POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Real time, dynamic cloud offloading for self-driving vehicles with secure and reliable automatic switching between local and edge computing

Supervisor

Prof. Fulvio RISSO

Candidate

Dario Paolo GULOTTA

July 2023

Summary

Autonomous vehicles are the subject of intense interest due to expectations that they will improve transportation. These automatic transport systems are quickly transitioning from being isolated to being connected, offloading an increasing number of tasks to external systems.

A vehicle that is truly autonomous is able to perceive its surroundings, make judgments based on what it sees and/or has been programmed to recognize, and then carry out an action in that environment. The ability of such autonomous machines to adapt to shifting environments is crucial, because it allows them to reach their goal regardless of the obstacle that arise on the way. Without human assistance, these vehicles can move around the workspace and avoid circumstances that could be dangerous to them or other people. These vehicles generate massive amounts of data from their sensors, including cameras, lidar, radar, and GPS. Robots with higher levels of sophistication can use the huge amount of data that comes from these sensors to locate objects in real time, and avoid them properly. This data needs to be processed and analyzed quickly because the surroundings could change very quickly, so rendering all that information is useless if the vehicle cannot make decisions about steering, acceleration, and braking in time.

In this scenario, the Edge Computing service delivery model enables the robots to delegate their computationally demanding tasks to potent computing infrastructure nearby. This distributed computing architecture brings computation and data storage closer to the source of the data, in this case, the autonomous vehicle. This allows for faster processing and analysis of data, which is critical for autonomous vehicles that need to make real-time decisions based on sensor data. However, there are also challenges associated with this type of deployment, including the need for dedicated communication protocols and infrastructure, as well as concerns about cybersecurity and privacy. In addition, the cost of deploying the necessary infrastructure, such as dedicated wireless communication, can be a barrier to widespread adoption.

Fully automated driving robots can only be effective when high-bandwidth, low-latency connectivity is enabled between vehicles, external servers, intelligent infrastructure, and other road users. 5G has the potential to enable the deployment of autonomous robots. It offers faster data speeds and lower latency than previous generations of wireless networks, which can make autonomous robots operate more effectively and efficiently. The combination of 5G, edge computing, and autonomous robots has the potential to transform the transportation system, by enabling new levels of efficiency and automation. However, many challenges will need to be addressed in order to ensure that the deployment of these technologies is functional and safe.

Table of Contents

List of Figures

Acronyms

ROS	Robot Operating System
DDS	Data Distribution Service
DCPS	Data-Centric Publish-Subscribe
QOS	Quality of Service
MOM	Message-Oriented Middleware
AMCL	Adaptive Monte-Carlo Localizer
BT	Behavior Tree
FSM	Finite State Machine
XML	Extensible Markup Language
JSON	JavaScript Object Notation

YAML

Yet Another Markup Language

HTTP

Hypertext Transfer Protocol

REST

REpresentational State Transfer

SEDIA

SEat Designed for Intelligent Autonomy

VM

Virtual Machine

RGB

Red Green Blue

RGBD

Red Green Blue-Depth

TOF

Time-of-Flight

LIDAR

Light Detection and Ranging

GPS

Global Positioning System

SLAM

Simultaneous Localization and Mapping

IOT

Internet of Things

K8S

Kubernetes

NAT	Network address translation
AMQP	Advanced Message Queuing Protocol
WAN	Wide Area Network
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
SPDP	Simple Participant Discovery Protocol
SEDP	Simple Endpoint Discovery Protocol
OMG	Object Management Group
VPN	Virtual Private Network
L2TP	Layer Two Tunneling Protocol
PPTP	Point-to-Point Tunneling Protocol
SSL/TLS	Secure Sockets Layer/Transport Layer Security
SSH	Secure Shell

DDSI

DDS Interoperability Wire Protocol

SNR

Signal-to-Noise Ratio

MEC

Multi-access Edge Computing

RAN

Radio Access Network

UPF

User Plane Function

SPI

Service Plugin Interface

PKI

Public Key Infrastructure

AES-GCM

Advanced Encryption Standard-Galois/Counter Mode

DH

Diffie–Hellman

KMIP

Key Management Interoperability Protocol

CA

Certificate Authorities

KDC

Kerberos Key Distribution Center

CLK

Clock

Chapter 1

Introduction

Autonomous robots are quickly transitioning from isolated to connected systems, offloading an increasing number of operations to third-party systems. Robots have always had the necessary intelligence built into them. Several historical factors contributed to this, including the limited availability of appropriate network connections capable of supporting the data transfer to and from the robot. By using the massive amount of data that is generated by the on-board sensors, they locate objects in real time and properly steer clear of them. In terms of energy use and computing power, processing the data from these sensors, however, can be very expensive. Robots generate a lot of data, which needs to be evaluated in real-time at every moment.

Today, these robots are quickly transitioning from isolated to connected systems, offloading an increasing number of operations to third-party systems. Offloading computational tasks can actually have a significant positive impact on computational capacity, allowing to handle larger workloads and process them more efficiently. In fact, by leveraging external resources (and specialized hardware), the overall computational power available can be increased. However, it's important to note that offloading typically involves some trade-offs, and one of the common trade-offs is increased latency. The extent of the latency trade-off depends on various factors, including the speed and reliability of the network connection, the distance between the local system and the offloading resource, and the complexity of the computations being offloaded. It's important to carefully consider the requirements and constraints of your specific use case and computational capacity, especially when latency and energy consumption minimization are required such as in collaborative robotics, where people and robots interact in dynamic environments.

In the scenario considered in this work, a mobile ROS2-based robot that can sense, compute, and communicate wirelessly, moves from a starting position to a target point in an operating environment, avoiding dynamic obstacles. For this robotic application, it was suggested a mechanism for offloading computation, designing and putting into place a local and remote switching system to accomplish this.

Multi-access edge computing (MEC) was the preferred choice for this use case, being a distributed computing architecture that brings computational capabilities closer to the network edge, specifically to the base stations or access points. This proximity reduces the distance between the connected agents (devices) and the data processing server, resulting in lower latencies. In the use case analyzed in this thesis, which involves, among others, the detection of lidar objects, low latency is crucial. The Edge Computing service delivery model enables the robotic agents to delegate their computationally demanding tasks to powerful computing infrastructure nearby. The decision to offload is based on the resources that are available at the network's edge. The aim is to use the resources, both on the local and edge sides, to meet the service level goals, which may include but are not limited to latency, bandwidth, reliability, and privacy. In contrast to on-board computers, edge platforms offer strong computation capacity, real-time data transmission, and the ability to handle enormous amounts of data at very high processing speeds. Additionally, because the distance between the edge-connected robots and the edge datacenters is much shorter, it will offer lower latencies.

Several methods were compared, which offer the connectivity required to close the gap in hybrid systems deployed between local and cloud. New architectural and technological developments have been examined, enabling seamless operation of containerized robotic applications at the edge or in the cloud. It is also provided an overview of the systems that enable secure and dependable high-bandwidth low-latency connectivity between automated vehicles and remote servers, ranging from extensions to the ROS 2 tooling to the integration of Kubernetes and ROS.

1.1 The Working Environment

The work of this thesis was carried out at Alba Robot, a startup that deals with indoor autonomous driving. In particular, it manages both the low level (hardware) and high level (software) in the creation of people movers, focusing on the ability to go from one starting point to another with the freedom to act autonomously. The part analyzed by this work focuses on the high level, and in particular on moving some parts, that make up the high level, to the cloud.

In addition to the real-world environment the company provided, one tool of particular importance was Crownlabs, a per-user virtual machine hosted in the Politecnico di Torino, which was used to test and validate most of the work done in this thesis. It is provided by a collection of virtual machines that are adequately furnished with the bare minimum of software needs. The lifecycle of user's remote desktops is easily manageble (e. g. , create, rebuild, destroy), making it possible to restore a clean environment as necessary.

1.2 Outline

This thesis is organized as follow:

Chapter 1: overview of the thesis's subject as well as the issue we are attempting to resolve in this chapter. It also describes the thesis's organizational structure.

Chapter 2: introduction to ROS2, DDS, Nav2, Kubernetes technology, and to the main concepts that are used in the thesis.

Chapter 3: a brief description of the autonomous system on which this paper is based.

Chapter 4: analysis of potential cloud offloading solutions and their implementation, along with the evaluation of the obtained results.

Chapter 5: analysis the chosen implementation approach for carrying out the suggested architecture.

Chapter 6: analysis of security strategies that can be integrated with the system to improve safety.

Chapter 7: conclusions regarding the thesis work, followed by a brief discussion of potential improvements.

Chapter 2

Background Technologies

2.1 Introduction to the ROS2 Project

Through the advent of robotic applications, such as autonomous driving or household robotics, the robot operating system (ROS) emerged as one of the most widely used software development frameworks. ROS 2 is a revamped version of the original ROS middleware that has been under development for over half a decade. It comprises a large set of interrelated software components that are commonly used to develop robotics applications. ROS2 and its predecessor share the same core concept. Therefore, we refer to software architectures in ROS1 or ROS2 as ROS systems.

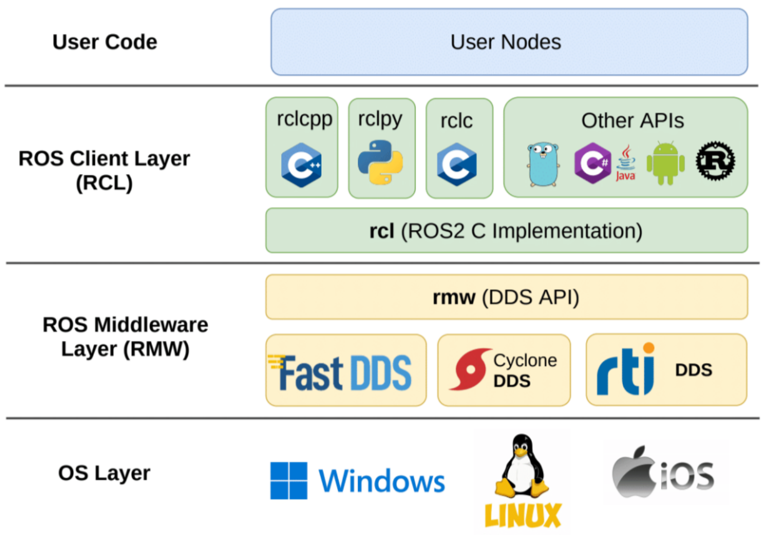


Figure 2.1: Ros2 design

ROS is not an operating system in the traditional sense of process management and scheduling; rather, it provides a structured communications layer above the host operating systems of a heterogenous compute cluster, to abstract hardware and low-level control from software application.

The design of ROS has been guided by a set of principles and a set of specific requirements. The following principles are asserted:

- Distribution: as with similarly complex domains, problems in robotics are best tackled with a distributed systems approach. Requirements are separated into functionally independent components, like device drivers for hardware, perception systems, control systems, executives, and so on. At run-time, these components have their own execution context and share data via explicit communication. This composition should be conducted in a decentralized and secure manner.

- Abstraction: to govern communication, interface specifications must be establish. These messages define the semantics of the data exchanged. A favorable abstraction balances the benefits of exposing the details of a component against the costs of overfitting the rest of the application to that component, thereby making it difficult to substitute an alternative. This approach leads to an ecosystem of interoperable components abstracted away from specific vendors of hardware or software components.

- Asynchrony: the messages defined are communicated among the components asynchronously, creating an event-based system. With this approach, an application can work across the multiple time domains that arise from combining physical devices with a host of software components; each of which may have its own frequency for providing data, accepting commands, or signaling events.

- Modularity: the UNIX design goal to 'make each program do one thing well' is mirrored. Modularity is enforced at multiple levels, across library APIs, message definitions, command-line tools, and even the software ecosystem itself. The ecosystem is organized into a large number of federated packages, as opposed to a single codebase.

All of these, but specially distribution, modularity, and asynchrony, are tightly related to the new reality of connected and decentralized robotic systems, moving from single robots to fleets, and moving from single-host computing to computing architectures naturally distributed across the edge-cloud continuum.

2.2 Ros2 Concepts

A typical ROS 2 system is composed of a number of distributed processes or nodes, ranging from sensor drivers to algorithm implementations, high-level decision-making, or external interfacing and control. At the heart of any ROS system is the ROS graph (see Figure 2.2). The ROS graph refers to the network of nodes in a ROS system and the connections between them by which they communicate.

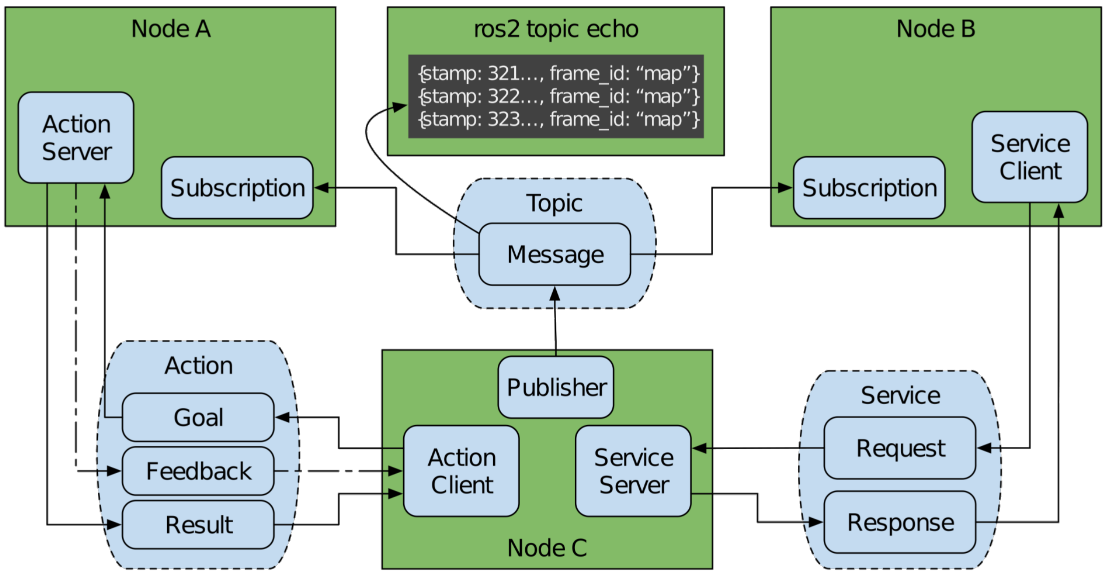


Figure 2.2: Ros2 graph: nodes, topics, services and actions

ROS nodes communicate to each other through available communication patterns: (i) topics, an asynchronous message passing framework with a pub/sub system; (ii) services, synchronous request-response patterns; and (iii) actions, asynchronous interfaces with a request-feedback-response architecture that fits particularly well physical actions requiring time and physical interaction with a robot's environment. Additionally, ROS provides a series of abstraction layers, with the communication middleware relying on implementations for the industry-standard DDS protocol.

2.2.1 Nodes

Nodes are the fundamental building blocks of ROS applications. They can be thought of as independent software components that work together to accomplish a particular goal. Nodes can communicate with each other by publishing and subscribing to messages on topics. This communication is asynchronous, which means that nodes can communicate with each other without blocking each other's processes. Each node in ROS has a unique name within the system, which is used to identify the node when communicating with other nodes. Nodes can be written in different programming language. This allows developers to choose the language that best suits their needs or to integrate existing software components into the ROS system. ROS nodes are typically organized into packages, which are collections of related nodes and resources. Packages provide a convenient way to manage and distribute ROS applications.

Nodes in ROS can be started and stopped independently, which makes it easy to add, remove, or replace nodes without affecting the rest of the system. This also allows you to distribute different nodes across multiple machines, which can improve performance and scalability.

2.2.2 Messages

Messages are used to communicate data between nodes in a robotic system. Messages are structured data types that consist of one or more fields, and they are sent and received to share different type of information. Messages can be used to represent a wide range of data types, such as numerical values, strings, images, and sensor readings. Messages can also be nested, which allows complex data structures to be represented.

ROS provides a set of standard message types, such as geometry_msgs/Twist for representing velocity commands, sensor_msgs/Image for representing images, and std_msgs/String for representing strings. Developers can also create custom message types to represent data that is specific to their application.

2.2.3 Topics

The simplest ROS2 graph is made up of nodes and topics. Topics are a fundamental communication mechanism used for interprocess communication (IPC) between nodes in a robotic system Topics allow nodes to send and receive messages(ref to messages), which are structured data types that contain information about the state of the system. They are created and managed by ROS2 nodes. A node that wants to publish messages on a topic creates a publisher for that topic, while a node that wants to receive messages on a topic creates a subscriber for that topic.

ROS 2 provides this publish-subscribe functionality focusing on using asynchronous messaging to organize a system using strongly typed interfaces. The publish-subscribe architecture allows many-to-many communication, which is advantageous: a node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.

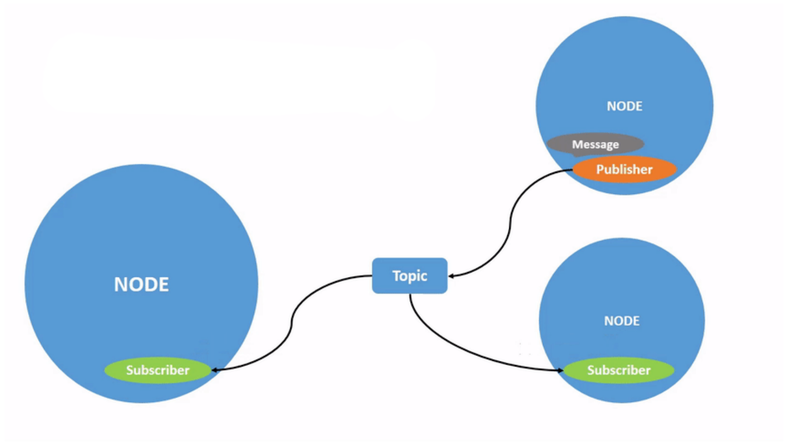


Figure 2.3: Ros2 topic communication pattern

2.2.4 Services

Asynchronous communication is not always the right tool. ROS 2 also provides a different style pattern, known as services. Services provide a request/response communication mechanism between nodes in a robotic system. Topics enable nodes to subscribe to data streams and receive continuous updates, whereas services only deliver data when specifically requested by a client.

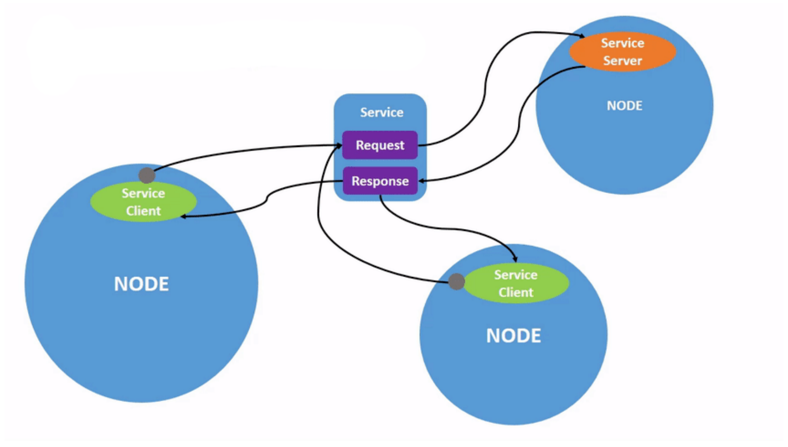


Figure 2.4: Ros2 service communication pattern

A service is a named entity that is composed of two parts: a service server and a service client. The service server provides a service that performs a specific task or provides information, while the service client sends requests to the server and receives responses. Services allow nodes to make requests for information or action, and receive responses that contain the results of those requests. Uniquely, ROS 2 allows a service client's process to not be blocked during a call.

Request-response communication provides easy data association between a request and response pair, which can be useful when ensuring a task was completed or received. For example, a service could be used to request the status of a sensor, or to control the movement of a robot arm.

2.2.5 Actions

A unique and more complex communication pattern of ROS 2 is the action. Actions are goal-oriented and asynchronous communication interfaces: this pattern is used in long-running tasks like autonomous navigation or manipulation, though it has a variety of uses.

An action is composed of three parts:

- Goal: we introduce the idea of a goal that can be sent to an ActionServer by an ActionClient in order to complete tasks using actions. It defines the ultimate state that it desires the system to achieve. For every goal that it accepts from a client, the action server keeps a state machine, which can be seen in Figure 2.5. It is important to notice that the state machine does not contain rejected goals.

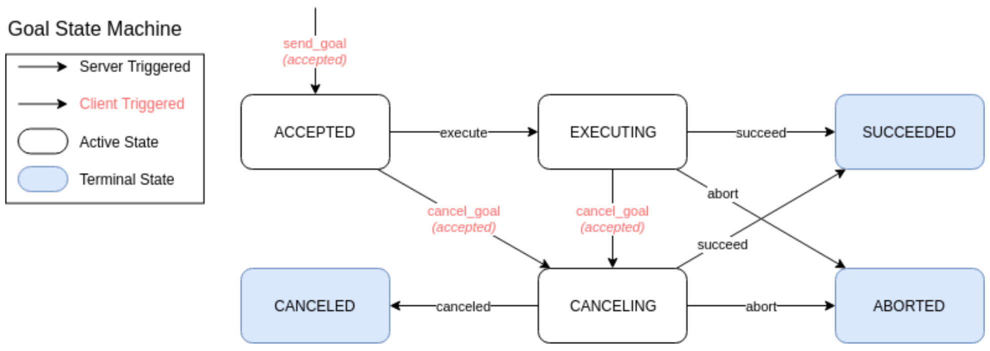


Figure 2.5: Ros2 goal state machine

- Feedback: it gives server implementers a way to inform an ActionClient, providing incremental progress updates that have been taken toward a goal during the task execution. It is a message that is sent on a regular basis to the Action Client while the Action is being completed. The Client has the right to halt an action during this stage.

- Result: on reaching the goal's completion, the ActionServer sends a result to the ActionClient. Since this is sent only once, it differs from feedback. It shows that the Action has been carried out completely and that the system has arrived at the desired end result.

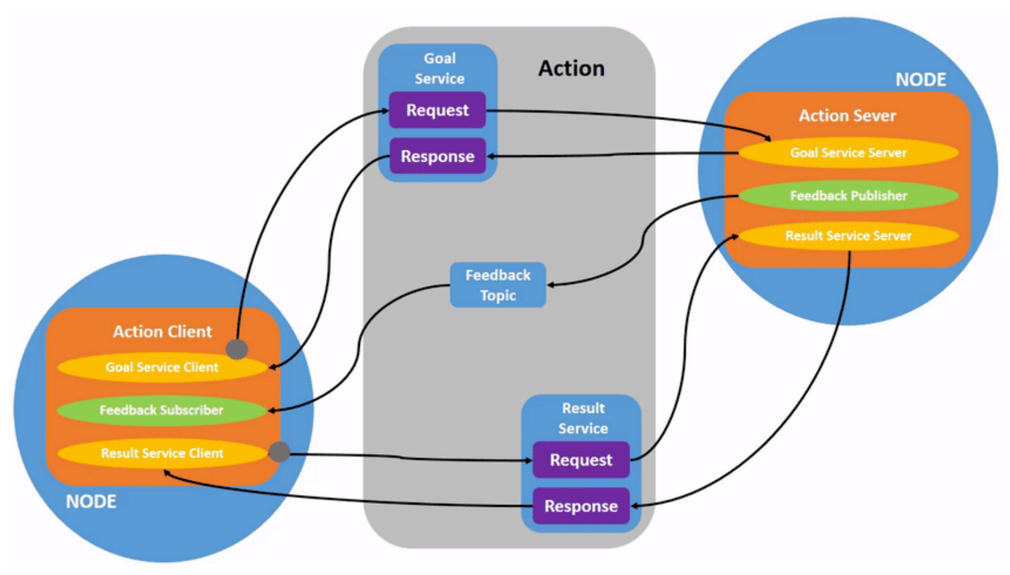


Figure 2.6: Ros2 action communication pattern

An action is initiated by sending a goal message to an action server, which is responsible for executing the task and sending back feedback and the result message. An action server has a name and a type, just like topics and services do. The name must be distinct across all action servers and may be namespaced. This indicates that different action servers with the same type running concurrently (under various namespaces) are possible. The action client sends a cancel message to the action server to cancel the ongoing task. There may be multiple clients on a single server, but the server will determine how to manage multiple clients' goals at once.

Their functionality is comparable to that of Services, with the exception that Actions are preemptable (they can be stopped mid-execution) and also offer consistent feedback. For as long as a goal is pursued, actions can maintain state; in fact, since each action goal is uniquely identified by its id, it is possible to maintain a separate state instance for each client when two action goals are being executed concurrently on the same server.

2.3 Managed Nodes

ROS2 introduced this particular and very useful feature that let the programmer to managed nodes via its lifecycle. Indeed, there is more control over the state of the

ROS system fully with a managed node life cycle. Before allowing any component to start carrying out its behavior, at the nodes startup we can easily verify that all components have been correctly instantiated thanks to a few services that enable external interaction with lifecycle nodes, that are automatically spawned by each node. The most crucial are called GetState and ChangeState, which allow to retrieve the state of the node and invoke transitions.

The key idea is that a managed node is essentially a black box that presents a known interface and follows a known life cycle state machine. This guarantees that any tools developed for managing nodes can be used with any compliant node and gives node developers flexibility in how they provide the managed life cycle functionality. The picture below provides a thorough overview of the two groups of lifecycle states that every managed node must adhere to:

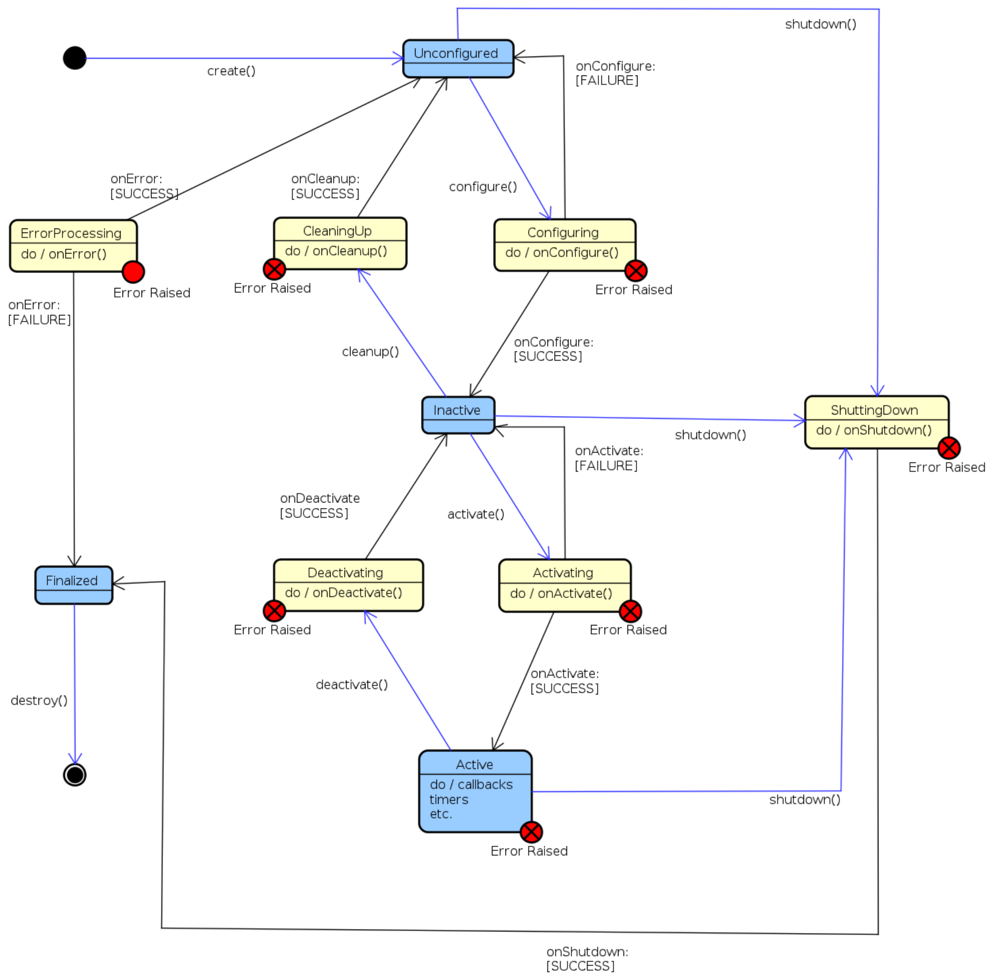


Figure 2.7: Lifecycle of a Ros2 managed node

Primary states

- Unconfigured: when a node is first instantiated, it is in this state of the life cycle. Additionally, after an error has occurred, a node may be tuned back to this state. Also there shouldn't be any stored state in this state.

- Inactive: a node in this state is one that is not currently processing anything. This state's main objective is to enable a node's (re-)configuration (configuration parameter changes, topic publications/subscription additions and deletions, etc.) without affecting the node's operation. The node will not be given any execution time while in this state to read topics, process data, respond to requests for functional services, etc. Any new data on managed topics won't be read, analyzed, or otherwise processed while the topic is in the inactive state. The topic's configured QoS policy will govern how long data is retained. Requests for managed services made to a node that is in the inactive state won't receive a response, and the caller will receive an immediate failure message.

- Active: the node's life cycle is currently in its primary state. The node performs any processing, responds to service requests, reads and processes data, and produces output while in this state.

- Finalized: the node's final state, before it is destroyed, is known as the finalized state. The only way out of this situation is to be destroyed because it is always terminal. Debugging and introspection are made possible by this state: in fact, instead of being destroyed immediately, a failed node will still be visible to system introspection and may be potentially introspectable by debugging tools. It is anticipated that the supervisory process will have a policy to automatically destroy and recreate the node if it is being launched in a respawn loop or has known reasons for cycling.

Transition states

- Configuring: the node's onConfigure callback will be activated during this transition state, enabling it to load its configuration and perform any necessary setup. A node's configuration will typically involve one-time tasks that must be completed during the node's lifetime, like acquiring permanent memory buffers and setting up topic publications/subscriptions that must be always the same throughout the life cycle. Regardless of whether it is active or inactive, the node uses this to set up any resources it needs to hold over its life. The initialization of configuration parameters, memory that is held continuously, and topic publications and subscriptions are a few examples of such resources.

- Cleaning Up: the node's onCleanup callback will be called during this transition state. This method should completely clear the node's state and put it back in the same operational state as when it was first created. Cleanup will move to ErrorProcessing if it is unsuccessful.

- Shutting Down: in this transitional state, the callback function onShutdown will be used. Any necessary cleanup before destruction should be handled by this method. Except for Finalized, any Primary State may be used to enter it and will receive the originating state as a parameter.

- Activating: the callback onActivate will be executed during this transition state. This approach is supposed to make any last preparations before beginning to execute. This might entail acquiring things like access to hardware, which are only held while the node is actually running. The callback shouldn't ideally be used for any preparation that takes a long time, like laborious hardware initialization.

- Deactivating: The callback onDeactivate will be executed in this transitional state. This method should undo the changes made by the onActivate method before it can begin executing.

- Error Processing: any error can be fixed in this state of transition. This state can be accessed from any other state where user code will be run. If error handling is successfully finished, the node can revert to the unconfigured state. If a complete cleanup is not possible, it must fail, in which case the node will move to the finalized state in anticipation of being destroyed. Error return codes in callback methods, callback methods themselves, uncaught exceptions, are all possible causes to moving to the ErrorProcessing state.

2.4 DDS

DDS, which stands for Data Distribution Service, is a middleware technology that provides a standard mechanism for distributed applications to exchange data in a reliable, scalable, and real-time manner.

DDS provides several features that make it an ideal middleware for robotics systems. First, it provides a data-centric publish/subscribe (DCPS) model, which allows data to be sent to multiple subscribers efficiently. Unlike the traditional publish/subscribe model, where publishers send data to the broker, and the broker forwards the data to subscribers, the data-centric model allows subscribers to express their interests explicitly. This may be its greatest strength, because reduces network traffic and improves scalability. DDS uses a data-centric approach, which means that the data itself is the focus of the communication, rather than the endpoints that produce or consume the data. This allows for a flexible and dynamic system, where new data sources and consumers can be added or removed without affecting the existing system.

DCPS defines the following entities—domain participant, publisher, data writer, subscriber, data reader, topic, and QoS policy, which makes a perfect match with the already discussed ROS2 concepts (see Section 2.2).

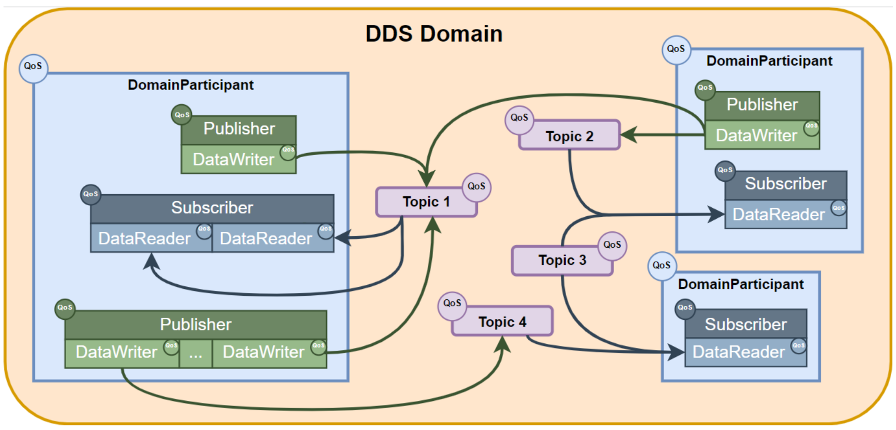


Figure 2.8: Structure of DCPS and data communication in a data domain

A domain participant refers to an application that interacts within the data domain. It is responsible for creating publishers, subscribers, and topics of its own. A publisher is an entity that publishes data to the network and is associated with one or more data writers that write data to be published. Once the data is written, the data writer notifies its publisher that the data is available. The publisher then serves as an interface between the application and the network.

A subscriber, on the other hand, receives published data from the network and is associated with one or more data readers that read in the received data. Once the data is received, the subscriber notifies its associated data reader, which makes the data available to the application. The data reader serves as an interface between the application and the network.

A topic represents a data object of a specific name and type. For a publisher and subscriber to communicate, the topic of the publisher must match the topic of the subscriber. Multiple publishers and subscribers can be associated with the same topic. A topic may also have multiple instances, each with a unique key.

Quality of Service (QoS) policies are used to specify the communication requirements. DDS defines 22 different QoS policies, and an individual entity can specify its own set of QoS policies. Applicable QoS policies may differ depending on the type of entities involved. For example, the durability policy can be applied to a topic but not to a publisher or subscriber.

DDS provides several benefits over traditional middleware technologies, such as message-oriented middleware (MOM) and remote procedure call (RPC). Firstly, DDS provides a high level of scalability and performance, as it uses a data-centric approach and supports multicast communication, which reduces network traffic and improves efficiency. Secondly, DDS provides a high level of reliability, as it supports fault tolerance and data replication, which ensures that data is not lost or corrupted in case of network failures or system crashes. Finally, DDS provides a high level of interoperability, as it supports a standard API and a standardized data model, which enables applications from different vendors to communicate with each other seamlessly.

2.5 Nav2

Since the development of the first robots, numerous mobile robot navigation frameworks and systems have been introduced. The service robots that are currently being proposed in factories, stores, and public spaces were made possible by these frameworks. Although many ideas have been put forth, very few have come close to matching the influence of the ROS Navigation Stack on the expanding mobile robotics industry. Conceptually speaking, the Navigation Stack is quite straightforward. It is an effective method for making sure mobile robots can move from one location to another. By analyzing information from the odometry, sensors, and environment map and producing velocity commands to be sent to the mobile base, the navigation stack's task is to give the robot a safe path to follow.

It can be challenging to use the navigation stack for an autonomous system, though. The robot's ROS installation, the presence of a tf transform tree, and the publication of sensor data using the appropriate ROS Message types are prerequisites for using the navigation stack. The performance of this navigation stack must be optimized, which calls for some fine-tuning of the parameters. In addition, the navigation stack must be configured in such a way that the morphology and dynamics of the robot perform at a high level.

Considering these requirements, ROS version 1 has encountered problems such as single robot system, single platform, poor real-time performance, poor stability, high network requirements etc., which gradually cannot meet the requirements of robot tasks in the current complex environment. The development and use of ROS2 offers the chance and capability for robots to get past the current development roadblock. The most recent ROS2 distributions are compatible with the new and improved ROS Navigation2 stack. While adding features like a behavior tree navigator and task-specific asynchronous servers to achieve high modularity, it inherits a set of basic implementations of algorithms. Planning, control, and recovery tasks are coordinated by Navigation2 using a customable behavior tree.

With the help of one of an increasing number of algorithm implementations, each behavior tree node in its federated model invokes a remote server to perform one of the aforementioned tasks. To make it simple to create and choose new algorithms or techniques at run-time, each server has implemented a standard plugin interface. This architecture utilizes multi-core processors and makes use of ROS2's real-time, low-latency capabilities.

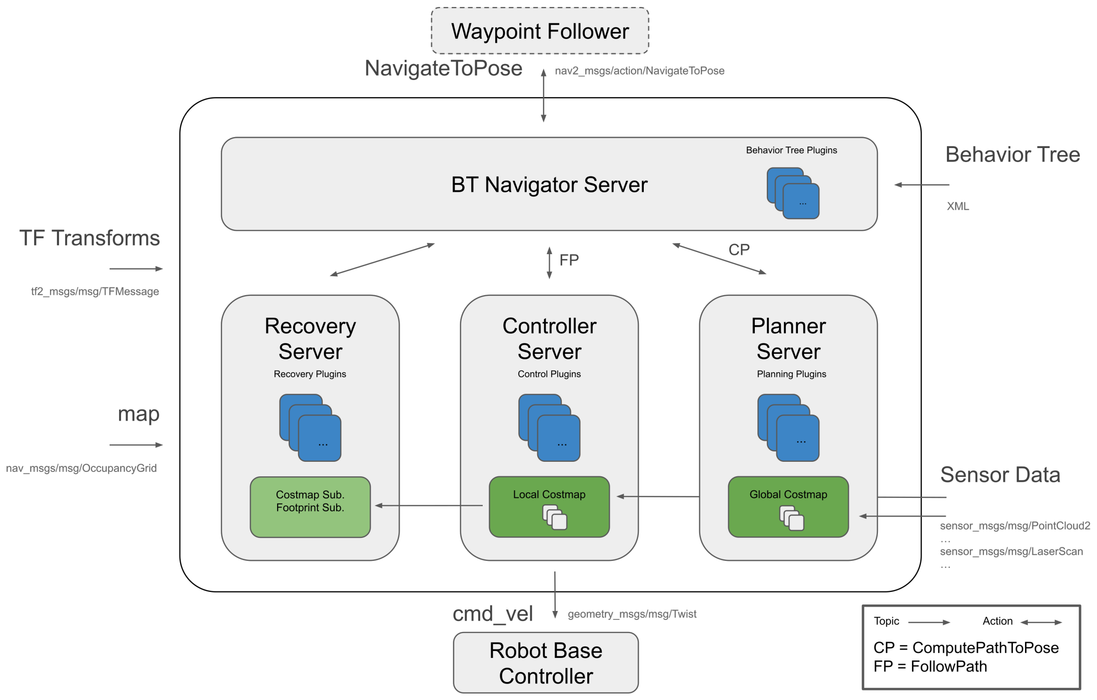


Figure 2.9: Nav2 stack

Navigation2 must be highly modular to develop a navigation system that can function with a wide range of robots in a variety of environments. This design strategy favored solutions that could be quickly reconfigured and chosen at run-time in addition to being modular. Figure 2.9 illustrates how Navigation2 achieved this by developing two design patterns: a behavior tree browser and task-specific asynchronous servers. Each particular server is a ROS2 node that houses algorithm plugins, which are dynamically loaded libraries that are used during runtime. Overall, the Nav2 stack consists of a group of instruments with the following navigational capabilities:

- Map Server: is used to load, serve, and store maps.
- AMCL: locate the robot on a map.
- Nav2 Planner: create a route from point A to point B that avoids obstacles.

- Nav2 Controller: direct the robot as it travels the path.
- Nav2 Smoother: plans for smooth paths will be more continuous and practical.
- Nav2 Costmap 2D: convert sensor data to a world cost map representation.
- Nav2 Behavior Trees and BT Navigator: utilize behavior trees to create intricate robot behaviors.
- Nav2 Recoveries: compute recovery actions in the event of failure.
- Nav2 Waypoint Follower: follow the waypoints in order.
- Nav2 Lifecycle Manager: manage the servers' lifecycles and watchdogs.
- Nav2 Core: plugins for enabling your own unique custom behaviors and algorithms.

Nav2, which implements navigation tasks for a robot, is based on ROS libraries and services. However, understanding ROS is not enough to fully comprehend Nav2 because it also introduces new concepts in addition to using ROS definitions.

2.5.1 BT Navigator

In challenging robotics tasks, behavior trees (BT) are becoming more prevalent. An autonomous agent, like a robot or a virtual character in a video game, can switch between various tasks using a Behavior Tree (BT). Figure 2.10 depicts a ball interaction task being carried out by a BT as an example.

BTs are a tool used to create modular systems that can respond to changes in their environment in a timely and effective manner. Reactive refers to the capacity to respond to changes in a timely and effective manner. We want a virtual game character to hide, run away from, or engage in combat if they become aware of an approaching enemy. We also want a robot to slow down and avoid colliding with a human if they get in the way of its intended trajectory.

The degree to which a system's components can be disassembled into smaller units and then reassembled is referred to as modularity. We desire a modular design for the agent so that individual components can be created, examined, and used independently. Working with individual components rather than the combined system is advantageous because complexity increases with size. Unlike a finite state machine (FSM), which could have dozens of states and hundreds of transitions, this is not the case. We can specify the desired behavior in modules by implementing task switching with BTs as opposed to FSMs. A behavior is frequently made up of a series of task-independent sub-behaviors, which can add more information and means that when designing one sub-behavior, the designer is not required to know which sub-behavior will be executed next.

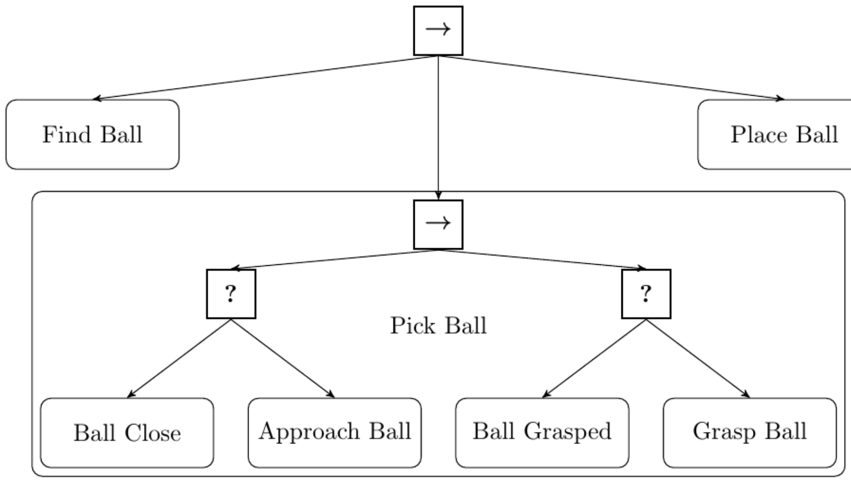


Figure 2.10: Behavior tree of a specific application

The planner, controller, and recovery servers are activated and their progress is monitored by the Behavior Tree Navigator, which uses a behavior tree to organize the navigation tasks. These servers are contacted by the behavior tree plugins using the ROS2 action interface. By altering a behavior tree that is stored as an XML file, distinct navigational behaviors can be produced. Different control flow and condition node types can easily be added to BTs without any programming knowledge. Making a new behavior tree markup file that is loaded at run-time is all it takes to alter the navigation logic. Because the behavior tree nodes in the navigator are based on the ROS2 actions, they can make long-running asynchronous server calls to processor cores. Thanks to this, the amount of compute resources that a navigation system can effectively use is significantly increased by using multi-core processors.

2.5.2 Recovery Server

To make the system fault-tolerant, recoveries are used to rescue the robot from challenging circumstances or make an effort to resolve a variety of problems. The recovery server executes them after receiving a request from one of the behavior tree's leaves. According to the default behavior, these actions are ranked from conservative to aggressive. Recovery attempts to deal with unknown or failing system conditions and manage them on their own.

It should be noted, though, that recovery behaviors can either be unique to its subtree (e.g. the global planner or controller) or system level in the subtree containing only recoveries in the event of system failures.

Three groups can be made up of them:

- Clear Costmap: In the event that the perception system fails, a recovery to clear the costmap layers is available.
- Spin: A recovery to free up space and push the robot out of any potential local failures, such as the perception that it is too trapped to back out.
- Wait: An alternative recovery in the event of a time-based impediment, such as traffic or the need to collect more sensor data.

Some examples include perception system flaws that cause the environmental representation to be cluttered with fictitious obstacles. The robot would then be able to move as a result of the clear costmap recovery being initiated. Another scenario would be if the robot became stuck because of moving obstacles or poor control. If possible, the robot can move from an unfavorable location into a space that is open and easy for it to navigate by backing up or spinning in place. To call an operator's attention for assistance in the event of a complete failure, a recovery may also be used.

2.5.3 Controller Server

When a robot is acting inappropriately, the way we follow the path computed globally or finish a local task in ROS1 is through controllers, also referred to as local planners. In order to determine what feasible control actions the base should take, the controller will have access to a representation of the local environment. Calculating the wheel's velocities and computing a valid control effort to follow the global plan is a controller's general task in Nav2. Controllers and local planners, however, come in a variety of classes. At each update iteration, many controllers will project the robot forward in space and calculate a locally viable path.

2.5.4 Planner Server

A planner performs path-finding by utilizing a variety of algorithms to find the shortest route while avoiding obstacles. It performs path-finding by utilizing a variety of algorithms to find the shortest path (depending on the nomenclature and algorithm chosen, the path may also be referred to as a route) while avoiding obstacles. While the full path is optimized by the global costmap and global planner, the local costmap and local planner are responsible for optimizing autonomous driving in close proximity. Sensor data will be buffered into and available to the planner from a global environmental representation. These elements work together to determine the best route given a navigational objective in the real world. There are numerous classes of supported plans and routes, though, which we will not go into detail here.

2.5.5 Waypoint Follower

The Nav2 waypoint follower is a basic feature of a navigation system that enables the robot to follow waypoints to get to multiple destinations. It includes a plugin interface for specific task executors, making it useful for completing specific tasks like taking a picture, picking up a box, or waiting for user input.

It can be used as a sample application in an on-robot solution; however, it can also be used for more complex scenarios, where the robot perform many complex tasks in complete autonomy. In this approach, the waypoint following application is more closely tied to the system autonomy.

The choice between the two approaches depends on the tasks the robot is completing, the environment, and the available cloud resources. Neither approach is better than the other, and the distinction is often very clear for a given business case.

2.5.6 Environmental Representation

The robot's perception of its surroundings is represented by the environmental representation. In order to combine different algorithms and data sources' information into a single space, it also serves as the central localization for those systems. The controllers, planners, and recoveries use this space to compute their tasks in a safe and effective manner. To keep track of obstacles in the real world, the navigation stack uses as a environmental representation the costmaps. In a costmap, each grid cell is given a cost that represents a certain value or distance between the robot and an obstacle.

There are two variations of the costmap in the Nav2 stack:

- Global costmap: this costmap is utilized for global planning, which entails developing long-term plans for the entire environment. Everything that the robot is aware from previous visits and stored knowledge is present here, for example the static map, which typically contains immovable elements like walls and other such things.

- Local costmap: It is used to plan locally and avoid obstacles. It reflect everything that can be learned about the present position using the sensors, such as every wall visible, along with moving people and other shifting things.

To buffer data into the costmap, various costmap layers are implemented as plugins. Using camera or depth sensors, costmap layers can be made to find and follow objects in the scene for collision avoidance. Before entering sensor data into the costmap layer, it might be a good idea to process the data.

2.6 Kubernetes

Kubernetes is a powerful container orchestration tool that allows you to manage, deploy and scale containerized applications in a distributed environment. It offers the control over how your applications should operate and how they should be able to communicate with one another and the outside world. Using techniques that offer predictability, scalability, and high availability, it is a platform created to fully manage the life cycle of containerized applications and services. Kubernetes can be used in cloud robotics to manage the deployment and scaling of robot software and applications. This can be useful in scenarios where a large number of robots need to be managed and updated simultaneously. Cloud robotics can also benefit from Kubernetes' ability to handle complex networking configurations, load balancing, and auto-scaling. This can help ensure that robots are always connected to the cloud, and can access the resources they need to operate efficiently.

In addition, Kubernetes can be used to deploy and manage heavy jobs used by robots to make decisions, analyze data, and perform tasks. This can help robots become more intelligent and adaptable, and provide better insights and feedback to their human operators.

2.6.1 Application Isolation

Application isolation refers to the partitioning of a single program or application stack from the rest of the active processes. The most traditional method for doing this is to simply run your application on a different physical machines, but that quickly becomes very expensive.

Bare metal, virtualized, and containerized are three different ways of running software applications.

Bare metal refers to running software directly on the underlying hardware without any intervening software layers. In this setup, the operating system and application run on the physical machine, giving direct access to hardware resources. However, there was no way to specify resource limits for individual applications in a physical server, which led to problems with resource distribution. In fact, if multiple applications are running on a physical server, there may be times when one application consumes the majority of the resources, which lowers the performance of the other applications. Running each application on a different physical server would be a fix for this. However, due to underutilization of resources and the high cost of maintaining a large number of physical servers, this did not scale.

Virtualization enables greater flexibility, scalability, and resource sharing than bare metal, making it popular in enterprise IT environments. Virtualization involves running multiple operating systems and applications on a single physical machine, using a virtualization layer that emulates the hardware. Virtual machines (VMs) can be created and destroyed quickly, and different workloads can be isolated from each other. Because information from one application cannot be freely accessed by another, virtualization allows for the isolation of applications between VMs and adds a layer of security.

Containerization is a lightweight form of virtualization that allows multiple applications to run on a single operating system kernel, without the need for a full operating system virtualization layer. Containers share the host operating system and underlying hardware resources, but are isolated from each other using namespace and cgroups. This results in greater efficiency and faster deployment times compared to virtual machines.

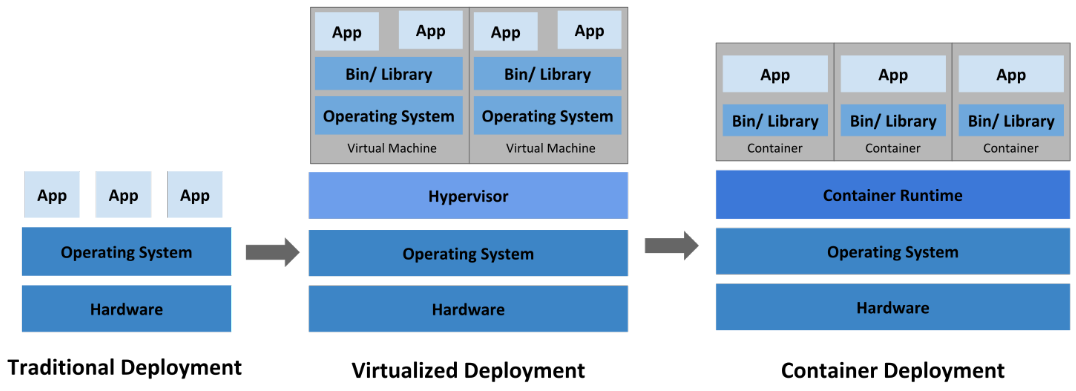


Figure 2.11: Evolution of application deployment

Containers are a good way to bundle and run applications, offering the best union of flexibility and scalability. However, there needs to be someone constantly monitoring containers running applications in a production environment to avoid downtime, to allow for another container to start up if one goes down.

Container orchestration is the automation of much of the operational effort required to run containerized workloads and services. This includes a wide range of automations involving provisioning, deployment, scaling (up and down), networking, load balancing and more. Kubernetes is the most popular open source platform for container orchestration: it handles application scaling and failover, offers deployment patterns, and does more.

2.6.2 Kubernetes Architecture

It is useful to get a sense of how Kubernetes is designed and organized at a high level in order to comprehend how it is able to provide these capabilities. One way to think of Kubernetes is as a multi-layered system, where each layer abstracts away the complexity of the one below it.

At its core, Kubernetes creates a cluster from separate physical or virtual machines by connecting them via a shared network. All Kubernetes capabilities, workloads, and components are configured on this cluster, which serves as its physical platform. Within the Kubernetes ecosystem, each machine in the cluster is assigned a specific function. The master server acts as the cluster's gateway and brain by providing an API for users and clients, monitoring the health of other servers, choosing how to divide and distribute work (a process known as "scheduling"), and coordinating communication between various components. The majority of Kubernetes' centralized logic is handled by the master server, which also serves as the cluster's main point of contact.

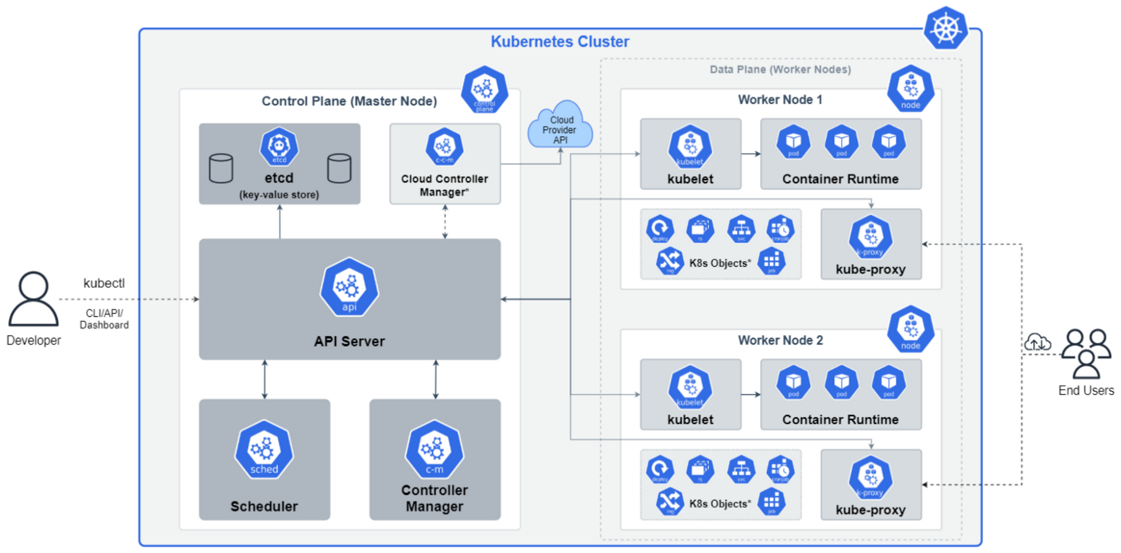


Figure 2.12: Kubernetes architecture overview

The additional machines in the cluster are known as nodes; these servers are in charge of accepting and managing workloads using both internal and external resources. Each node in Kubernetes must have a container runtime installed (such as Docker or containerd), as Kubernetes runs applications and services in containers to aid in isolation, management, and flexibility. The node modifies networking rules to properly route and forward traffic after receiving work instructions from the master server and creating or destroying containers as needed.

2.6.3 Master Server

The master server serves as the main control plane for Kubernetes clusters, as we have already discussed. It acts as the primary point of contact for administrators and users and offers a variety of cluster-wide systems for the relatively simple worker nodes. Overall, the different components of the master server cooperate to accept user requests, choose the most effective times to schedule workload containers, authenticate users and nodes, modify cluster-wide networking, manage scaling and health checking duties, and so forth.

These supporting elements ensure that the desired state of the applications corresponds to the cluster's actual state. When launching an application or service, a declarative plan outlining what should be created and how it should be managed is sent in JSON or YAML. The master server then takes the plan and, after analyzing the system's requirements and current state, determines how to implement it on the infrastructure. One machine can have these components installed, or they can be spread across several servers. In the following sections, we will examine each of the parts that make up master servers.

etcd

A globally accessible configuration store is one of the essential parts that Kubernetes requires to operate. The etcd project is a simple, distributed key-value store that can be set up to operate across multiple nodes. Each node in the cluster can access configuration data stored by Kubernetes in etcd. This can be used for service discovery and to help components configure or reconfigure themselves in accordance with the most recent information. With features like distributed locking and leader election, it also aids in maintaining cluster state. The interface for setting or retrieving values is very simple because a straightforward HTTP/JSON API is provided.

Like the majority of other control plane components, etcd can be set up on a single master server or distributed across a number of machines in real-world applications. The only prerequisite is that every Kubernetes machine can access it via the network.

kube-apiserver

An API server is one of the most significant master services. Since it enables users to customize Kubernetes' workloads and organizational units, this serves as the cluster's primary management point. Additionally, it is in charge of ensuring that the service information for deployed containers and the etcd store are in sync. In order to maintain cluster health and spread knowledge and commands, it serves as a link between various components.

Since the API server uses a RESTful interface, a wide range of tools and libraries can easily communicate with it. In order to communicate with the Kubernetes cluster from a local computer, a client called kubectl is available by default.

kube-controller-manager

The controller manager performs a variety of general services. It primarily oversees various controllers that regulate the cluster's state, manages workload life cycles, and carry out routine tasks. A replication controller, for instance, makes sure that the number of replicas (identical copies) specified for a pod corresponds to the number of replicas that are actually deployed on the cluster. The specifics of these operations are entered into etcd, where the controller manager keeps an eye out for modifications via the API server.

When a change is detected, the controller reads the fresh data and puts the procedure into action to create the desired state. To do this, an application may need to be scaled up or down, or its endpoints may need to be modified.

kube-scheduler

The scheduler is the process in charge of actually allocating workloads to particular cluster nodes. This service takes the operating requirements for a workload, examines the current infrastructure environment, and assigns the workload to a suitable node or nodes.

In order to prevent scheduling workloads that exceed the resources that are available, the scheduler is in charge of monitoring the capacity that is currently available on each host. The total capacity of each server as well as the resources that have already been assigned to running workloads must be known to the scheduler.

cloud-controller-manager

In order to comprehend and manage the state of resources in the cluster, Kubernetes can be installed in a variety of environments and interact with a range of infrastructure providers. Despite the fact that Kubernetes can operate with generic representations of resources like attachable storage and load balancers, it still needs a way to translate these to the actual resources offered by heterogeneous cloud providers.

Cloud controller manager is the glue that holds Kubernetes' interactions with providers of various capabilities, features, and APIs together while internally maintaining relatively generic constructs. This enables Kubernetes to create and use additional cloud services to fulfill the work requirements submitted to the cluster. Kubernetes can then update its state information in accordance with data gathered from the cloud provider, modify cloud resources as changes are required in the system, and adjust cloud resources as needed.

2.6.4 Node Server

Nodes are computers in Kubernetes that run containers to carry out tasks. For interacting with master components, setting up container networking, and managing the actual workloads assigned to them, node servers must meet a few requirements.

Container Runtime

A container runtime is the first requirement for each node. Containerd or cri-o are two alternatives to Docker, which is typically used to fulfill this requirement.

Applications enclosed in a largely isolated but lightweight operating environment are called containers, and the container runtime is in charge of starting and managing these applications. At the most fundamental level, each cluster work unit is implemented as one or more containers that need to be deployed. The part of each node that actually executes the containers specified in the workloads sent to the cluster is known as the container runtime.

kubelet

A tiny service known as kubelet serves as each node's primary point of contact with the cluster group. This service is in charge of communicating information to and from the control plane services as well as interacting with the etcd store to read configuration information or write new values.

To log into the cluster, receive commands, and perform tasks, the kubelet service interacts with the master components. A manifest that specifies the workload and operational parameters is used to receive work. The responsibility for maintaining the state of the work on the node server is then transferred to the kubelet process. When necessary, it launches or destroys containers by controlling the container runtime.

kube-proxy

On each node server, a tiny proxy service called kube-proxy is run in order to control individual host subnetting and make services available to other components. This process forwards requests to the appropriate containers, performs crude load balancing, and is generally in charge of ensuring that the networking environment is predictable and accessible, but isolated when necessary.

2.6.5 Objects and Workloads

Kubernetes uses additional abstraction layers over the container interface to provide scaling, resiliency, and life cycle management features, even though containers are the underlying mechanism for deploying applications. Instead of directly managing containers, users define and communicate with instances made up of different primitives offered by the Kubernetes object model. The various kinds of objects that can be used to define these workloads will be covered in more detail below.

Pods

Kubernetes works with pods as its most fundamental unit. There is no host assignment for containers themselves. As an alternative, a pod encloses one or more containers that are closely coupled.

An application that should be controlled as a single pod typically consists of one or more containers. Pods are made up of containers that work together closely, have a common life cycle, and are always best scheduled on the same node. They all share the same environment, volumes, and IP space and are managed as a single entity. They should generally considered as a single, monolithic application despite their containerized implementation to best conceptualize how the cluster will manage the pod's resources and scheduling.

Pods typically consist of a main container that fulfills the overall purpose of the workload and optionally some helper containers that facilitate closely related tasks. These are applications that would benefit from running and managing themselves in separate containers but are closely related to the main application. One container running the main application server, for instance, and a helper container that pulls down files to the shared filesystem when changes are found in an external repository, are examples of a pod. On the pod level, horizontal scaling is typically discouraged because other higher level objects are better suited to the task.

In general, users shouldn't manage pods themselves because they lack some of the features commonly required by applications (like sophisticated life cycle management and scaling). Instead, higher level objects are recommended for use by users because they add functionality while still using pods or pod templates as their foundation.

Replication Controller and Replication Sets

Managing clusters of identical, replicated pods is a common task when using Kubernetes as opposed to managing a single pod. These are built using pod templates, and controllers called replication controllers and replication sets can scale them horizontally.

A replication controller is an object that specifies a pod template and control parameters to scale identical replicas of a pod horizontally by increasing or decreasing the number of running copies. It's simple to increase availability and distribute load using this native Kubernetes technique. Because there is a template embedded in the replication controller configuration that closely resembles a pod definition, the replication controller knows how to create new pods as needed. Making sure that the number of pods deployed in the cluster and the number of pods specified in its configuration is the responsibility of the replication controller. It will launch new pods to make up for any failures caused by underlying hosts or pods. A controller either starts up or kills containers to match the desired number if the configuration for the number of replicas changes. In order to lessen the impact on application availability, replication controllers can also carry out rolling updates, which involve upgrading a group of pods one at a time.

Replication sets are an improvement on the replication controller design that offers more flexibility in how the controller determines which pods it should manage. Because they can choose more replicas, replication sets are starting to take the place of replication controllers. However, unlike replication controllers, they cannot perform rolling updates to cycle backends to a new version. Replication sets are designed to be utilized within additional, higher level units that offer that functionality. Replication controllers and replication sets, like pods, are seldom the units you work with directly. They build on the pod design to add horizontal scaling and reliability guarantees, but they lack some of the fine-grained life cycle management capabilities found in more complex objects.

Deployments

One of the most popular workloads to directly create and manage is deployment. Replication sets are used by deployments as a building block to add versatile life cycle management functionality. Despite the fact that deployments created using replication sets may seem to duplicate the functionality provided by replication controllers, they actually address many of the problems associated with the implementation of rolling updates. Users are required to submit a plan for a new replication controller that would replace the existing controller when updating applications that use replication controllers. When using replication controllers, it is either difficult or left up to the user to perform tasks like tracking history, recovering from network failures during the update, and rolling back undesirable changes.

The deployment is intended to make managing the life cycle of replicated pods easier. By simply changing the configuration, it is simple to modify deployments. Kubernetes will then adjust the replica sets, handle the transition between different application versions, and, if desired, maintain event history and undo capabilities automatically. Due to these features, deployments are most likely the Kubernetes high level object type to work with most frequently.

Stateful Sets

Uniqueness and ordering guarantees are provided by stateful sets, a type of specialized pod controller. These are primarily utilized when you need more precise control because of unique requirements for deployment ordering, persistent data, or reliable networking. Stateful sets, for instance, are frequently linked to data-oriented applications, such as databases, which require access to the same volumes even when scheduled to a different node.

By generating an exclusive, numeric name for each pod that will endure even if the pod needs to be moved to another node, stateful sets offer a reliable networking identifier. When rescheduling is required, persistent storage volumes can also be moved with a pod. To stop accidental data loss, the volumes remain even after the pod has been deleted.

Stateful sets operate in accordance with the numbered identifier in their name when deploying or adjusting scale. This provides more control and predictability over the execution order, which can be advantageous in some circumstances.

Daemon Sets

Another specialized type of pod controller are daemon sets, which execute a copy of a pod on each node in the cluster (or a subset, if specified). This most frequently comes in handy when deploying pods that assist with maintenance and offer services to the nodes directly.

Daemon sets are frequently used for tasks like gathering and forwarding logs, averaging metrics, and running services that expand a node's functionality. Daemon sets can get around pod scheduling restrictions that prevent other controllers from allocating pods to particular hosts because they frequently provide fundamental services and are required throughout the fleet. Because of its particular responsibilities, the master server, for instance, is frequently configured to be unavailable for regular pod scheduling. However, daemon sets have the ability to override the restriction on a pod-by-pod basis to ensure that essential services are running.

Jobs and Cron Jobs

All of the workloads we've discussed so far have long-lasting life cycles, such like services. Kubernetes uses a workload called a job to provide a more task-based workflow where the running containers are expected to exit successfully after some time once they have finished their work. Jobs come in handy if you need to run batch or one-off processing rather than a continuous service.

Cron jobs are built upon jobs. Cron jobs in Kubernetes provide an interface to run jobs with a scheduling component, just like the traditional cron daemons on Linux and Unix-like systems that execute scripts on a schedule. You can use cron jobs to plan an operation to run at a specific time or repeatedly. Cron jobs in Kubernetes are essentially a reimplementation of the standard cron behavior using the cluster as a platform rather than a single operating system.

2.6.6 Other Kubernetes Components

Kubernetes offers a number of other abstractions, in addition to the workloads that can be run on a cluster, to assist in managing the applications, controlling networking, and enabling persistence. Here, we'll go over a few of the more useful components.

Services

To describe long-running, frequently network-connected processes that can respond to requests, we have been using the term "service" in the traditional, Unix-like sense up until this point. Instead, a service in Kubernetes is a component that serves as a basic internal load balancer and pod ambassador. To present them as a single entity, a service logically groups together pods that carry out the same function. By doing so, you can set up a service that will be able to track and direct traffic to all backend containers of a specific type. Just the service's stable endpoint is all that internal consumers require to function. As you need it, you can scale back or replace the backend work units thanks to the service abstraction. Regardless of modifications to the pods that a service routes to, its IP address remains constant. Container designs can be made simpler and they can easily gain discoverability by deploying a service.

Configuring a service is recommended whenever is needed to give access to one or more pods to another application or outside consumers. For instance, a service will offer the necessary abstraction if there is a collection of pods running web servers that should be reachable from the internet. Similar to this, there should be set upped an internal service to grant access to the database pods through the web servers if they need to store and retrieve data.

Services can be made available outside of the cluster by selecting one of several approaches, even though they are typically only accessible via an internally routable IP address. The way the NodePort configuration operates is by opening a static port on each node's external networking interface. An internal cluster IP service will automatically direct traffic to the proper pods from the external port. As an alternative, the LoadBalancer service type uses the Kubernetes load balancer integration of a cloud provider to create an external load balancer to route traffic to the service. The appropriate resource will be created by the cloud controller manager, who will then configure it using the internal service service addresses.

Volumes and Persistent Volumes

In many containerized environments, it can be difficult to share data trustworthily and ensure its availability between container restarts. Runtimes for containers frequently offer a way to attach storage that lasts longer than the lifetime of the container, but implementations are frequently rigid.

In order to address this, Kubernetes uses its own volumes abstraction that enables data to be shared by all containers within a pod and to remain available until the pod is terminated. Therefore, without the need for complicated external mechanisms, tightly coupled pods can easily share files. Access to the shared files will not be impacted by container failures within the pod. It is not a good solution for truly persistent data because once the pod is terminated, the shared volume is destroyed.

The abstraction of more durable storage that is unrelated to the pod life cycle is made possible by persistent volumes. As an alternative, they enable administrators to set up storage resources for the cluster that users can use to request and claim for the running pods. A persistent volume's reclamation policy governs whether it is immediately removed with the data after a pod has finished using it or is kept around until it is manually deleted. In addition to allocating more storage than is physically possible locally, persistent data can protect against node-based failures.

Labels and Annotations

Labeling is a related but distinct organizational abstraction in Kubernetes from the other ideas. A label in Kubernetes is a semantic tag that can be applied to Kubernetes objects to identify them as being a part of a group. These can then be chosen when management or routing is being applied to various instances. To identify the pods that each controller-based object should operate on, for instance, labels are used. Labels are used by services to determine which backend pods they should direct requests to. Simple key-value pairs are used to represent labels. There can be multiple labels for a single unit, but there can only be one entry for each key per unit. Typically, a "name" key is used as a general purpose identifier, but objects can also be categorized according to other factors like development stage, public accessibility, application version, etc.

The ability to add arbitrary key-value data to an object is provided by annotations, a similar mechanism. While annotations are more free-form and can contain less structured data, labels are best used for semantic information that is necessary to match a pod with selection criteria. An object that is not useful for selection can generally have rich metadata added to it using annotations.

Chapter 3

SEDIA: SEat Designed for Intelligent Autonomy

This paper focuses on the SEDIA project, which is a cutting-edge mobility platform developed by Alba Robot srl. The project aims to create a self-driving wheelchair capable of functioning as a fully autonomous vehicle. One of the key objectives of the SEDIA project is to enable users to travel safely and efficiently from one location to another.

The autonomous navigation system integrated into the wheelchair, built on top of ROS2 and Nav2, consists of two main groups: the hardware architecture and the software architecture.

The hardware layer includes all the physical components of the autonomous guide, such as sensors, actuators, motors, and other perception technologies to perceive the surrounding environment. These hardware components control the movements of the guide and capture data from the environment, providing the necessary inputs for the software layer to process and make decisions. On the other hand, the software architecture focuses on the higher-level algorithms, decision-making processes, and real-time coordination of the robotic system that enable the system to navigate autonomously. This includes algorithms for obstacle avoidance, path planning, object recognition and others.

The development of the SEDIA platform is ongoing, and the team at Alba Robot is constantly working to improve its functionality and performance. The platform is being developed using the latest technologies and is expected to become an important tool for people with limited mobility.

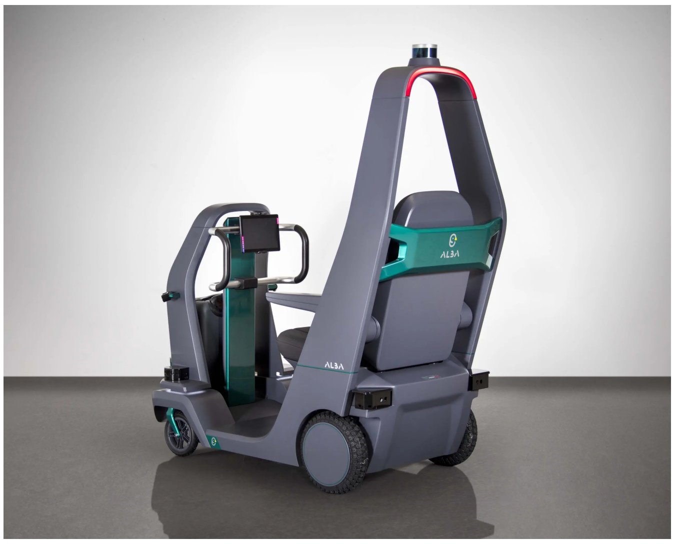


Figure 3.1: SEDIA prototype model, developed by Italdesign

The hardware and software architectures are designed to work together seamlessly, with the software providing commands to the hardware based on its perception of the environment. This allows the guide to navigate through different types of terrain and obstacles, making real-time adjustments as needed.

3.1 Hardware Layer

The Low Level's objective is to collect information about the environment surrounding the vehicle for analysis by the High Level component. To accomplish this task, the vehicle utilizes a diverse array of sensors to accurately perceive its surroundings.

3.1.1 Depth Cameras

Digital cameras typically produce images in the form of a 2D pixel grid, where each pixel is associated with values for Red, Green, and Blue (RGB). By combining thousands to millions of pixels together, we can create the photographs that we are all familiar with. However, a depth camera uses pixels with a different numerical value associated with them - the distance from the camera, or "depth". Some depth cameras have both an RGB and depth system, which allows for the capture of pixels with all four values, known as RGBD. Depth cameras use a range of technologies to capture depth information. Some of the most common technologies used in depth cameras include:

- Time-of-Flight (ToF): This technology measures the time it takes for a beam of infrared light to travel to an object and back. By analyzing the time it takes for the light to return, the camera can calculate the distance to each object and create a depth map of the scene. This results in greater data density, more precise calibration, and narrower shadows behind objects because each pixel corresponds to one light beam projected by the device. These sensors are sensitive to various surface types, including those that are highly reflective or very dark, because they use light projection. In these scenarios, the image frequently contains invalid data. In contrast to stereo sensors, which rely on the lighting of the scene, they are much more resistant to low light or dim conditions.

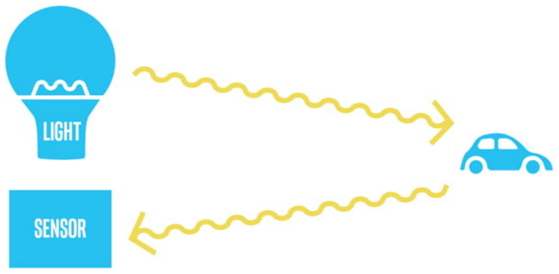


Figure 3.2: Operating diagram of a ToF sensor

- Structured light: This technology uses a pattern of infrared light projected onto the scene. By analyzing the distortion of the pattern as it reflects off objects in the scene, the camera can calculate the distance to each object. The way the pattern deforms is then used in constructing the depth map. Due to their low robustness to sunlight and the possibility of interference with the projected light pattern, this type of sensor is typically used indoors and does not require an external light source.

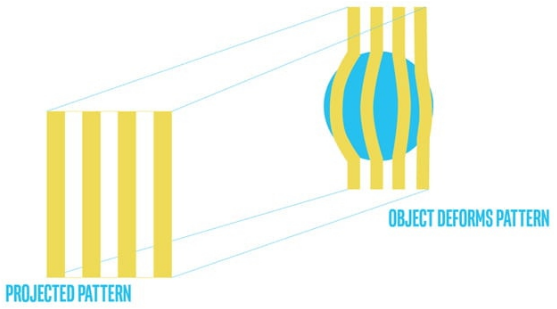


Figure 3.3: Operating diagram of a structured light sensor

- Stereo vision: try to mimic human vision by using two cameras to capture images of the same scene from slightly different angles. By analyzing the differences in the images, the camera can calculate the distance to each object and create a depth map of the scene. It is simple to obtain the depth map using the disparity information, which has an inverse relationship with depth. This also implies that some sensors of this type typically only function in environments with a lot of features and have issues in environments with few features (e. g. a pure wall). On the other hand, this type of sensor typically enables higher depth image resolutions than other types of cameras. Due to the fact that these computations for feature detection and matching are performed on the camera itself, they typically demand more processing power than alternative techniques. Furthermore, because the error grows quadratically with object distance from the sensor, the stereo working distance is constrained by the baseline between the cameras.

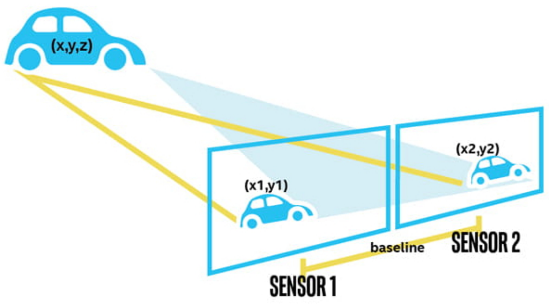


Figure 3.4: Operating diagram of a stereo vision sensor

3.1.2 LiDAR

Light detection and ranging (LiDAR) sensors are similar to ToF but with scanning, making them more accurate than ToF since they use multiple IR pulses instead of just one like ToF sensors typically do. The basic principle of lidar is simple: a laser emits a pulse of light that travels through the air and bounces off an object in its path. The sensor then measures the time it takes for the light to bounce back, allowing it to accurately calculate the distance to the object. By repeating this process many times per second, the lidar can create a detailed 3D map of the object or environment. The laser beam is split by rotating hexagonal mirrors in current lidar systems. The lower beams are used to detect lane markings and other road features, while the upper three beams are used to detect vehicles and other obstacles up ahead.

The main benefit of using lidar is that it provides spatial structure, and this data can be combined with that from other sensors to gain a better understanding of the environment around the vehicle in terms of the static and dynamic characteristics of the objects there.

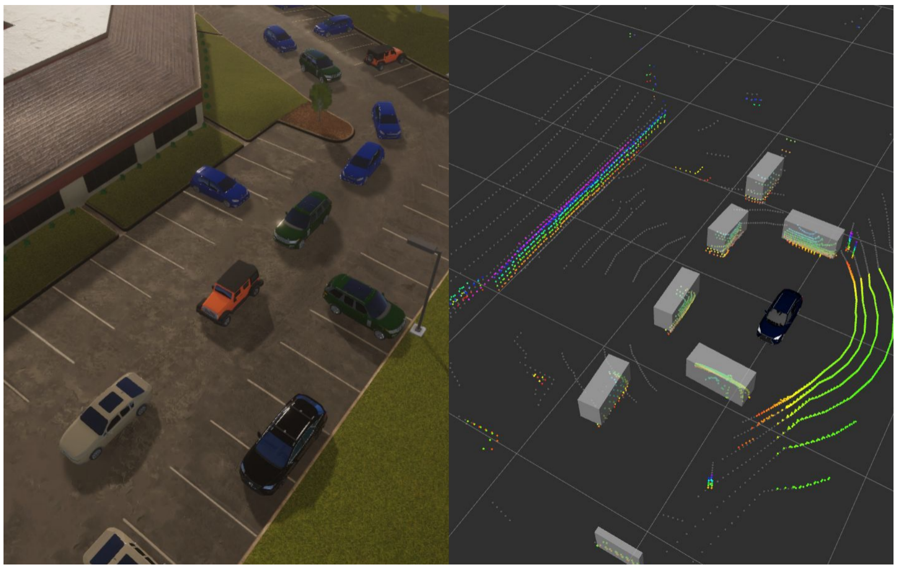


Figure 3.5: Lidar wave simulation in a road environment

Lidar sensors can capture details as small as a few millimeters, providing a level of detail that is not possible with traditional surveying methods. This accuracy is particularly useful in applications such as autonomous driving, where accurate maps are crucial for making informed decisions about moving and fixed obstacles.

While lidar technology has many benefits, there are some drawbacks. One of the most significant challenges is cost. Lidar sensors can be expensive, making it difficult for smaller companies or organizations to adopt the technology. There are also challenges related to data processing, as lidar sensors can generate enormous amounts of data that can be difficult to manage and analyze.

3.2 Software Layer

Here lies the logic of the autonomous vehicle, which is made up of ROS2 nodes that work together to give life to autonomous driving itself.

In the SEDIA software layer, multiple ROS2 nodes work together to accomplish various tasks such as perception, planning, control, and communication. Here are some of the ROS2 nodes used in the vehicle and their functionalities:

- Sensor nodes: These nodes capture data from various sensors such as LiDAR, cameras, and GPS, and publish them as ROS2 messages.

- Perception nodes: These nodes process the sensor data and extract information such as object detection, localization, and mapping. Perception nodes can use various algorithms such as computer vision, machine learning, and SLAM.
- Planning nodes: These nodes use the information obtained from the perception nodes to plan a trajectory for the vehicle. Planning nodes can use various algorithms to create a path for the robot to move from one point to another.
- Control nodes: These nodes use the planned trajectory to generate control signals for the vehicle's actuators such as steering, throttle, and brakes.
- Communication nodes: These nodes enable communication between different components of the autonomous vehicle, such as sending commands from the planning node to the control node or sending sensor data from the sensor node to the perception node.

All these nodes work together in a coordinated manner to achieve the goal of autonomous driving.

The complex algorithms that these nodes run, together with the large amount of data that the sensors provide at the high level, give room to the switching technology illustrated in the following sections, allowing the connection of the freestanding vehicles and the offloading of the more demanding tasks to the cloud infrastructure.

Chapter 4

Mobile Cloud Offloading

The IoT and robotics domains are becoming more intertwined as connected robotic systems become more prevalent. There is a lot of potential at the intersection of autonomous robots with the edge and cloud domains because these robotic systems have a tendency to be power and data intensive. To cut down on energy consumption and manufacturing costs, autonomous mobile robots like SEDIA will need to have relatively few computational resources on board. Autonomous robotic systems might also have storage restrictions in addition to computation restrictions, due to the high cost of the main memory. However, at the same time, the computational demand for robotic perception and decision-making algorithms is rising.

When compared to the processing demands of smart mobile devices' intensive applications, mobile cloud offloading can be seen as a link that bridges the gap between their constrained computing resources. The advantages of using edge and cloud computing paradigms and solutions for robotic systems are indeed enormous.

Real-time latency and bandwidth are important factors that are taken into consideration during the orchestration of computational offloading. Low-latency and high-throughput computational offloading are made possible with the introduction of 5G, 6G, and beyond mobile networking solutions by utilizing the computational resources of servers deployed across the RAN layer. The term "mobile" places more emphasis on the on-site infrastructure, which means that because it is more constrained than cloud infrastructure, it is more crucial to manage or orchestrate how it is shared among services.

In spite of that, widespread adoption requires further integration into the current tools and frameworks, particularly with ROS, the de facto standard middleware for the design and development of robotic systems. Therefore, this section aims to discuss novel design approaches, architecture, and technologies within the aforementioned context, that take advantage of the growing connectivity and pervasiveness of robots.

4.1 ROS2 Middlewares in Kubernetes

By moving computation to the cloud, we will have to interact with the problem to look into a suitable solution that is based on a software layer that enables multiple actors to communicate with each other. This layer is known as ROS2 middleware and enables communication between various parts of a robotic system. It offers a collection of tools and services for message passing, data serialization and deserialization, and inter-process communication. There are several middleware options available for building robotic systems and it is the objective of this section to choose the most suitable one to integrate with the cloud world.

According to Chapter 2.6, Kubernetes (K8s) is the most popular orchestration platform for containerized applications. It helps to manage the applications by scaling them up and down, executing updates and rollbacks, and performing self-healing, among other management functions. Although K8s makes use of Docker as its default container engine, it handles networking in a different way than Docker does by default. Pods in a K8s cluster are assigned a unique IP address within a private network, but these IP addresses are not directly accessible from outside the cluster. Kubernetes uses the concept of a Service to expose pods to external or internal network traffic, and its networking model imposes the following fundamental requirements:

- All containers can communicate with each other without the use of NAT.
- All nodes and all containers can communicate without NAT (and vice versa).
- A container's IP address is the same as what other containers see.

Therefore, in a Kubernetes environment, a networking middleware has a number of options for creating a useful bridge to enable communication between pods in the cluster. Different options are analyzed below in such a way as to be able to have a general picture and to choose the one that integrates more with the SEDIA environment.

4.1.1 ZeroMQ

Building distributed applications is made possible by sockets offered by ZeroMQ, a lightweight and high-performance messaging library. An endpoint for communication that can send and receive messages is a socket. These sockets are intended to offer a straightforward and lightweight abstraction over a number of networking protocols, including TCP, in-process, inter-process, multicast, and WebSocket.

Each of the socket types offered by ZeroMQ is intended for a particular messaging pattern. The various socket types are:

- REQ/REP(request/reply): This socket type is designed for synchronous request-reply communication between two endpoints. The server receives a message from the client containing a request, and the server responds with a message.
- PUB/SUB(publish/subscribe): This socket type is made for one-to-many communication in which a single endpoint (the publisher) sends messages to a number of subscribers.
- PUSH/PULL: This socket type is made for parallel processing and load balancing. Messages are transmitted from the push socket to a number of pull sockets, which process them concurrently.
- DEALER/ROUTER: This socket type is intended for load balancing and routing. The message is sent from the dealer socket to a number of router sockets, which then route it to the correct location based on the message's content.
- PAIR: Designed for peer-to-peer one-to-one communication, this socket type enables two endpoints to exchange messages.

ZeroMQ is made to be a high-performance messaging library that can process lots of data quickly. It accomplishes this by doing away with the overhead of conventional message queuing systems and using a lightweight messaging model. In addition to being very extensible and modular, it offers a wide range of language bindings and protocols.

Additionally, it offers a number of messaging patterns, which are guidelines for how messages are transmitted and received between sockets. These messaging patterns include:

- Request-reply: This pattern is employed when two endpoints need to communicate synchronously. A request message is sent from the client to the server, which replies with a message.
- Publish-subscribe: This communication pattern is used one-to-many. The publisher simultaneously sends messages to many subscribers, all of whom receive them.
- Pipeline: For load balancing and parallel processing, use this pattern. The pipeline is made up of a push socket that broadcasts messages to numerous pull sockets, which then process them concurrently.
- Exclusive pair: This pattern is used for one-on-one communication. Peer-to-peer messaging is used between the two endpoints.

4.1.2 AMQP

AMQP stands for Advanced Message Queuing Protocol. It is an open standard application layer protocol for message-oriented middleware. The protocol enables communication between different applications or components, even if they are written in different programming languages or running on different operating systems. At a high level, AMQP is a messaging protocol that enables the exchange of messages between different software applications or components. It establishes a common format for messages to be transmitted and provides features for dependable message delivery, routing, and queuing. Messages are encoded in a small binary format that can be effectively transmitted over a network because the protocol is, at its most basic level, a binary one. Because of its transport-agnostic design, the protocol can be used with a wide range of network protocols, including TCP/IP and WebSocket.

Clients connect to a message broker in order to send and receive messages using the client-server architecture of AMQP. The broker serves as a message hub and is in charge of directing messages to the appropriate destination. Clients can subscribe to receive messages from particular queues or topics as well as send messages to those queues or topics. AMQP's support for various messaging patterns is one of its key features. In point-to-point messaging, for instance, a message is sent from one client to another. Additionally, it supports publish/subscribe messaging, in which a message is sent to a topic and multiple clients can subscribe to receive messages on that topic.

Additionally, AMQP offers features for reliable message delivery, which ensures that messages will reach their intended recipients despite network outages or other disruptions. This is accomplished by combining message acknowledgments, message redelivery, and transactional messaging. Additional security features supported by it include encryption, authentication, and authorization. This makes it possible for applications to securely communicate over an untrusted network.

4.1.3 Kafka

Kafka is a distributed streaming platform that is designed to process and store data streams in a scalable, fault-tolerant manner while handling large amounts of real-time data. It is fundamentally based on a publish-subscribe model, where subscribers consume data that publishers produce. Data is arranged into topics, which are essentially named feeds or channels that represent a stream of records. Each record is a key-value pair that includes some data as well as optional metadata, like timestamps and headers.

Topics are further divided into partitions, which are the units of parallelism in Kafka. Each partition consists of a linearly consumable, immutable sequence of records in the ordered and immutable order. Kafka can manage large amounts of data while maintaining high throughput and low latency because partitions are distributed across a cluster of servers.

Producers are in charge of adding data to topics, and consumers are in charge of reading that data. Consumers can read from one or more partitions in parallel, and can choose where to start reading from within a partition (i.e. the offset). This enables use cases like real-time processing, stream processing, and batch processing by allowing consumers to process data in a flexible and scalable way.

Additional features and abilities offered by Kafka include:

- Replication: Kafka can duplicate data between various brokers (i. e. high availability and fault tolerance are provided by servers). Because replicas are kept in asynchronous fashion, writes to a single broker can be acknowledged right away.
- Retention: Data can be replayed in the event of failures or processing errors because Kafka can store data for a configurable length of time or size. Each topic can have its own retention settings.
- Data compression: Kafka can use data compression to lower network bandwidth and storage needs. Compression can be configured on a per-topic basis.
- Security: SSL/TLS encryption, Kerberos and OAuth-compliant authentication are supported by Kafka. Access control lists (ACLs) can be used to limit access to topics and partitions.
- Ecosystem: Kafka has a robust ecosystem of libraries and tools for working with data streams, including connectors for integrating with different data sources and sinks and frameworks for creating stream processing applications.

4.1.4 MQTT

MQTT is a simple publish-subscribe messaging protocol that works well for interacting with other systems and IoT devices. It is made to be low-overhead, which makes it perfect for devices with limited resources, like smart sensors, wearables, and other IoT devices, which typically have to transmit and receive data over a network with limited resources and bandwidth. Since MQTT is simple to set up and effectively transmits IoT data, these IoT devices use it for data transmission. MQTT implements the publish/subscribe model by defining clients and brokers as follow: any device that uses an MQTT library, from a server to a microcontroller, is referred to as an MQTT client. When a client sends or receives messages, it takes on the roles of a publisher and a receiver, respectively. A MQTT client device is essentially any device that uses MQTT to communicate over a network. The backend system that manages messages between the various clients is known as an MQTT broker. Receiving and filtering messages, locating clients who have subscribed to each message, and sending the messages to them are all duties of the broker. Additionally, it is in charge of handling missed messages and client sessions as well as authorizing and authenticating MQTT clients. An MQTT connection is used to establish communication between clients and brokers. Clients connect to the MQTT broker by sending a CONNECT message. The broker responds with a CONNACK message to verify that a connection has been made. The broker and the MQTT client both need a TCP/IP stack to communicate. Only the broker and the clients ever communicate.

Described below is a general overview of how MQTT operates: An MQTT client establishes a connection with the MQTT broker. Once connected, the client has the option of publishing messages, subscribing to particular messages, or doing both at once. When the MQTT broker receives a message, it forwards it to interested subscribers.

The MQTT protocol has become a standard for IoT data transmission because it delivers the following benefits:

- Scalable: MQTT implementation requires very little code and uses minimal amount of power when in use. In-built features in the protocol enable communication with a large number of IoT devices. So you can connect with millions of these devices by using the MQTT protocol.
- Reliable: Many IoT devices connect over unreliable cellular networks with low bandwidth and high latency. The time it takes an IoT device to reconnect to the cloud is shortened by built-in MQTT features. For the purpose of ensuring reliability for IoT use cases, it also defines three different quality-of-service levels: exactly once, at least once, and once but no more than once.
- Secure: MQTT enables developers to quickly encrypt messages and authenticate users and devices using a variety of contemporary authentication protocols, including OAuth, TLS, Customer Managed Certificates, and others.
- Well supported: the MQTT protocol implementation is well supported in a number of languages, such as Python. Therefore, it can be easily implemented by developers with little to no coding in any kind of application.

4.1.5 DDS

DDS is a data-centric publish-subscribe middleware, as shown in Chapter 2.4, that enables real-time communication between nodes in a distributed system. It is made to be fault-tolerant, scalable, secure, and meet the performance needs of real-time and embedded systems. The DDS pub/sub interaction model promotes loose coupling between applications with respect to time (i.e., the applications need not be present at the same time) and space (i.e., applications may be located anywhere).

Data-centricity is the central idea of DDS. Data serves as the medium for interaction with a datacentric middleware. The middleware understands the structure of the data it manages and it is aware of the contents (i.e., values). This enables DDS to extensively optimize performance.

4.2 The Chosen Middleware

Considering the key features of all the protocols mentioned above, we can summarize:

Building distributed applications is made simple and accessible with the help of the lightweight messaging library ZeroMQ. It is made for messaging that is low-latency and high-throughput. However, some of the more sophisticated features offered by DDS, like content-based filtering and automatic discovery, are absent from it.

The messaging protocol AMQP offers distributed systems and dependable and secure method of exchanging data. It supports a number of messaging patterns, including point-to-point and publish-subscribe. On the other hand, real-time performance is not optimized.

Kafka is a distributed streaming platform designed for creating real-time data pipelines and streaming applications. For managing large amounts of data, it offers a scalable and fault-tolerant infrastructure.

The lightweight messaging standard MQTT was created for Internet of Things applications. It offers a quick and effective way for devices to communicate with one another over data.

DDS is a middleware created for distributed systems that require real-time and mission-critical applications. It offers high-performance, scalable, and fault-tolerant communication infrastructure. DDS supports a selection of QoS policies that can be altered to satisfy particular application needs. Additionally, it offers a wealth of features that are already built-in, such as content-based filtering, data sharing, and automatic discovery, that make it simple to create and maintain distributed systems.

Although all middleware have many advantages, none of them are as well-suited to real-time performance or large scalability as DDS is. In fact, DDS offers a more complete set of features that are tailored for real-time systems and is a standardized middleware solution. Furthermore, DDS discovery in a cloud environment can function without an external discovery service if a K8s networking plugin supports multicast. The K8s networking model is a better fit for this middleware: DDS participants exchange their IP addresses for peer-to-peer communications, and therefore DDS works better over a network without NAT, where pods can find each other. Pods have unreliable IP addresses because they are dynamically assigned when they are created. As a result, pods are typically attached to a K8s service that has a trustworthy IP address and DNS name. The network traffic for the joined backend pods is then load balanced by a K8s service. Pods can find and connect with each other by topic using the DDS discovery service, which abstracts IP-based communications. As a result, the IP unreliability issue is fixed and DDS pods can now discover and communicate without the need for a K8s service.

Having said that, it is obvious how DDS could be the best choice as a middleware in a Kubernetes environment if there is a need for both reliability, real-time performance, and the need to be as consistent as possible with the ROS architecture. In fact, it has attributes like guaranteed delivery, low latency, and programmable Quality of Service policies that make it ideal for dependable and real-time communication. DDS can be a perfect solution for the SEDIA architecture because it is already used as the main infrastructure for communication.

However, real-world scenarios can make it difficult to integrate DDS into a Kubernetes cluster because CNI support for multicast isn't always feasible. The DDS middleware's multicast discovery would need to be turned off, preventing its features from being fully utilized. It is necessary to find a solution that can be applied to the most generic case study because the cluster might be managed by outside parties. Actually, as we will see in the test section later, it is necessary in a sense to disable the DDS's distributed discovery capability in favor of a centralized one, in order to merge the middleware, ROS, and K8s.

4.3 Connecting to the Cloud

The first architectural challenge we will look at is how to move SEDIA's autonomous navigation architecture's intelligence from the onboard computer of the vehicle, where it currently runs locally, to the cloud. A lot of complexity is introduced when attempting to offload deeply interconnect nodes, primarily due to the underlying communication protocol. Actually, the current DDS is built for transmission over a single local network domain, with UDP/IP as the default communication method over LAN for all DDS participants. However, when DDS applications interact with one another across various network domains, serious issues may arise. The following factors are the root causes of these issues. First off, the majority of WAN ISPs forbid multicast and UDP flows. Second, since TCP ports are typically used to forward outgoing traffic flows, there might be a lot of TCP connections between publishers and subscribers.

A DDS routing approach, which can offer effective data distribution over WAN, is here proposed as a way to get around these restrictions.

The first DDS router-based solution, developed by eProsima, and the second, created by Eclipse, are the ones we'll be looking at in-depth. They both conceptually address the same issue, but their actual technical and architectural implementation varies. Because of this, it becomes necessary to investigate these applications in order to modify each router configuration to work with the SEDIA architecture.

4.4 eProsima DDS Router

The cross-platform, non-graphical DDS Router application, created by eProsima and powered by Fast DDS, enables the construction of a communication bridge between two DDS networks that would otherwise be isolated. The primary function of the DDS Router is to connect two DDS networks that are physically or virtually apart and are connected to different LANs (see Figure 4.1). This allows the components of each network to simultaneously publish and subscribe to local and remote topics.

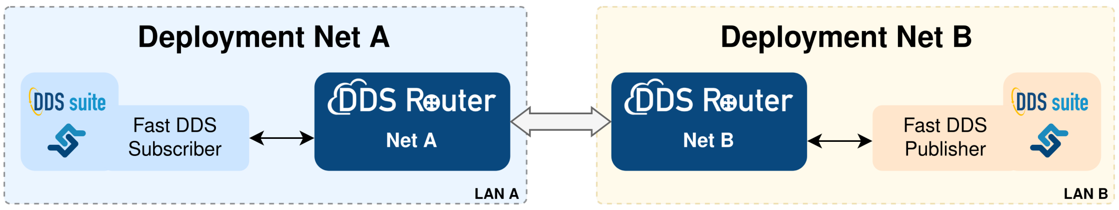


Figure 4.1: Two different LANs, connected through the eProsima DDS Router

Participants are an abstraction of DDS DomainParticipants, and they are used by the DDS Router application to run internally (see Chapter 2.x). Each one of these Participants each serve as a communication interface, like a "door" to a particular DDS network configuration.

Each participant configuration has a unique Participant Name that shouldn't be used more than once during a DDS Router execution and is specified as a different item of the participants array. Each Participant Kind (Table 4.1) is corresponding to one or more names or aliases that serve as a representation of it. The kind tag in the .yaml configuration file can be used to specify which kind of Participant to use. The Participant won't be created and the execution will fail if the kind is not one of the accepted aliases.

Participant Kind	Aliases	Description
Echo Participant	echo	Used to see in stdout the data that is being relayed by the router, as well as information regarding discovery events. All the data received by any of the Participants of the router will be printed (if data is true) with its topic and source guid, along with the payload
Simple Participant	simple local	This Participant will discover all Participants deployed in its own local network in the same domain via multicast communication,and will communicate with those that share publication or subscription topics.
Local Discovery Server Participant	discovery-server local-ds ds	This Participant will work as discovery broker for those Participants that connect to it (clients or servers). This is highly useful in networks that do not support multicast communication; or to reduce the number of meta-traffic packets exchanged in discovery, reducing the network traffic in the discovery process.
Discovery Server Wan	wan-discovery-sever wan-ds	This Participant will work as bridge for every Participant working locally in the LAN and any other LAN that has a DDS Router with an active Discovery Server WAN Participant. Each of the networks to be connected require a running DDS Router, and the messages will be relay from one to another depending on the topics filtered by each of them.
WAN Participant	wan router initial-peers	This Participant will work as bridge for every Participant working locally in the LAN and any other LAN that has a DDS Router with an active WAN Participant.

Table 4.1: Types of participants available in DDS Router configuration.

These Participants enable simultaneous connections between the application and various DDS networks. When one of these Participants receives a message from the DDS network to which they are linked, they pass the information and the message's source along to the other Participants. The initial DDS Router configuration determines how the DDS Router will operate and what topics it will cover. A yaml configuration file is used to set up a DDS Router. The DDS Router configuration, including settings for topics filtering and participants, is all contained in this file. If a file is not provided, the application will attempt to load a file named DDS_ROUTER _CONFIGURATION that must be in the same working directory where it is executed in order for the DDS Router to accept this configuration file. If no file is provided and there is no such default file in the current directory, the router application will not run. The YAML Validator tool can be used to quickly validate configuration files. The following is a general, non-exhaustive example of a configuration file:

```yaml
version: v3.0

specs:
  threads: 10
  max-depth: 1000

builtin-topics:
  - name: rt/chatter
    type: std_msgs::msg::dds_::String_
  - name: HelloWorldTopic
    type: HelloWorld
    qos:
      reliability: true
      durability: true

blocklist:
  - name: "rr/*"
  - name: "rq/*"

participants:
# Simple DDS Participant in domain 3
  - name: Participant0                # Participant Name
    kind: local                       # Participant Kind (local = simple)
    domain: 3                         # DomainId = 3

# Discovery Server DDS Participant with ROS GuidPrefix so a local ROS 2 Client could connect to it
  - name: ServerROS2                  # Participant Name
    kind: local-discovery-server      # Participant Kind
    discovery-server-guid:
      id: 1
    listening-addresses:              # Local Discovery Server Listening Addresses
      - ip: 127.0.0.1                 #Transport = UDP (by default)
        port: 11600

# Participant that will communicate with a DDS Router in a different LAN.
  - name: Wan                        # Participant Name
    kind: wan-ds                      # Participant Kind
    discovery-server-guid:
      id: 2                           # Internal WAN Discovery Server id
    connection-addresses:             # WAN Discovery Server Connection Addresses
      - ip: 8.8.8.8
        port: 11666
        transport: udp
```

Figure 4.2: Sample configuration file for eProsima DDS Router

Let's dissect this in more detail:

- version: a value used to specify the configuration version used to parse the file. This setting maintains compatibility with upcoming releases by allowing use of the same YAML file in an outdated configuration format.
- specs: The specs optional tag contains a number of options pertaining to the overall configuration of the DDS Router instance to be used. The values that can be configured are:
  - thread-pool's internal maximum thread count can be set using the optional value "threads" by the user. The number of threads that the application can spawn can be restricted using this ThreadPool. As a result, data transmission between Participants operates more effectively.
  - max-depth: a choice value that controls how long Fast DDS internal entities keep records of events. The depth of every RTPS History instance is set to 5000 by default, which limits the number of samples that a DDS Router instance can send to late joiner Readers set to TRANSIENT_LOCAL DurabilityQosPolicy kind. When the sample size and/or number of created endpoints (which rise with the number of topics and DDS Router participants) become so large as to result in memory exhaustion issues, its value should be reduced. Additionally, if enough memory is available and you want to send more samples to late joiners, you can choose to increase this value.
- built-in topics: in addition to the dynamic creation of endpoints in DDS Topics discovery, using the built-in topic list can speed up the discovery phase. The DDS router will create the DataWriters and DataReaders during router initialization by defining topics in this list. Additionally, this feature enables the manual forcing of a topic's QoS, ensuring that any entities created within that topic adhere to the specified QoS rather than the one that was initially discovered.
- topic filtering: The DDS Router enables topic filtering, which lets users specify which DDS Topics the application will relay. In DDS Router, a set of rules can be defined in this manner to filter data samples that the user does not want to forward. A configuration file does not necessarily need to contain this set of rules. In this scenario, a DDS Router will forward all data published under the topics that it automatically discovers within the DDS network to which it is connected. There are three lists available to define these data filtering rules according to the Topics to which they belong:
  - allowlist is a list of topics that DDS Router will forward, i.e. DDS Router will forward any data published under topics that match the allowlist's

  expressions. Data will be forwarded for all topics if no allowlist is provided (barring blocklist filters that exclude certain topics).
  - blocklist: This is a list of topics that the DDS Router will block, meaning that any data published under these topics that matches the filters listed in the blocklist will be ignored by the DDS Router and not relayed. Moreover, when a topic matches an expression in both the blocklist and the allowlist, the blocklist takes precedence, and the data for this topic is discarded.

- participant configuration: there are various Participant configurations that participants can handle. Every participant must specify the type of participant and is given a special participant name. Any number of Participants and multiple participant types are possible. The Domain Id of a particular Participant is configured by the tag domain. Network Addresses are components that may be set up for particular Participants.

  The following are the characteristics of an address:

  - ip: the host's IP (public IP in WAN communication cases).
  - port: where the Participant is listening.
  - external-port: a public port that is only open to external entities using TCP. In the absence of this value, the external port is taken to be identical to the internal one.
  - transport-protocol: UDP or TCP are availabe. If it is not set, the Participant Kind will decide by default.
  - ip-version: either v4 or v6. If it is not set, the IP string format would be used to make a decision.
  - domain-name: use a distinct domain name to query a DNS server for the relevant IP. If ip is specified, this field is ignored.

- A DDS GuidPrefix is a Global Unique Identifier shared by a Participant and all of its Sub-Entities that is required by a Discovery Server. Uniquely identifies a DDS Participant. (so that other Participants can connect to it.). An id, with which the GuidPrefix will be calculated automatically, or a string, which represents a group of 12 hexadecimal numbers separated by a point, can be used to configure this prefix.

- The network addresses that this Participant will use to listen for remote Participants are configured by the tag listening-addresses.

- Configuring a connection with one or more distant Discovery Servers can be done using the tag connection-addresses. This key controls an array whose elements each have a GuidPrefix referencing the Discovery Server to connect to and a tag addresses setting up the addresses of that Discovery Server. Each component of addresses must adhere to the network address configuration.

Let's now examine how the router can be utilized now that its features have been presented. Providing effective data distribution over WAN is the DDS router's primary function, as was already mentioned. A DDS Router instance will be present on each end of the communication to achieve this. The Figure 4.3 depicts the actual testing environment accurately by demonstrating how two distinct network domains are linked together via a WAN link.

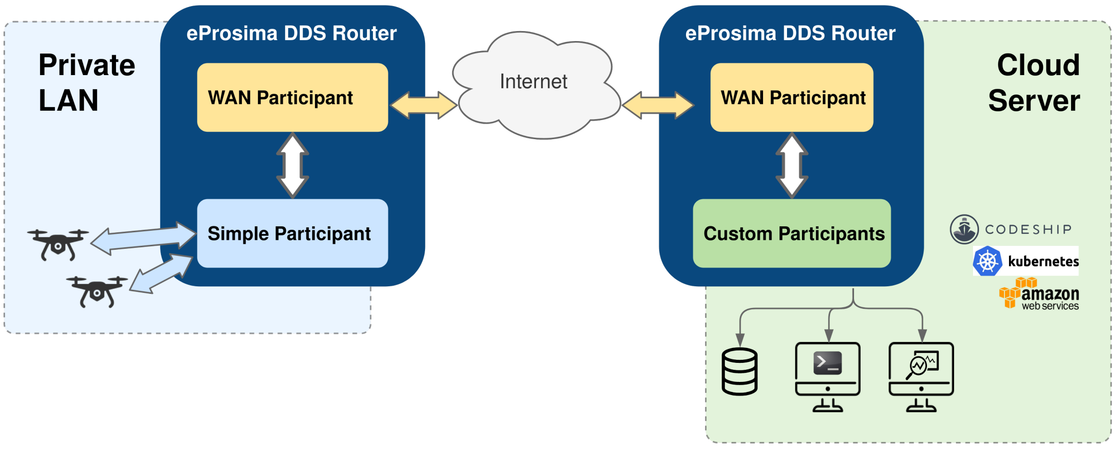


Figure 4.3: Local Side and Remote Side communicating through shared network via eProsima DDS Router

Looking into the two sides by using this architecture as a point of reference:

Private LAN: The network domain where SEDIA nodes are deployed is referred to as local side. These nodes are operating unaltered and flawlessly; they are aware of the DDS router's presence thanks to the DDS discovery mechanism, and they communicate with it in the same manner as any other participant in the SEDIA.
The router must be set up with two participants: the Wan Participant will act as the client in the discovery of the remote participant, and the SimpleParticipant will discover the participants deployed in its own local network. To be able to hear all local data exchanged by the ROS2 nodes, the Simple Participant must be set up to listen at the DDS domain (domain tag) that corresponds to the ROS_DOMAIN_ID of the SEDIA nodes. Once we have confirmed that the remote DDS Router we wish to connect to is operational, we must write down its listening Network Address, which consists of an ip, a port, and a transport protocol (TCP/UDP); as we will see, these parameters

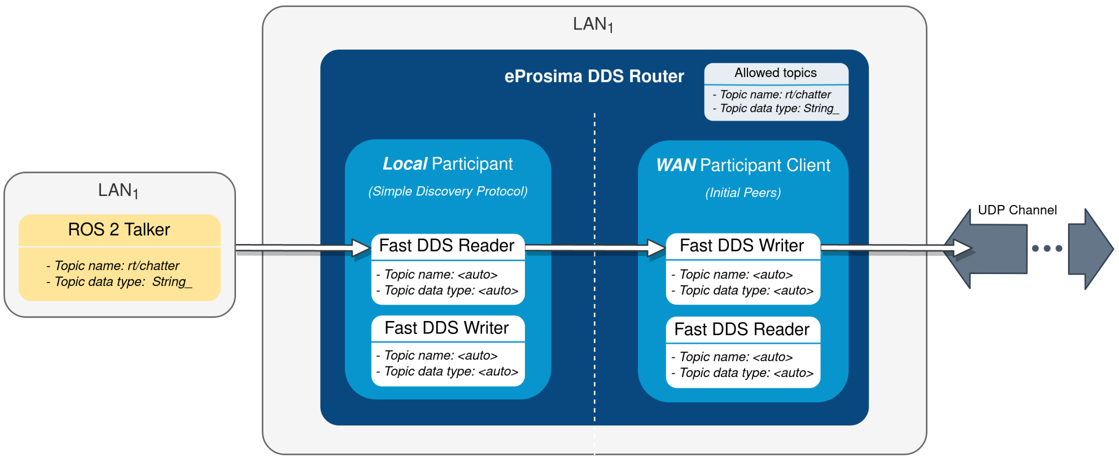


Figure 4.4: Inner view of the Local Side

depend on the configuration of the cloud side. Then, using a configuration file similar to the one previously displayed in Figure 4.2, we can use that information to configure our own DDS Router instance that we want to run in our LAN, whose tags need to contain the appropriate values.
The local participant will begin to receive messages published by the ROS 2 talker node and will then forward them to the WAN participant once they have acknowledged each other's existence through Simple DDS discovery mechanism (multicast communication). Then, these messages will be sent to a different participant located on a K8s cluster to which it connects using WAN communication over UDP/IP.

Cloud Server: The duplicate SEDIA node instances that were chosen as being able to be moved to the cloud server are located here on the cloud side. A second DDS Router instance that is set up similarly to the local one must be launched as well. The Server instance only needs a listening Network Address (listening-addresses tag) and the Discovery Server Guid (id tag) in this instance, so no connection-addresses tag is necessary.
The WAN Participant will establish a connection with the local side, making the pairing request, and will act as a server to that client, receiving all data sent by it. The Local Discovery Server Participant will then transmit these data to the configured ROS_DOMAIN_ID, which publishes all data into the cloud DDS network. The discovery server address, which the DDS protocol uses to forward all messages and from which it can discover other participants, must be set up in order to enable the ROS2 Pod to communicate in this architecture.

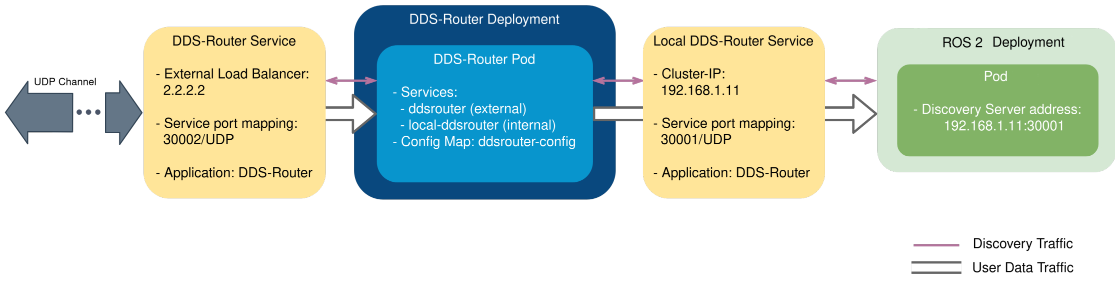
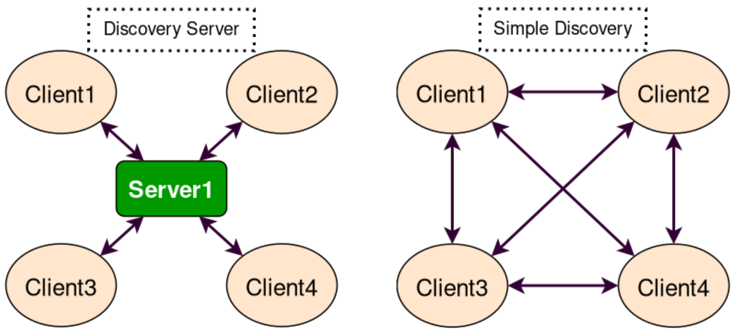

UDP Channel
DDS-Router Service
- External Load Balancer: 2.2.2.2
- Service port mapping: 30002/UDP
- Application: DDS-Router
DDS-Router Deployment
DDS-Router Pod
- Services:
- ddsrouter (external)
- local-ddsrouter (internal)
- Config Map: ddsrouter-config
Local DDS-Router Service
- Cluster-IP: 192.168.1.11
- Service port mapping: 30001/UDP
- Application: DDS-Router
ROS 2 Deployment
Pod
- Discovery Server address: 192.168.1.11:30001
Discovery Traffic
User Data Traffic

Figure 4.5: Inner view of the Cloud Side

Because multicast routing in cloud environments is difficult to enable, a Local Discovery Server, rather than a Simple Participant, is used to communicate with the active pods.
This is the key component of the adopted solution. In most cases, multicast is supported within the same Local Area Network, just like broadcast traffic, so all intended destination nodes can be reached and they will receive and recognize the traffic published in the active topics. However, in a cloud environment like Kubernetes, for example, multicast traffic between pods is frequently not supported. This behavior is determined by the CNI (Container Network Interface) that is used in the cluster. Since the cluster may be managed by outside parties, it is frequently impossible to tailor it to the needs of the application. Solving this problem can be very difficult. The CNI determines whether multicast support is available or not because it controls how the network resources are used. The DDS Discovery Server Protocol feature offers a solution to this issue.



(a) Discovery Server (b) Simple Discovery

Figure 4.6: Architecture comparison between Discovery Server and Simple Discovery

As opposed to the distributed mechanism used by DDS by default, it offers a centralized dynamic discovery mechanism. This mechanism is built on a client-server discovery paradigm, where the metatraffic (message exchange among DomainParticipants to identify each other) is managed by one or more server DomainParticipants (Figure 4.6.a), as opposed to simple discovery (Figure 4.6.b), where the metatraffic is exchanged using a message broadcast mechanism like an IP multicast protocol.

In addition to the two deployments already mentioned, two K8s services are needed to direct dataflow to each of the pods, as shown in the Figure 4.5. A ClusterIP service will provide the network address through which messages flow from the local Discovery Server to the pods that are waiting for them, and a LoadBalancer will forward messages reaching the cluster to the WAN participant of the cloud router.

4.4.1 Experiments and Results

A number of tests were run, gathering a ton of data on various parameters, to validate the viability of such an approach. A lot of attention was paid to the amount of computing power required by the instances of the DDS router needed to connect the two LANs, as well as the time it took for a set of messages to be sent and received between ROS nodes deployed on the local side and the cloud side, respectively.

4.4.2 Single Host

Figure 4.7 illustrates the latency caused by the DDS protocol when sending messages to nodes that are directly running on the same host's bare metal. It also depicts a different scenario in which a DDS router instance is deployed and serves as a message broker for the local nodes. The publication rate is constrained to 2 messages sent every 1 second, but it is parameterized and changeable as desired.

Measurements demonstrate that the DDS protocol, which is a low-latency middleware, succeeds at message delivery in less than 1 ms on average. When validating all of the results, this is the order of magnitude that we are aiming for. The proposed solution can only be successful if it operates as closely as possible to this value.

In order to determine whether the processing time added 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. As stated earlier in this section, the goal is to have ROS2 nodes communicate through an instance of the eProsima DDS Router. The router has been specifically set up to connect two different ROS Domains because there is only one host in this instance, simulating the behavior of a WAN scenario.

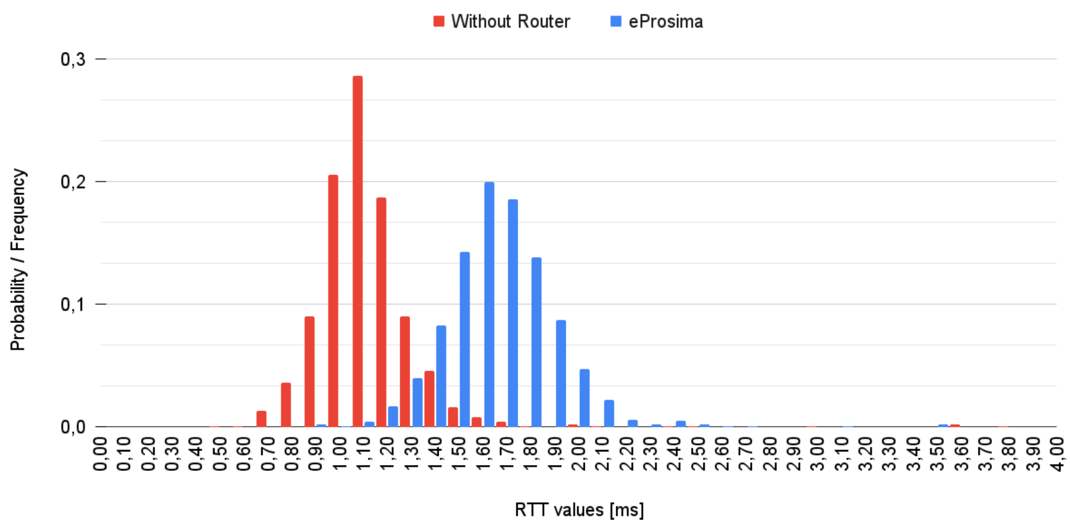


Figure 4.7: In red, the RTT of a message without crossing the router. In blue, the RTT of a message managed by eProsima

We essentially see no difference in trend and obtained values, other than a slight shift, when we compare this result to the bare DDS protocol, where the same data were exchanged inside the same exact machine without the router. This indicates that a small amount of time is added to the message exchange due to the DDS Router processing, but overall, this time is less than 1 ms, and the order of magnitude is unchanged.

To put it another way, it is obvious that the DDS Router does not significantly increase the computation of latency. However, this is merely a simulated environment with two Simple Participants to allow communication between ROS2 nodes using different ROS_DOMAIN_IDs. We'll see that the way a WAN Participant is used may alter the results.

4.4.3 Remote Hosts

This is a more complex scenario where the local nodes are connected to the server's network interface thanks to two instances of the DDS Router, one local and one server-based. The local WAN Participant gathers local messages and sends them to the cloud nodes. Naturally, a Simple Discovery Server Participant was sufficient for local servers communicating in the same LAN, but a Local Discovery Server Participant was required for cloud servers.

Different scenarios were taken into account in order to gather information about how reliable a router-centric solution could be. However, it is important to note that the execution environment of these tests does not accurately reflect the real situation, particularly because the two machines, on which the local nodes and the cloud nodes, respectively, are installed, are close to one another and are connected via a physical LAN, which has greater stability and reliability than a wireless connection.

4.4.4 Multiple Nodes

Five different configurations have been chosen to test the DDS Router's capacity to handle the discovery and message-forwarding of multiple nodes. Ten simple chatter nodes, provided by the demo_node package of the ROS2 stack, for instance, were deployed on the local side in the first instance. It is important to note that each node instance has a cloud counterpart that is deployed there as well. A message is exchanged every second between each pair of these nodes. At the same time, a custom node is measuring the change in the round-trip time between the local and cloud, while also taking the router's overhead into account.

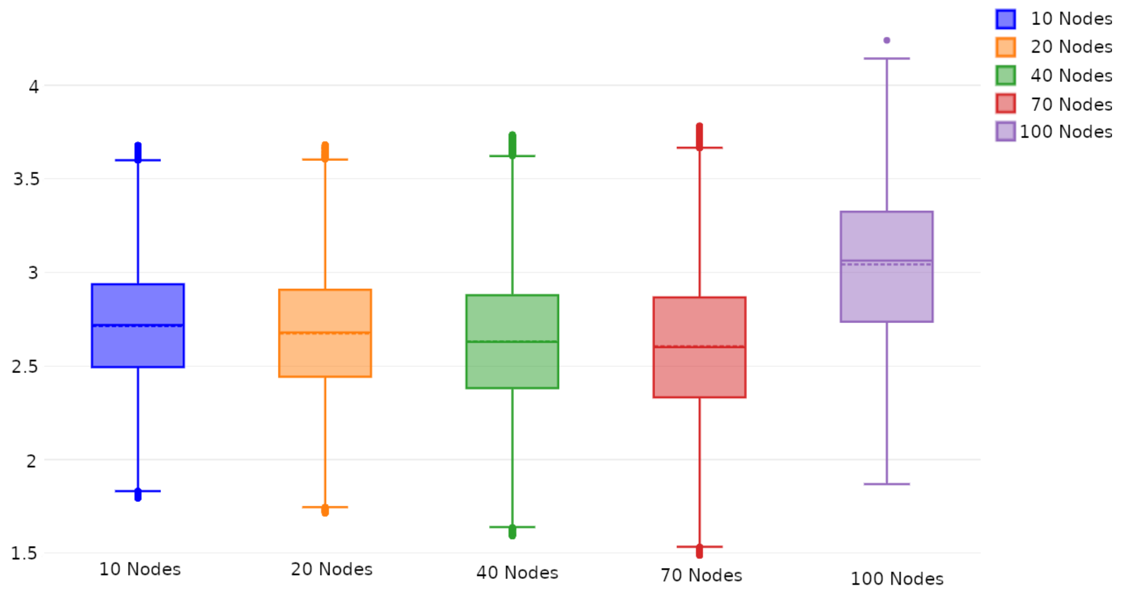


Figure 4.8: RTT analysis when multiple nodes are present

A boxplot of the data distribution obtained for all the scenarios we have taken into consideration is a useful tool for comparing the results previously described. Figure 4.8 displays the final outcomes.

In every scenario taken into account, the router responds very well. Particularly in the first four cases, the latencies are practically equivalent. The router didn't change much until it was stressed beyond its capacity. In fact, running 100 nodes proved to be a little more difficult, but overall the test was successful because the order of magnitude remained the same on average.

We can only fully comprehend what is happening, though, when we take into account the outliers. Figure 4.9 shows a very different pattern of behavior than before: while many messages report a low RTT, the delay has occasionally increased to tens of milliseconds.

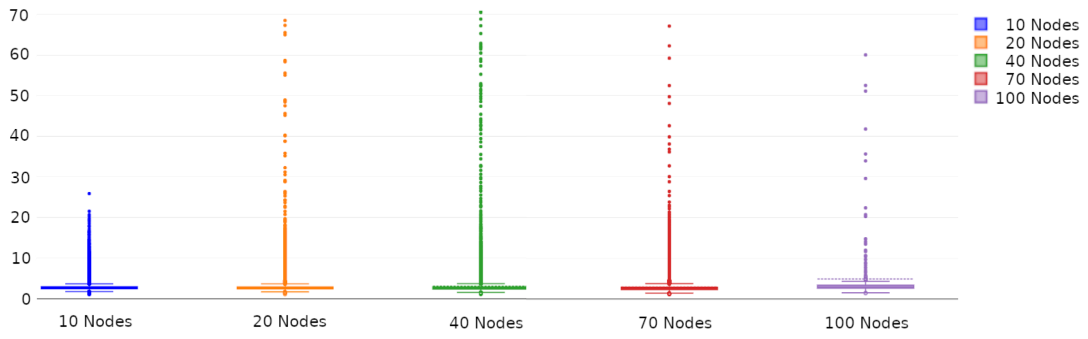


Figure 4.9: RTT outliers with multiple nodes

It can be challenging to explain this trend, but it makes sense when you look at the router's overall CPU usage (see Figure 4.10). Suddenly, everything is made clear. Prometheus, a flexible monitoring tool for capturing and processing numerical time series, produced this graph. Prometheus is an excellent fit for Kubernetes because it is scalable, simple to integrate with, has a rich query language, and offers strong monitoring and alerting capabilities.

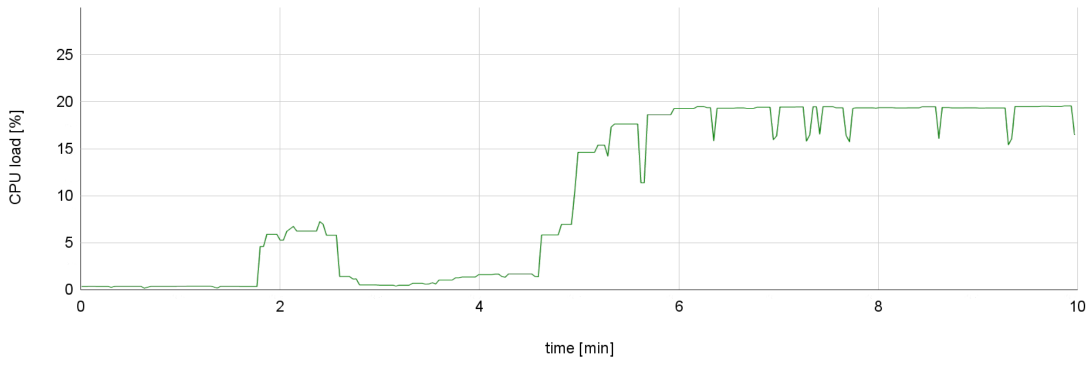


Figure 4.10: CPU load of 10 chatter nodes

The CPU usage of the router during the time frame associated with the creation of 10 cloud-side nodes is depicted in the first half of the figure. As previously mentioned, the router establishes a shadow participant for each node currently connected to the domain. This comes with a price, and it depends linearly on the quantity of active nodes. When there are many nodes, there are also many domain participants, which results in many shadow copies being created by the router. Once all nodes have been located and these copies have been made, however, this cost cancels out.

According to Figure 4.11, as the number of nodes rises, the initial peak has a strong impact on the CPU load. When 100 nodes were running simultaneously, a critical behavior took place. Due to the sheer number of nodes that need to be found, the router rapidly reaches CPU saturation before finding every node. This led to the conclusion that the number of nodes is just as crucial to the viability of this solution as latency.

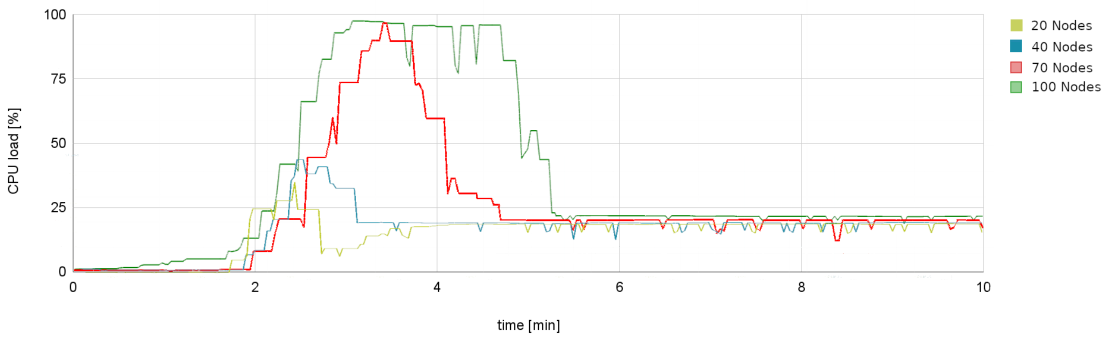


Figure 4.11: CPU load comparison between different deployment scenario

The right side of the graphs shown above will now be the focus. Since all domain participants have been found at this time, the router is now free to perform message forwarding. It can be seen that the trend does not change as much as it did during the discovery phase by comparing all the cases mentioned above. In contrast, even as the number of nodes rises, the CPU load essentially stays the same. Only a very slight increase is noticeable when there are a lot of participants, but this difference in load is almost irrelevant when compared to the total number of participants.

4.4.5 Varying Delays

As previously stated, a real-world scenario where the local side and cloud side are connected via a wireless connection may behave differently than the one taken as an example before. This is why it became necessary to use a tool called netem. By simulating the characteristics of wide area networks, it offers network emulation functionality for testing protocols. Netem is a Linux traffic control facility enhancement that allows addition of delay, packet loss, duplication, and more other characteristics to packets passing through a chosen network interface. Here, we've examined the scenario where a connection experiences various network delays.

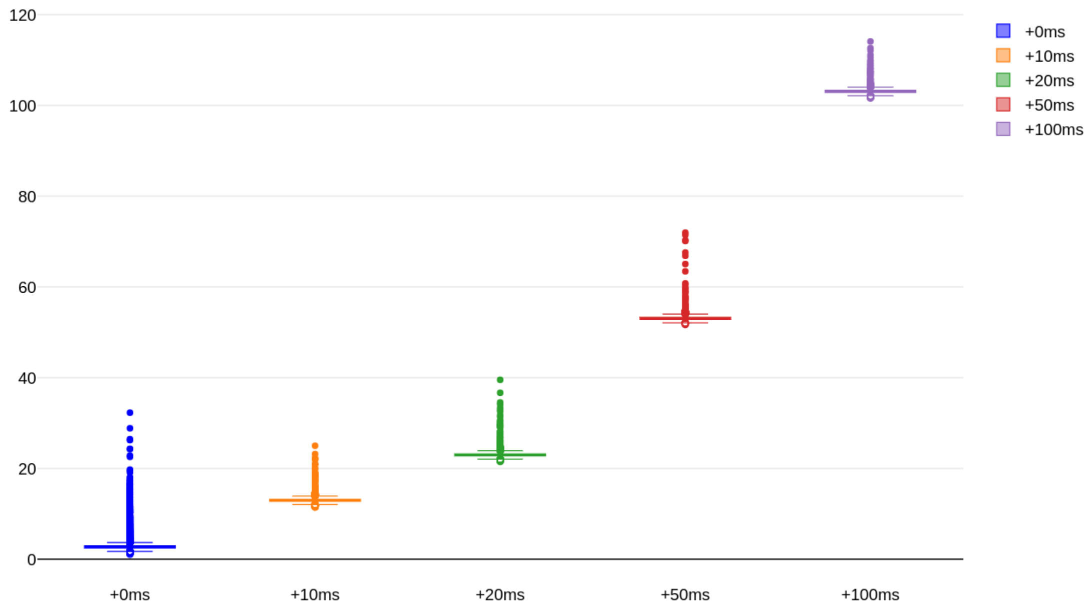


Figure 4.12: RTT with the addition of increasing latencies

The data on display was obtained using a variety of configurations, from the default setting in which there was no communication delay added, to the limit scenario in which a 100ms delay was simulated. The outcomes demonstrate that the router is able to control the message exchange over a link with a potentially high delay without any issues. However, because DDS is a real-time protocol and SEDIA is based on this concept specifically, this does not imply that a practical solution is possible in which the RTT between local and cloud is very high (e.g. 100ms). Too much delay would completely harm the performance and especially fundamental operations of the autonomous system.

4.4.6 Varying Packet Drop Rate

Finally, once more with the aid of the netem tool, it was examined how the router functions when it must communicate over a network that introduces some degree of packet loss. Generally, when a network packet does not get to where it should, information is lost and packet loss occurs.

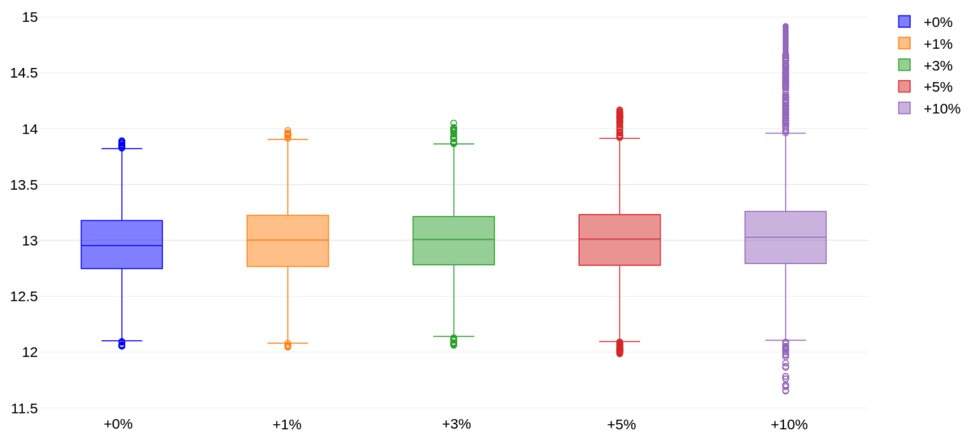


Figure 4.13: RTT with the addition of increasing packet drop rate

Figure 4.13 displays five distinct scenarios, each with a rising percentage of packet loss. Taking into account a 10ms delay between the local and cloud sides, the RTT in every case is, on average, around 13ms. It appears that experiencing packet loss has no impact on how well the router works. Similar to the first test case, the true behavior is only understood when outliers are taken into account (see Figure 4.14).

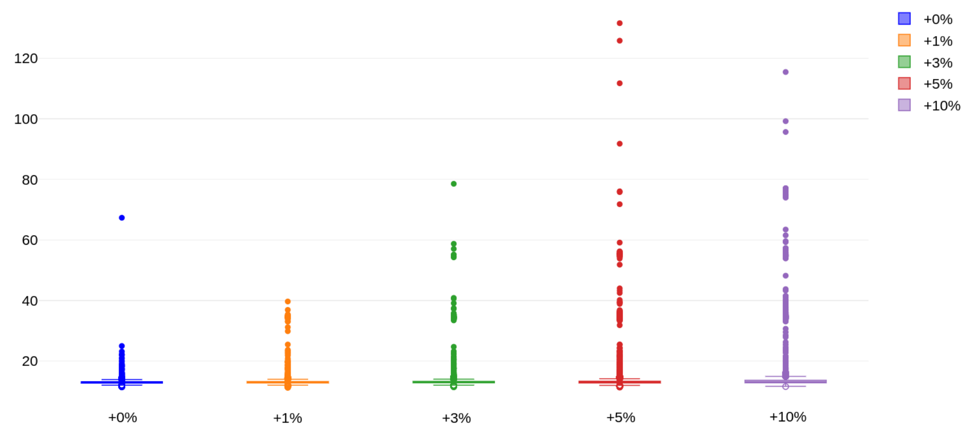


Figure 4.14: RTT with the addition of increasing packet drop rate, zoom out

The distribution of the gathered data is depicted in great detail on the graph above, which emphasizes how the message delay increases as the quality of the connection deteriorates. Because of how the router's QoS is configured (which, in turn, has an impact on the DDS middleware's QoS), dropped messages are still able to travel to distant nodes and back. In fact, it is possible to fine-tune these policies, choosing, for example, whether to treat a message as RELIABLE or BEST_EFFORT. In the first instance, the router ensures that the message is delivered to the remote side; in the second instance, it does not.

Although the best_effort version is undoubtedly the quickest, most effective, and least resource-demanding method of obtaining the most recent/latest value for a topic (in terms of CPU and network bandwidth), there is no assurance that the data sent will be received in a WAN scenario.

4.5 Zenoh Router

As was previously mentioned, ROS2 has adopted DDS as the method for data exchange between nodes inside and possibly across a robot. The execution of analytical comparisons between different DDS implementations is motivated by their existence. However, scaling DDS communication across a WAN or multiple LANs is not as simple as it might seem because of some of the fundamental tenets that form the basis of the DDS wire-protocol, in addition to the fact that it uses UDP/IP multicast for communication. We concentrate particularly on an architecture, which employs Zenoh Router, which is conceptually very similar to that pertaining to eProsima (see Figure 4.15).

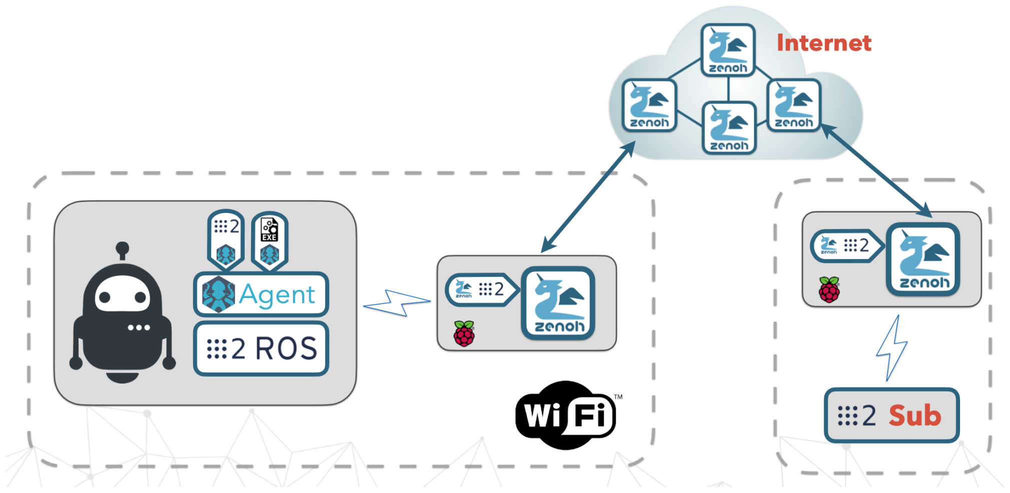


Figure 4.15: Overview of a possible use of the Zenoh router in a WAN scenario

Since its launch, Zenoh has been built to work at Internet scale. It combines traditional pub/sub mechanisms with geodistributed remote computation to provide a unified data management that can scale up quickly. When compared to DDS, Zenoh takes a very different approach to discovery. In fact, DDS exchanges every writer and reader who is available for a given topic's tiniest, most minute details, as well as the topic's information if that information is available.

The SPDP (Simple Participant Discovery Protocol) and SEDP (Simple Entity Discovery Protocol) are two of the discovery protocols used by DDS. In essence, the SPDP handles locating nearby individuals; in DDS jargon, it locates domain participants. When a new domain participant is found, the SEPD steps in to initiate a mutual exchange of the complete list of data-readers, data-writers, and optionally topics. As a result, the quantity of discovery data produced by real-world applications increases quadratically with node count. More specifically, if we have a system with n domain participants, each of whom has r readers and w writers, then the volume of discovery traffic scales with n*(n-1)*(r+w).

Different rules apply in Zenoh. DDS experiences a significant difference depending on whether a writer matches one or one million readers in terms of the amount of discovery traffic generated, reliability protocol overhead, and discovery data to store in memory. In order to lessen this unfavorable side-effect, Zenoh can predict the resource used by an application despite the quantity of "matching" readers and writers. Another characteristic of this router is that the protocol (along with the configuration given to it) may choose to make generalizations, raising the level of abstraction and attempting to distance itself from the specifics. This is a crucial mechanism for discovery data compression and for making sure the data can withstand Internet-scale applications.

This router will be used to connect the local and cloud-based ROS2 node instances in this section, and its architecture will be described, along with how to work with the DDS middleware and Kubernetes clusters.

How to transparently bridge the ROS2/DDS communication using Zenoh is the main issue to be solved. In actuality, Zenoh uses a custom protocol that is not directly compatible with the DDS standard. To transparently bridge DDS communication over Zenoh and vice versa, the Eclipse Zenoh Project provides the zenoh-plugin-dds. The OMG Data Distribution Service is implemented by CycloneDDS, which is used by the Zenoh/DDS bridge to find the DDS readers and writers declared by the ROS2 application. The bridge builds a mirror DDS-entity for each discovered DDS participant; in other words, it builds a reader when discovering a writer and vice versa. The bridge also makes the appropriate declarations and maps the DDS topics read and written by the identified DDS entities on zenoh resources. This service can be used to interact with DDS or ROS2 applications through the zenoh ecosystem or to route DDS traffic over zenoh, for example, to remotely control a robot. It can be dynamically loaded by a zenoh router because it is an Eclipse zenoh plugin, but there is also a standalone version.

By default, the bridge does not route DDS discovery traffic to the remote bridges through Zenoh. This means that the DDS entities found in one domain won't be advertised in the other domain if you use two Zenoh-bridge-DDS to connect two DDS domains. Therefore, only if matching readers and writers are independently declared in the two domains will the DDS data be routed between them.

The ROS graph on one side of the bridge might not accurately reflect all of the nodes from the other side of the bridge as a result of this default behavior. There's a chance that the ros2 topic list command won't include every topic that has been declared on the other side. The ROS graph is also restricted to the nodes in each domain.

However, all bridges behave differently when the *-fwd-discovery* option is used:

- Each bridge will transmit local DDS discovery data to the distant bridges via Zenoh (in a more efficient manner than the original DDS discovery traffic).

- Each bridge that receives DDS discovery data via Zenoh will build a copy of the DDS reader or writer with a comparable QoS. The ROS2 nodes will find those replicas, which will serve the route to and from Zenoh.

- Each bridge will send the ros_discovery_info data to the distant bridges (in a less intensive manner than the original publications). Upon receipt, the remote bridges will change the GIDs of the original entities into the GIDs of the corresponding replicas and republish the ros_discovery_info on DDS. The ROS2 nodes on each host can then find the entire ROS graph.

4.5.1 Experiments and Results

4.5.2 Single Host

In order to have enough information to compare the Zenoh router with that of eProsima, the same set of tests as previously reported were carried out. As a result, only the outcomes that are most pertinent are reported.

First, we looked at how much latency the router brings to a situation where there are two nodes that live in different domains, running on the same host and communicating through the router.

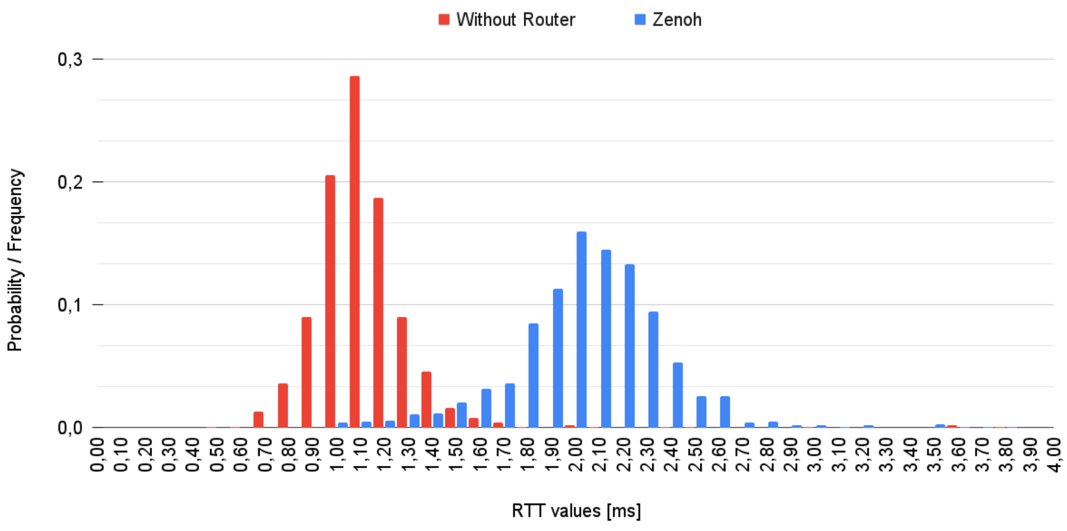


Figure 4.16: In blue, the RTT of a message without crossing the router. In red, the RTT of a message managed by Zenoh

The outcomes are exactly what we expected. The router adds the least amount of latency (on average, around 1ms). The result is truly acceptable and consistent with the test's behavior as previously reported.

4.5.3 Remote Hosts

This configuration resembles that of the eProsima router pretty closely. The local side is perfectly the same, while the architecture chosen for the cloud side and shown below is very similar to the one previously presented, however, it has been modified to work with the Zenoh protocol.

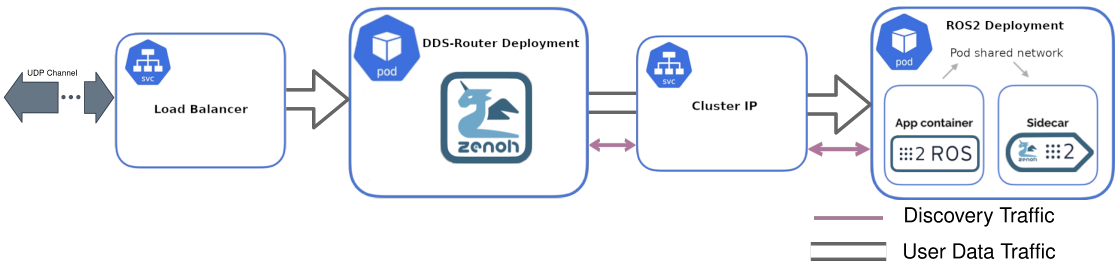


Figure 4.17: Overview of the deployment of Zenoh router in Cloud Side

In fact, an instance of zenoh-bridge-dds has been inserted into each of the Kubernetes pods in which the ROS nodes run to ensure that they can communicate with one another and with the outside world. The bridge and the ROS node are both containerized applications that live in the same pod. There are situations like this one, where enclosing multiple containers in a pod is required, although configuring a single container in a pod is typically the use case. This typically happens when there are two or more containers that are closely coupled and demand the same resources.

This pattern is known as sidecar containers. They are additional containers that run alongside the primary container. The main and sidecar containers also share the pod network, and the pod containers can communicate with each other on the same network using localhost or the pod's IP, reducing latency between them.

Therefore, the bridge connects each pod, through a Cluster IP, to the zenoh router, which in turn connects the cloud side to the local side thanks to the Load Balancer. The test's results are shown in Figure 4.18.

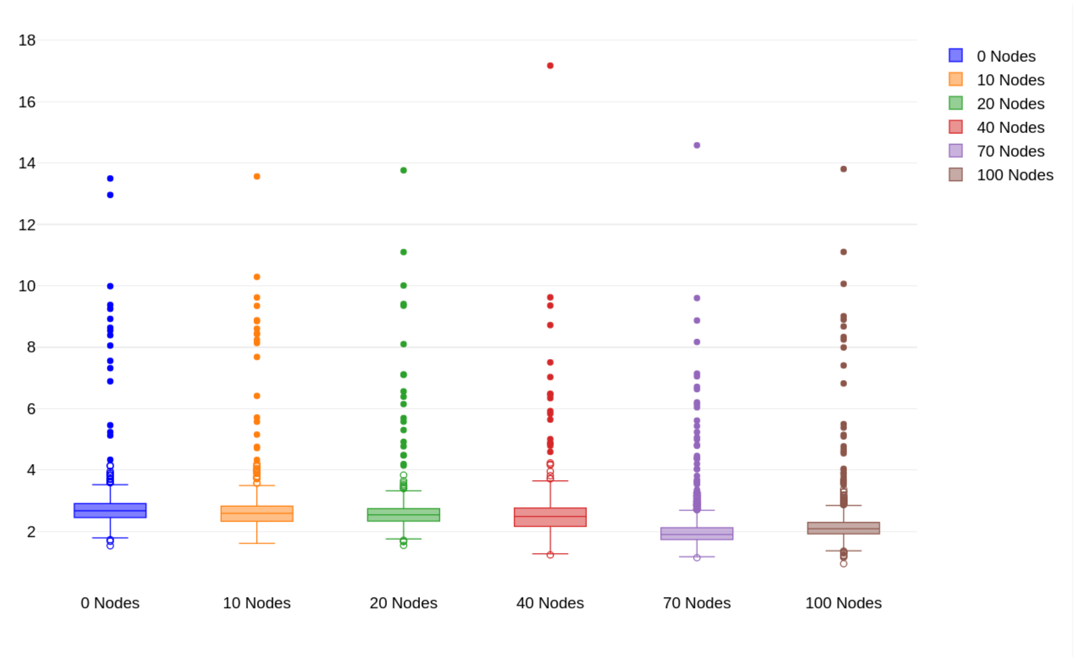


Figure 4.18: RTT with multiple nodes, detailed view

The ability of Zenoh to successfully extricate himself when it's time to find the various nodes in the domain is something we wanted to emphasize. The same scenarios as in the earlier tests were taken into consideration, with special attention paid to measuring the RTT of messages traveling across the wan and returning to the local side.

The router performs significantly better overall, and especially during the discovery phase, as shown by the graph. The router as a whole became much

more stable and effective because those peaks of many tens of ms were no longer observed (in fact, in this case, a threshold of 20 ms is never exceeded).
Given the excellent results of these tests and the fact that, at the time this thesis was being written, the SEDIA team decided to work with the cyclonedds middleware, we can say that, of the two suggested solutions, the Zenoh router is clearly the better option.

4.6 Wireguard VPN

The third and last solution that this thesis has explored consists in exploiting a VPN between the local and the cloud side.

A virtual private network (VPN) is a method for establishing a secure connection over an insecure communication channel between two networks or between a computer and a network. An encrypted tunnel is used by a VPN connection to safely transfer data over the open internet rather than relying on costly hardware to create closed-off networks.

By establishing connections over the Internet, a VPN expands a private local network. In other words, by using a VPN, a device can remotely connect to a private network and behave exactly as if it were physically connected to that network: that device will be a member of the private LAN and have a proper private IP address. To reach the mentioned LAN, the private traffic will travel over the internet.

Two endpoints are needed for a conventional VPN. A remote endpoint is one, and a local endpoint is another. Both endpoints must be set up and configured to send and receive data using a VPN protocol before the VPN connection can be established. Third-party clients, integrated OS features, and network-based implementations are a few of the different ways VPN functionality can be implemented. Anyhow, the VPN must be identical to or compatible with the VPN setup employed on the other endpoint. Once both endpoints are set up and configured, they establish a connection known as a VPN tunnel. The connection may always be active or may be dynamically activated by the user or specific circumstances.

When connecting remote endpoints, various VPN types can be used:

- site-to-site (s2s): when connecting multiple local networks to a virtual communication network using a public transportation medium, a site-to-site VPN is used. For instance, connecting various corporate locations to one another is a possible scenario. As an alternative, a corporate network can also be used to realize location networks. Corporate networks are built on a private fixed connection, but in order to use it, businesses must first rent the necessary infrastructure. On the other hand, a VPN connection is dependent on a public network. Here, the only expense is for an internet connection. A VPN router is necessary to establish the link between the VPN tunnel and the local network when setting up a site-to-site VPN.

- end-to-end (e2e): the end systems themselves serve as the tunnel endpoints rather than VPN Gateways, which are edge devices. As the tunnel endpoints are the actual devices, a VPN connection is established between two devices in this instance rather than two networks. This type of VPN can be thought of as a device-to-device connection and can be used with different mobile devices.

- remote access: A remote access virtual private network encrypts all traffic that users send and receive, allowing users who are working remotely to securely access and use applications and data that are stored in the corporate data center and headquarters. By building a "virtually private" tunnel between a company's network and a remote user, even when the latter is in a public setting, remote access VPN achieves private communication. The reason for this is that the traffic is encrypted, rendering it incomprehensible to any interceptor. Remote users can safely connect to and use their company's network in much the same way as they would if they were physically present in the office. When using a remote access VPN, data can be sent without the organization having to worry about the communication being hacked or intercepted.

VPNs employ tunneling protocols, which serve as rules for sending the data across the network. It offers comprehensive guidelines on how to package the data and what checks to run once it gets to its destination. The process speed and security are directly impacted by these various methods. These are the most well-liked ones:

- Internet Protocol Security (IPSec): data exchange is protected by IPSec, a VPN tunneling protocol, which requires session authentication and data packet encryption. The encrypted message is contained within a data packet that has also been encrypted twice, making it a two-way encryption process. Due to its high compatibility, the IPSec protocol is frequently used in site-to-site VPN configurations for added security.

- Layer 2 Tunneling Protocol (L2TP): in order for two L2TP connection points to communicate securely, a tunnel must be created. After being established, it uses a different tunneling protocol to encrypt the data sent. High security of the exchanged data is made possible by L2TP's intricate architecture. Another common option for site-to-site configurations, especially when more security is required.

- Point-to-Point Tunneling Protocol (PPTP): a tunnel using a PPTP cipher is created by PPTP, another tunneling protocol. The computing power has, however, grown exponentially since the development of the cipher in the 1990s. It wouldn't take long to break the cipher using brute force in order to see the transmitted data. This makes this cipher rarely used in technology; it is preferable to use a replacement that uses more secure tunneling protocols and sophisticated encryption.

- SSL and TLS: the same standard that encrypts HTTPS web pages is known as Transport Layer Security or Secure Socket Layer. In this way, the user's access is restricted to particular applications rather than the entire network, and the web browser serves as the client. No additional software is typically needed because nearly all browsers support SSL and TLS connections. SSL/TLS are typically used by remote access VPNs.

- OpenVPN: adding more cryptographic algorithms to the SSL/TLS framework through OpenVPN will increase the security of your encrypted tunnel. Due to its high security and effectiveness, it is one of the preferred tunneling protocol. As you won't be able to install it natively on many devices to create router to router VPN networks, compatibility and setup can be hit or miss. As a result, the performance may change. Versions of it are available in User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) formats. TCP is slower but better at preserving data integrity, while UDP is faster because it uses fewer data checks. In general, OpenVPN is a well-rounded and secure tunneling protocol that is well-liked for both remote access and site-to-site virtual private network uses.

- Wireguard: the most recent widely used tunneling protocol is less complicated than IPSec and OpenVPN, but it is also a lot more effective and secure. In order to achieve the best performance with the smallest amount of error, it depends on extremely streamlined code. Although it is still in the early stages of adoption, many technologies are using Wireguard-based site-to-site connections.

The type of application that will be integrated will largely determine which of these options is the best fit. Due to its lightweight qualities and the fact that it is also incorporated into other cutting-edge solutions that operate in the ROS environment, such as FogRos2, Wireguard was chosen in this instance.

4.6.1 Examining in Depth

WireGuard is a very straightforward VPN. In contrast to IPsec, it aims to be quicker, easier, more efficient, and more useful while avoiding the excruciatingly tedious setup process. It aims to perform significantly better than OpenVPN. In order to be suitable for a variety of situations, WireGuard is made to be a general-purpose VPN that can be used with both supercomputers and embedded interfaces. It was originally made available for the Linux kernel, but it is now cross-platform (Windows, macOS, BSD, iOS, and Android) and widely deployable. Although it is still undergoing extensive development, it may already be regarded as the most secure, user-friendly, and straightforward VPN solution available.

As with SSH, WireGuard aims to be simple to deploy and configure. A VPN connection is established by simply exchanging very basic public keys, just like exchanging SSH keys, and the rest is transparently handled automatically. There is no need to manage daemons, state, connections, or anything else that goes on behind the scenes.

It is worth noticing that IPsec is the industry-standard solution in Linux for encrypted tunnels. However, this solution has a disproportionately high level of complexity and code. Security labeling and firewalling semantics for IPsec packets are entirely different for administrators. While separating the transformation layer from the interface layer is correct from a networking standpoint, and separating the key exchange layer from the transport encryption layer is wise from a semantic standpoint, this strictly intricate layering approach increases complexity and makes correct implementation and deployment prohibitive.

WireGuard merely provides a virtual interface (wg0, for instance), which can then be managed using the common *ip* and *ifconfig* utilities. The interface must be set up with a private key, possibly a pre-shared symmetric key, and all of the public keys of the peers it will securely communicate with. After that, the tunnel is ready to use. The administrator does not need to be concerned with the specifics of key exchanges, connections, disconnections, reconnections, discovery, and so on, because they take place transparently and reliably behind the scenes. In other words, the WireGuard interface seems stateless from the perspective of administration.

These principles are implemented by this protocol, which also emphasizes simplicity and an auditable codebase. Despite this, WireGuard is incredibly fast and suitable for a variety of environments. In order to achieve a valid engineering solution that is both more practical and more secure, WireGuard does indeed violate traditional layering principles by combining the key exchange and the layer 3 transport encryption into one mechanism and using a virtual interface rather than a transform layer.

At this point, let's focus on how to integrate the VPN in the message exchanged between the SEDIA nodes and a kubernetes cluster.

Figure 4.19 depict a general example on how to use Wireguard. Kubernetes exposed a nodeport that is protected by a VPN, allowing remote peers to connect to running Kubernetes pods and exchange information securely.

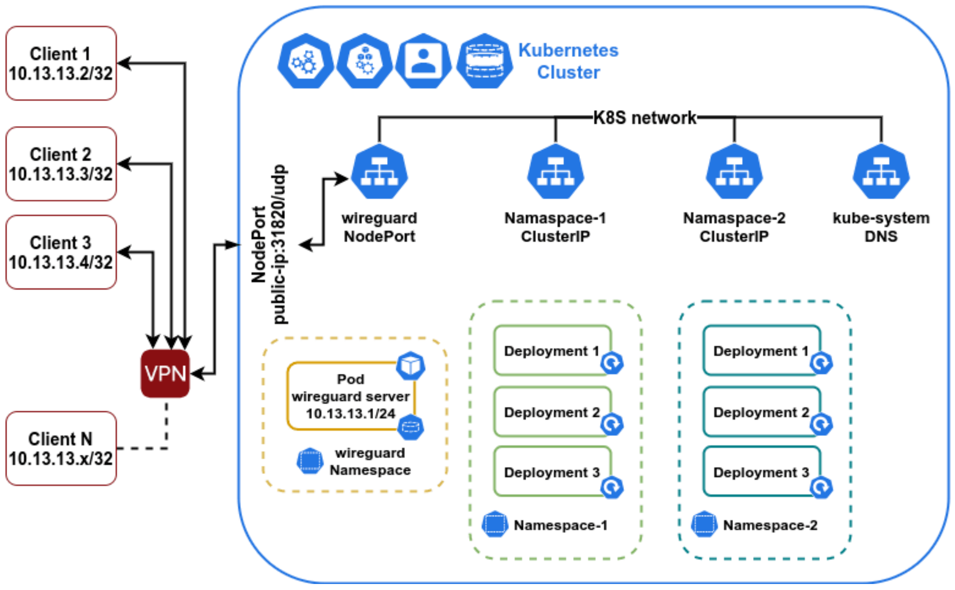


Figure 4.19: Wireguard in Kubernetes, securing pods exposed via NodePort

The architecture feasibility, however, is strictly bounded to the middleware in use. In fact, the domain id, a non-negative integer, is how DCPS (data-centric publish-subscribe communications) uniquely identifies a domain. This domain id is mapped to UDP/IP port numbers in the UDP/IP mapping, which ensures that accidental cross-domain communication is impossible with the default mapping. These port numbers are particularly important for the discovery protocol.

In a domain, every DCPS domain participant is also mirrored as a DDSI participant. Through the discovery protocols, these DDSI participants drive the identification of other participants, readers, and writers. By default, every DDSI participant has a distinct network address in the form of a separate UDP/IP socket with a distinct port number. The DDSI specification refers to the Eclipse Cyclone DDS implementation as being stateful. Only discovered readers and writers exchange data. Eclipse Cyclone DDS internally creates a proxy participant, reader, or writer for every remote participant, reader, or writer. Based on the topic and type names, as well as the QoS settings, writers and readers are matched during the discovery process.

In the DDSI discovery protocol, Cyclone only advertises the address corresponding to one network interface, the preferred interface, for the purpose of transmitting multicast packets. It is possible to override the default interface by setting the General/Interface option, adding the name of the interface in use, in order to choose the wg0 interface created during the Wireguard setup process. A remote participant who publishes multiple addresses in an SPDP message will only choose one address to use when contacting that participant. The first eligible address that is either on the same network as the locally selected interface or that is on a network that corresponds to any of the other local interfaces is chosen. By specifying peers in the Discovery/Peers section, any number of unicast addresses can be configured as addresses to be contacted in addition to (or instead of) the multicast-based discovery. All of these addresses receive SPDP messages every time they are sent.

Using the Discovery Server (see Section 4.4) is one option when using FastRTPS. Although it functions as intended, there is a drawback: Adding new ROS2 nodes to the network necessitates editing the XML DDS configuration file (configured with the environment variable FASTRTPS_DEFAULT_PROFILES_FILE), by adding a new record in the known peer section. Therefore, the primary challenge is to change this XML configuration file on each robot. There is a need for a solution that dynamically adds new robots to the VPN network and makes them automatically discoverable by existing ones: the SIMPLE DDS discovery process automatically detects ROS2 nodes if they are running on devices connected to the same local area network (LAN), as was stated above. This is the straightforward solution. It is obvious that a virtual LAN works the same way, so all nodes connected to the virtual private network built using a VPN will automatically find one another. The middleware settings must be changed in this instance as well, similar to the last, though.

Due to the developer side's complete lack of effort, this solution undoubtedly appear attractive. In fact, unlike the previous one, a configuration of the DDS middleware is required, even though the architecture remains the same. An additional layer, made up of the VPN establishment, is also required, which has nothing to do with the application's actual nature. As a result, this solution has the benefit of being straightforward, presuming that we have a wireguard-like effortless method for establishing a VPN connection between the server and SEDIA, regardless of the applications that are running on them. Although there are many open-source solutions for setting up a VPN, the main issue is actually not technical: allowing this simple multicasting is not regarded as a workable solution to our project.

Multicasting is actually workable for relatively small fleets in simple networks where every device is linked to the same WiFi router. One multicast service discovery message is sent when a new device is added to an existing network, and this message is then forwarded to all ROS 2 nodes in the LAN network by a router. These nodes respond by sending the newly added device their own service discovery messages. However, if we needed to make multicast work over WAN instead of LAN using a peer-to-peer VPN network, each packet would need to be multiplied in the source by the proportionate amount of the VPN network's elements.

This enormous volume of messages will eventually become unmanageable and completely saturate the links connecting the local and cloud sides, severely impairing message exchange performance and ultimately the operation of SEDIA.

Chapter 5

Dynamic Switching

In cloud networked multi-robot systems, cloud offloading is essential for utilizing remote infrastructure's support for computation and reaping the rewards of the highly evolved cloud network infrastructure. Making the best choices for offloading, however, is not simple given the delay constraint, additional costs of data transmission, and remote computation. Due to their network connectivity and on-demand computational needs, which have a significant impact on the communication links between robots and clouds, task offloading for robots is particularly more complicated.

Additionally, for this kind of system, a suitable workload distribution between local computation (robot) and the cloud is needed in order to maximize resource efficiency. For modeling systems that can handle these higher level of complications, it is imperative to establish more thorough offloading schemes. The offloading mechanisms are viewed as switches, and when the network connection to the cloud side is good enough, the ROS nodes elected as switchable are moved remotely.

5.1 Distributed Navigation System

We want to put into practice a distributed navigation system with a switching mechanism that can behave in the same way both locally and in the cloud. The system would use cloud when the network can sustain it and local when it cannot. If the network is completely lost, navigation cannot be carried out in the cloud because the nodes running inside the cloud server will not be reachable at all. Another possibility is that the network connection is active but isn't robust enough to support autonomous navigation. These situations unquestionably necessitate a switch to the local computation.

The ability to effectively avoid obstacles is one way to gauge how effective a navigation system is. Obstacle avoidance is, in fact, the most safety-critical process included in an autonomous driving application, and it necessitates low latencies to be carried out smoothly and safely. Other processes include a priori path calculation, obstacle detection, and many others. It takes a lot of computational power, but more importantly, very fast data forwarding, which means that the obstacle' data, acquired in the detection phase, must be processed very quickly and the system must react to the their presence as quickly as possible. This is necessary to identify objects or people in the path and force the vehicle to avoid them.

As discussed in Chapter 2.5, Nav2 is the (default) navigation stack in ROS2, and the one used by SEDIA. It needs a low data transmission latency between local and cloud nodes to ensure a good obstacle avoidance process. So, we need to comprehend how the Nav2 obstacle avoidance system operates in order to define a factor allows the switching mechanism to trigger.

The Controller Server is the first element we must take into account in order to examine Nav2's method of avoiding obstacles in more detail. It is a critical component, which is responsible for producing velocity commands, controlling the robot in following a designated path and ensuring it reaches its destination safely and efficiently. The controller transforms velocity commands that can be sent to the robot's motion controller from the path plan created by the Nav2 planner. To generate the velocity commands, this controller can use a variety of algorithms, including PID controllers, geometric controllers, and model predictive controllers. Depending on the kind of robot and the task requirements, a particular algorithm is employed. Additionally, it adjusts the velocity commands in accordance with the robot's position and velocity, which are current state variables. In order to avoid collisions and stay on course, it can also handle error circumstances like obstacles or malfunctioning sensors and adjust the velocity commands.

The controller operates in an infinite loop, like the majority of the Nav2 stack's components. It is independent of the sensors and cameras' detections because there are other nodes that perform algorithms and that are interposed between the detection of environment and the knowledge of the obstacles by the controller itself; instead, it acts autonomously, periodically producing its output and basing its computation solely on the data kept in the Local Costmap. In fact, it simply reads the data that is already present in such a map, coming from earlier surrounding detections, in order to compute the local path and generate the velocity commands as mentioned. This is done at every clock cycle: if the controller is done with its work, it sleeps until the timer expires, instead it executes its algorithm when the timer goes off, using only the information present in the Local Costmap at that precise moment. The frequency at which this timer runs can be set by the developer; at the time of writing, the Controller Server of SEDIA runs at a rate of 25Hz, which means that its working algorithm is executed every 40ms. The robot's immediate surroundings are more precisely depicted by the Local Costmap. The controller server uses it to generate velocity commands to avoid obstacles and stay on course.

It is updated continuously using sensor data. Information on obstructions, open space, and other features in the robot's immediate surroundings are provided by the local costmap. It is based on a grid representation of the environment, where each cell represents a particular area and is given a cost depending on whether or not there are obstacles, terrain features, or other factors present. The grid cells of the costmap can be updated based on sensor data by configuring the costmap to use various algorithms and parameters.

Therefore, it is obvious that moving these nodes to the cloud side can result in a noticeable performance improvement. In fact, because of the increased computing power, these components actually provide the output data in a much shorter amount of time. In order to provide more accurate and up-to-date data, the server controller can read data from the costmap more frequently, reducing the waiting time. With that in mind, the purpose of this section is to create a novel task offloading decision-making scheme that takes the following factors into account all at once: (i) selection of task for offloading, (ii) continuous monitoring of network connectivity, (iii) manage the correctness of the application behavior when offload tasks.

5.1.1 Node Manager

The Lifecycle nodes concept from ROS2 is the foundation of the cloud/local switching solution. As further detailed in Chapter 2.3, for each node, which behaves in a manner that is very reminiscent of a state machine, the ROS2 project defines a precise lifecycle. The only node with a self-managed lifecycle and in charge of managing every other managed node present in the same DDS network is the node_manager (see Figure 5.1), who controls the state transitions of every node. Thanks to this manager, every node in a ROS2 navigation system, such as the one used by SEDIA, goes through a series of lifecycle transitions to move from the Unconfigured state to the Active state, where it is prepared to carry out its tasks.

In order for each local node to have its shadow copy in cloud side, we need to generate one replica for every node (but only those that can run in the cloud, maintaining the SEDIA's proper operation). The existence of another indentical replica of itself has no effect on the two nodes, which are totally agnostic. This is in line with the principle we want to establish, which states that the proposed switching solution shouldn't significantly alter the architecture as it is and that only some additional logic should be used to orchestrate the switching. Furthermore, regardless of how the two side are linked together, all nodes, both local and cloud instances, must be connected to the same DDS network in order for each node in the system to find every other node.

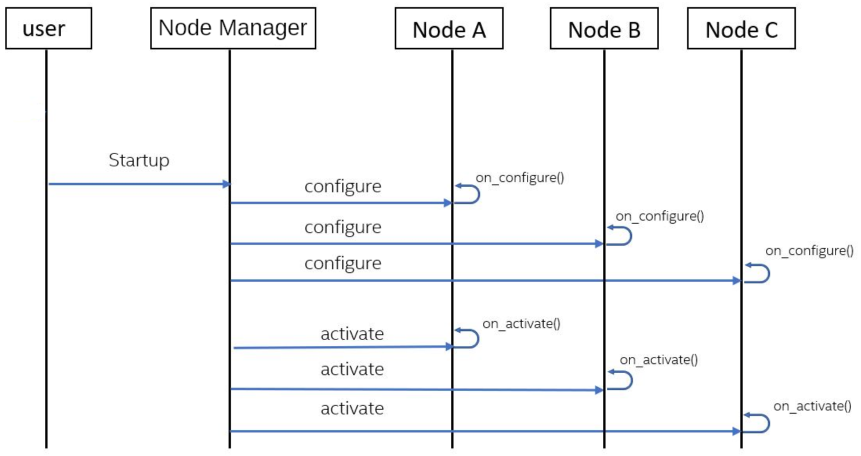


Figure 5.1: Node Manager process of activating nodes

As a result, it is very simple to control which of the two local nodes must be performing its duties at a given time when each local node is taken into account along with its corresponding cloud replica. Only one of the two nodes must be Active, and the other must be Inactive. In other words, we use the node_manager's power to control node lifecycles to activate all cloud nodes when a certain condition is met while deactivating local nodes in the interim. When that condition is not yet met, the switching is carried out the other way around, by simply activating the local nodes while deactivating the cloud nodes. The main or the shadow copy of that specific node must always be in the Active state, carrying out its duties and publishing data in its topics, while the other node must be in the Inactive state, lying dormant and not taking part of the navigation.

This is the fundamental idea that will enable us to construct an appropriate response to our switching issue. Of course, it is not sufficient in and of itself, as further thought must be given to its implementation.

As we already mentioned, the switching mechanism is based on the requirement that only one node of each couple, which is made up of a local node and its cloud replica, must be active at a given time. This node is known as the node_manager and is responsible for managing the lifecycle of all the others. In order to support the activation and deactivation of the set of nodes belonging to each respective side, the node_manager has been modified. In fact, the local node_manager and the cloud node_manager will initiate the state transitions of the local and cloud nodes, respectively, of the navigation architecture. Actually, they are unaware of the reasoning behind why they need to activate a switch; instead, they simply receive a request to do so through some ROS2 services run by the Network Controller (which will be discussed in more detail in the following section) and carry it out.

The changes to the local node_manager consist of both new logical behaviors and additions to its data structures and services:

- The node_manager must first be informed that some of the nodes it is responsible for managing may have a cloud replica. Only the local nodes' lifecycle will be managed by the local node_manager; however, depending on whether or not those local nodes are the only ones present in the navigation system, the behavior of the node_manager will vary. Because of this, a new parameter that determines whether a particular node has a cloud replica or not has been added to the node_manager's configuration file. If this variable is set to true, the node_manager will treat that node differently from the others even though there is no guarantee that a cloud version of the node has been defined (the developer must be careful to actually define it). When a node has the same parameter set to false, the node_manager continues to behave normally because the local copy of the node_manager is the only one that can function in that situation. An example of this would be a node that is unable to be moved to the cloud because it is strictly related to the SEDIA hardware layer.

- Additionally, there have been added two services: activate_local_nodes and deactivate_local_nodes. The functions linked to these services are called when one of the two services receives a request. In deed, the network controller can ask the node_manager to trigger a switch using these services.

- The node_manager, which is called by the State Holder, now has a third service request_transform. As we will see in Section 5.1.3, this service enables crucial data to be injected both locally and in the cloud when the nodes are reactivated. This knowledge permits the nodes' states to converge more quickly, favoring the switching mechanism's responsiveness.

These are the adjustments made to the node_manager that were required to support the use of the switching mechanism presented in this thesis. Because they are merely additions built on top of the node_manager's original operating principles and compatible with its pre-existing base behaviors, it should be emphasized that these modifications don't affect system functioning. To put it another way, while some modifications are required to support a local/cloud switching system, it was not considered that they would significantly alter the architecture from how it had previously been designed. Additionally, only the node_manager's hardcode has been slightly altered, leaving the architecture's other nodes unaffected.

5.1.2 Offloading Decision Controller

The main logic that determines whether or not to initiate switching is represented by this controller. It must be known that the switching procedure entails choosing not to use cloud nodes and, in the event that the network forbids it, to fall back on local ones. The system goes back to the cloud nodes once the network condition are good enough. To accomplish this idea, a ROS2 node is required, which can constantly assess the network's state and determine whether it is appropriate for the system as a whole to perform a switching process. Since this controller is responsible for this function, it monitors the network and instructs the node_manager to act accordingly. Physical distances and the best network routing protocols limit the amount of network delays and variability. The boundary cannot be crossed due to physical restrictions; they can only be minimized. As a result, we are only able to mitigate network delays and variations; they will never completely disappear. Furthermore, when cloud computing environments are used to support these type of applications, network delays unpredictability becomes critical. Building a trustworthy real-time controller that can handle such unpredictable network delays is made more difficult as a result.

As previously stated, the controller must decide whether or not to offload the computations necessary for identifying and tracking a moving object. When making a choice, it takes into account the estimated network link conditions. Network monitoring is carried out by injecting test traffic into a network and monitoring its path to a destination.

Such adaptive switching protocols must be based on a method that can both quickly adjust to the temporal dynamics of the network, while also accurately estimating the quality of wireless links in terms of a quantitative measure.

As well as for ascertain whether the remote host is reachable, it is useful to monitoring some performance indicators, such as:

- Packet loss: occurs when data packets fail to reach their intended destination. It can be caused by network congestion, equipment failure, or other factors. It is usually measured as a percentage of packets lost over the total number of packets sent. High packet loss can affect the performance of the real-time application and can cause data corruption or loss, which can lead to data retransmission, reducing the efficiency of the network.
- Latency: is the time delay between the transmission and receipt of data. It is usually measured in milliseconds (ms) and can be affected by network congestion, distance, and the quality of network equipment. High latency can cause delays in data transfer, which can affect the flow of the application.
- Jitter: is the variation in time delay between when a signal is transmitted and when it's received over a network connection, measuring the variability in

ping. This is often caused by network congestion, poor hardware performance and not implementing packet prioritization. Good connections have a reliable and consistent response time, which is represented as a lower jitter score. The higher the jitter score, the more inconsistent response times are.

- Bandwidth: refers to the amount of data that can be transmitted over a network in a given period of time, usually measured in bits per second (bps). The available bandwidth can affect the network speed and overall performance. If the bandwidth is low, the network may be slow, causing delays in data transfer, which can affect the user experience. Bandwidth can be increased by upgrading network hardware or by optimizing the use of the available bandwidth.

- SNR: the performance of a wireless communication link can be finely measured by taking another parameter into consideration, the Signal-to-noise ratio (SNR). However, due to the higher variance of SNR on the non-line-of-sight channel, it is known that neither instantaneous signal-to-noise ratio (SNR) nor mean values are sufficient to fully characterize the quality of the channel.

It was decided to use a method based on the Kalman filter in order to prevent the switching mechanism from being accidentally activated due to the potential measurement errors of the parameters just mentioned. It is an algorithm used to infer the state of a system from inconsistent and noisy measurements over time. With each iteration, the estimate is improved upon by using a mathematical model to predict the state and new measurements to update the estimate. Even with noisy or insufficient measurements, the filter is still able to determine the system's state with accuracy. This is accomplished by taking into account the measurement and state prediction uncertainties.

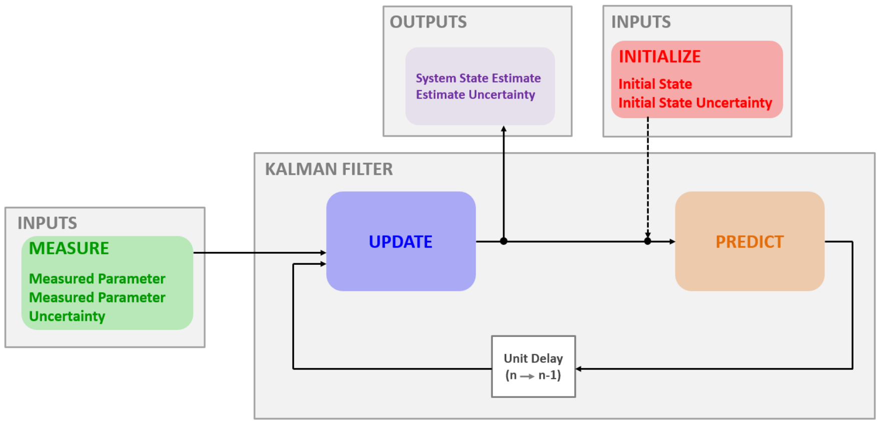


Figure 5.2: Kalman filter algorithm

The Kalman filter's fundamental principle is to continuously update the estimated state of the system based on new measurements. It does this by using a prediction-correction process. The algorithm has two stages: prediction and correction.

1. Prediction phase: In this phase, filter uses the mathematical model of the system to predict the current state of the system based on the previous state and a control input, if applicable. The prediction includes an estimate of the uncertainty or noise associated with the state prediction.
2. Correction phase: In this phase, the filter updates the state estimate based on the new measurements. The filter computes the difference between the predicted and the measured state, and then updates the state estimate based on the magnitude of the difference and the estimated uncertainty in the prediction.

Monitoring takes a proactive approach to network troubleshooting by highlighting potential problems before they impact the functioning of the autonomous system, invoking consequently the switching mechanism. However, there are other issues to take into account when deciding whether to offload the computation in addition to network uncertainty. In fact, it's important to consider whether or not it's practical at the application level, defining a cutoff point above which to activate the switching. As was stated earlier, the distinguishing characteristic in the case of SEDIA is to guarantee accurate obstacle detection. The times to consider are therefore those that are related to the Controller Server and, of course, network latency. The total response time is affected by two factors when the controller decides to perform the tasks on the robot itself: $\mathrm{T_{clkLocal}}$ is the interval between two iterations of the local Controller Server, and $\mathrm{T_s}$ is the interval between two scans of the Lidar sensor. In contrast, if the controller chooses to offload the computation operations to the cloud side, the total execution time or response time of the offloaded computation operations is specified as follows: $\mathrm{T_{clkCloud}}$ is the interval between two iterations of the cloud Controller Server, $\mathrm{T_{nl}}$ is the network latency (which is the sum of all the previously mentioned network parameters), and Ts is the interval between two scans of a single sensor. The requirement is that the time needed to avoid an obstacle in the cloud be less than or equal to the time needed to do so locally. From this definition, it is possible to extrapolate the following formula, which serves as our logical switch triggering condition, by taking into account the time intervals discussed in order to accomplish this.

$$\mathrm{T_s + T_{nl} + T_{clkCloud} < T_s + T_{clkLocal}}$$

Since $\mathrm{T_s}$ is strictly related to the hardware that the physical sensors use, it is obvious that it cannot be modified and can therefore be excluded from the inequality. T_{nl} is entirely dependent on network conditions and is not within our control. Therefore, the only variable we can influence is $T_{clkCloud}$. By controlling computational resources and, as we've already mentioned, increasing the frequency at which the algorithms run, we can offer the SEDIA application intelligent dynamic switching.

We can, for instance, compare the computing time for the worst-case scenario for the most significant nodes in the object detection chain, both when these nodes are deployed locally and when they are deployed in the cloud. Each of these nodes operates according to its own clock reference; they complete their tasks during each clock cycle. While the Lidar clock cycle is 10Hz, the controller and the local costmap run at 25Hz in the local side (as per the SEDIA specification). Thanks to the higher computational power, the frequency can be increased up to 50 Hz on the cloud side for both the controller and the costmap. As we previously stated, we are unable to increase the frequency of the Lidar sensors because they are strictly hardware-constrained.

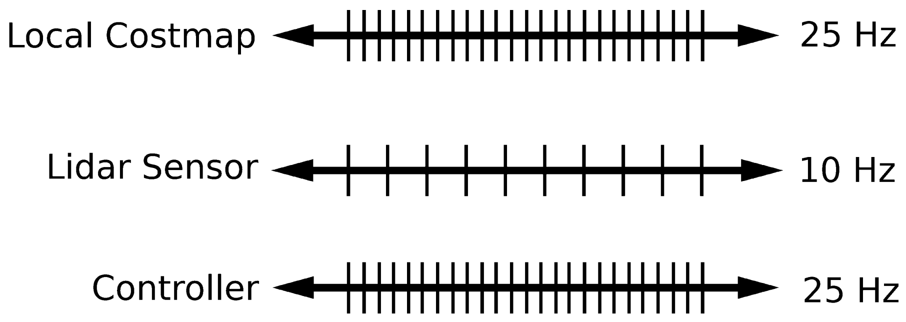


Figure 5.3: Comparison between Local Costmap, Lidar and Controller clock cycle

The worst scenario is when a obstacle appears in front of the sensors just as the sensor clock is about to expire. As a result, the Lidars must wait until after the subsequent cycle before it can see the obstruction. We can state that as soon as the obstacle is seen, i.e., ignoring the delay in data transmission between all of the nodes, the data are transmitted to the Local Costmap after approximately 100 milliseconds. Given that also in this case the costmap clock just elapsed, there will be an additional 40ms delay because it must wait until the next clock cycle to update its internal state. After that, the controller, whose clock has just been cycled, must wait an additional 40 ms. Therefore, after $100ms + 40ms + 40ms = 180ms$ the object detection chain is complete.

On the other hand, we can examine the same route using nodes deployed in the cloud, where their CPU power is significantly greater. The latency of the Lidar, for obvious reasons, doesn't change, and the second and third each (respectively local costmap and controller), get cut in half. $100ms + 20ms + 20ms = 140ms$ is the total amount of time needed for cloud object detection to complete, which is a $40ms$ improvement over the local side case. Then if the network RTT between local and cloud side is less than $180ms - 140ms = 40ms$, we can take advantage of cloud offload. The discrepancy between the local and cloud systems is therefore caused by different performance and latency, both of which are higher in the cloud than locally, as is evident from the results. It is important to note that this calculation does not account for all nodes that are actually present or the delays that each one adds. Consequently, the example should not be seen as a reference to the switching time, but only as an explanation of the given problem.

5.1.3 State Holder

When a local side node is duplicated and its cloud side replica is active, the local side nodes are not updated in real time with the state changes that the system is publishing and instead are in a standby state. Bringing a node from the Inactive to Active state, if it is necessary to switch the nodes for the reasons previously mentioned, without giving it a context to align itself with the rest of the system could cause the system itself to malfunction.

It is therefore important to maintain the state of the node when reactivating a copy node, both in the case of passing from local to cloud and vice versa. Maintaining node state in this type of architecture is crucial because it allows the nodes to keep track of its own data and to ensure that it is consistent with the rest of the system. Node state refers to the current state of itself at a specific point in time. This state can include things like configuration settings, map caches, and localization information.

If a node loses its state, it can have significant consequences for the overall system. For example, if a node loses its configuration settings, it may not be able to connect to other nodes or perform its intended function. Similarly, if a node loses its localization information related to the SEDIA, it may not be able to maintain interlaced context or track other nodes behavior. However, not all the nodes in the SEDIA system have an internal state, and only a subset of these are crucial in maintaining the state of the wheelchair.

It was decided to develop a ROS2 node that retains the absolute minimum amount of data necessary to safely reactivate every node while ensuring that switching occurs without any issues. Maintaining state, in ROS jargon, means becoming a subscriber of messages that carry this essential information. The state_holder does just that: it receives this information periodically and maintains it. To keep this data it was decided to use WitheDB. It is an open-source NoSQL database that was created to provide a fast and efficient in-memory data storage and retrieval system. Unlike traditional disk-based databases, Whitedb stores data directly in memory, allowing for extremely fast read and write performance.

Whitedb uses a memory-mapped file to store data, which is one of its distinguishing characteristics. This file is directly mapped into the process address space, enabling the operating system to manage memory access. Due to the lack of costly disk I/O operations, data access in Whitedb is extremely quick. It supports a variety of data types, including integers, floating-point numbers, strings, and binary data. Data is stored in tables, which can be created and modified dynamically. It also includes support for indexes, which can be used to optimize queries and improve performance. Its support for atomic transactions is another key feature. To ensure that all changes are either committed together or rolled back together in the event of an error, operations on the database can be grouped together into a single transaction. Data consistency and dependability are enhanced as a result.

All things considered, Whitedb is a strong and adaptable database system that offers quick and effective data storage and retrieval capabilities; in fact it is made to be very effective in terms of performance, achieving very high throughput rates even handling large datasets and. Indeed, it is the ideal fit for this node's objective thanks to its support for atomic transactions, indexes, and high performance.

5.2 Physical Constraints

The foundation of this solution is to guarantee responsive behavior during dynamic switching to and from the cloud. It is necessary to clarify what is meant by "cloud side" for this reason. Although it may be close to the facility where the SEDIA vehicle is active, moving the nodes to a remote public cloud datacenter is not practical because the latencies introduced would be too high for the kind of real-time application that we want to manage.

Real-time robotic applications can benefit greatly from edge computing. To make decisions and take actions in real-time, robots frequently need low-latency and high-bandwidth communication with the computing infrastructure. Instead of being housed in a centralized data center, computing resources are located at the network edge. By moving computing resources closer to the robots, edge computing can offer such low-latency communication. It decreases latency and boosts data processing efficiency by bringing computation and data storage closer to the point of need.

In order to reduce the amount of data that needs to be transmitted to a centralized cloud or data center, edge computing's primary objective is to process data nearer to the source. This can be especially helpful for applications including lidar sensors, that demand high bandwidth or low latency. Edge computing can also increase data security and privacy by processing data locally rather than sending sensitive information over a network to a distant data center.

5.2.1 MEC - 5G UPF

A few interconnected technologies that can be used to boost the performance, dependability, and security of mobile computing include Mobile Edge Computing (MEC) and 5G networks.

Multi-access Edge Computing (or Mobile Edge Computing) is a technology that makes it possible to deploy computation, storage, and network resources closer to where data is generated and consumed, at the network's edge. Instead of using distant cloud servers, MEC enables applications and services to run on local edge servers. It enables low-latency data processing, lessens network load, and boosts security. MEC architecture involves the deployment of computing resources such as micro data centers or edge servers at the edge of the network. Different mobile network architectures, including 5G, and Wi-Fi 6, can access these resources, which may be owned by the service provider or a third party. In order to build and deploy applications at the network's edge, it offers a standardized environment for application developers. The following are just a few advantages of MEC:

- Low Latency: it can significantly reduce the communication latency between the robots and the computing infrastructure. The robots can send sensor data to the edge node for processing and receive the results in real-time.
- Improved Reliability: it can improve the reliability of robotic applications by providing redundant computing resources and backup connectivity options. In case of any network failure, the robots can switch to a nearby edge node and continue their operations.
- Efficient Resource Utilization: it can optimize the usage of computing resources by offloading some of the processing tasks to the edge nodes. This can reduce the burden on the central cloud infrastructure and improve the scalability of robotic applications.
- Edge Analytics: it can provide real-time analytics capabilities on the edge, allowing the robots to make decisions based on the analyzed data quickly. This can significantly improve the efficiency and effectiveness of robotic applications.
- Improved Security: enables data processing to be performed on the device or at the edge of the network. This reduces the need to transmit sensitive data to remote cloud servers, which improves data privacy and security.
- Scalability: enables the deployment of distributed computing resources at the edge of the network. This allows for the scaling of resources based on demand.
- Cost-Effective: reduces the amount of data transmitted over the network and enables the deployment of distributed resources, which reduces the cost of data processing.

MEC is an edge computing use case for service providers. As service providers move away from physical appliances and towards a service-based architecture the result is a decoupling that allows for a broader ecosystem where mobility workloads can run. This makes it possible for real-time applications to benefit from lower latency, especially in light of recent developments in 5G.

End devices must connect through radio access networks (RAN); authorized application developers can access RAN through MEC implementations, enabling them to use edge computing at both the application level and at the lower level of network functions and information processing.

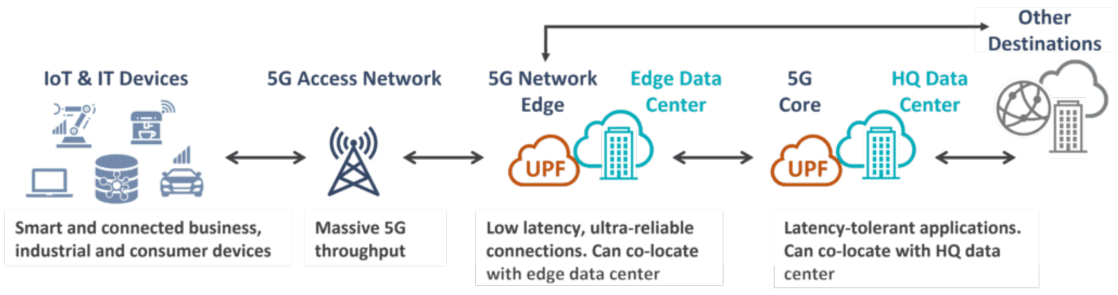

Other Destinations
IoT & IT Devices
5G Access Network
5G Network Edge
Edge Data Center
5G Core
HQ Data Center
UPF
UPF
Smart and connected business, industrial and consumer devices
Massive 5G throughput
Low latency, ultra-reliable connections. Can co-locate with edge data center
Latency-tolerant applications. Can co-locate with HQ data center

Figure 5.4: Abstract view of integrated 5G and edge-computing systems

Due to its ability to provide low latency and high throughput, the User Plane Function, or UPF, is a crucial part of the 5G network (see Figure 5.4). To enable packet processing and traffic aggregation to be carried out in the core or closer to the network's edge, it decouples control and user plane functions.

Mobile autonomous vehicles like SEDIA can offload computationally intensive tasks to nearby edge servers, which can process the data quickly and efficiently, by deploying computing resources at the edge of the network and combining them with the high-speed, low-latency connectivity offered by 5G networks. By doing this, the vehicles use less energy, perform better, and can offer new services and applications that demand more processing power than the onboard computing module is capable of providing.

Chapter 6

Secure Middleware Across the Network

When the Robot Operating System was first developed, its primary focus was on providing a flexible and efficient framework for prototyping and developing robotic applications. At that time, security was not a major concern, and the early versions of ROS did not include many security features. However, when combining ROS robots with edge computing, sending messages through the network, the need for security became increasingly important.

With the growing use of robots to the real-world industrial applications, the potential consequences of security breaches in robotic systems could be severe. As a result, the ROS community has been actively working to improve the security of ROS over the years. This has included the development of security guidelines and best practices, as well as the implementation of security features including authentication, encryption, access control, secure communication channels, and vulnerability management.

The Data Distribution Service communication protocol, which enables quick communication between numerous nodes, is essential to the security of the Robot Operating System 2. Historically, DDS lacked any authentication, encryption, or access control safeguards, leaving it open to security threats like data type manipulation, unregistered Publishers and Subscribers, and impersonation attacks.

Each DDS node has a minimum of one Publisher or Subscriber who sends and receives messages using specific Topics. While this enables real-time communication, security is sacrificed because Publishers' messages can be read by Subscriber nodes if they make the same Topic reference, leaving vulnerabilities that could be exploited.

Security is of utmost importance because any interruption to these robot networks could compromise the safety of people as well as the environment.

6.1 Security Countermeasures

The DDS-Security specification adds security enhancements to the DDS specification by defining a Service Plugin Interface (SPI) architecture, a set of builtin implementations of the SPIs, and the security model enforced by the SPIs. There are five SPIs defined: Authentication, Access control, Cryptographic, Logging, and Data tagging. ROS 2's security features currently utilize only the first three SPIs. Authentication is central to the entire SPI architecture, Access control deals with defining and enforcing restrictions on DDS-related capabilities, and Cryptographic handles encryption, decryption, signing, hashing, etc.

ROS 2 uses the builtin authentication, access control, and cryptographic plugins, which use Public Key Infrastructure (PKI) and AES-GCM. The rationale for using the builtin plugins is that they are the only approach described in detail by the spec and mandatory for all compliant DDS implementations to interoperably support, which makes the ROS 2 security features work across vendors with minimal effort.

When a DDS participant wants to establish a secure connection with another participant, it initiates the handshake process. The security standard for DDS implements a three-way handshake consisting of HandshakeRequest, HandshakeReply and HandshakeFinal messages. The Sending Participant sends an HandshakeRequest message, containing its certificate information, a DH public key, and a random nonce. The receiving participant responds with a HandshakeReply message, indicating its readiness to establish a secure connection. It contains Recieving Participant's certificate information, a DH public key, Sending Participant's random nonce, as well as another random nonce. The initiating participant then sends a HandshakeFinal message, confirming the establishment of the secure connection. It contains both nonces and is signed by Sending Participant. During this process, both participants negotiate the security parameters of the connection, including the encryption algorithms, key lengths, and authentication mechanisms to be used. Once the secure connection has been established, data can be transmitted between the participants with confidentiality and integrity.

Depending on the specific requirements of the application, there may be security features or combinations of features that can be implemented to ensure that data is transmitted securely and that only authorized participants can access it.

Additional security considerations could benefit the protection of data integrity for both ROS 2 and the default DDS middleware. Following are presented two different approach to secure the DDS messaging protocol that traverses the network.

6.1.1 SROS2

SROS2 is an extension of the Robot Operating System 2 (ROS 2) framework that provides security features for robotics applications. It was developed to address security concerns related to communication, access control, and data protection in distributed robotic systems. SROS2 is designed to be modular and extensible, which allows users to customize and adapt the security features to their specific needs. It provides a set of configurable security policies that can be used to control the behavior of the security features, as well as a set of tools for managing security-related tasks such as key generation and certificate management.

One of the main security concerns in robotics is the secure communication between different nodes in a distributed system. SROS2 addresses this issue by providing message encryption, which ensures that the data exchanged between nodes is protected from unauthorized access.

In addition to message encryption, SROS2 also provides authentication and authorization mechanisms to ensure that only authorized nodes can access the data. Authentication is achieved through the use of digital certificates, which are issued by trusted certificate authorities (CA) and used to verify the identity of the nodes. Authorization is implemented through access control policies, which specify the permissions that are granted to different nodes based on their roles and responsibilities.

Another important security feature of SROS2 is identity management. This feature allows nodes to securely exchange identity information, such as public keys and certificates, and establish trust relationships with each other. SROS2 uses a centralized identity management system based on the Key Management Interoperability Protocol (KMIP), which allows nodes to securely store and manage their identity information.

SROS system uses Transport Layer Security to secure socket-level communication between nodes, without requiring any modifications to the client libraries. SROS employs a keyserver to generate and distribute Public Key Infrastructure elements, including asymmetric keys, certificate authorities (CA), and signed certificates, to simplify the use and development of SROS-enabled systems. The keyserver can be run separately from ROS and is designed to integrate seamlessly with the rest of the SROS workflow. When bootstrapping an existing ROS application to use SROS, the keyserver generates the required CAs and signs its own nodestore, which is then used to securely connect with nodes and distribute PKI elements. The keyserver within the SROS system enables effortless integration with the other components of the SROS workflow. Moreover, it features prudent default settings that make the process of learning PKI more manageable.

Hence, SROS2 provides a comprehensive set of security features that address the security concerns related to communication, access control, and data protection in distributed robotic systems. Its modular and extensible design allows users to customize and adapt the security features to their specific needs, making it a versatile and flexible solution for securing robotic applications.

6.1.2 DDS Cerberos

One of the primary security issues in DDS middleware is unauthorized access. DDS entities (publishers and subscribers) may be deployed across different security domains, and it is essential to ensure that only authorized entities can access the data being transmitted over DDS. Additionally, DDS middleware is susceptible to message interception and replay attacks, which can compromise the confidentiality and integrity of the data being transmitted.

To address these security issues, the DDS-Cerberus project proposes a solution that integrates Kerberos with DDS middleware. Kerberos is a widely used network authentication protocol that uses tickets to verify the identity of users or services. Kerberos operates on the principle of a trusted third party, where a Kerberos Key Distribution Center (KDC) issues tickets to entities that can be used to authenticate and authorize themselves with other entities in the network.

In the DDS-Cerberus solution, the KDC issues tickets to DDS entities that are registered with the KDC. These tickets contain information about the entity's identity and permissions, as well as a secret key that can be used to authenticate and encrypt messages. When a DDS entity wants to communicate with another entity, it presents its ticket to the other entity as proof of identity and permission to access the data being transmitted.

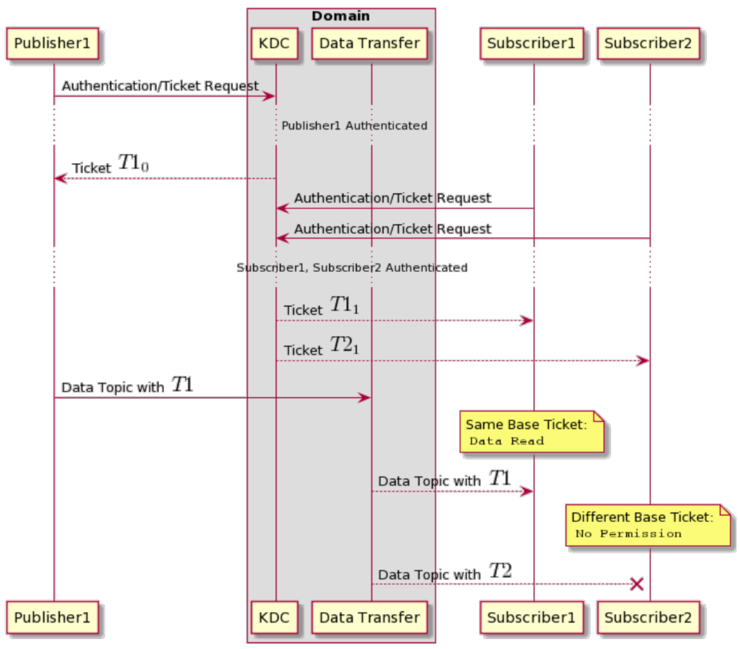

Domain
Publisher1
KDC
Data Transfer
Subscriber1
Subscriber2
Authentication/Ticket Request
Publisher1 Authenticated
Ticket $T1_0$
Authentication/Ticket Request
Authentication/Ticket Request
Subscriber1, Subscriber2 Authenticated
Ticket $T1_1$
Ticket $T2_1$
Data Topic with $T1$
Same Base Ticket: Data Read
Data Topic with $T1$
Different Base Ticket: No Permission
Data Topic with $T2$

Figure 6.1: Diagram of the DDS Cerberos sequence, showing the functionality of topic with tickets

The integration of Kerberos with DDS middleware provides several benefits for security. First, it provides a standardized and secure mechanism for authentication and authorization of DDS entities. Second, it enables secure communication between DDS entities across different security domains, as long as the KDC is trusted by all domains. Third, it provides protection against message interception and replay attacks by encrypting messages using the secret key in the Kerberos ticket.

However, there are also some potential drawbacks to this solution. One issue is the overhead of managing Kerberos tickets, which can add latency and complexity to the system. Another issue is the potential for the KDC to become a single point of failure or attack, which can compromise the security of the entire DDS system. To address these potential issues, the DDS-Cerberus project proposes a number of enhancements to the Kerberos protocol. One enhancement is the use of lightweight tickets that contain only the necessary information to authenticate and authorize DDS entities. Another enhancement is the use of hierarchical KDCs to distribute the load and reduce the risk of a single point of failure or attack.

In addition, the DDS-Cerberus solution proposes the use of secure DDS profiles that specify the security requirements and policies for DDS systems. These profiles can be used to ensure that DDS systems are configured and deployed in a secure manner, and can be customized to meet the specific security needs of different industries and applications.

Overall, DDS-Cerberus is an interesting research project that proposes a solution to improve security in DDS middleware using Kerberos tickets. While there are some potential drawbacks to this solution, it is an important step towards enhancing the security of real-time systems that rely on DDS middleware.

Chapter 7

Conclusions and Future Work

The cloud offloading approach to robotics developed in this thesis involves the use of cloud computing resources to enhance the capabilities of mobile robots. Rather than relying solely on the processing power and storage capacity of the robot itself, it offload some of the computational tasks to a remote server or cluster of servers, typically located in a data center physically close to the facility.

There are several advantages to using cloud offloading. One of the primary advantages is that it allows the mobile robot to conserve its resources, including processing power, memory, and battery life. By offloading computational tasks to the cloud, the robot can perform more advanced tasks without being limited by its own processing power. This also makes it possible to build smaller, lighter, and more energy-efficient robots that can perform complex tasks.

Another advantage of cloud robotics is that it enables remote access to the robot's data and capabilities. This can be particularly useful for monitoring and controlling the robot's actions, as well as for conducting collaborative tasks with other robots or humans. For example, a robot in one location could send data to a remote server, which could then be accessed by another robot or human in a different location.

Cloud offloading also enables algorithm updates to be pushed to the robot from the cloud. This means that as new algorithms are developed, they can be easily implemented on the cloud and pushed to the robot, without requiring any additional processing power on the robot itself. This makes it possible to update the robot's capabilities quickly and easily, without the need for physical modifications or upgrades.

As shown before, there are some challenges associated with cloud robotics. One of the main challenges is latency, or the delay that occurs when data is sent to and from the cloud. This can be a problem for real-time applications, such as autonomous driving, where delays can have a significant impact on performance. We saw that latency can be greatly mitigated with edge computing, which involves performing some processing tasks locally on the robot, while offloading more computationally intensive tasks to the cloud.

Another challenge is security, as sending data to the cloud involves potentially sensitive information. Cloud robotics systems must be designed with robust security measures to protect against data breaches or other types of cyberattacks.

Despite the numerous advantages and the more advanced capabilities enabled by the cloud offloading implementation discussed in this thesis, there could be some improvements to this scenario, such as:

- Robot Fleet Management: involves the coordination, deployment, maintenance, and optimization of a group of robots working together to accomplish a set of tasks. It typically includes a variety of tasks, such as monitoring the robots' performance, ensuring their maintenance and repair, scheduling their tasks, and optimizing their movements and energy consumption. gestire un flotta di robot e non un solo veicolo. Having numerous connected vehicles is a challenge both in terms of managing computational resources in the cloud and in terms of networking, as it is necessary to mitigate a possible single point of failure at the entrance to the cluster.

  With a large number of vehicles generating data, there is a need for efficient processing and analysis of the data received. This requires managing computational resources in the cloud to ensure that there is enough processing power and storage capacity to handle the massive amounts of data generated by connected vehicles.

  Additionally, networking is critical to ensure that data is transmitted securely and efficiently between the vehicles and the cloud. The network architecture should be designed in such a way as to avoid creating a single point of failure, which could bring down the entire system. This can be achieved through the use of redundant links, load balancing, and failover mechanisms.

- Distributed Cloud: it is possible to exploit the computing power of stand-by vehicles on which to offload demanding tasks from other active vehicles.

  However, it implies heavily modifying the very functioning of the ROS nodes. Currently each node collaborates with the others to manage the autonomous driving of a single vehicle. Being able to have a cloud distributed among all the vehicles present implies that the algorithms that run on the nodes of each vehicle are decoupled from the state of the vehicle itself. therefore migrating more towards a microservices architecture, in which each node answers a call and therefore acts independently of the vehicle that requests the service at that moment.

- Network Analyzer: there is room for improvement on the network analyzer algorithms, in particular by calibrating the parameters considered before. Fine-tuning the parameters of the algorithm is an essential step in improving the performance of the system.

  It can be possible to further improve the switching mechanism letting the system know a priori that the wireless connection will drop to critical level in a specific area. In order to do so, the cloud side can aggregate the signal level data sent by the vehicles in real-time, producing a heatmap of the wireless signal. Each vehicle can interact with this map and switch to the local/cloud side accordingly to the state of the network.

  By using these methods, the switching mechanism can be improved, and the vehicle can have a more seamless and uninterrupted wireless connectivity experience.

Bibliography

[1] ROS 2 Docs, *https://docs.ros.org/en/humble/index.html*

[2] Kubernetes Docs, *https://kubernetes.io/docs/home/*

[3] Fast DDS Docs, *https://fast-dds.docs.eprosima.com/en/latest/index.html*

[4] Eclipse Cyclone DDS Docs, *https://cyclonedds.io/docs/cyclonedds/latest/*

[5] Nav2 Docs, *https://navigation.ros.org/*

[6] eProsima Docs, *https://eprosima-dds-router.readthedocs.io/en/latest/*

[7] Zenoh Docs, *https://zenoh.io/*

[8] A Kalman Filter Based Link Quality Estimation Scheme for Wireless Sensor Networks (2007). Murat Senel, Krishna Chintalapudi†, Dhananjay Lal†, Abtin Keshavarzian† and Edward J. Coyle. *IEEE GLOBECOM 2007 - IEEE Global Telecommunications Conference*

[9] The Marathon 2: A Navigation System (2020). Steve Macenski, Francisco Martín, Ruffin White, Jonatan Ginés Claver. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*

[10] Distributed Robotic Systems in the Edge-Cloud Continuum with ROS 2: a Review on Novel Architectures and Technology Readiness (2022). Jiaqiang Zhang, Farhad Keramat, Xianjia Yu, Daniel Montero Hern, Jorge Peña Queralta, Tomi Westerlund. *2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC)*

[11] FogROS 2: An Adaptive and Extensible Platform for Cloud and Fog Robotics Using ROS 2 (2023). Jeffrey Ichnowski, Kaiyuan Chen, Karthik Dharmarajan, Simeon Adebola, Michael Danielczuk, Vıctor Mayoral-Vilches, Hugo Zhan, Derek Xu, Ramtin Ghassemi, John Kubiatowicz, Ion Stoica, Joseph Gonzalez, Ken Goldberg. *2023 Proceedings IEEE International Conference on Robotics and Automation*

[12] DDS-Cerberus: Improving Security in DDS Middleware Using Kerberos Tickets (2022). Andrew T. Park, Richard Dill, Douglas D. Hodson, Wayne C. Henry *Department of Electrical and Computer Engineering Air Force Institute of Technology, Wright-Patterson Air Force Base, OH, USA*

[13] Security and Performance Considerations in ROS 2: A Balancing Act (2018). Jongkil Kim, Jonathon M. Smereka, Calvin Cheung, Surya Nepal.