and the most important basic K8s object. A Pod may be composed by one or more containers, with shared storage and network resources, and a specification for how to run such containers. A Pod is mandatorily executed on a single node and it represents in K8s what a process is in an OS.

- **ReplicaSet**: its purpose is to maintain a stable set of replica Pods running at any time. It is indeed often used to guarantee the availability of a specified number of identical Pods for resiliency reasons. The idea is to have a "controller" that checks continuously that a certain amount of replicas are up and running.

- **Deployment**: usually, users do not directly manage ReplicaSets, but rely on a higher-level object, which is the Deployment. This object encapsulates the declaration of a ReplicaSet of Pods, giving it additional metadata: users basically only declare this object to tell the K8s control plane to deploy a resilient set of Pods to run their applications.

- The **K8s network** model: every Pod in a cluster gets its own unique cluster-wide IP address. This means users do not need to explicitly create links between Pods and they almost never need to deal with mapping container ports to host ports. Kubernetes imposes the following fundamental requirements on any networking implementation:

  – Pods can communicate with all other pods on any other node without NAT.

  – Agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node.

  These requirements must be supported by an external network provider, which has to follow the Container Networking Interface (CNI) specification. This is a minimal specification for adding and removing containers to networks, which are the basic operations any network provider must implement. The CNI defines the interface between container runtime (e.g., Docker) and plugins,

and such plugins enable the Kubernetes' networking model used. Example of providers are Calico, Flannel, Romana, Cilium, etc...

- **Service**: it is an abstract way to expose a stable access point to an application running on a set of Pods as a network service, regardeless of their location or number. Kubernetes Pods are nonpermanent resources and in a Deployment, as a set of Pods running at a certain time could be different than the one being running a moment later. For this reason, we need a Service, which is an abstraction defining a logical set of Pods and a policy by which to access them. It doesn't matter if some are replaced by others with new IP addresses: the logical grouping defined by a K8s Service will take care of redirecting the requests to the intended ones and this process will be transparent to the user.
- **ConfigMap**: it is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume and mainly use them to set configuration data separately from application code. A ConfigMap allows to decouple environment-specific configuration from container images, making applications easily portable.

All these concepts are going to be useful in our considerations towards the goal of having the cloud side of our autonomous driving application running inside a K8s cluster. However, before actually deploying our ROS2 nodes in such cluster, there is another important thing we must mention, described in the next section.

6.3.1 ROS2 Discovery problem

We already know that the DDS middleware has its own peculiar ways of implementing communication between ROS2 nodes: for instance, the SIMPLE discovery mechanism of DDS allows nodes inside the same DDS network to be automatically discovered and put in mutual communication only by means of multicast discovery traffic. As previously discussed, a DDS Network does not always have a 1:1 match with a LAN, as it is also defined by the DomainID (`ROS_DOMAIN_ID` in our case), the Transport protocol (TCP/UDP) and the Discovery mechanism used in the network (SIMPLE, Discovery Server, etc...).

Generally, multicast is supported inside the same Local Area Network, exactly as broadcast traffic, hence all the intended destination nodes can be reached and they will receive and recognize the traffic published in the active topics. But what if multicast cannot be used?

This is exactly the case of most cloud environments: often, inside a Kubernetes cluster, the CNI used (e.g. Calico) **does not support multicast**, and this represents a problem for our system, as the first step to build our ROS2 cloud architecture

is to have running nodes who can be discovered and put in communication. Since it is not statistically possible to assume that we have full control over the Kubernetes cluster we're using, hence we cannot always implement whatever CNI supports multicast (it is the case of this thesis), how can we implement ROS2 communication inside K8s? In other words, how can we let a ROS2 node running inside a Pod listen to the data published by another node inside another Pod in the same cluster?

The solution is provided by eProsima itself (the Fast DDS vendor as previously mentioned), and it is the **Discovery Server** Discovery mechanism, an alternative to the basic SIMPLE one. The Fast DDS Discovery Server provides a client/server architecture that allows nodes to connect with each other using an intermediate server: each node functions as a discovery client, sharing its info with one or more servers and receiving Discovery information from it. This reduces discovery-related network traffic, and it does not require multicasting capabilities. Figure 6.3 compares a Discovery phase performed using the Fast DDS SIMPLE protocol with the one we have when a Discovery Server is deployed for the same reason.

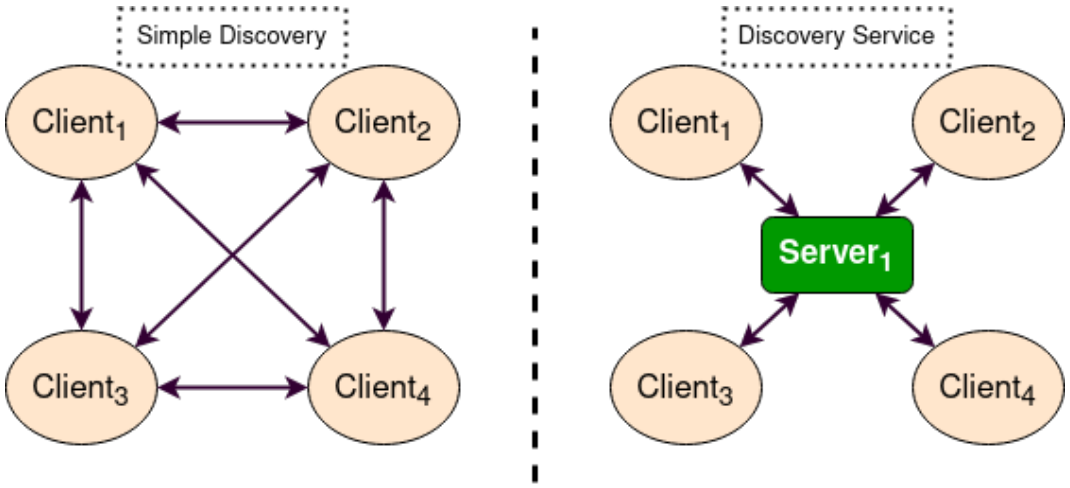

Figure 6.3: Fast DDS Discovery Server

In few words, this paradigm **avoids the use of multicast traffic for the Discovery phase**. It calls for **unicast** Discovery traffic coming from every node, which will be delivered to this central server and not directly from one node to all the others. Then, the server will distribute such traffic itselfto let the DDS middleware discover all nodes in the network, which is defined by the usage of another Discovery mechanism (as well as a DomainID and a Transport Protocol). Also, more than one Discovery Server can be used in the same DDS Network, in order to increase the resiliency of the application, since a central server for

Discovery is a *SPoF* (Single Point of Failure).

The Discovery server, though, does not come pre-installed with the Fast DDS middleware, hence any user who wants to use it must download and build it on its machine priorly. In our case, we will see that there is no need of actually installing a Discovery Server in our cloud system based on K8s, as it will be already included in the DDS Router installation.

6.3.2 ROS2 remote communication with Kubernetes cloud-side

Once we have clarity on how to produce the Docker images for the ALBA Robot architecture nodes and how we can execute such nodes inside Kubernetes Pods, the next and final step is to **link the cloud and local parts of the system together**, through ROS2 communication over remote LANs. Let us suppose, in this sense, that we have selected all the nodes feasible of being executed in the cloud, created an image for each of them and pushed it to Docker Hub or any other private or public repository (that we have access to of course), and started a K8s cluster in the cloud servers we have at our disposal (if only one, we can leverage some tool to virtualize the main K8s nodes into one server, like *Minikube*). How can we make the cloud and local nodes talk to each other in this case?

We already presented in Section **??** the solution proposed for ROS2 remote communication, which consists in leveraging the capabilities of the eProsima DDS Router. However, in those considerations we have described only nodes running in "normal" Linux environments, not executed on top on any cloud-native architecture: the deployment of the DDS Router was therefore pretty simple and symmetrical. We have already mentioned, though, that a **different configuration** must be applied cloud-side if a container orchestrator or other forms of cloud-native platforms are deployed: which configuration, precisely? The DDS Router official documentation by eProsima provides this answer straight away, as they face the same exact problem.

"Apart from plain LAN-to-LAN communication, Cloud environments such as container-oriented platforms have also been present throughout the DDS Router design phase": hence, eProsima themselves already took care of the integration of their technology with cloud platforms such as container orchestrators. The differences in configuration w.r.t. a non-cloud platform are not many, but some key additional considerations must be done if our nodes are running inside K8s Pods (it'll be the same if we're using different cloud technologies, such as Amazon Web Services or others). In the example presented, both a Kubernetes network and a local environment are set up in order to establish communication between a pair of ROS nodes, one sending messages from a LAN *(talker)* and another one

(listener) receiving them in the cloud. This can clearly be extended to any number of nodes, therefore we will talk in general terms and focus on the important things, which are the DDS Router's configuration details.

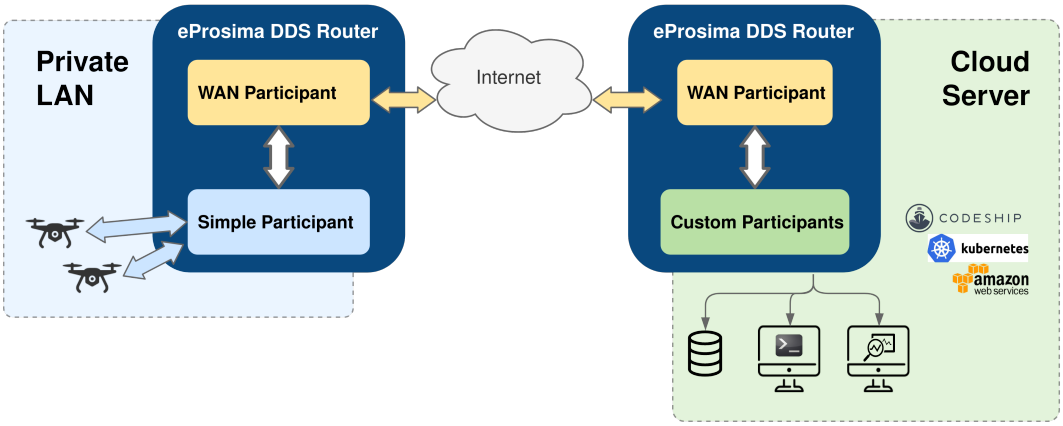

Figure 6.4: WAN communication with the DDS Router

Figure 6.4 (which we already presented in Section 4.3.3) shows the overall setup to let local ROS2 nodes reach cloud ones executed in a remote server (the internet network is in the middle), inside K8s Pods:

- The local instance of DDS Router only requires a **Simple Participant** and a **WAN Participant** that will play the client role in the discovery process of remote WAN Participants. After having acknowledged the existence of other nodes in the local private LAN through the SIMPLE DDS Discovery mechanism (multicast), the local participant will start receiving messages from those nodes and forward them to the local WAN Participant. Following, this Participant sends such messages to its remote peer, hosted in the K8s cluster to which it connects via WAN communication over UDP/IP. We already know this mechanism from previous sections.

- Cloud-side, two different Participants are required in the DDS Router configuration, each in a different K8s Pod:

  – A **WAN Participant** that receives the messages coming from the local LAN through the mentioned UDP communication channel.

  – A **Local Discovery Server Participant** that propagates them to any ROS2 node hosted in other K8s Pods.

  Indeed, the "Custom Participant" shown in Figure 6.4 (as already said in Section 4.3.3) is actually a Local Discovery Server one, which addresses the

difficulty of enabling multicast routing in cloud environments, discussed in the previous section.

Of course, the Pods in which the DDS Router and all the cloud nodes are executed will be obtained through K8s Deployments: furthermore, for every Pod, a K8s Service is required in order to direct dataflow to each respective one. Indeed, a *LoadBalancer* Service will forward to the WAN participant all messages reaching the cluster and a *ClusterIP* will be in charge of delivering those messages from the Local Discovery Server Participant to any other Pod.

These are the base concepts needed to implement a ROS2 architecture in which remote nodes must communicate with each other, and the cloud side of the system runs inside a K8s cluster (or any other cloud-native platform). In this section we did not go in the slightest detail on how to produce the proper Kubernetes configuration files needed to deploy the described Pods and Services, as the official documentation of the DDS Router explains it thoroughly. Also, the resulting system will be **agnostic to the possible deployment of a switching mechanism** such as the one described in Section 4.4, as that will be built on top of this architecture and it only uses ROS2 nodes to implement its working algorithm, assuming they can freely communicate.

Chapter 7

Testing the system

At this point, we have a complete view on the solution adopted by this thesis to enable an autonomous robot transparently access services locally and on an edge server. This chapter is meant to describe some test performed in order to acquire some data to validate the proposed approach.

First of all, we need to give a brief description of the **testing environment**, which is a crucial element for the determination of the actual registered latencies: the position of the cloud server w.r.t. the wheelchair is one key factor to define which values are in play, but not the only one. This distance is to be intended in network terms, hence in which networks the server and the vehicle are located, if the public internet has to be trespassed in the communication between the two entities and how many hops separate them. We need the lowest latency values possible, hence we need the server and the vehicle to be as close as possible. Another key element, though, is the time required for data processing, which can be very heavy dependeing on the applicaton used.

There is an important thing to specify: none of these tests are going to include the actual wheelchair of the *SEDIA* project together with the switching mechanism described in Section 4.4. Such switching system is indeed not ready to address the problems arisen when we create duplicates of a whole set of nodes of the High Level architecture and execute them in the cloud while their replicas run locally. There are issues that must be faced first in order to have a smooth result, and they're treated in this thesis as future work to be done when any developer wants to implement a real cloud autonomous driving solution out of this work (see Chapter 8). For now, we limit our testing to some simple configurations.

7.1 The testing environment

The project this thesis is inserted in includes the usage of a cloud server, belonging to *TIM* and located in a datacenter in Turin, Italy. This work was authorized to leverage this specific server's functionalities through a web platform developed by *Capgemini*. Let's see the component of the experimentation in order:

- The local side of the tests is represented by the ***Nvidia Jetson Xavier NX*** SBC, which we talked about in Section 3.1.2. This is the board *ALBA Robot* were currently running their High Level architecture into at the time of this thesis, hence it represents a good candidate for the base platform in which our local nodes run.

- The NX board is connected to the network of the server through a **5G router** provided by *TIM* for these experiments. In few words, this network is not public: it is in fact a 5G network belonging to *TIM* itself, with private IP addresses assigned to any device connected to it. Hence, the router has been configured with a specific APN to reach that network and receive an address in that pool, which is 10.193.0.0/16.

  Our NX board has to be connected to such router, operation that can be done in Ethernet or Wifi. Since using the Wifi board embedded in the NX kills its performances, it was preferred to use Ethernet to connect it to the 5G router, which uses DHCP to assign private in its internal network, which is 192.168.225.0/24. Of course, it exposes a NAT to allow the internal traffic reach the outside world, which in this case is the private *TIM* 5G network. Needless to say, a 5G coverage must be available in the location the router is acting, otherwise it won't be able to reach the server.

- At the other side, the **cloud server** used in these experiments has an interface in the already mentioned network, with an address in the 10.193.0.0/16 pull. A **Kubernetes cluster** has already been created among a certain number of servers in the datacenter, but the user cannot have direct access to it. There is only one entrypoint to such cluster, which is the a web server exposing a certain port. *Capgemini* has in fact developed an UI (User Interface) to allow a developer deploy Kubernetes Pods in the underlying cluster, by abstracting the presence of K8s itself.

  Figure 7.1 shows a screen of this interface, in which two main operations are permitted:

  – **Onboarding**: it is the process by which a Docker image (or any other form of containerized application) can be fetched from a remote registry, either public or private, and stored in the datacenter's internal registry.

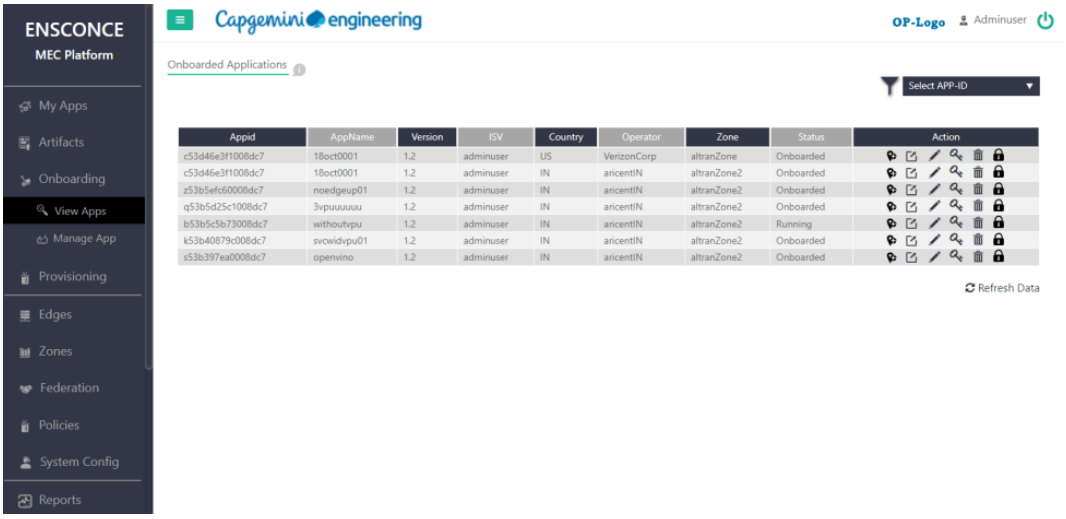

Figure 7.1: *Capgemini* UI for K8s cluster management

These images are not yet instantiated, of course. The Onboarding process requires some key specifications, namely which port(s) will be exposed by the corresponding container, some possible environmental variables that must be exported when instantiated, in which region and zone this container must be executed, the amount of RAM and number of CPUs to be reserved for it, some metadata about the type of the containerized application, as well as of course the registry the image has to be fetched from (with proper credentials for authorizations).

– **Provisioning**: once Onboarding is complete, the Provisioning process consists in the actual creation of a Kubernetes Pod for the corresponding onboarded image. The system automatically assigns an AppID to every provisioned application, and is logs can be viewed in the Troubleshooting section. As said, these processes abstract every Kubernetes behaviour, and the user is nearly not aware of the presence itself of a K8s cluster underneath. There is no possibility of interacting with such cluster by means of the K8s CLI (kubectl), as the only connection point with it is the *Capgemini* interface itself.

Considering this whole setup, some tests have been prepared.

7.2 Test1: registered latencies for message transmission

The first kind of test focuses on computing the values of **latency** required to send a certain set of messages between ROS2 nodes located in different spots, namely the time that passes from sending each message to receiving it at its destination. Specifically, the following set of nodes have been created (written in C++ using the ROS2 Client Library):

- **10 publisher nodes**, each one of which sends a message of a certain data type every 100ms for 1 minute, resulting in 600 messages in total. Each node uses a specific topic, which is tied to its messages' data type, as we already know. Every message type used by these nodes has a *Header* structure in its definition, in which two fields (`message.header.stamp.sec` and `message.header.stamp.nanosec`) are reserved for the representation of a timestamp: these fields are set by the publishers with the measurement of the time at which each message was sent.
- **10 subscriber nodes** are then coupled with the publishers, each listening to the respective peer's topic. The role of every subscriber is to acquire the 600 messages published from the other side of the topic, read their `message.header.stamp.nanosec` field and compare each of them with the current time. The time value required to send a message from a publisher to the crespective subscriber is obtained by subtracting the time value found in the *Header* to the current time.

Actually, for the scenario implying latency computations between the server and the NX (Scenario 5) a different approach has been used, due to the non-synchronization of the clocks of the two machines. In this case, publishers send the same type of data to the subscribers while internally saving the current time: subscribers do not perform any computation and just send a different message back as soon as they receive one from the respective publisher. When this message arrives, the receiving publisher computes the difference between the previously saved time and the current one, and divides the result by two (it was a round-trip exchange and we only need the latency for a one-way sending). This algorithm provides a way of avoiding the mentioned synchronization problem, as times are not retrieved in the message themselves, but computed by only one of the two peers of a publisher/subscriber couple. The same approach is used in Scenario 4 (server only).

The data types used by these nodes are chosen among the `sensor_msgs` and `geometry_msgs` packages, as they represent the kind of messages most commonly used by the actual High Level nodes of *SEDIA*. In this way, the computation of

the required latency for their sending will be reportable to the real use-cases in which *SEDIA* works.

These two sets of nodes will be executed at the same time in different scenarios: in the same machine or in different ones, in the same network or in remote ones etc... At the same time, a system inspection has been done in the machine(s) in which these nodes have been executed, by means of the *mpstat* tool, in order to retrieve information about the CPU usage during the whole time those data were exchanged (1 minute). These tests have taken in consideration the `%usr` percentage in the `mpstat` command output (ROS2 is seen as a user application by the system of course), and the overall value for all the CPUs present in the machine, which is defined by the tag *all* in the same output.

Scenario 1: latency on NX

This is the simplest scenario, in which all 20 publishers and subscribers are executed at the same time inside the Nvidia Jetson Xavier NX. We expect the corresponding latency to be very low, as the communication between nodes does not have to go out of the physical machine's Hardware. Lower latency values can only be possible if we computed them inside a more performing machine, such as a server.

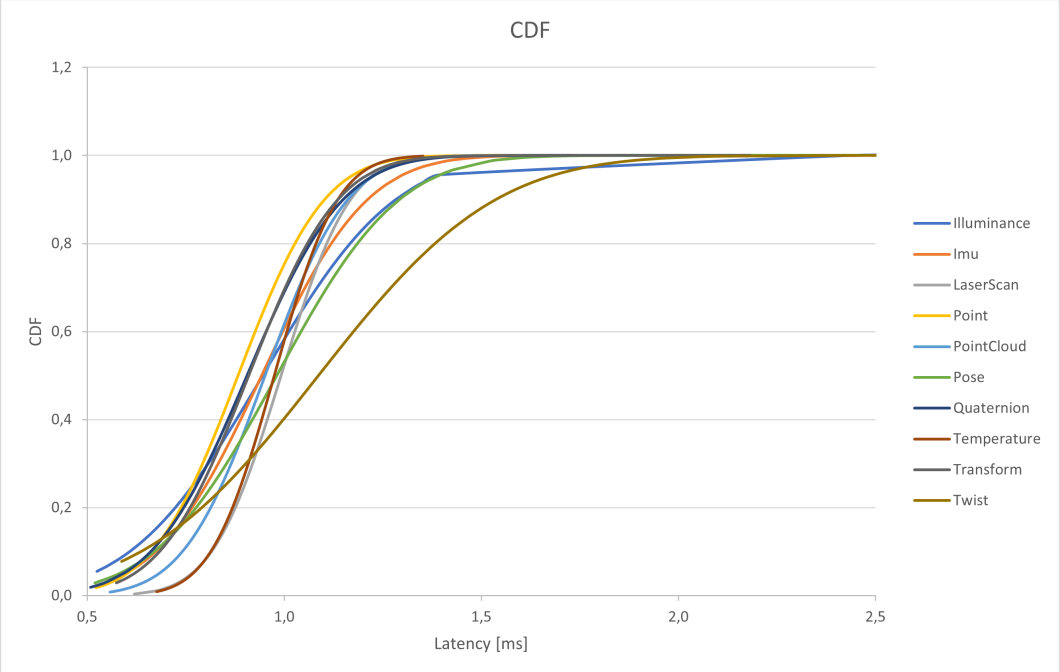

Figure 7.2: Latency on NX

Figure 7.2 represents the CDF *(Cumulative Distribution Function)* for the data

collected in this scenario. Every curve represents a different kind of messages exchanged, namely a different publisher/subscriber couple. The graph can be interpreted in this way: each CDF value in the `y` axis represents the probability that the latency required to send the message in this scenario is less than the corresponding value in the `x` axis. For instance, the probability that this latency is less than 1ms is about `60%`.

As expected, the computed values are quite low, because nearly all our data are sent and received in less than 1,5ms. Only one curve (the one corresponding to the Twist messages) slightly differs from the others, perhaps due to the different nature of the considered messages. In fact, it is expected that different kind of messages require different times for their sending, because of the gap in their processing times: these times are comparable, but some differences can be clearly seen. However, these differences can be aleatory, as they may depend on the NX's computational load at the time of the experiment.

Following, the CPU usage of the NX is shown in Figure 7.3: we can see that its range goes from nearly 26% to 27%, meaning that the execution of even 20 nodes at the same time does not have a strong impact on the board's performance.

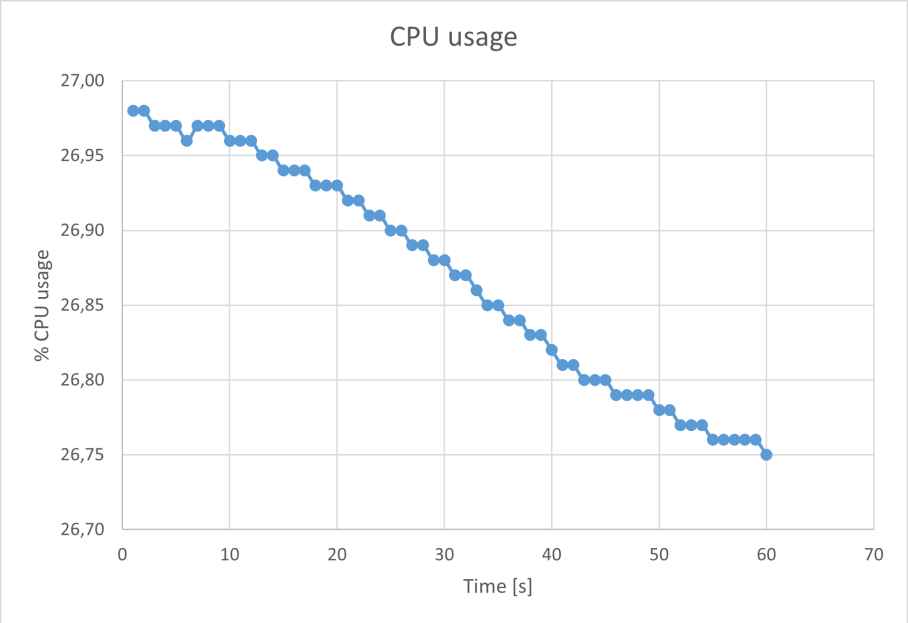

Figure 7.3: CPU usage on NX

Scenario 2: Latency on NX with the eProsima DDS Router

Our final goal is to have ROS2 nodes communicating between remote LANs through an instance of the eProsima DDS Router, as explained earlier in this thesis. In

order to see if the processing time brought by the instantiation of such additional element has an impact on the system's performances, and consequently on the latency required to send some messages, a scenario with a local instance of the DDS Router has been considered. Specifically, we leverage an instance in the NX board to make our 10 publishers communicate with our 10 subscribers over different ROS2 domains (see "Change Domain Example" in the DDS Router documentation).

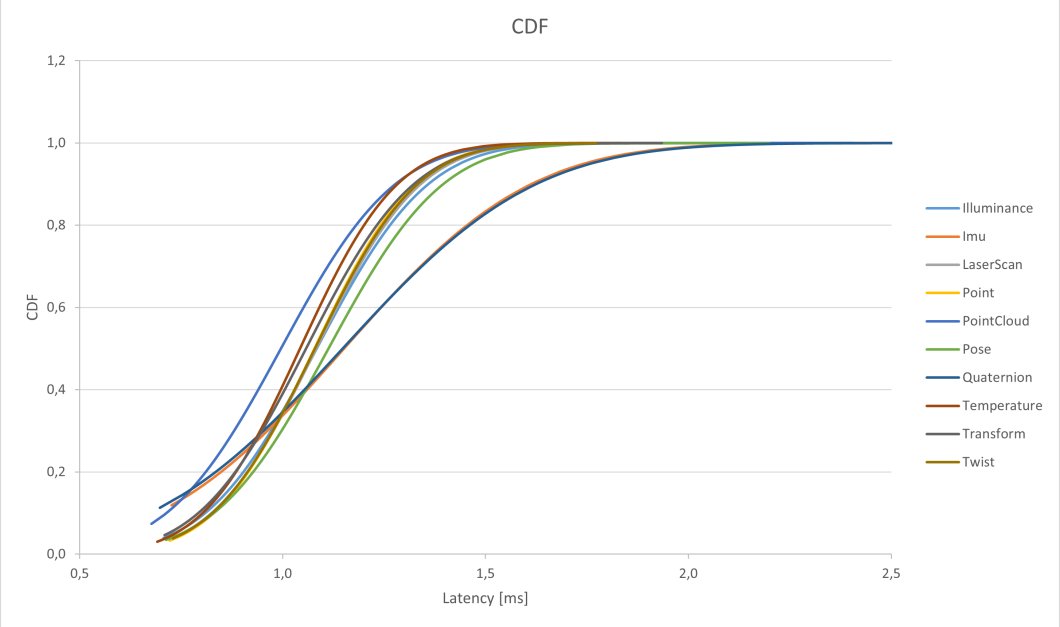

Figure 7.4: Latency on NX with DDS Router

Figure 7.4 shows, as usual, the CDF of the latency values collected in this scenario: if we compare with Scenario 1, in which the same data were exchanged inside the same exact machine, we basically see no difference in trend and obtained values, other than a slight shift to the right for all curves. This means that a certain amount of time is added by the DDS Router processing in the message exchange, but it is less than 1ms overall and the order of magnitude remains exactly the same.

In other words, we can clearly see that the presence of the DDS Router does not mean much addition in the latency computation, only few microseconds that could be required, for instance, to forward the message between its internal participants (even though a zero-copy mechanism is implemented for this) or for some sort of message processing. Let's keep in mind, however, that this is only a Change Domain instance, with two Simple Participants to make ROS2 nodes talk over different `ROS_DOMAIN_ID`s. The usage of a WAN Participant, as we'll see, might be very different.

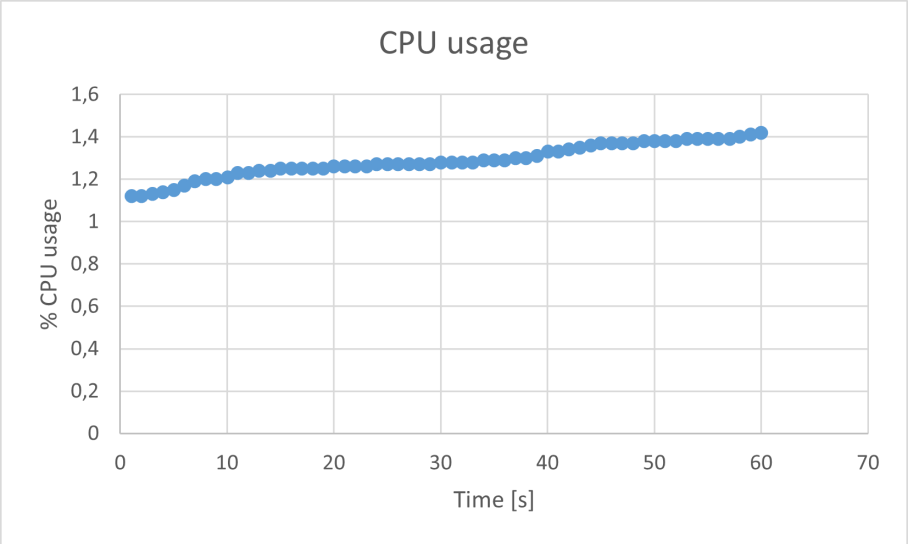

Figure 7.5: CPU usage on NX with DDSRouter

The CPU usage on the NX is pretty much constant, as it goes from nearly 1% to 1,4%. These values, though, are unexpectedly low, which could mean that the ones registered in Scenario 1 were randomly high due to some other process running in the machine at the same time.

Scenario 3: Latency between NX and a laptop

In this case, the same nodes are launched in two different machines: one is again the *Nvidia Jetson Xavier NX*, while the other is a laptop with Ubuntu 20.04 (the same running in the NX and the one supporting ROS), in which ROS2 is installed. Here, the goal is to see how the latency values differ if messages exit one machine and enter another, passing through an L2 network. Not surpassing the LAN boundaries should maintain the computed latency pretty low, but still, it should be higher than the previous scenario.

Figure 7.6 shows the CDF of the latency computations obtained by executing the 10 publishers in the NX and the 10 subscribers in the laptop. As expected, the obtained values are an order of magnitude higher than the previous scenarios: all of them are less than 12ms, and the majority goes from 8 to 9ms. In this case, no particular difference is seen between the curves: this means that the diverse behaviours registered in the previous scenario could have been random, probably caused by some different computational status of the board.

The CPU usage of the NX and the laptop are shown in Figure 7.7 and 7.8 respectively. The values registered in this case are pretty low in both machines: the NX has a much lighter computational load than before, as it is running half

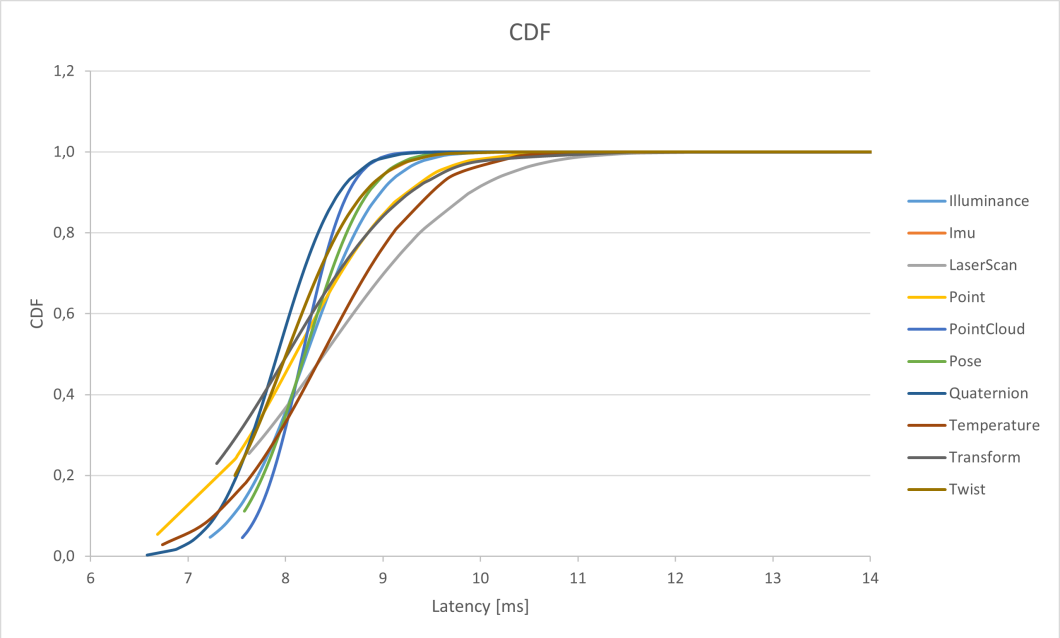

Figure 7.6: Latency between NX and laptop

of the nodes (the publishers), and similarly the laptop is running the other half, having also a much more performing Hardware at its disposal.

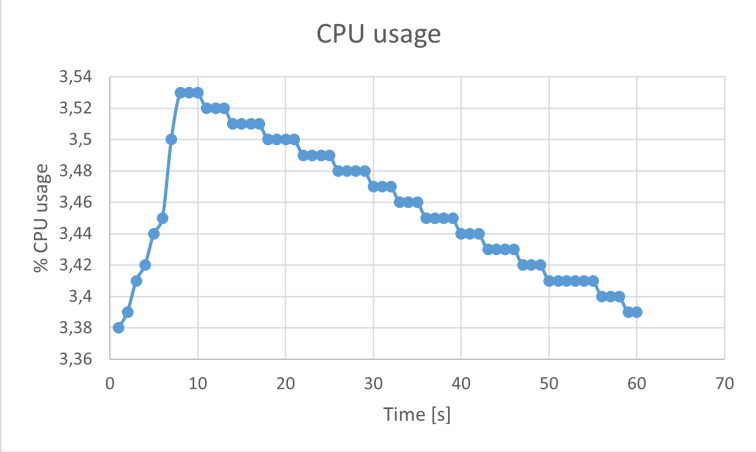

Figure 7.7: CPU usage of the NX board

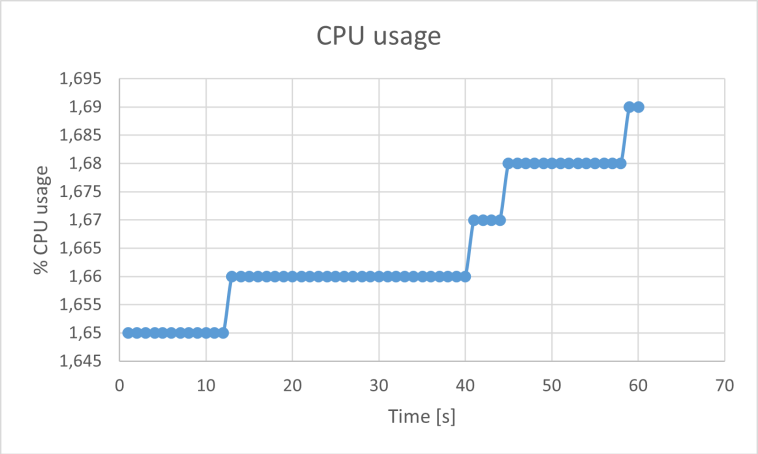

Figure 7.8: CPU usage of the laptop

Scenario 4: latency inside the server

It would be useful to compute how much time is required to exchange the same exact messages between Kubernetes Pods inside the server, as a final architecture could definitely include cloud nodes talking to each other. The values we expect are strongly tied to the actual resources assigned to the various containers at the moment of their creation, which can be configured in the Onboarding phase (as discussed in Section 7.1). In our case, 1 CPU and 1GB of RAM has been reserved to each container (Pod), not in exclusive mode (meaning that idle CPU clocks can be used by another Pod when necessary.

The curves represented in Figure 7.9 are not as clean as the ones obtained in the NX scenarios (some nodes received less messages than the ones expected): this cloud depend on some possible problems in the server internal communication, probably due to the mechanisms implemented by the platform to connect the Pods (K8s Pods can usually communicate automatically, but in the platform used in these experiments some additional effort must be done in order to connect them).

In any case, we can see that the trend is the same as the one inside the NX, but the latencies registered are much lower: the maximum one is nearly 0,6ms for all kinds of messages, except for Quaternion, which encountered some problem.

The CPU usage can be retrieved through the Capgemini platform itself, and a screen is shown in Figure 7.10, regarding in particular the Illuminance Pod. Every microservice has indeed 1 CPU at its disposal, as said, but a little amount of that (0,1%) was used to perform the node's job.

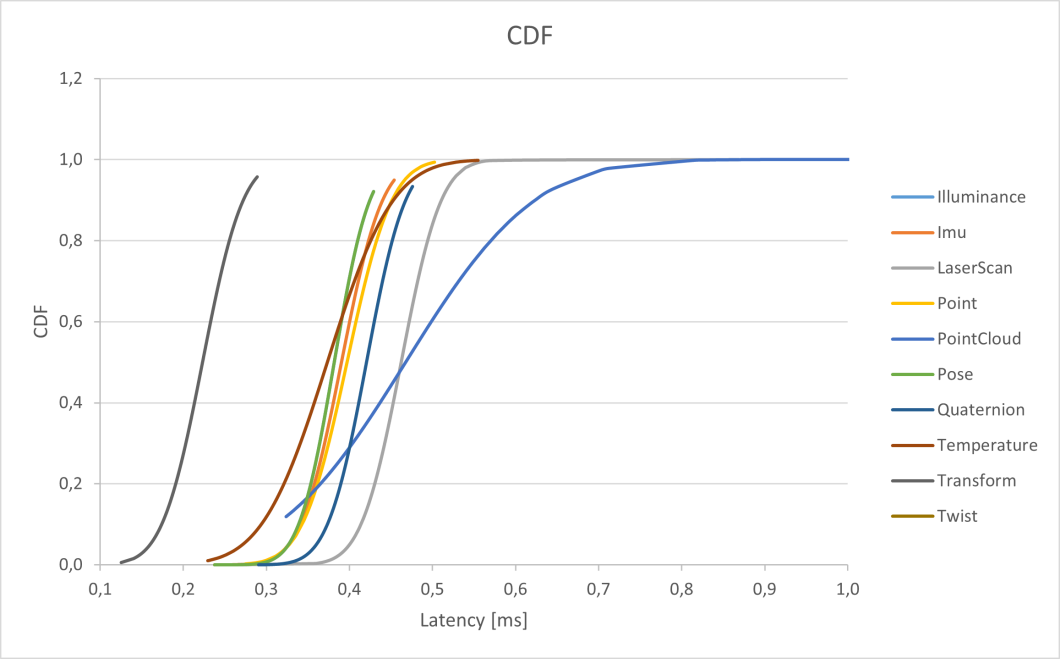

Figure 7.9: Latency inside the cloud server

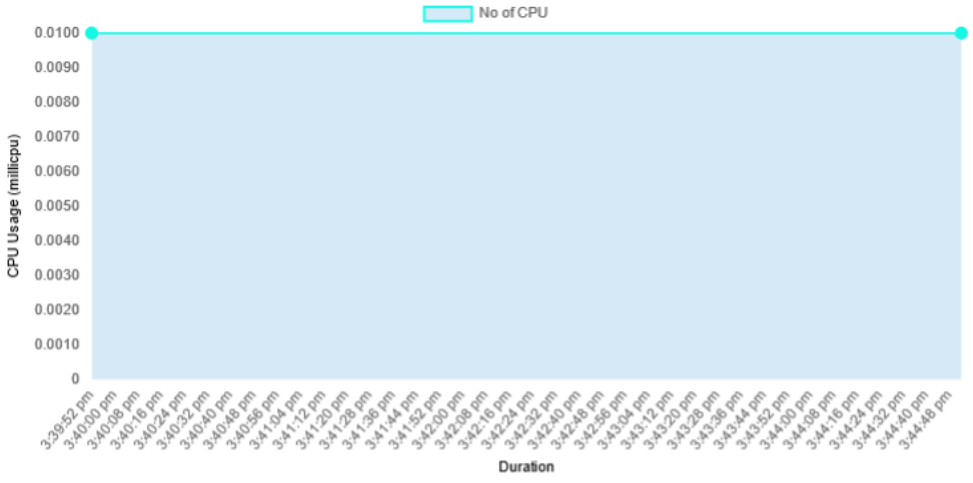

Figure 7.10: CPU usage of the server for one microservice

Scenario 5: latency between NX and server

This is the most complex scenario, which uses a WAN Participant in each of the two instances of the DDS Router, one of which is local and one in the server, which connect the NX board with the server's network interface: the local WAN Participant collects the local messages and delivers them to the cloud nodes. Of course, a Local Discovery Server Participant was necessary cloud-side, while only a Simple one in the NX was enough.

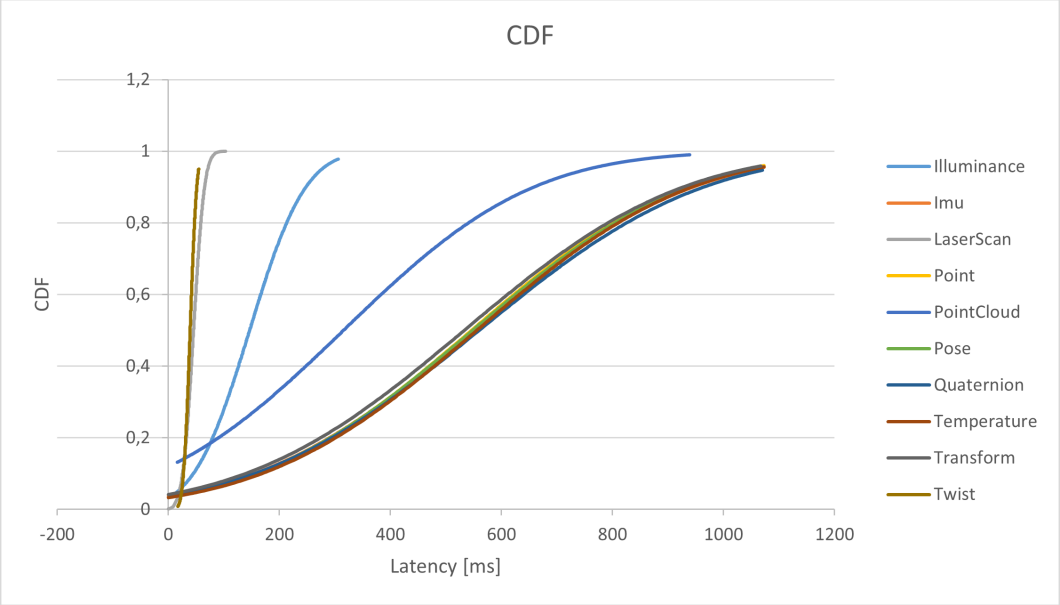

Figure 7.11: Latency between NX and server

In this case, the computed latencies are significantly higher, not even comparable to the previous ones. Except for some message kinds, which could be delivered in a little time, the majority of messages took even 1 second to be delivered, with a mean of around 500ms. This is a monstrous amount of time for the goals of an autonomous driving application, but the possible reason to explain these data is that the bare DDS Router instance used is not capable of sustaining a real-time exchange of data, given by the large amount of messages sent between the two sides (600 messages*10 nodes). It represented a huge bottleneck in the communication, condition that was made worse by the usage of the debug mode while executing the experiment (to solve communication problems encountered) and the presence of an Echo Participant that printed all messages. Given that the server was reachable in very low times (*ping* sessions gave an RTT with an order of magnitude of 10ms), all the additional latency is very likely brought by the processing performed by the

DDS Router, which brings us to the consideration that it is not designed for real-time ultra low-latency applications: a different approach may be more appropriate. In this sense, **alternative frameworks** have been developed, built on top of the DDS Router: they can be the answer to lowering these latencies as much as possible.

The usage of CPU in the NX was actually not high, and this means that all the problem encountered in message sending were generated by the cloud DDS Router instance: for this amount of messages, 1 CPU and 1GB of RAM were not enough, perhaps.

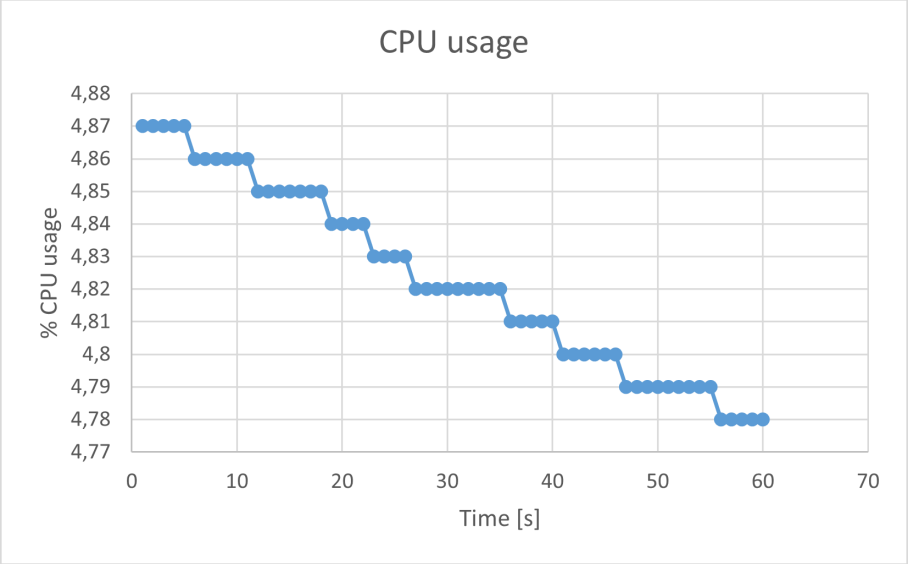

Figure 7.12: CPU usage of NX

An additional test has been performed, in which the computational resources given to the Pod executing the DDS Router server-side have been sensitively increased (8 vCPUs and 8GB of RAM), in order to see if the previous problems were actually generated with the DDS Router's data processing overhead. Only one type of message (Illuminance in particular) has been considered, as particular differences have not been seen in the latency behaviour among different message types, if not random.

The latency values registered in this case, shown in Figure 7.13, actually confirm the assumption we made earlier: with more computational power, the latency reaches a mean value of about 100ms, with peaks of 160ms (and some outliers reaching even 1s). These results bear out that the actual bottleneck is data processing performed by the DDS Router, and it is not a network problem: latency is still high, though, for the requirements of an autonomous driving application,

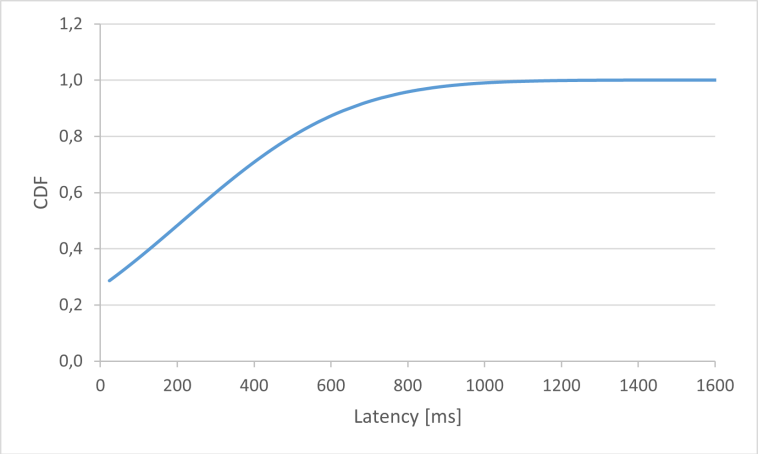

Figure 7.13: Latency between NX and server with enhanced performance

which further confirms that a different technology should be considered when making remote ROS2 nodes communicate.

Comparison between scenarios

A good way of comparing the previously described results is a boxplot of the data distribution obtained in all scenarios we have presented.

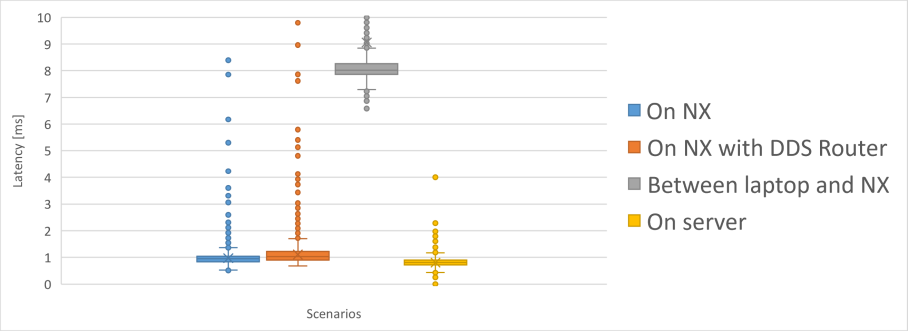

Figure 7.14: Latency distributions without remote communication

Figure 7.14 shows this comparison in the first four cases, without considering any machine-server communication. The distributions obtained are definitely comparable, meaning that any scenario which does not imply remote communication gives nearly the same exact results: the order of magnitude is 1ms, with a slighter improvement in the server case (the yellow box in the plot), as seen.

The case of Scenario 3 (publishers in the NX board and subscribers in a laptop in the same LAN) is nearly an order of magnitude above, considering that

data must trespass a physical network and physical devices (such as switches) in order to reach the destination, hence the grey box is a little bit above the other ones.

A completely different graph is obtained by also considering the latency measured in the fifth scenario: the data distribution in that case is not even comparable with the previous ones, clearly shown by the blue box in Figure 7.15, which is much higher w.r.t. the others and pretty large, meaning that diverse sets of data have been registered. This shows the results we discussed in Figure 7.11 in a better and more straightforward way.

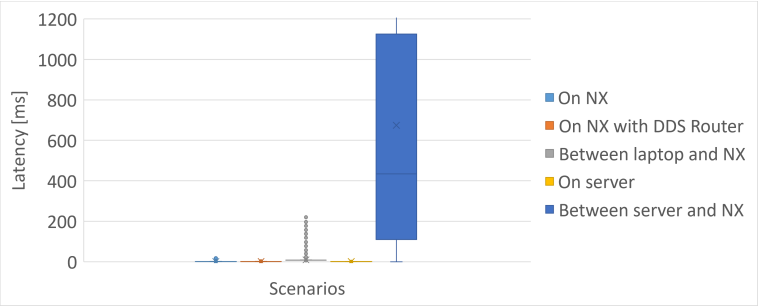

Figure 7.15: Latency distributions with remote communication

As done previously, we can compare the previously presented data with those obtained by assigning more computational resources to the DDS Router instance, shown in Figure 7.16: in this case, the last scenario presents much lower latencies than the ones in the previous graph, but they're still high for real-time applications such as autonomous driving ones, as previously said.

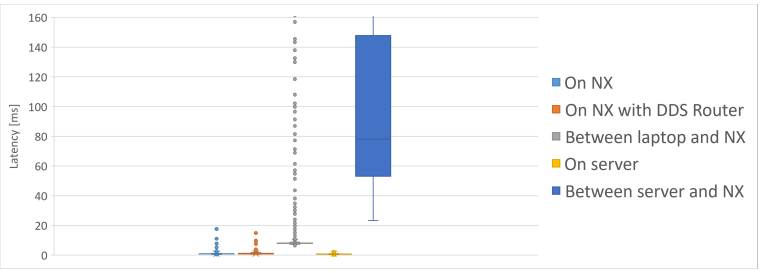

Figure 7.16: Latency distributions with remote communication with enhanced performances

7.3 Test 2: latency required for nodes' activation and deactivation

Another type of test that was considered to acquire some data to validate the autonomous driving solution in cloud consists in **measuring the time required by ROS2 nodes for their activation and deactivation**. Indeed, as explained in Section 4.4, the solution proposed by this thesis strongly leverages the concept of ROS2 lifecycle to have only one replica (either local or cloud) of each node in Active state at a certain moment. Hence, it would be useful to have an idea of how much time is required to perform this particular state transition.

For these measurements, a `lifecycle_manager` node has been leveraged, taken from the ROS2 lifecycle examples GitHub repository and adapted to each intended scenario. This node has the capability of triggering the state transitions of any other managed node, and it was configured to activate and deactivate example nodes in sequence for a total of 10 times, while measuring the time required for these state transitions. We highlight that only the `on_activate()` and `on_deacivate()` are considered, as they're the ones actually used by the system: in these particular transitions, the managed node allocates (or deallocates) only the required elements to start performing its tasks, hence they're much faster than the `on_configure()` one, for example, as in that case everything needed by the node in terms of Hardware and data structures has to be reserved.

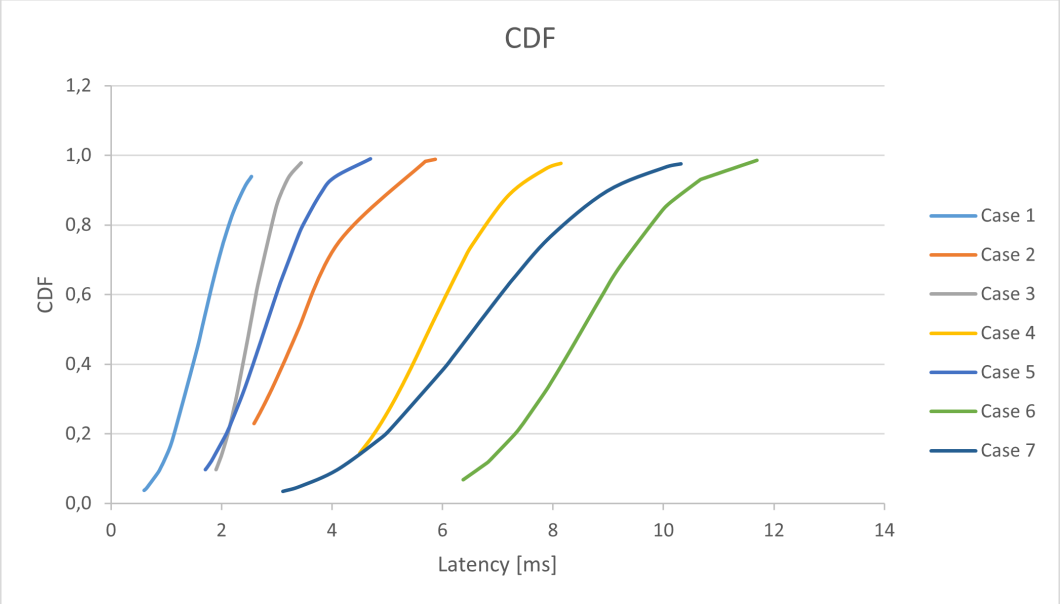

Figure 7.17: Latency required for nodes activation

Figure 7.17 represents the CDF for the acquired data about the latency required to trigger the activation of the considered node, in terms of message exchange and processing time, in 7 different cases:

- **Case1**: a `lifecycle_talker` (example managed node whose lifecycle we want to control) is executed in the same LAN and in the same machine as the `lifecycle_manager`.
- **Case2**: the `lifecycle_manager` is inside the NX board, in which all other ROS2 nodes are in Unconfigured state. The `map_server` node was chosen as an example target this time.
- **Case3**: same as Case2, but `alba_v2_localization` is taken as target.
- **Case4**: The `lifecycle_manager` is inside the NX (like Case2 and Case3), but the other nodes inside it are up and running (Active state), no longer Unconfigured. The `map_server` is taken as example target.
- **Case5**: same as Case4, but `alba_v2_localization` is taken as target.
- **Case6**: the `lifecycle_manager` runs in a laptop and the `map_server` is executed in the NX, where all the other nodes are Active.
- **Case7**: same as Case6, but the `alba_v2_localization` node is taken as target.

We can see that the data range, considering all the 7 cases together, goes from nearly 1ms to 12ms at maximum. There are differences in the retrieved data, which can be clearly seen by the fact that the curves representing the last considered conditions are shifted towards the right w.r.t. the ones representing the previous cases. This is because the scenarios implying two physical machines (NX and laptop) present some additional latency given by the time spent to go from one device to another, while the other cases are not affected by this problem.

We can clearly say, then, that no substantial processing time is required to activate a node, even in a context in which other nodes are present (a little bit more latency is registered when they are Inactive w.r.t. when they're Unconfigured, compare Case2 and Case4). The activation latency has the order of magnitude of 1ms, hence designing a switching solution based on it should not represent a real problem, according to these data.

Figure 7.18 represents, instead, the CDF on the latency data taken in the process of deactivation of the nodes, in the same exact scenarios (indeed, nodes have been activated and deactivated in loop and the measurements have been taken

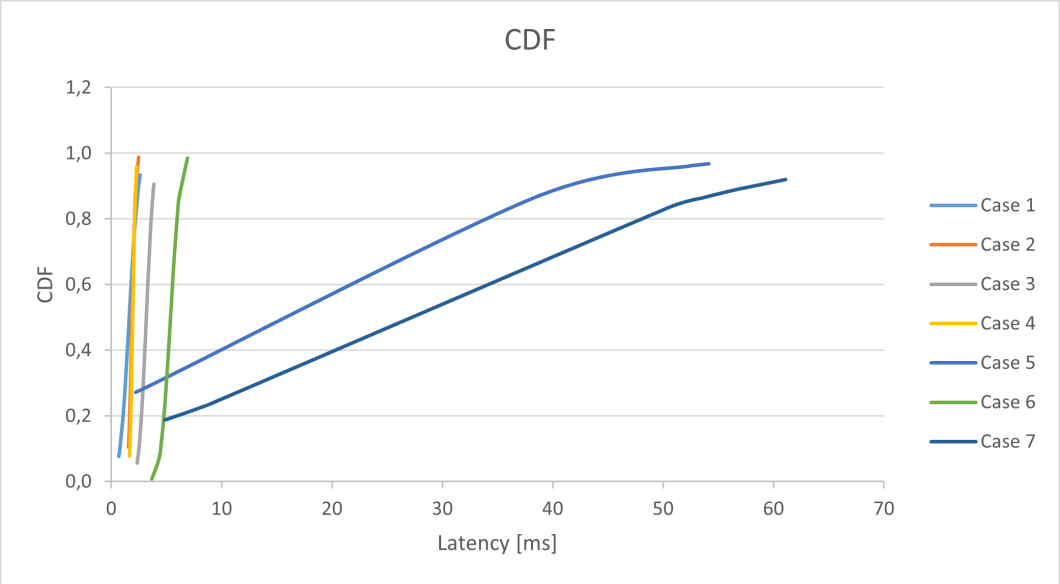

Figure 7.18: Latency required for nodes activation

separately). One important thing emerges from this graph: in most situations, the latencies are equal to the ones required for activation (see the curves on the left), namely with an order of magnitude of 1ms. However, there are some cases in which a consistently higher amount of time is required to deactivate a certain node, which can be clearly seen for Case5 and Case7. Indeed, the manager spent up to 60ms to deactivate the considered node, which was `alba_v2_localization` in both cases: this means that activation and deactivation times are very variable and strongly depend on the considered node, in terms of how many operations have to be performed when activating or deactivating it, for instance.

In this experiments, the `alba_v2_localization` node had to go through a lot of processing before being deactivated, while not that much in the opposite transition, as witnessed by the previous graph. The slight difference in the two curves of Case5 and Case7 is given by the additional latency required for execution in different machines (Case7), but the greatest gap with the other cases is generated by the difference in processing time, as mentioned.

In conclusion, we can say that even the activation or deactivation phases, which are at the very core of the switching solution proposed by this thesis, can bring much additional latency that can heavily affect the overall system. This strongly depends on the node whose transitions we're triggering, and must be considered case by case in order to decide if this cloud autonomous driving solution can be feasible or not.

Chapter 8

Future work

The system developed by this thesis in order to move the robot's computational load to the cloud is not complete: as already discussed, additional effort must be applied in order to produce a working prototype of a cloud solution for autonomous driving out of this thesis, especially for its switching capabilities. Some possible future additions, which could be useful in delivering such prototype, are to be considered for both problems faced in this work:

1. ROS2 Remote communication: the solution to this problem, which in this thesis is represented by the DDS Router technology, can be definitely improved. As the data itself collected in Section 7.2 confirm, a DDS Router instance cannot live up to the real-time latency constraints of this kind of application, as it is not optimized to work with its strict requirements. However, there are different implementations of bridges capable of connecting DDS Networks available as open-source plugins, such as Eclipse Zenoh for *Cyclone DDS*. These alternatives can be the final solution to this problem and they have not been explored here.

2. Cloud/local switching: the proposed architecture is designed from scratch, and it still has a lot of faults that must be taken in consideration to have a working prototype of a switching mechanism for a ROS2 autononmous system. They've already been discussed in Section 4.4.1 (in which also proposed solutions are described), but we list them here once again:

    - Synchronization: the activation of local instance and deactivation of the remote one (and vice versa) must be synchronized. Each node needs some time to be activated or deactivated, hence we can end up having both cloud and local replicas in the same lifecycle state (either Active or Inactive) or two exactly equal nodes publishing the same information to a third node at the same time. Most likely, the majority of nodes will not

have any problem in receiving the same information twice, but it could be an issue for some of them.

- Actions: a very common scenario generated by the switching system presented in this work is the one in which a cloud node starts an action and a switching is triggered in the meantime: in this case, there will be no trace of it in the system, as the local node is the one in charge of performing that action, and it does not know anything about it.
- Stateful nodes: some nodes need past data to work properly, namely data they have computed previously. When a switching is triggered, the newly activated instance needs the same data, otherwise it cannot start performing the same task its respective replica would have performed.
- Also, the switching logic presented in Chapter 5 could be improved, adding complexity to increase the efficiency of the decision-taking algorithm.

These are the main problems spotted in the design of the proposed architecture, and they are not of difficult address at first sight. Of course, the solutions mentioned in this Section and Section 4.4.1 are just theoretical, and completely different ones can be thought by any developer who wants to continue this work.

Of course, upcoming work must take in consideration that new releases of ROS2 will most likely be leveraged in the future, with additions which have not been part of this system's design: any future solution shall be flexible enough to adapt to what's coming next in the robotic word. Even the usage of ROS2 itself, which has been at the core of this work, could be considered obsolete in the future, in favour of more cloud-friendly technologies that could be better integrated with new cloud platforms.