POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Automatic and Seamless Switching of Cloud Native Services Running in the Cloud Continuum

Supervisor

Prof. Fulvio RISSO

Candidate

Giuseppe GALLUZZO

April 2024

Summary

Cloud Continuum has recently been defined as an extension of traditional Cloud towards multiple entities (e.g., Edge, Fog, IoT) providing processing, storage, and data generation capabilities. Such fluid ecosystem represents a continuum of aggregated resources, distributed from the network edge to the cloud/datacenter, making all the nodes capable of hosting on-demand services.

In the context of Cloud computational resources, a typical application involves a first part running at the edge (e.g., to acquire sensor data), while other components run in the cloud (e.g., to carry out heavy processing faster). The goal is thus to understand "where happens what", that is addressing the problem of where the computation must be performed at each moment in order to ensure that the overall application can always run in a seamless way. When offloading a critical task this way, network outages cause the components to be no longer able to exchange data. This severely needs a prediction algorithm, which must make the application able to temporarily switch to a "backup service" in the shortest possible time.

The thesis aims at investigating this problem by comparing Robot Operating System (ROS) application-specific solutions based on the LifecycleNodes abstraction and more general cloud-native solutions. The main technology addressed here to implement a computing continuum is the Liqo.io project, started at Politecnico di Torino. Liqo enables a straightforward way to manage Kubernetes multi-cluster topologies and allows to design a switching algorithm in terms of network traffic control among nodes (i.e., through NetworkPolicies resources). This has led to a prototype of a seamless-switching working mode for a general application, including ROS tasks in this specific use case.

By exploring these different technologies, it was important to realize how latency represents the main challenge when it comes to inter-cluster communications, which obviously worsen with the more critical the tasks to be offloaded become. This study might later involve robot fleets possibly in their design phase, making it possible to build them lighter, more energy-efficient and able to perform complex tasks. The contribute to this distributed Liqo-based cloud model can finally lead to the use of external resources and computing power either from a private cloud or from nearby stand-by robots, even from servers of a partner provider.

Acknowledgements

Vorrei innanzitutto ringraziare il Professor Fulvio Risso, mio relatore, per i preziosi insegnamenti ricevuti dentro e fuori dall'aula, e per avermi trasmesso passione in questo campo. Ringrazio Daniele Cacciabue, correlatore e ottimo compagno di viaggio: infiniti i consigli e le ore che hai dedicato a me e alla mia tesi, le quali sono state soprattutto stimolanti occasioni di scambio. Inoltre ringrazio Enrico Ferrera e Francesco Aglieco dell'istituto di ricerca LINKS Foundation per la particolare disponibilità e per avermi fornito materiale essenziale alla realizzazione della tesi.

Sarebbe impossibile esprimere a parole il sentimento di gratitudine che provo nei confronti di Mamma, Papà e mia sorella, dei miei zii e di tutta la mia famiglia. Sicuramente un pensiero speciale va a tutti i miei nonni che hanno sempre dato e continuano a dare un senso a tutto questo, insegnandomi che "è più facile se la vedi come un gioco!"

Indispensabile è stato il sostegno e il continuo incoraggiamento di tutti i miei amici. Liborio, Riccardo, Luca, Jimy, Giovanni, Paolo e Arianna: alla vostra pazienza e al vostro affetto non rinuncerei mai, grazie per essere stati sempre lì anche solo ad incrociare le dita per me.

Ringrazio infine Iacopo, solo tu hai idea di come ogni tuo abbraccio abbia contribuito a questo lavoro. Grazie per l'inesauribile forza che mi hai dato e perché mi stai sempre accanto. A tutti voi dico ancora un forte grazie, per esserci stati e perché so che ci sarete sempre.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 What is the Cloud Continuum

The concept of Cloud Continuum has gained significant traction among researchers and industries. This growing interest underscores the need for a clear and unified definition of this new concept to figure out the full extent of the Cloud and its actual capabilities.

For many years, managing computer infrastructures revolved around physical hardware servers. The primary focus was on "how to do" things, that is configuring, maintaining and troubleshooting individual machines often still deployed on premise. This hardware-centric approach limited scalability and flexibility.

A turning point came with the emergence of computing virtualization. This technology revolutionized the IT world by allowing multiple virtual machines to consolidate on a single physical server, significantly improving resource utilization and efficiency. Virtualization fundamentally changed the way to think about infrastructure, moving the focus from "how to do" to "what to do": instead of managing individual servers, IT professionals could now focus on the specific services and applications running on top of the virtualized infrastructure.

However, virtualization also introduced a new challenge: managing a growing number of geographically dispersed physical servers. **Cloud computing** thus emerged as the next-level solution for accessing generic resources over the internet, absolutely on-demand and at scale, thanks to datacenters. This brought to the brand-new concepts of “serverless” and software-as-a-Service (SaaS) which definitively abstracts the underlying physical hardware, allowing users to access computing resources and any kind of application by eliminating the need to manage physical infrastructure entirely, something that looks like, precisely, as a service.

Today the Cloud is often viewed as an endless pool of resources on which to build and scale applications for various purposes. With that been said, if the Cloud is the resource pool, Cloud Continuum can be seen as an extension of such resources and of computational capabilities, widely spanning from a datacenter to the very edge of the network, promising scalability and reliability.

The concepts of **Cloud, Fog and Edge** computing thus complete the picture of a **seamless continuum** of computing resources and services, forging terms like "*Cloud-to-anything*" opposed to independent application resource pools. This new way of conceiving computing has emerged from the need of applications which continuously produce large amounts of data (e.g., IoT, robots, mobile devices, sensor nodes) to achieve lower latencies and better processing and storage capabilities. These requirements, together with resilience, self-healing and security, certainly represent the main challenges to face, in order to make all related deploying and orchestration operations actually seamless.

Over the years, there has been a shift in autonomous robots from operating independently to relying on connection with external systems. Traditionally, limited network capabilities restricted robots reliance on external processing. Now, with better connections, robots can "**offload**" complex tasks, freeing up their own computing power. This improves efficiency but introduces latency as data travels. Minimizing latency is especially crucial in situations where robots and people are used to working together. One specific solution is using edge computing to bring processing power closer to the robots, enabling faster communication and real-time decision-making, thus reducing latency.

Considering a distributed application spanned across such fluid infrastructure, this thesis presents a possible Cloud continuum architecture. This has been implemented using a combination of containerization technologies and a distributed multi-cloud resource-management framework such as **Liqo** and **Kubernetes**, focusing in particular on a **ROS** (Robot Operating System) application. Liqo makes this process much easier by creating a unified view of both the local and remote resources, resulting in a single "virtual cluster" which aggregates processing power and services. This innovative approach allows to explore how the offloading of critical tasks can be realized, focusing on real-time complex application that in particular involve autonomous navigation systems in robotics, with the aim of improving efficiency and scalability of such applications. The methodology chapters will delve into the possible implementations using a stateless Cloud prototype environment, developed in collaboration with LINKS Foundation research institute.

Similar service-oriented technologies might represent a new base for the implementation of a transparent **switching algorithm,** not only towards the Cloud but, if necessary and possible, on something that remains close to the edge device, which could be a nearby robot temporarily on stand-by or some edge server of a provider RAN network, according to the approach called Multi-access Edge Computing (MEC). Future implementations could thus provide the robots with discover abilities to find partner servers, upgrading the switching algorithm and making it a choice of the best solution among the suitable servers where to offload tasks to, effectively reducing delays by shortening the distances.

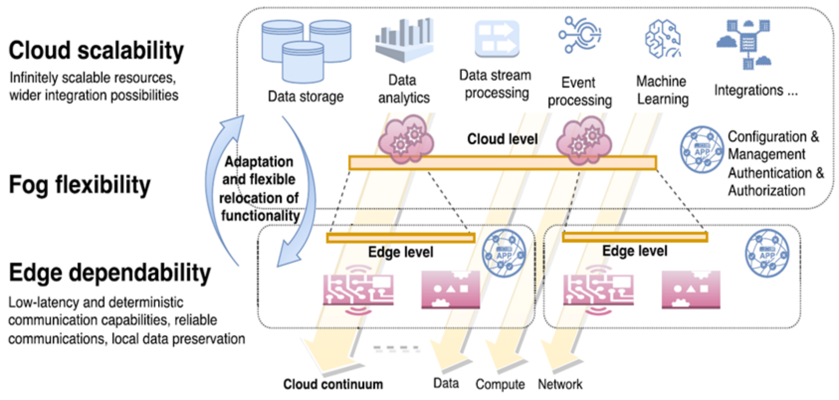

Cloud scalability
Infinitely scalable resources, wider integration possibilities
Fog flexibility
Edge dependability
Low-latency and deterministic communication capabilities, reliable communications, local data preservation
Data storage
Data analytics
Data stream processing
Event processing
Machine Learning
Integrations ...
Cloud level
Configuration & Management Authentication & Authorization
Adaptation and flexible relocation of functionality
Edge level
Edge level
APP
Cloud continuum
Data
Compute
Network

Figure 1.1: Architecture of Cloud Continuum [1]

1.2 Thesis structure

This thesis is organized as follows:

- **Chapter 1 – Introduction**
- **Chapter 2 - Related Works:** a brief overview of some previous works related to this thesis same research field
- **Chapter 3 - Background:** a presentation of tools and technologies used for this work such as ROS, Kubernetes, Zenoh and Liqo
- **Chapter 4 - Cloud Offloading Architecture:** examination of the overall architecture of a possible Cloud offloading system and its expected behavior
- **Chapter 5 - Implementation:** deep immersion into the dynamics of some possible implementations for the switching algorithm, each one with their components, practical implications and challenges
- **Chapter 6 - Experimental Evaluation:** explanation of the results of some tests executed on the most promising of the implementations, presented with charts and guided by a state machine for benchmarking each of the algorithm critical operations
- **Chapter 7 - Conclusions:** reflections on the research process, the outcomes and some suggestions for possible future works and improvements

Chapter 2

Related Works

This chapter will give a brief overview of the research landscape surrounding the task offloading concept, describing the results of some previous works developed in Politecnico di Torino which paved the way for the development of a new Cloud continuum architecture presented later in this thesis.

2.1 ROS2 Dynamic Switching

The previous work aimed at demonstrating that it is possible to efficiently perform Cloud Offloading of certain tasks, taking as use case the autonomous navigation in the robotics field. It specifically analyzed the Robot Operating System (ROS) navigation module and its obstacle avoidance critical process for a physical moving robot [2][3]. To do this, it was studied the possibility of creating a distributed system by performing dynamic switching from some local ROS nodes at the edge (i.e., on the robot itself) towards certain nodes deployed on the Cloud. The nodes on the remote data center must carry out the most CPU-intensive job associated with the internal controller server. This node is responsible for processing all the data received from a set of sensors (e.g. lidar), periodically analyzing a map generated through this data, then finally decide which command (i.e., speed and direction) to give instantly to the waiting robot.

Relying on technologies such as **Kubernetes, Zenoh and ROS Lifecycle nodes**, which will be described in detail in the next Background section, the earlier work outlines its architecture composed by several components distributed across the robot and the remote Cloud server.

ROS and Kubernetes (K8s) were the obvious choices since they have become the de facto standards in their respective fields: robotics and orchestration. In this robotics dynamic system in fact, there is the necessity of an orchestrator to manage the lifecycle of the various micro-services, so Kubernetes has proved to

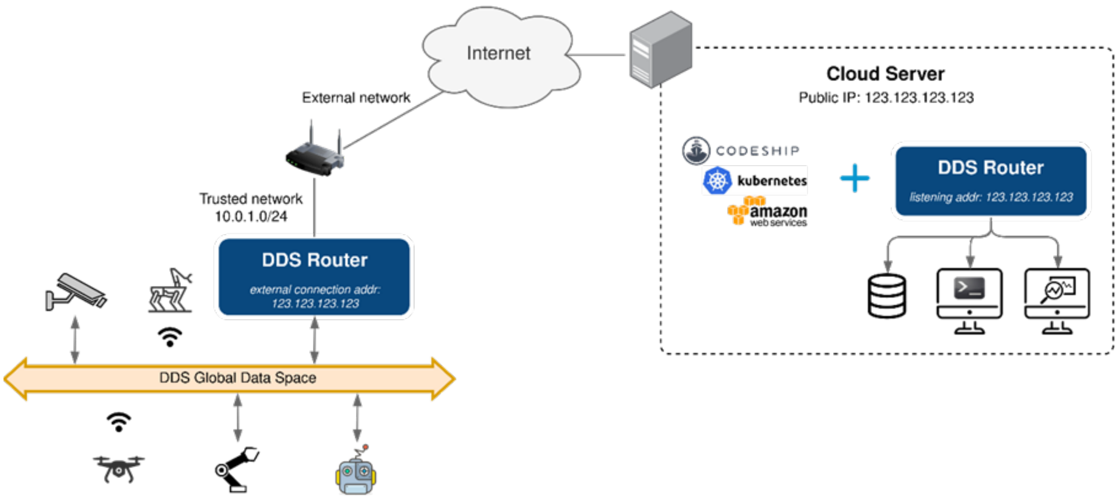


Figure 2.1: Overview of a simple distributed DDS network

be the smartest choice even for embedded devices which are usually non-Cloud systems but can adapt to such Cloud environments, thanks to the availability of versions with low resource demand like K3s.

Zenoh is a very promising protocol for distributed robotic systems. It leverages the publish/subscribe communication model, enabling robots to collaborate effectively in real-time with minimal resource usage [7]. It has recently been announced as the official next ROS2 middleware surpassing DDS in terms of performance and reduced bandwidth consumption, making it one the best option for future internet-based robotic applications.

The **dynamic switching** for the previous theses consists in a process of activating/deactivating target nodes on the robot or on the Cloud respectively. The solution is based on the use of a certain class of nodes (and related interfaces) provided by the ROS environment and called Lifecycle Nodes: they can be configured and activated via simple commands which essentially allow the get/set of the status of a node. According to this architecture, there exist a local and a remote copy of each target node, but only one at a time will be kept active, depending on whether the network conditions in that moment are good enough to make (and maintain) the task offloaded on the Cloud. Finally, the switching operation proves to be convenient when the computational resources on the Cloud server allow the navigation task to run at a higher frequency than the maximum possible locally.

This thesis considers the results of such work as its starting point depicting the context for a further investigation. In the next chapters the research for the best computing continuum architecture will be addressed from a higher-level point of view, providing a Kubernetes cluster also for the robot, in order to take advantage of the potential of some existing tools, such as Liqo, that facilitate the orchestration of multi-cluster architectures, aiming for a solution which is more generic, automated and reliable.

Chapter 3

Background

This chapter provides a comprehensive overview of the tools and technologies that play a central role in this research, such as ROS, Kubernetes, Zenoh and Liqo. The main concepts and functionalities will be covered, in order to better understand how these tools can work together as the base for developing an efficient Cloud-offloading algorithm.

3.1 ROS2: Robot Operating System

The increasing complexity of robotic applications, from self-driving cars to intelligent home robots, made evident the need for a robust software development framework. In response, the Robot Operating System (ROS) emerged as a popular choice, offering a collection of tools and libraries to foster development. ROS2 represents a significant evolution of the ROS framework, built upon the core concepts of its predecessor but offering enhanced performance, scalability, and real-time capabilities [13]. These core concepts, such as distributed systems and message-passing communication, represent the foundation for building robotic applications within ROS 2, referred generally as ROS systems, making it the de-facto standard framework for developing software for robots. This section delves deep into the core functionalities of ROS 2, exploring nodes, messages, topics, services, actions, and the powerful concept of Lifecycle Nodes.

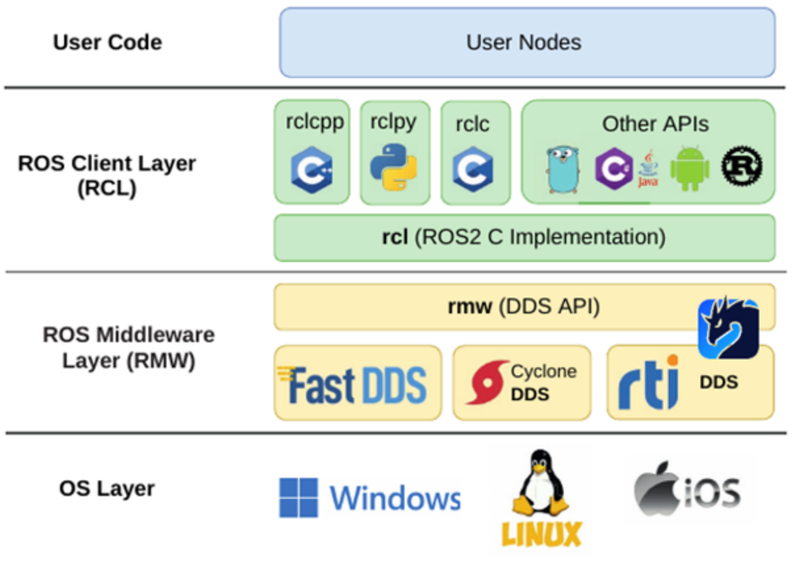


Figure 3.1: ROS2 infrastructure layers

3.1.1 Core building blocks: nodes, messages, services and actions

Nodes

Nodes are the fundamental building blocks of ROS 2 applications. Each node represents an independent process with a specific functionality or task. Nodes communicate with each other to exchange data and collaborate for achieving system goals. Here are some key features of ROS 2 nodes:

- **Independent Processes:** Nodes run as separate processes within the operating system, allowing for modularity and parallel execution.
- **Process Communication:** ROS 2 provides mechanisms (topics and services) for nodes to communicate and exchange data.
- **Language Agnostic:** Nodes can be written in various programming languages, promoting flexibility and leveraging existing developer expertise.
- **Unique Names:** Each node within a ROS 2 system has a unique name for identification and interaction with other nodes.

- **Functionality Encapsulation:** Nodes encapsulate specific functionalities, promoting code organization and reusability.

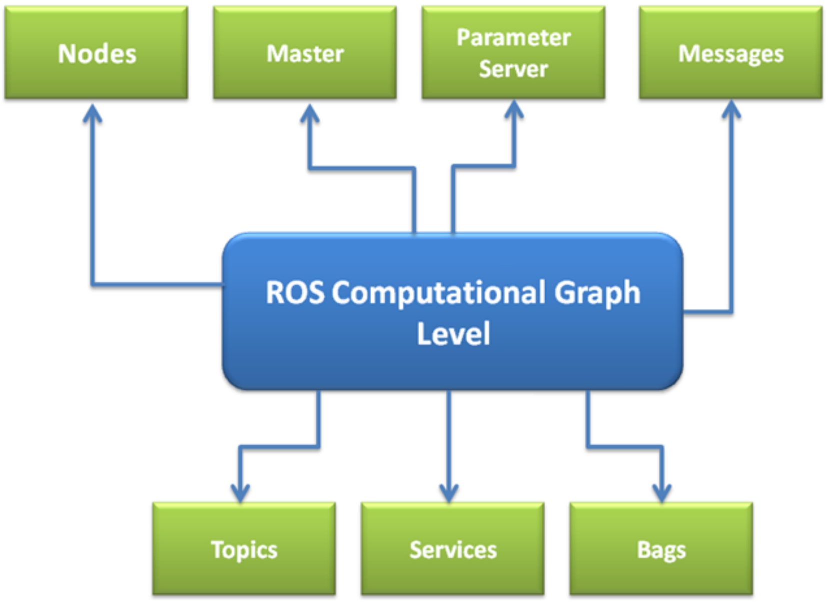


Figure 3.2: ROS2 core building blocks

Messages

Messages are the data structures used by ROS 2 nodes to exchange information. These data types define the format and content of the information being communicated. Messages consist of named fields, each with a specific data type (e.g., string, integer, sensor data). Here are some aspects of ROS2 messages:

- **Structured Data Format:** Messages ensure type safety and clarity in communication, enhancing code maintainability and reducing errors.
- **Customizable Message Types:** Developers can create custom message types tailored to their specific application needs.
- **Standard Message Library:** ROS 2 provides a library of pre-defined message types for common data like sensor readings, robot commands, and navigation information.

Pub-Sub model overview

The publisher-subscriber (pub-sub) model is a widely used architectural pattern. It can be used in software development to enable communication between different components in a system.

In particular, it is very common in distributed systems, where different parts of the system need to interact with each other but don't want to be tightly coupled. This model involves publishers and subscribers, making it a messaging pattern. Specifically, the publishers are responsible for sending messages to the system, while subscribers are responsible for receiving those messages. Mainly, the pub-sub model is based on decoupling components in a system, which means that components can interact without being coupled directly [14].

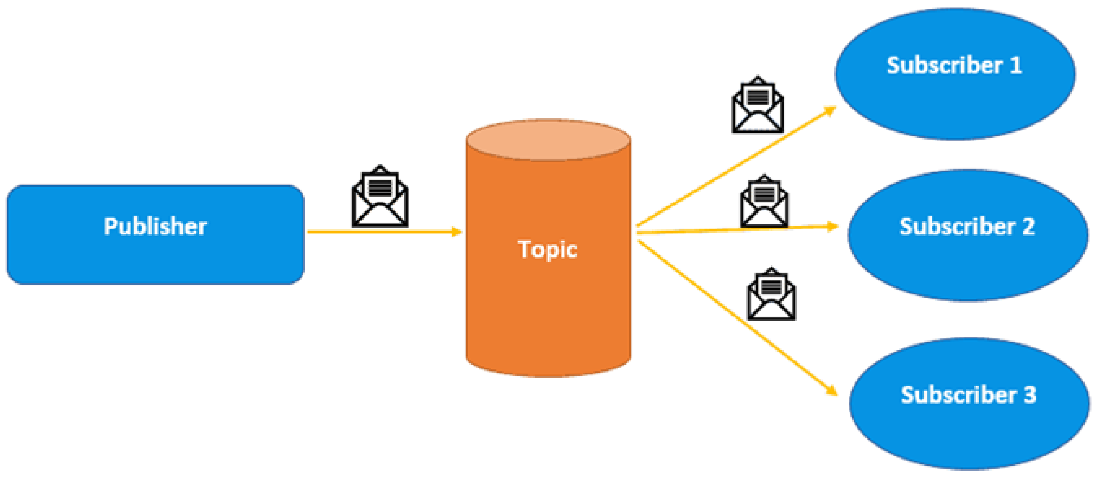
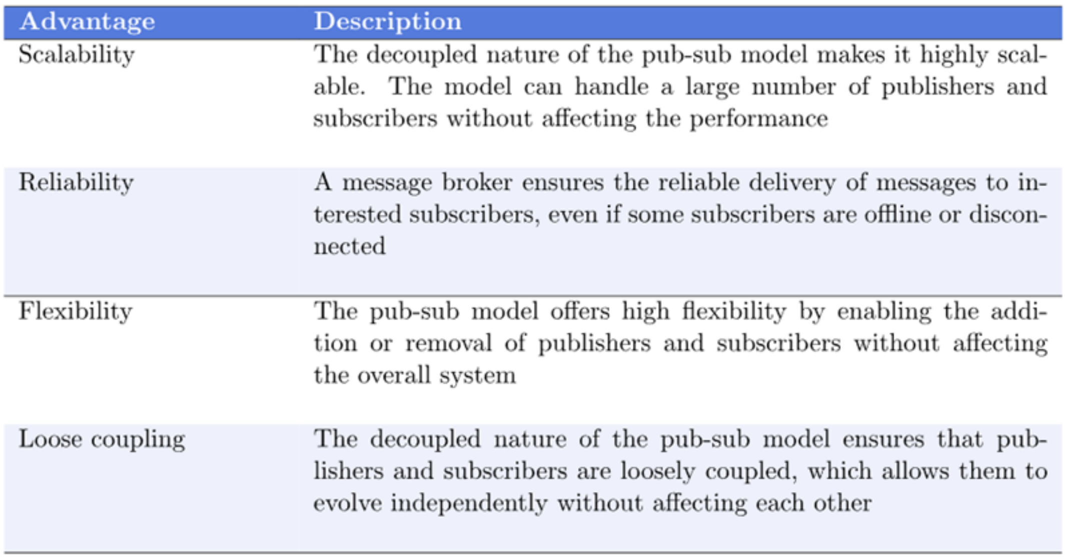


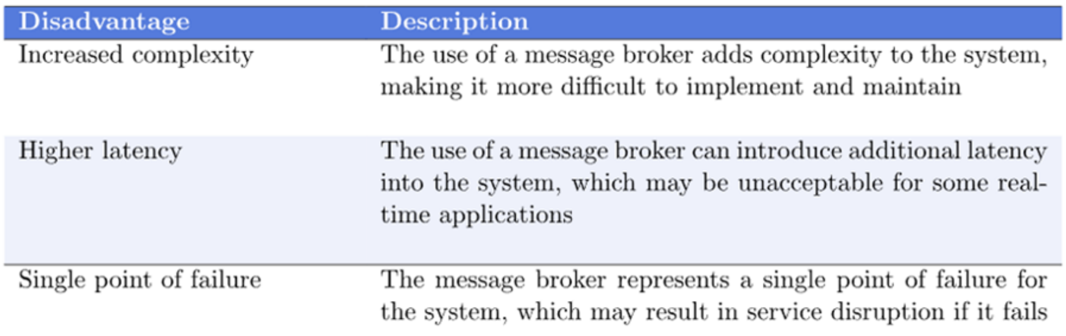
Figure 3.3: pub-sub model architecture

The pub-sub model has several benefits. The following table summarizes its main advantages:



Advantage	Description
Scalability	The decoupled nature of the pub-sub model makes it highly scalable. The model can handle a large number of publishers and subscribers without affecting the performance
Reliability	A message broker ensures the reliable delivery of messages to interested subscribers, even if some subscribers are offline or disconnected
Flexibility	The pub-sub model offers high flexibility by enabling the addition or removal of publishers and subscribers without affecting the overall system
Loose coupling	The decoupled nature of the pub-sub model ensures that publishers and subscribers are loosely coupled, which allows them to evolve independently without affecting each other

However, the pub-sub model also has some drawbacks. The following table shows its main drawbacks:



Disadvantage	Description
Increased complexity	The use of a message broker adds complexity to the system, making it more difficult to implement and maintain
Higher latency	The use of a message broker can introduce additional latency into the system, which may be unacceptable for some real-time applications
Single point of failure	The message broker represents a single point of failure for the system, which may result in service disruption if it fails

ROS2 nodes leverage three primary communication patterns: **topics, services and actions**.

Topics (Publish-Subscribe)

Topics facilitate asynchronous data exchange between nodes. A node can publish messages to a specific topic and any other node can subscribe to that topic to receive those messages. This many-to-many communication allows for loose coupling between nodes, as publishers and subscribers don't need to be aware of each other's specific details.

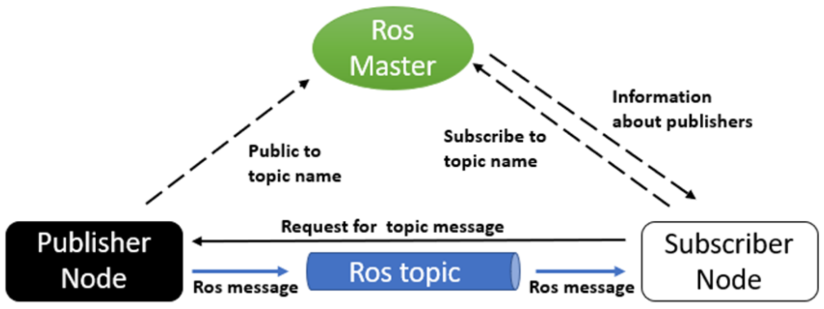


Figure 3.4: pub-sub model applied to a ROS2 system

Services (Request-Response)

Services provide a synchronous request-response communication pattern. A node (client) can send a service request message to another node (server) specifying a desired action or information. The server node processes the request, performs the action, and sends a response message back to the client. This pattern is useful for scenarios requiring guaranteed data association between requests and results. These communication pattern, applied to ROS2 system of nodes, enable efficient data exchange and collaboration between them, allowing developers to build complex robotic functionalities.

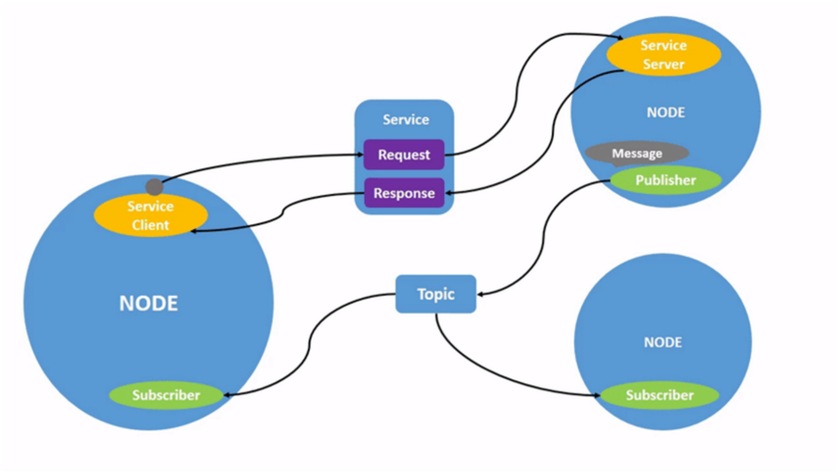


Figure 3.5: ROS2 services

Actions

Actions provide a more structured approach for handling complex tasks that require progress updates and feedback during execution. Actions follow a request-feedback-response pattern:

- **Request:** The client node sends a request message specifying the desired action and potentially initial parameters.
- **Feedback:** The server node can send feedback messages during execution, providing progress updates or intermediate results to the client.
- **Response:** Upon completion, the server sends a final response message indicating success or failure, along with any relevant results.

This pattern is beneficial for long-running tasks where the client needs to monitor progress and potentially react to intermediate results.

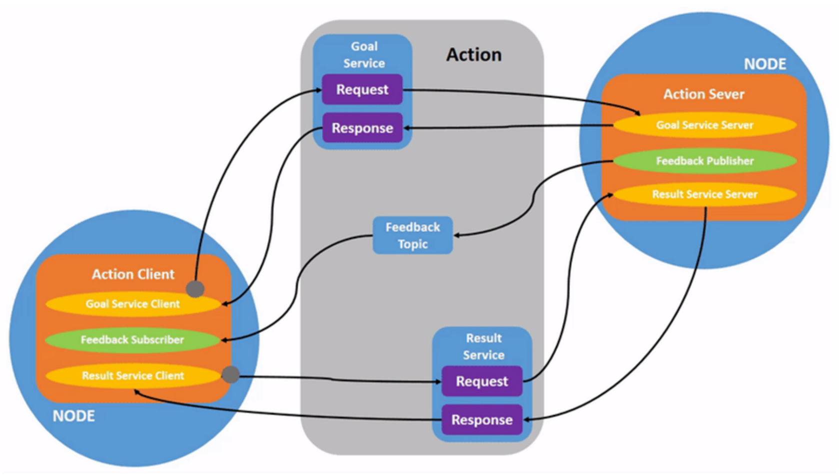


Figure 3.6: ROS2 actions

3.1.2 Lifecyle Nodes

ROS2 Lifecycle Nodes offer a powerful abstraction for managing the lifecycle of individual nodes within a ROS2 system. This concept goes beyond simple startup and shutdown, allowing developers to define and control the various stages a node can transition through during its operation. Here's a comprehensive breakdown of Lifecycle Nodes:

- **Improved System Control:** Lifecycle Nodes provide a structured approach to managing node startup, shutdown, configuration changes, and error handling. This leads to more predictable and robust behavior in ROS 2 systems, especially for complex applications with multiple interacting nodes.
- **Dependency Management:** Lifecycle Nodes allow developers to specify dependencies between nodes. A node can wait for other required nodes to transition to an "active" state before activating itself. This ensures a well-defined startup sequence for complex systems, preventing errors due to missing dependencies.
- **Graceful Error Handling:** Lifecycle Nodes facilitate graceful handling of node failures. If a node encounters an error, it can transition to an appropriate state (e.g., "inactive") to allow for investigation and recovery. This promotes system resilience and easier troubleshooting.
- **Dynamic Reconfiguration:** Lifecycle Nodes enable developers to perform dynamic configuration changes while a node is running. This allows the behavior of a node to be adjusted without requiring a complete restart, improving system flexibility and adaptability to changing conditions.
- **Centralized Control:** External entities can trigger state transitions for Lifecycle Nodes by sending specific ROS2 service requests. This enables centralized control and management of robotic systems, particularly for large-scale deployments with many nodes.

Lifecycle States

ROS 2 defines a set of standard lifecycle states for nodes:

- **Unconfigured:** The initial state after node creation. No configuration has been loaded.
- **Inactive:** The node is not actively processing data or participating in communication.
- **Active:** The node is operational and performing its intended functionality.
- **Shutting Down:** The node is gracefully terminating its execution and cleaning up resources.
- **Errored:** The node has encountered an error and is no longer functioning properly.

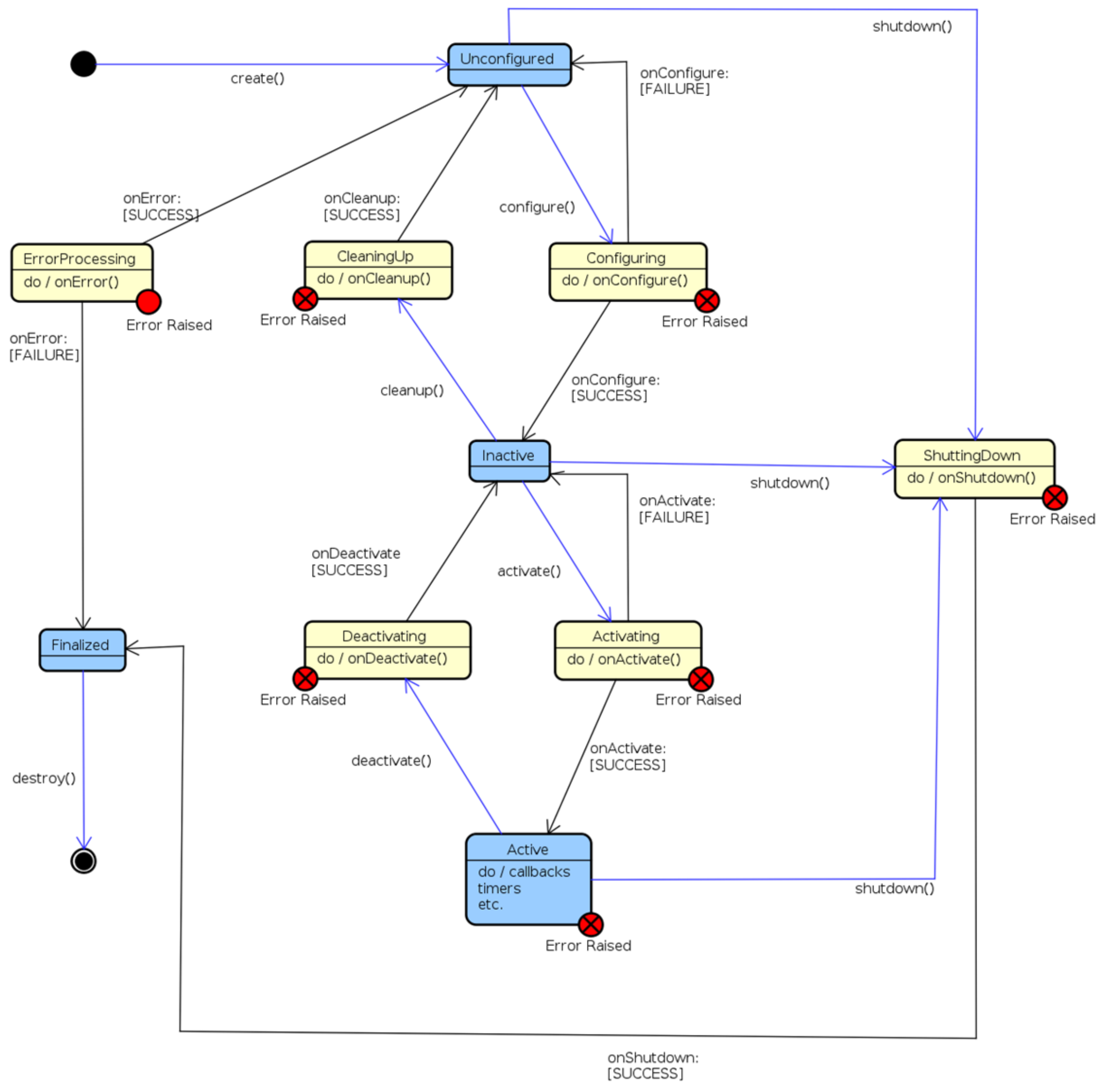


Figure 3.7: ROS2 Lifecycle state machine

Lifecycle Transitions

Lifecycle Nodes can transition between these states based on specific events or conditions triggered from within the node itself, external service requests, or system events. Developers define these transitions and the corresponding logic within their node code using the Lifecycle Node API.

- **Structured Lifecycle Management:** Lifecycle Nodes provide a structured framework for managing the lifecycle of each node, promoting cleaner and more maintainable code.

- **Improved System Reliability:** By handling dependencies, errors, and configuration changes effectively, Lifecycle Nodes contribute to more robust and reliable robotic systems.
- **Flexibility and Adaptability:** The ability to perform dynamic reconfiguration and centralized control through state transitions enhances the adaptability of robotic systems to changing requirements.

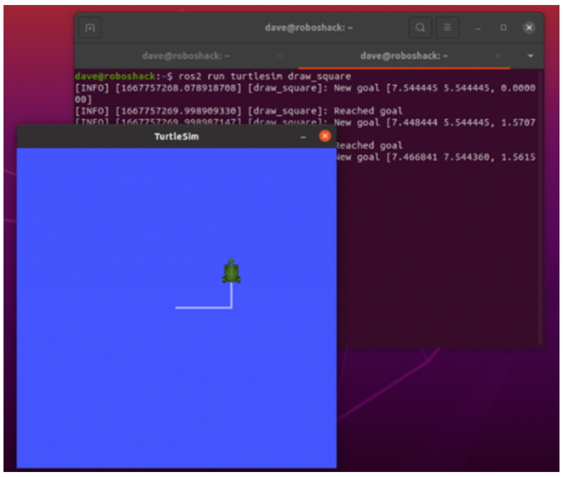
In conclusion, ROS2 Lifecycle Nodes are a valuable tool for developers building complex and reliable robotic systems. By leveraging this abstraction, developers can define a well-controlled lifecycle for each node, leading to more predictable, robust, and adaptable robotic applications.

3.1.3 Turtlesim package

Turtlesim is a popular and fundamental tool used for learning the basics of ROS2 systems. It is a simple, 2D graphical simulator featuring turtles that you can control and manipulate. Turtlesim serves as a great introduction to ROS2 because it is simple and easy to understand and avoids complexities of real robots, allowing you to focus on core ROS2 concepts like nodes, topics, and messages. The graphical interface provides immediate feedback on your actions, making it easier to grasp how ROS2 works. You can experiment with controlling the turtles, spawning new ones, and even drawing shapes through ROS2 commands. The Turtlesim package consists of two main nodes:

- turtlesim_node: This node launches the Turtlesim simulator window, which displays the turtles and their environment.
- turtle_teleop_key: This node allows you to control the turtles using your keyboard. By publishing commands on a specific topic, it directs the turtles to move forward, backward, turn left, or right.

Using ROS2 Turtlesim typically involves two steps: installation by using your ROS distribution's package manager, and launching to open the simulator window. In order to control the turtle, you have to execute the turtle_teleop_key node and gain control your keyboard. Exploring further, you can spawn additional turtles, create more complex motion patterns, and even visualize sensor data.



```
dave@roboshack:~$ ros2 run turtlesim draw_square
[INFO] [1667757268.078918708] [draw_square]: New goal [7.544445 5.544445, 0.0000
00]
[INFO] [1667757269.998909330] [draw_square]: Reached goal
[INFO] [1667757269.998987147] [draw_square]: New goal [7.448444 5.544445, 1.5707
                                              Reached goal
                                              New goal [7.466841 7.544360, 1.5615
```

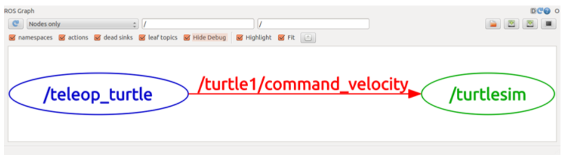


Figure 3.8: ROS2 Turtlesim package example and its ros_graph

3.2 Zenoh

Zenoh emerges as a new contender in the realm of distributed system communication, offering a comprehensive approach that goes beyond the traditional publish-subscribe (pub/sub) paradigm. It aims to unify how to handle data in motion (real-time streams), data at rest (stored data), and computations across geographically dispersed systems. Here's a closer look at Zenoh core functionalities and its potential benefits for various applications.

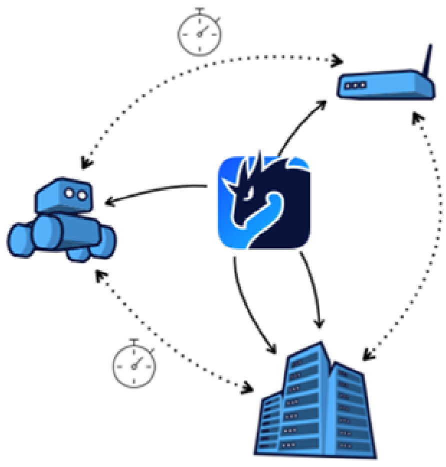


Figure 3.9: Zenoh typical interactions

A unified vision for data

Zenoh builds upon the familiar pub/sub concept, enabling efficient data exchange between applications. However, it takes this concept a step further by offering greater flexibility. Publishers can disseminate data with various granularities, allowing them to share entire datasets or specific subsets as needed. Subscribers, on the other hand, can express interest in specific data patterns or filters, ensuring they only receive the information truly relevant to their tasks.

This pub/sub foundation is further enhanced by Zenoh seamless integration with storage mechanisms. Data can now persist across geographically dispersed locations, making it readily accessible from diverse edge devices and cloud resources within a distributed system. Zenoh empowers applications to not only exchange data in real-time but also query it efficiently, both in active streams and within the persistent storage layer. This allows for targeted retrieval of relevant information based on specific criteria, regardless of the data's location within the system.

Efficiency and scalability

Zenoh is designed with efficiency in mind, allowing it to operate effectively on resource-constrained devices at the network edge as well as on powerful server-grade hardware. This makes it a suitable choice for a wide range of real-time application requiring fast response times. Furthermore, Zenoh offers a flexible data model that supports various data types, promoting interoperability by working seamlessly and transparently with existing communication protocols and middleware solutions on already existing infrastructures. A key strength of Zenoh lies in its decentralized approach. Data and computations are distributed across the network, enhancing system resilience and fault tolerance. There's no single point of failure, as the system can continue to operate even if individual components experience issues.

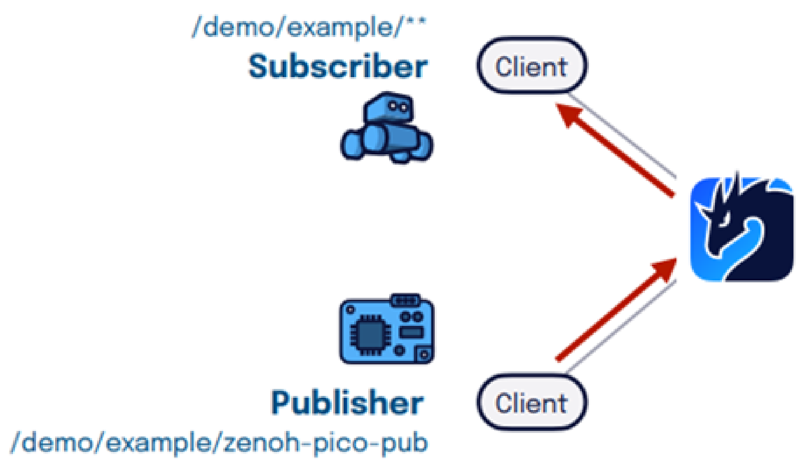


Figure 3.10: pub-sub pattern with the mediation of Zenoh

Potential applications for Zenoh

Zenoh capabilities make it a perfect choice for various applications, including:

- **Industrial Automation:** Real-time data exchange, edge computing, and seamless integration with diverse industrial sensors and devices
- **Internet of Things (IoT):** Efficient data collection, processing, and analysis from a large number of geographically dispersed IoT devices
- **Autonomous Systems:** Real-time communication, sensor data fusion, and on-board computation for autonomous robots and vehicles. Low-latency data exchange for high-frequency applications

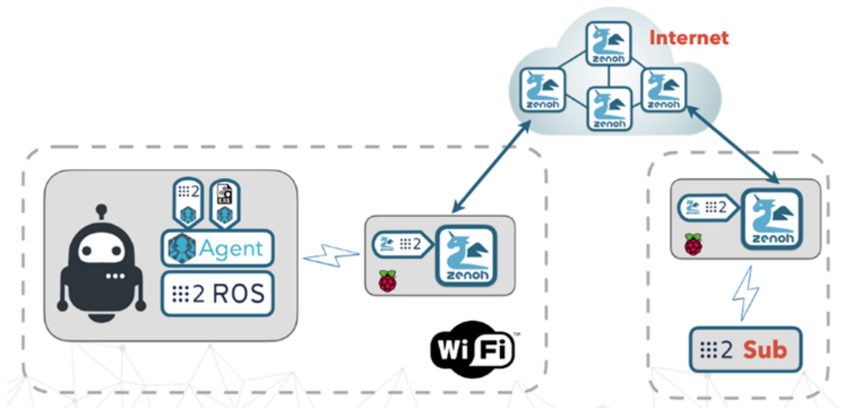


Figure 3.11: a wide range of applications can work at internet-scale through Zenoh

Zenoh vs DDS

Zenoh extends beyond the core strengths of DDS by offering a unified approach that encompasses data in motion, data at rest, and computations. This broader vision allows developers to manage not just real-time data streams but also persistent storage and embedded computations directly within the data flow. While the Data Distribution Service (DDS) with its DCPS (Data-Centric Publish-Subscribe) protocol remains a dominant force in real-time data exchange for distributed systems like ROS 2, Zenoh reveals as a potential challenger offering a broader set of functionalities. Both Zenoh and DDS are designed for scalability, allowing them to handle complex distributed systems effectively. However, Zenoh lightweight design makes it potentially well-suited for resource-constrained edge devices, a growing consideration in modern robotic deployments.

Optimizing Data Transmission

"Discovery overhead" refers to the network traffic and processing resources consumed by ROS2 nodes to find available topics and services. Here are some strategies to minimize it:

- **Logical Namespaces:** Organizing nodes and topics within logical namespaces reduces the overall search space for discovery requests.
- **Content-based Filtering:** Subscribers can specify filters in their subscription requests, allowing them to receive only relevant data, reducing unnecessary data transmissions.

- **Leveraging DDS Features:** Exploring features like "lazy discovery" can further optimize discovery by deferring it until a node attempt to publish or subscribe

ROS 2 leverages a concept called "content-based publish-subscribe." When a node declares a publication, it specifies the data type and any associated metadata. With Zenoh, declaring a publication allows to not send data on the wire when there are no matching subscribers, thus saving bandwidth from unnecessary transmissions. If no subscribers exist, the node can optimize its behavior and avoid sending unnecessary data packets over the network, conserving bandwidth and improving network efficiency.

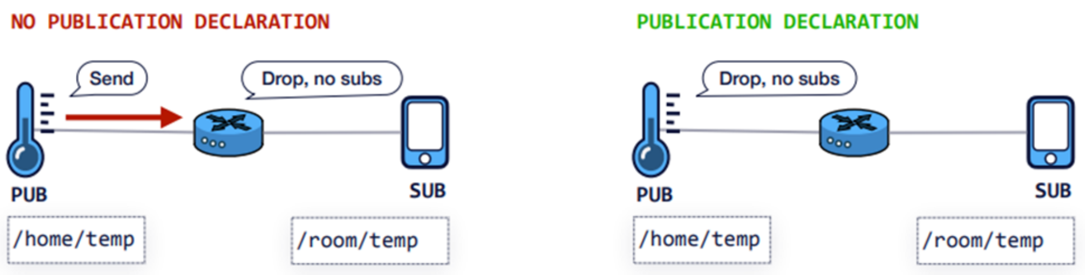


Figure 3.12: minimizing discovery overehead

3.2.1 Zenoh plugin for DDS: bridges and routers

Zenoh offers a plugin that acts as a gateway or bridge between Zenoh and DDS systems. This functionality can facilitate communication between ROS 2 nodes and applications using other DDS implementations that are not natively compatible with ROS 2, by transforming or filtering data streams before they are passed between two different systems. ROS 2 utilizes bridges, routers to facilitate data flow across a distributed network:

- **Bridges:** Connect separate ROS 2 networks, allowing data exchange between nodes in each network. They operate at Layer 2 of the OSI model, forwarding all data packets received from one network to the other
- **Routers:** Operate at Layer 3 and make intelligent routing decisions based on destination addresses, improving efficiency and potentially isolating network segments for security

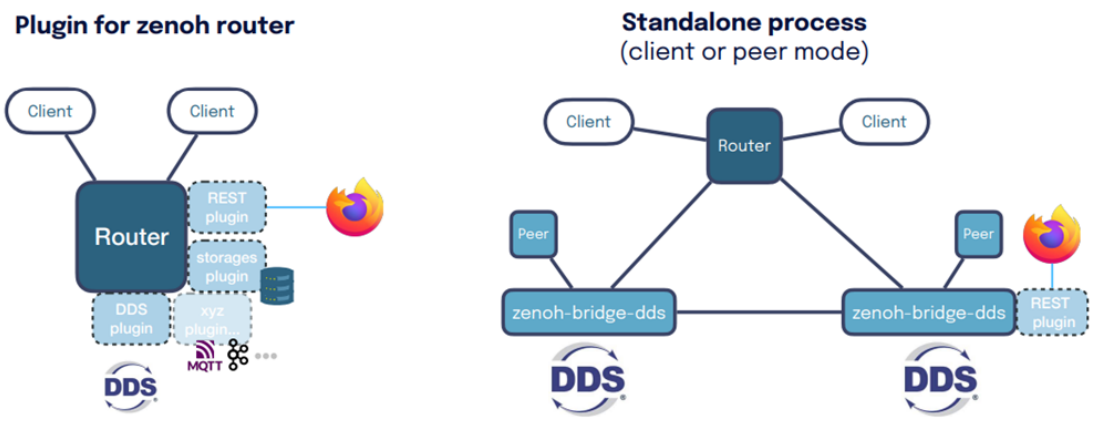


Figure 3.13: two Zenoh configurations for working with DDS

While both bridge-router communication and router-router communication serve the purpose of data routing within Zenoh, they operate at different levels and handle data movement in distinct ways (see figure 3.13).

Bridge-Router communication

Bridge-router communication acts like a network bridge, functioning at Layer 2 essentially connects separate Zenoh networks, forwarding data packets it receives from one network to all connected nodes on the other network. Unlike routers, bridge-routers don't make intelligent routing decisions. They simply relay all data they encounter. These bridges are able to connect ROS nodes belonging to different ROS_DOMAIN_IDs: this approach is suitable for straightforward network extensions where all nodes on the receiving network require access to all data from the sending network. However, bridge-router communication can become inefficient, especially in large and complex Zenoh deployments. Since it blindly forwards all data, it can lead to unnecessary network traffic if not all nodes on the receiving network actually need the data.

Router-Router communication

Router-router communication, on the other hand, operates at Layer 3 and focuses on connecting and facilitating communication between multiple Zenoh networks on the Internet. Unlike bridge, routers act as intelligent traffic directors. They maintain routing tables that specify the most efficient path to reach specific network segments. When a data packet arrives at a router, its destination address is examined. By consulting its routing table, the router determines the optimal route for the packet to reach its intended recipient. This intelligent routing approach is essential for large and complex Zenoh deployments. It ensures that data reaches only the nodes that require it, reducing network congestion and improving overall system performance.

Choosing the right approach:

The decision between bridge-router and router-router communication depends on the specific needs of your Zenoh network. If you have a simple network extension where all nodes require access to all data, bridge-router communication might be enough. However, for larger and more complex deployments possible spanning across geographically distributed servers, efficient data routing becomes crucial, and router-router communication is the preferred approach, thanks to its ability to optimize data flow and minimize unnecessary network traffic.

In conclusion, while DDS remains a robust tool for real-time communication in ROS 2, Zenoh offers a broader set of functionalities that developers can explore for optimizing data management within robotics systems. Understanding discovery overhead, Zenoh plugins, bridge/router concepts, and data transmission optimization techniques empowers developers to build more efficient and scalable robotic applications, deciding on the most suitable communication approach based on the specific needs of the project.

3.3 Kubernetes

Kubernetes, often shortened to K8s, has become the de facto standard for containers orchestration in the modern application development world. This open-source platform automates the deployment, scaling, and management of containerized applications across clusters of machines. It excels at managing the lifecycle of containerized applications, automating tasks like deploying containers across a cluster, scaling deployments up or down based on demand, and restarting failed containers to ensure high availability. This translates to robust and resilient applications that can adapt to changing demands. Kubernetes also comes with K3s, a lightweight K8s distribution designed for resource-constrained environments, perfect for bringing container orchestration to places where traditional Kubernetes might be too heavy.

Before Kubernetes, applications used to be deployed as monoliths, requiring entire servers (bare metal) just for them. Virtualization came along, letting multiple applications share a single server, like splitting the machine resources into smaller sections. This was far better, but applications still carried the weight of a full operating system. Now, with containerization, applications are broken down into tiny, focused microservices, each in its own lightweight container, each needing only its specific function. This allows for, faster deployments, better scaling and easier development, all helping to focus on building the service, not on the single server (see figure 3.14).

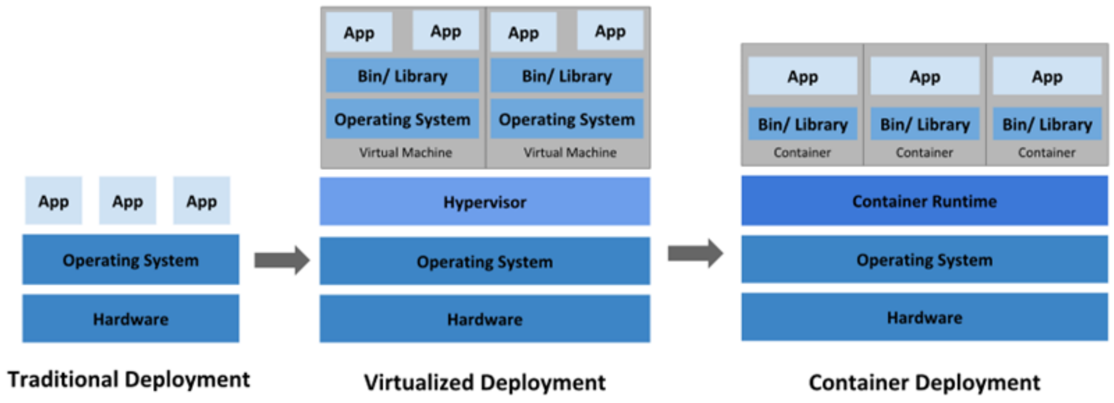


Figure 3.14: applications developing evolution

Kubernetes creates a service abstraction that hides the underlying server details and automatically directs incoming traffic to healthy instances within the deployment. Additionally, Kubernetes offers built-in load balancing, ensuring traffic is distributed evenly across multiple containers, preventing any single container from becoming overwhelmed. Kubernetes also boasts self-healing capabilities. It constantly monitors the health of containers (in their pods) and machines (worker nodes) within the cluster. If a container malfunctions, rescheduling the containers running on that machine to other healthy nodes, ensuring minimal downtime for applications.

Declarative Management

As far as the creation of resources for deploying the applications, Kubernetes utilizes a declarative approach. Developers specify the desired state of their application deployments using YAML files or custom objects. Kubernetes then acts as the orchestrator, working to maintain that desired state, automating the deployment and management processes. Additionally, Kubernetes employs a sophisticated scheduling algorithm to place containers on appropriate machines within the cluster, considering factors like resource availability, machine health, and any affinity/anti-affinity rules defined by developers to optimize performance and resource utilization [5].

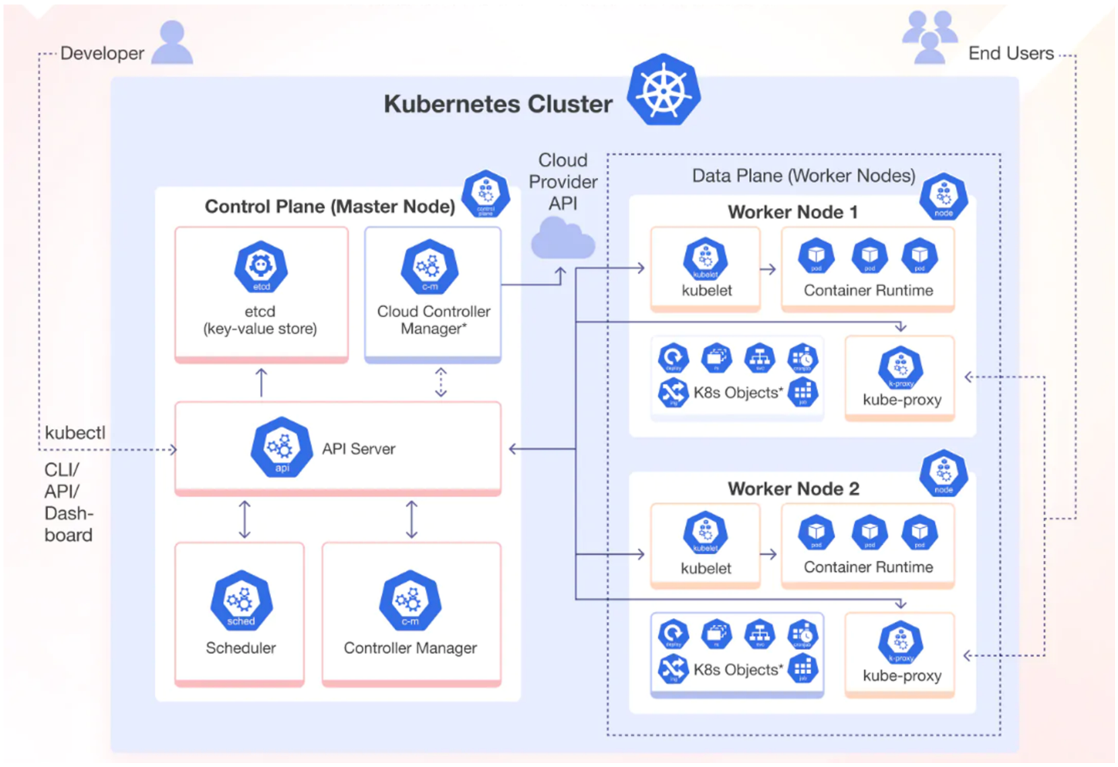


Figure 3.15: Kubernetes Architecure Overview

The Need for Container Network Interfaces (CNI)

While Docker provides basic container networking functionalities, Kubernetes relies on CNI plugins for advanced network configuration within a cluster. Here is where CNI plays a crucial role:

- **Flexibility and Customization:** CNI plugins offer a diverse range of networking options, allowing developers to choose the most suitable solution for their specific needs. Think of them as different musical styles – each CNI plugin offers a unique way to connect containers within the cluster.

- **Multi-Host Networking:** CNI plugins enable seamless communication between containers running on different machines within the cluster. This is critical for distributed applications where microservices need to interact with each other

- **Policy Enforcement:** CNI plugins can be used to implement network security policies within the cluster

3.3.1 Cilium and NetworkPolicies

Cilium is one the available CNI plugins for Kubernetes which goes beyond traditional networking solutions. It operates at the granular level of individual containers, providing a comprehensive approach to both security and connectivity. It enforces security policies to safeguard the cluster environment, while simultaneously facilitating efficient communication between containers. This ensures seamless data flow and robust protection, all without the need for complex configurations or unwanted network overlays. Cilium's lightweight design and deep integration with container technologies make it a valuable asset for modern application development [15].

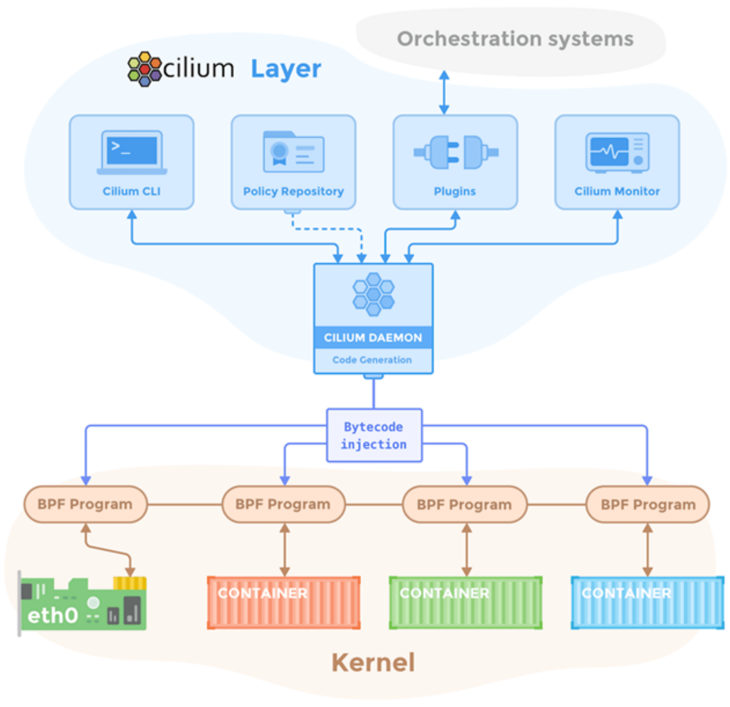


Figure 3.16: Cilium architecture overview

Ensuring secure communication between these containers remains a critical challenge. **K8s Network Policies** emerge as a powerful solution, offering the possibility to control network traffic within a Kubernetes cluster, allowing to achieve several key objectives:

- **Enhanced Security:** By restricting unnecessary network access for containers, K8s Network Policies significantly reduce the attack surface within a cluster. This approach minimizes potential security vulnerabilities

- **Enforced Application Logic:** Network policies can be configured to enforce the intended communication patterns within an application. This ensures containers interact only with specific resources and services they require for proper functionality. This prevents unintended data leaks or disruptions in communication flow

- **Simplified Network Management:** K8s Network Policies offer a centralized approach for managing network security across the entire cluster. This eliminates the need for manual configuration of firewalls for individual pods, streamlining network management tasks

While K8s Network Policies offer a powerful security tool, their effectiveness strongly depends on careful planning and configuration. Overly restrictive policies can interfere with application functionality by limiting necessary communication channels. Striking a balance between security and functionality is crucial for successful implementation.

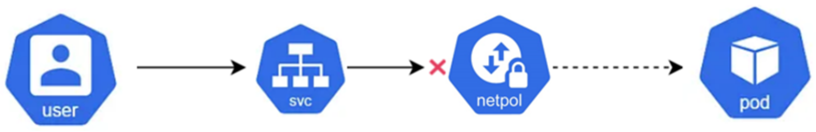


Figure 3.17: how a network policy works

Cilium, as well as other cni, comes with its own network policies, and is able to enforcing control over network traffic with the finest granularity (see figure 3.18). These **CiliumNetworkPolicies** come in different flavors, each offering a specific level of detail:

- **Endpoint-based**: These policies are like bouncers at the container door, controlling communication between individual containers.

- **Label-based**: Imagine assigning security tags to your containers. Label-based policies act like group permissions, restricting traffic flow based on these shared labels.

- **Network-based**: These policies work like traditional firewalls, controlling access based on IP addresses or CIDR ranges.

- **Service-based:** Think of services as departments within your application. Service-based policies manage secure communication channels between these departments.

- **Entity-Based**: Cilium network policies can be configured to target specific entities, such as the kube-apiserver. This allows you to define fine-grained access controls, specifying which pods or namespaces are authorized to communicate with the kube-apiserver. By restricting access to the kube-apiserver only to authorized entities, you significantly reduce the attack surface and potential security vulnerabilities within your cluster.

Moreover, Cilium also offers policies that may have control at the cluster level called CiliumClusterwideNetworkPolicies and act as "global" security rules, applying to all containers across the entire cluster, ensuring consistent security across your entire application landscape.

With this range of policy options, Cilium empowers you to tailor network security to your specific needs, creating a safe and efficient environment for your containerized deployments.

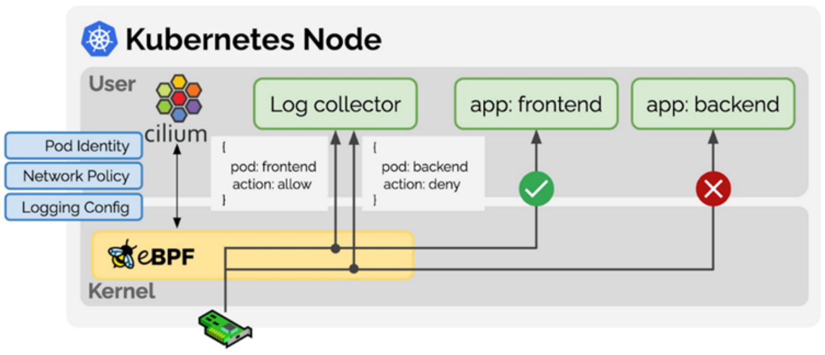


Figure 3.18: Cilium Network Policies enforcing example

3.3.2 ServiceAccounts and ClusterRoles

Kubernetes implements a robust access control system using Role-Based Access Control (RBAC). This system, consisting in a well-defined set of regulations, governs how entities within the cluster interact with resources and ensures only authorized actions are performed. Kubernetes RBAC, with its roles, service accounts, and role bindings, provides a comprehensive framework for access control within containerized environments. This system ensures only authorized services can interact with resources, fostering a secure and controlled environment for running containerized applications at scale. The RBAC framework operates on three core components:

- **Roles:** These represent predefined sets of permissions within a specific namespace (a logical grouping of resources). Roles define the allowable actions (e.g., read, write, delete) on specific resources within the namespace (e.g., deployments, pods, secrets)

- **Service Accounts:** Unlike human users who can directly interact with the Kubernetes API server, applications in Kubernetes run as services. Service accounts function as unique identities for these services, enabling them to authenticate and request access to resources from the Kubernetes API server
- **Role Bindings:** These establish the connection between roles and service accounts. Administrators can bind specific roles to relevant service accounts, effectively granting them the necessary permissions to perform their designated tasks within the cluster. This binding process acts as an authorization step, ensuring services only have access to the resources required for their specific functionalities

3.3.3 K8's Client Libraries

The Kubernetes API server acts as the central nervous system of the container orchestration platform. It receives commands, manages resources, and orchestrates the activities of containerized applications. There are two primary methods for accessing the Kubernetes API:

- **Direct Access:** This method involves using tools like curl or custom code to interact with the API server directly. However, for direct access, RBAC plays a crucial role. You'll need the necessary API credentials (token or certificate) associated with a service account that has been granted the appropriate RBAC role for the desired action. This approach offers granular control but requires a deeper understanding of the Kubernetes roles, bindings and service accounts
- **Kubernetes Client Libraries:** These libraries, available in various programming languages, provide a more user-friendly and secure way to interact with the Kubernetes API. The libraries handle authentication and authorization behind the scenes, without the need of directly managing service accounts and RBAC configurations. Developers simply need to specify the desired actions on resources, and the library takes care of the secure communication with the API server. While less granular, client libraries simplify development and ensure applications adhere to established RBAC policies

In conclusion, Kubernetes serves as a powerful platform for managing containerized applications at scale. Its functionalities for deployment, scaling, service discovery, resource management, and self-healing automation make it a crucial tool for modern application development. The ability to leverage CNI plugins allows for flexible and customizable network configurations within the cluster. For resource-constrained environments, K3s offers a lightweight alternative, delivering the core functionalities of Kubernetes with a reduced footprint and simplified management, making it a valuable option for edge computing and IoT deployments.

3.4 Liqo

The world of container orchestration is a dynamic landscape, constantly evolving to meet the demands of modern applications. Managing workloads across multiple Kubernetes clusters, however, presents a significant challenge. Traditional approaches often rely on complex network configurations or cumbersome VPN solutions, creating bottlenecks and decreasing agility. Liqo emerges as a brand-new open-source project, offering a game-changing solution for multi-cluster deployments.

Inspired by the concept of "**liquid computing**", Liqo establishes connections between disparate Kubernetes clusters, enabling them to share resources and services. This creates a dynamic pool of resources which adheres perfectly to the concept of Cloud Continuum. With Liqo, clusters can in fact contribute their unused resources at any given time, reducing infrastructure costs for others and creating new possibilities for the edge-to-anything computing.

One of Liqo key strengths lies in its ability to seamlessly extend the standard Kubernetes APIs. This transparency ensures minimal modifications are required for existing applications to operate within a Liqo environment. Resources familiar from single-cluster deployments are not only applicable but often enhanced for Liqo specific objectives.

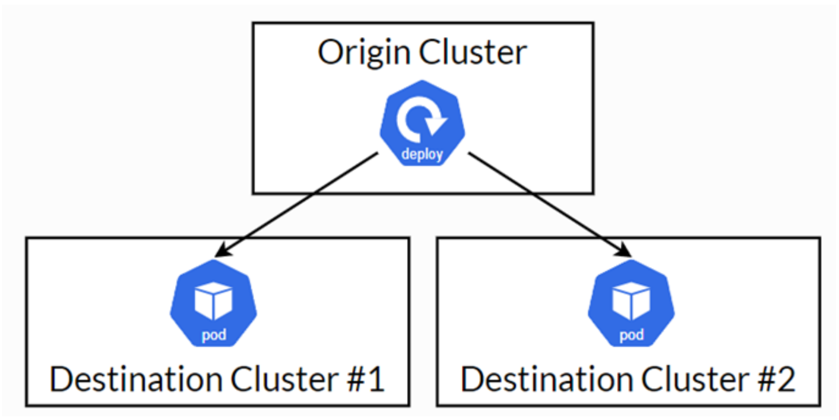


Figure 3.19: typical multi-cluster scenario achieved thanks to Liqo

Key components and operations

Liqo architecture relies on several critical components to enable its functionalities:

- **liqoctl:** it is the CLI tool for the installation and management of Liqo. Specifically, it abstracts the creation and modification of Liqo defined custom resources by wrapping the corresponding Helm commands and automatically retrieving the appropriate parameters based on the target cluster configuration. It also establishes and revokes peering relationships towards remote clusters, configures workload offloading and retrieves the status of Liqo

- **Peering:** unidirectional resource and service consumption relationship between two Kubernetes clusters, with one cluster (i.e., the consumer) granted the capability to offload tasks to a remote cluster (i.e., the provider), but not vice versa. In this case, the consumer establishes an "outgoing peering" towards the provider, which in turn is subjected to an "incoming peering" from the consumer

- **Virtual Kubelet:** the "kubelet" is a critical program running on each Kubernetes worker node. It manages pods, monitors their health, and interacts with the container runtime to ensure your containerized applications run smoothly. The "virtual kubelet" therefore similarly interacts with the Kubernetes API server as if it were a physical node. This allows Liqo to manage resources in a granular and efficient manner within the federated clusters

- **Foreign Cluster:** this component represents a remote cluster within the local cluster's context, holding essential information for establishing and maintaining connections.

- **Virtual Node:** it is the result of the peering process within the local cluster's context, corresponding to the representation of a remote cluster resource. This allows the local cluster scheduler to deploy pods on the remote cluster as if it were a local node

- **Namespace Offloading:** this process enables the transparent execution of pods in remote clusters, facilitating resource optimization across the Liqo environment

- **Reflection:** this is the main Liqo feature that enables the mirroring of essential resources for offloaded workloads. Thanks to reflection, Liqo handles the complexities of the offloading operation, like remapping information and makes the pods work seamlessly across clusters

3.4.1 Peerings and Foreign Clusters

Peering is a fundamental pillar of Liqo. This process establishes secure connections between clusters, each playing a specific role:

- **Consumer Cluster:** Requests resources from other clusters
- **Provider Cluster:** Offers unused resources to consumer clusters

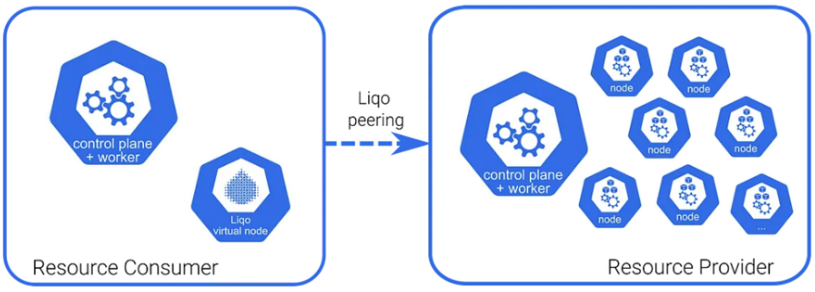


Figure 3.20: Liqo peering between two clusters

The peering process itself involves three key steps:

1. **Authentication:** Clusters validate each other's identities to ensure security and trust.
2. **Networking:** Clusters discover each other's IP ranges and configure Network Address Translation (NAT) rules to establish efficient communication channels.
3. **Resource Sharing:** Clusters negotiate the quantity and type of resources they wish to exchange, enabling the core functionality of resource sharing.

Users can manually add clusters by specifying a specific peering command which can be automatically generated, together with the associated authentication token, from the command line of the remote cluster. After the peering, each cluster keeps maintaining complete control over the resources it shares and the entities it interacts with. It is important to note that the remote cluster can always decide to limit the percentage of its resources that can be actually granted. The final result of the peering process is the creation, from the local cluster point of view, of a **"virtual node"**, that from now on will act as a normal worker node where to schedule pods on, except for the fact that these pods will be actually scheduled on the remote peered cluster. This all is possible thanks to a particular component called "virtual kubelet".

This peering process enables:

- **Dynamic Workload Placement:** thanks to Liqo “reflection” mechanism, the peering allows you to offload workloads (pods) to a foreign cluster based on predefined criteria. This fosters a dynamic and elastic environment where workloads can be automatically distributed across available resources in different clusters.

- **Transparent Service Consumption:** Services deployed in one cluster (local cluster) can be seamlessly consumed by applications running in another peered cluster (foreign cluster). This eliminates the need to redeploy services across all clusters, simplifying application management and promoting an efficient and transparent resource utilization

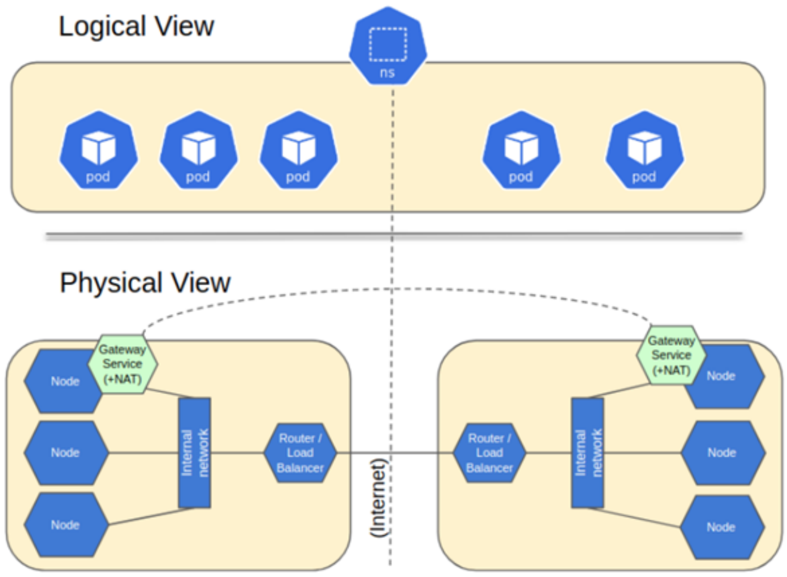


Figure 3.21: Liqo peering is transparent to the local cluster logical point of view

The peering operation also leads to the creation of a "**ForeignCluster**" resource within the local cluster. This resource acts as a representation of the remote cluster, holding vital information for the established connection. As already described, this process is asymmetric and thus allows for maximum flexibility. Bidirectional peerings combinations are supported as well. For example, the same cluster can play the role of provider and consumer in multiple peerings.

3.4.2 Namespace Offloading

Kubernetes resources created in one cluster become accessible to pods running in the peered clusters. This is achieved through another fundamental operation called "namespace offloading."

Liqo namespace offloading functionality takes multi-cluster management a step further. It allows you to designate specific namespaces within your local cluster for offloading pods to a foreign cluster, freeing up valuable resources in your local cluster for more demanding workloads. This capability enables:

- **Resource Optimization:** by offloading namespaces, Liqo allows you to optimize resource utilization across your entire Kubernetes infrastructure
- **Cost Efficiency:** efficient resource utilization translates to cost savings, especially when leveraging cloud-based Kubernetes environments with pay-as-you-go models

When a resource is deployed within the namespace designated for offloading, Liqo creates a "shadow" copy of that resource in the remote cluster. Pods in the remote cluster transparently interact with this shadow copy, which then forwards communication to the original service in the source cluster. This seamless mechanism extends service reach across the whole **single virtual cluster** thus obtained.

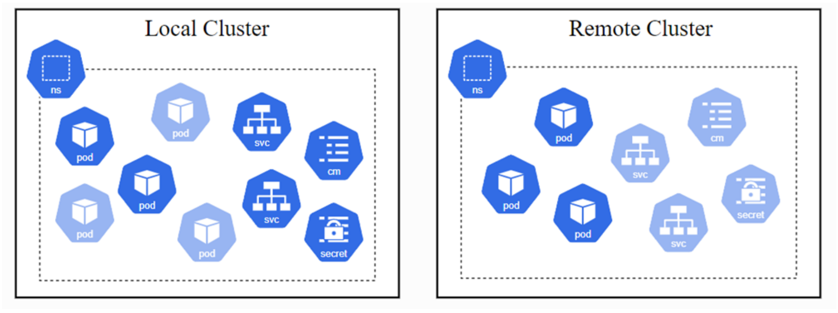


Figure 3.22: offloaded resources actually run on the remote foreign cluster

Liqo, with its innovative approach to multi-cluster networking and resource sharing, offers an excellent alternative to traditional solutions. By fostering a dynamic and resource-efficient environment, Liqo empowers organizations to unlock the full potential of their Kubernetes deployments and embrace the possibilities of multi-cluster data centers.

Chapter 4

Cloud offloading architecture

This chapter presents a technical examination of the problem previously introduced, proposing different design ideas. The overall architecture of a possible Cloud offloading system will be presented in details.

4.1 Overview

As far as the general architecture of this Cloud Continuum system, when deploying a distributed application it is important to focus on how to achieve the communication between the local (edge) and the remote (Cloud) components . Considering the simplest Cloud scenario, these components are divided onto at least two different Kubernetes clusters: in this use case there is a main listener ROS node on the "local cluster" (i.e. the robot itself) and a custom publisher ROS node on the "remote cluster". The multi-cluster topology will be managed using Liqo to finally get a seamless-switching working mode for the application. This configuration aims at relieving the robot from some cpu-intensive tasks, charging some other process on a data center remote machine, where computational resources are supposed to be more suitable for this kind of task, so that it will be possible to work at higher frequencies.

Liqo is an excellent tool to achieve a transparent switch from the main local cluster to the one where the pod with the remote copy of the publisher node will be deployed. With Liqo all the nodes are really part of a resource continuum, perfect integrated with Kubernetes and easy to manage: once the peerings have been established and the namespace offloading policy set, it is immediately evident the result of a fluid apparent single-cluster infrastructure. Offloading a service to a remote node (e.g. another robot, a public Cloud server, or an edge server) requires a certain number of operations such as "hooking" the remote Cloud and orchestrating the service transparently between the local Cloud and the remote one. Liqo greatly simplifies this operation as it allows to create a full "virtual cluster" that aggregates both resources and services, creating a single virtual space. It therefore simplifies the orchestrator logic by delegating the responsibility of creating and managing the virtual space to this component.

This thesis takes advantage of the **turtlesim** ROS2 package as software prototype of a physical robot navigation system. It has been then customized with the help of LINKS Foundation for this purpose. Turtlesim is a lightweight simulator for learning ROS, very helpful to understand how to interact with the main ROS tools, topics and services concepts, as well as for testing purposes.

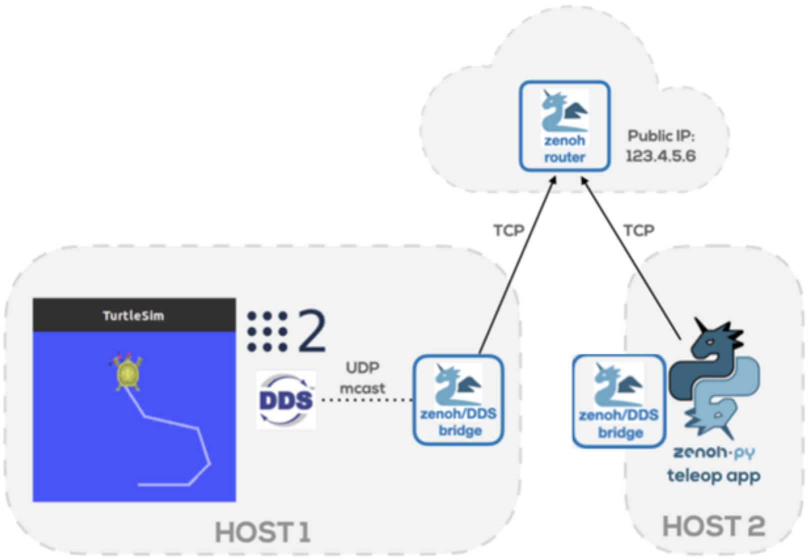


Figure 4.1: basic configuration for turtlesim<–>teleop remote communication with zenoh

Considering a basic environment in which the turtlesim and its "teleop" publisher are directly mutually reachable (i.e. in the same subnet), they would normally communicate by means of UDP multicast, since by default ROS2 uses DDS as its middleware. However, when those nodes are part of a distributed system (like in figure 4.1) of course internet routers do not let multicast traffic pass through. Moreover, assuming this ROS nodes are deployed as containers in as many pods of a Kubernetes cluster, another problem has to be faced: K8s CNIs don't usually come with multicast support, so the only basic configuration which can allow containers to talk to each other directly is the one where they are coupled as "sidecars" of a main container in a single pod. There exists some specific technology that may be capable of enabling multicast traffic within the cluster or a at least at the namespace level (e.g., Weavenet, DiscoveryServer), but they won't be further explored in this thesis because of a set of limitations due to this particular use case, as it will be explained in the implementation section.

As in the figure above, the communication is possible thanks to **Zenoh**: this technology allows a much better performance comparing to basic DDS, drastically decreasing discovery traffic among nodes, and this is a strongly required feature for real-time applications [7]; it also automatically manages connection losses and it is totally transparent to ROS nodes. Finally, in order to overcome the multicast limitation, each distributed node has to be coupled with a zenoh-bridge, on its turn connected to a Zenoh router realizing the overall inter-cluster communication. Multiple configurations are possible through zenoh [6]: the one selected for this work presents a single zenoh-router on the remote cluster, because for a real scalable application it should be only a client (robot) responsibility to contact the server (zenoh-router) anytime its necessary (e.g., after a disconnection); besides a potentially moving robot could not have a static IP address.

4.2 Real-Time Seamless switching

The goal now is to try to figure out the complete scenario and find the best way to enable the switch from the local cluster to the remote cluster when the network connection is stable: this operation must be efficient and absolutely transparent to the robot, whenever the Cloud offloading of a certain task is requested and possible. In particular, anytime the robot wants to offload the "navigation" task, there must be a way to exclude the local publisher in favor of the remote one while, when the network link drops or the set of network parameters are overall not to be considered good enough, the system has to switch again to the local configuration as a backup service. This all brings to define a **LOCAL_MODE** and an **OFFLOADING_MODE** as the two operation modalities for the system.

What it has to be explored is how to switch seamlessly from one mode to the other:

- one possibility is to imagine an architecture providing a way to *power off* the local publisher component when in offloading mode (and vice-versa)
- a reasonable alternative seems to be the *isolation* of the local cluster when in local mode from a network point of view

Before discussing the chosen solution, it is important now to have a look at the different configurations both on the edge and in the Cloud. Crownlabs has been used as a working environment for the virtual machines. It is a remote desktop technology hosted in Politecnico di Torino providing immediate access to personal computing labs, allowing to select the most suitable virtual machine for one's work with some already-installed software and an easy-to-manage VMs lifecycle.

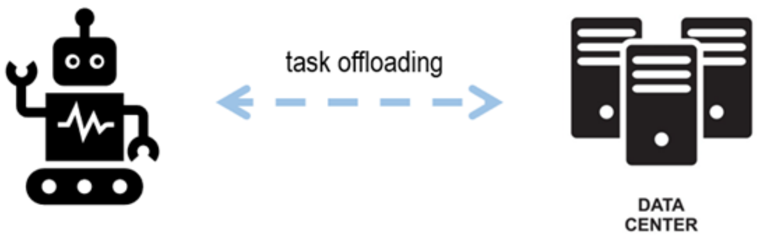


Figure 4.2: Cloud offloading interactions

4.2.1 Local cluster

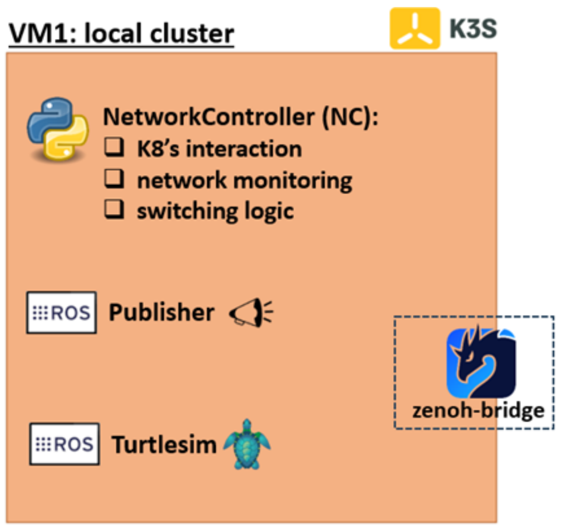


Figure 4.3: local cluster configuration

The local VM has to host a single-node K3s cluster with four main components, as described in the table below:

Component	Function
Turtlesim	It represents the “robot” itself, simulating a turtle waiting for navigation commands inside a window
Turtlesim_lifecycle_publisher (TLP)	It acts as the local version of a remote control for the turtlesim, implemented as a ROS custom publisher associated with the turtle specific movements topic
NetworkController (NC)	It is the most important node and it carries the main switching logic
Zenoh-Bridge	The Zenoh local component acting as a bridge for every message captured in its ROS domain

The local cluster reflects the normal functioning scheme of a robotic navigation system: a turtle robot waits for some commands to move around inside its playground. When running in local_mode the turtle must always be able to survive in its independent system, that's why there is also a local version of the publisher. On the contrary, when in offloading_mode it will be the Zenoh-bridge to enable a communication channel with the outside.

The **Network Controller (NC)** is the brain of the switching operation and incorporates the glue logic of the two worlds: Kubernetes and ROS. This is a ROS node which, regardless of its implementation, must present 3 specific modules which respectively concern:

- interaction with a Kubernetes cluster
- network conditions monitoring
- switching logic

Interacting with Kubernetes is necessary to order the cluster API-server a sort of isolation of the local cluster and this means having some K8s or Liqo resource created or modified on-demand. To this purpose Kubernetes documentation already provides a way for API direct access or a set of client libraries for some programming languages which appropriately manage kubeconfig (i.e., K8s configuration files enabling the communication with api server) and permissions issues for the user, incorporating the corresponding REST APIs, as it was described in the Background section.

At the same time the switching can be requested only when all the relative network parameters are suitable for the operation, that requires monitoring the network state or even some prediction/correction logic that would perfectly suit this use case (e.g., by using a Kalman filter).

4.2.2 Remote cluster

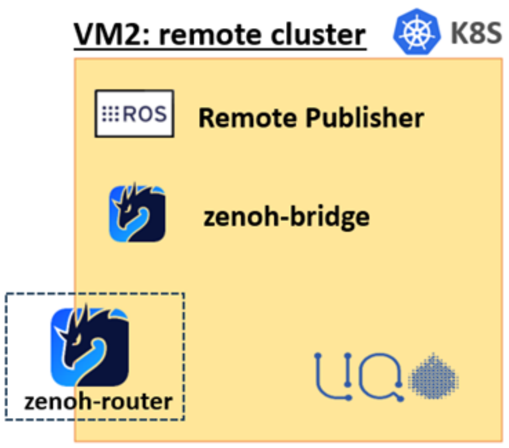


Figure 4.4: remote cluster configuration

The remote VM has to host a K8s cluster with three components, as described in the table below:

Component	**Function**
Remote-publisher	It acts as the remote version of a remote control for the turtlesim, implemented as ROS custom publisher associated with the turtle specific movements topic.
Zenoh-Bridge	The zenoh remote component acting as a bridge for every message captured in its ROS domain
Zenoh-Router	Zenoh component that enables the inter-cluster communication by means of the zenoh-bridges

A generic remote cluster is "enabled" when chosen as Liqo remote-peer and when it is possible to make the switch, according to the decision logic of the NC which monitors the network condition. It is the Zenoh-router the one component which a zenoh-bridge can "attach to" and send every message that the turtlesim and the remote-publisher need to exchange.

In figure 4.5 it is shown the remote configuration from the Kubernetes point of view. The main deployment (on the right) with the offloaded application task has the zenoh-*bridge* as its sidecar and talks with it by means of multicast. On the other hand, there is the zenoh-*router* deployment (on the left) exposed internally by a ClusterIP service in order to make it reachable from the zenoh-bridge, and externally by a LoadBalancer service to get a public IP reachable from the outside. This IP will be used by Liqo only in the peering phase, then intra-cluster traffic will take advantage of the internal VPN.

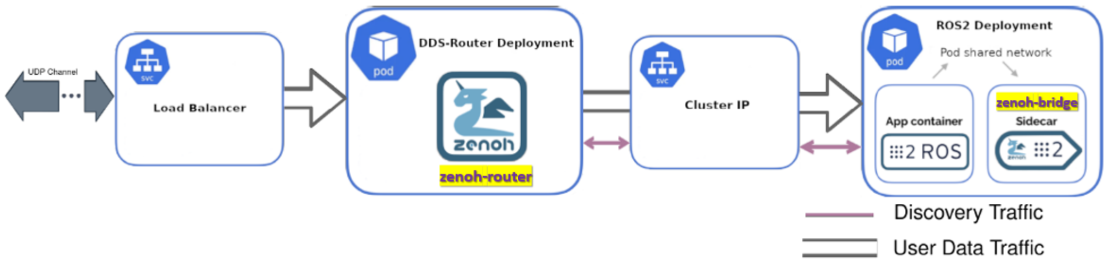


Figure 4.5: Cloud side Kubernetes configuration

The most generic scenario in figure 4.6 finally suggests how the main cluster may potentially have more than one Liqo peer where to offload its heavy tasks. However, this particularly complex configuration is not treated here as this work focuses on deeply exploring a single-peer scenario with one remote cluster only. In this case the local (robot) K3s cluster is the one that acts as main cluster exploiting the remote peer resources.

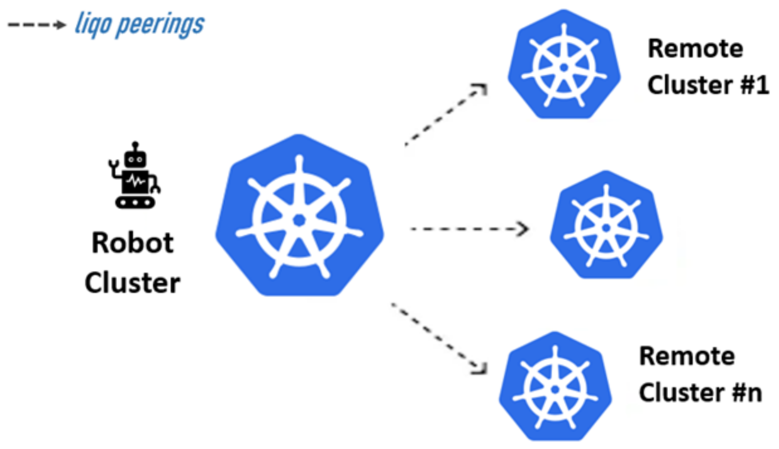


Figure 4.6: potential scenario with multiple Liqo-peers

Chapter 5

Implementation

In order to put into practice such a distributed offloading system this chapter will go deep into the actual dynamics of some possible solutions, providing a comprehensive understanding of all the roads travelled, each one with its own practical implications and challenges, component description and the overall system behavior.

5.1 Affinity, migration and switching

What this thesis aimed at from the very beginning was to prove it is possible using Liqo as a tool to achieve a transparent switch from local to remote clusters, in order to efficiently perform Cloud offloading of specific tasks.

As described in the background section, once the peering with an external cluster is done, one of the main features of Liqo is that of migrating workloads, which is possible to combine with different namespace offloading policies. There are three kinds of policies for the offloading: Local, Remote and LocalAndRemote. Considering such Liqo potential to manage where pods have to be scheduled across the peers' cluster nodes, it is easy to figure out the possibility of migrating any pods of a generic application just playing with nodes label selectors.

Labels are key/value pairs that are attached to objects such as Pods and Nodes. They are intended to be used for identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Labels can be used to organize and select subsets of objects, attaching to them at creation time and then subsequently added or modified at any time. Each object can have a set of key/value labels defined and each key must be unique for a given object.

The concept of **affinity** comes now in handy: through the labels attributes Kubernetes provides a flexible and functional way that is seems to be exactly the one feature to exploit for the offloading purpose. Node affinity is a set of rules the Kubernetes scheduler uses to determine where a pod can be placed: pods with a certain label can be scheduled on a node to which is assigned the same label (see figure 5.1); but there are many other possible combinations of labels and operators which correspond to different configurations. This all means that, while a pod is running on node, if that node affinity rule is changed, that pod could be moved away on another node if available, or stay in a "pending" state if it is not. Note: when talking about pod and node affinity, since the naming may be misleading, it is important to realize that both features are meant from the pod's perspective. Node affinity attracts pods to nodes, and pod affinity attracts pods to pods.

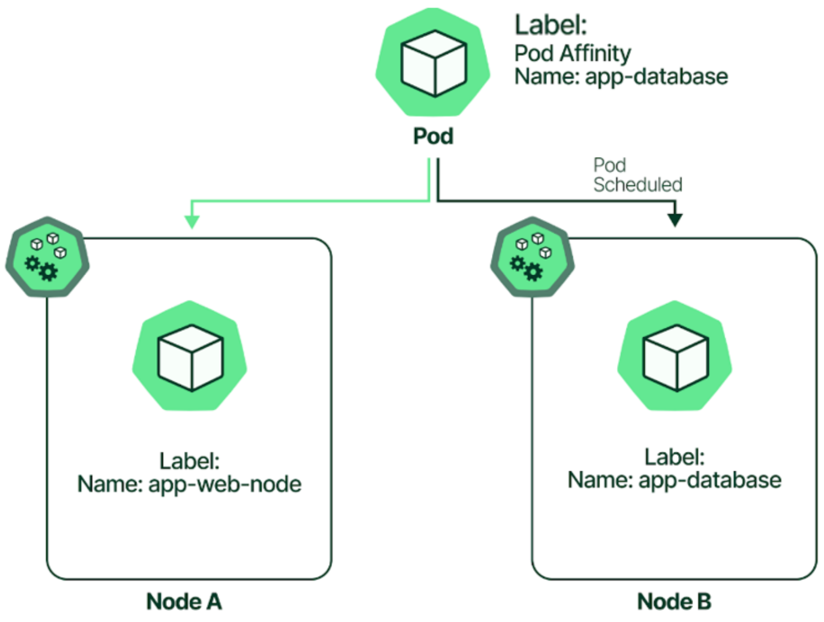


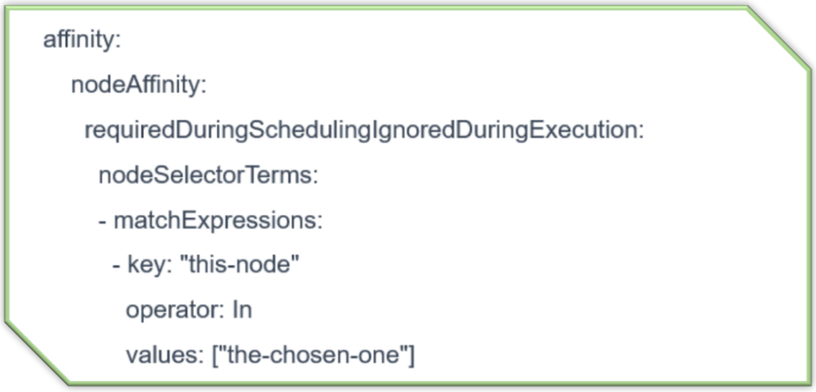
Figure 5.1: how node affinity works [4]

As a consequence of a generic affinity set of rules, it is clear now that turning on and off a service, previously described as a possible implementation for the switching algorithm, can be implemented as an operation of service "displacement". Therefore, it is important now to discuss the differences between service "**migration**" and task "**switching**".

The idea with Liqo is to exploit the possible node/pod affinity Kubernetes configurations in order to force the scheduler towards a specific virtual node, of course reacting to the NetworkController decisions based on the network conditions, in real-time. That would be more a form of migration.

However, migration is not a simple concept and some problems have to be discussed:

1. Lack of support for live-update of affinity labels: the main type of node affinity provides the value “requiredDuringSchedulingIgnoredDuringExecution” in its configuration file, which forces the scheduler to apply this rule at scheduling time (i.e., when the pod is not running yet) but ignores every changing in pod labels once it is running (see figure 5.2). The opposite affinity value should have instead a value ending with “RequiredDuringExecution”. The problem is that, at the time of writing, this emerges as a feature announced but not yet implemented in Kubernetes. A possible workaround for this problem could be not to rely on node affinity but rather to use “taints and tolerations”. While node affinity is a property of pods that attracts them to a set of nodes (either as a preference or a hard requirement), taints are applied to nodes and are the opposite of node affinity: a taint repel pods from being scheduled to a specific node. Tolerations are applied to pods and allow the scheduler to schedule pods with matching taints. Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. This way all constraints can now be expressed as a sort of node anti-affinity rule. Moreover, taints come with some parameters to configure such as: “key”, “operator” and “effect”. If the effect is set to “NoExecute” value the taint will finally affect pods that are already running on the relative node, solving the affinity labels "rolling update" problem.



```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: "this-node"
          operator: In
          values: ["the-chosen-one"]
```

Figure 5.2: example of node affinity specification in a pod yaml file

2. **Rescheduling latency**: given that this is how affinity works, real-time offloading reactions to network conditions changes actually translates into a periodic rescheduling operation which would absolutely be a problem if too frequent, since it takes a lot of time. This high amount of time would prove to

be too much in most application cases, and generally it is not even a parameter under the developer control, especially on the remote cluster side when part of a public Cloud. The duration of a deployment rolling update depends on several factors:

(a) Number of pods: the scheduler needs to find suitable nodes for each new pod during the update. With a larger number of pods, scheduling can take longer.

(b) Pod startup time: the scheduler waits for new pods to be running healthy before terminating old ones. If pods have a long startup time, the update takes longer.

(c) Available resources: the scheduler searches for nodes with enough resources to accommodate new pods. Limited resources can lead to slower scheduling.

Finally, it is clearly important to focus on the conditions of the “repatriation” of the microservices. Indeed, the first thing to monitor is the network condition, the state of the link and the wireless signal quality, but there is a big problem to cope with, and that is about how critical the task to offload actually is. Real-time applications severely need stable low latencies, first of all as a primary safety requirement, but also as the minimum functional service conditions that must be satisfied. In particular a ROS navigation task requires a lot of computational power because of the high frequency at which sensor data are emitted (e.g., to detect obstacles) and then processed: the overall **system reaction time** is a crucial parameter so it directly affects implementation choices.

At this point it was evident that this affinity-based solution it is not the one that best suits this thesis use case: it is necessary to design another one providing a transition towards a **temporary "local backup" configuration** (i.e., the one previously proposed as “local cluster”) before the next switch to the remote copy of that service laying on a Liqo virtual node, otherwise the time required for rescheduling all pods onto another cluster would make such switching algorithm anything but seamless. In conclusion, migration is not the purpose of this work although it is definitely supported by Liqo. Such "displacing" operation has certainly the advantage of being able to rely on a single copy and configuration of the service, making the offloading algorithm behave as a choice of the best machine onto move a service as-is at each moment, but it is probably not the right choice for guaranteeing no downtimes for an overall real-time critical application.

Regarding the architectural point of view, from now on there will be a set of manual operations to be performed a priori at the same orchestration level, in addition to the creation of the local cluster deployment:

- creation of the two Kubernetes cluster
- Liqo installation, peering and namespace offloading
- creation of the remote deployment on Liqo offloaded namespace, as a consequence of the switching concept just described, avoiding additional scheduling delays (this would not be needed in case of migration)

5.2 Three different switching strategies

Once established it is not necessary to move any node and assumed the simplest scenario with only two Liqo-peers (one local and one remote), the objective is to develop a method for managing the lifecycle of both the instances of the service to offload, corresponding to the two publisher nodes for the turtlesim ROS application. This method should guarantee a seamless switch between the local and the remote version of the publisher and no downtimes for the turtlesim (i.e., the turtle never stopping), at the same time reducing resource consumption on the robot and achieving better performances on the Cloud.

Reviewing the previous concepts of OFFLOADING_MODE and LOCAL_MODE:
• in local mode, the robot performs all the tasks locally listening to the local publisher until network conditions will be good for switching
• in offloading mode, the turtlesim will instead listen to the remote publisher node running on the remote cluster, as long as conditions remain good enough

The main possible strategies to implement this mechanism are three and will be further explored in the next sections.

5.2.1 ROS2 Lifecycle solution

The transition from the local node to the remote one can be seen as an alternation of switching off and on operations performed respectively on the local and on the remote publisher.

ROS2 introduced the concept of managed nodes, also called **Lifecycle Nodes**. Managed nodes contain a state machine with a set of predefined states. These states can be changed by invoking a transition id which indicates the succeeding consecutive state (see figure 5.3). Actually, lifecycle nodes abstraction is implemented via a ROS2 service, so the invocation of a transition corresponds to a service call which sends the service-server a message with the appropriate format. For a more verbose explanation on the applied state machine, please refer to the ROS2 background section which provides an in-detail discussion about each state and transition.

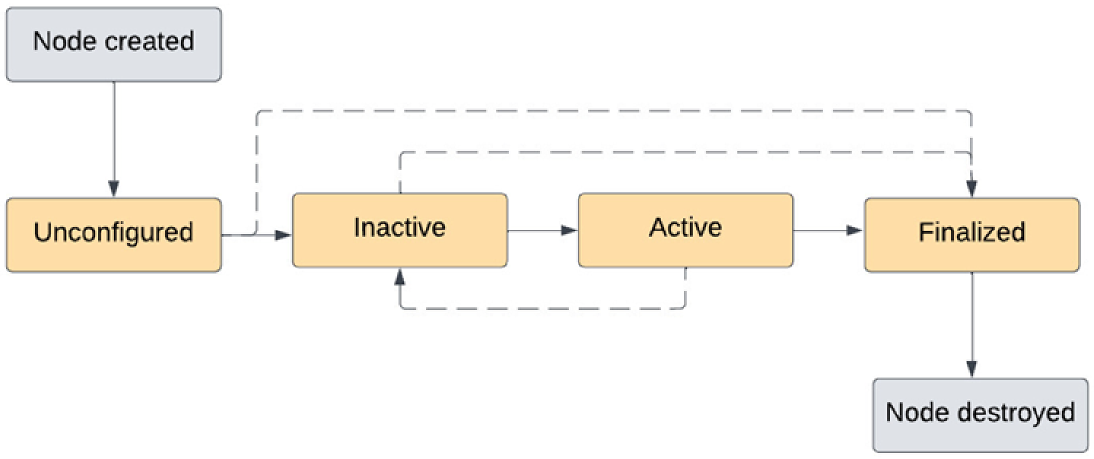


Figure 5.3: primary states of a lifecycle node

The most important primary states and transitions are therefore the ones that allow to go from the “inactive” state to the “active” one. No messages are getting published while a publisher node is still not active and vice versa.

The Network Controller is supposed to use such lifecycle abstraction in order to activate and deactivate the right nodes in their respective clusters, according to the system operating mode at each moment. Local and remote publishers are lifecycle nodes acting as servers waiting for the notifications that the NC, which behaves as a client for both services, sends them at every state change (see figure 5.4).

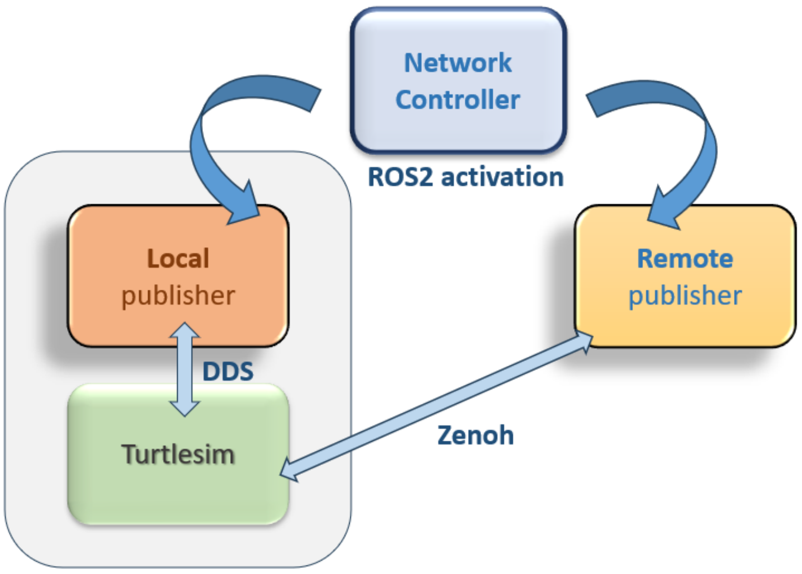


Figure 5.4: lifecycle-only solution

As far as the technical implementations choices, the main language used for ROS2 nodes is typically C++. However, for this work it would be convenient to write the NetworkController node in Python, because of its need for an internal interaction module with a Kubernetes cluster which, as it will be further described, is far convenient to implement by means of the existent Python client libraries for K8s. Actually, this specifc module is not required for this pure ROS2 Lifecycle version of NC for the switching algorithm but, since part of the NC code can be reused and extended for the two other versions, the final language choice has been Python. Further details about coding with Kubernetes will be given in the next sections.

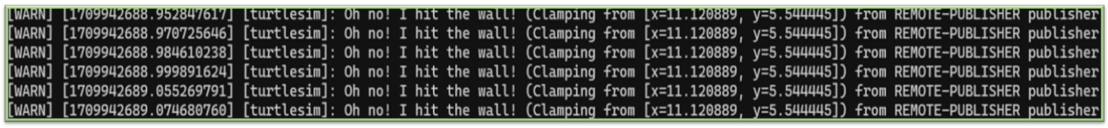
Regarding the **organization of software classes,** the ROS2 packages are so called: network_controller (NC), turtlesim and turtlesim_lifecycle_publisher (TLP). For the remote publisher the same node of the local publisher has been used, taking advantage of the "remapping" ROS feature: all ROS nodes take a set of arguments that allow various properties to be reconfigured, like topics, parameters, and services, each identified by Names [13]. Names are hard coded in ROS nodes, but they can be changed at runtime through remapping. Without remapping, every instance of a node would require changes in code. All ROS-specific arguments have to be specified after a "–ros-args" flag, e.g.:

ros2 run <my_package> <node_executable> –ros-args -r <old name>:=<new name>

Note that these remappings are "static", in that they apply for the lifetime of the node. “Dynamic” remapping of names after nodes have been started is not yet supported.

The **remapping** feature also allows to solve a problem that occurs with such configuration with two identical publisher lifecycle nodes (and their identical service names): the two zenoh-bridges automatically advertise the same redundant information. In a ROS environment it is in fact very important to maintain distinct names: not doing this may have a zenoh-bridge occasionally forwarding captured messages, in particular service calls like the ones sent by the NewtorkController, causing spurious misbehaviors. Having the possibility to set different names through the remapping function has helped to solve this problem without any code changing.

In addition to all the nodes described so far, another package has been created with the name of turtlesim_msgs. Its function is that to provide a custom topic through which the turtlesim and the publisher can communicate. Normally they would use the traditional cmd_vel **topic for movement directives**, but with the help of LINKS Foundation this has been adapted to what best suited the visualization of a final demo. On Kubernetes containers in fact, it is not very easy achieving the same visual result of the usual turtlesim movements on its graphic interface, so for test the turtlesim is running in headless mode. Headless mode means the container is running in the background without any foreground elements visible as there is no screen or console. Therefore, the typical way to check the application is running is by attaching that container a shell (i.e., by the kubectl "exec" command). Another possible choice is to watch the pod containers logs from the outside and follow the node updates. In particular, the turtlesim is by default not particularly verbose when simply “walking” around the playground, instead it is used to send constant information of its position when it hits the window walls: this data corresponds to the kind of print on screen which can been modified in the code and extended with the information of the current publisher which in fact, depending on the operating mode, will be now either “from Local-Publisher” or “from Remote-Publisher” (figure 5.5).



```
[WARN] [1709942688.952847617] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.120889, y=5.544445]) from REMOTE-PUBLISHER publisher
[WARN] [1709942688.970725646] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.120889, y=5.544445]) from REMOTE-PUBLISHER publisher
[WARN] [1709942688.984610238] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.120889, y=5.544445]) from REMOTE-PUBLISHER publisher
[WARN] [1709942688.999891624] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.120889, y=5.544445]) from REMOTE-PUBLISHER publisher
[WARN] [1709942689.055269791] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.120889, y=5.544445]) from REMOTE-PUBLISHER publisher
[WARN] [1709942689.074680760] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.120889, y=5.544445]) from REMOTE-PUBLISHER publisher
```

Figure 5.5: turtlesim example of clamping log messages

ROS2 Service interaction

The typical interaction with a ROS2 lifecycle service when using the command line interface has this format: ros2 service call <node_name/service_name > <service_msg_structure> <msg>

This command actually hides a set of operations that should be always done when dealing with services from the code of a ROS node service client. In order they are:

1. creating a lifecycle service client
2. waiting until the service is up
3. preparing a callback for the asynchronous service-call
4. sending activation/deactivation request

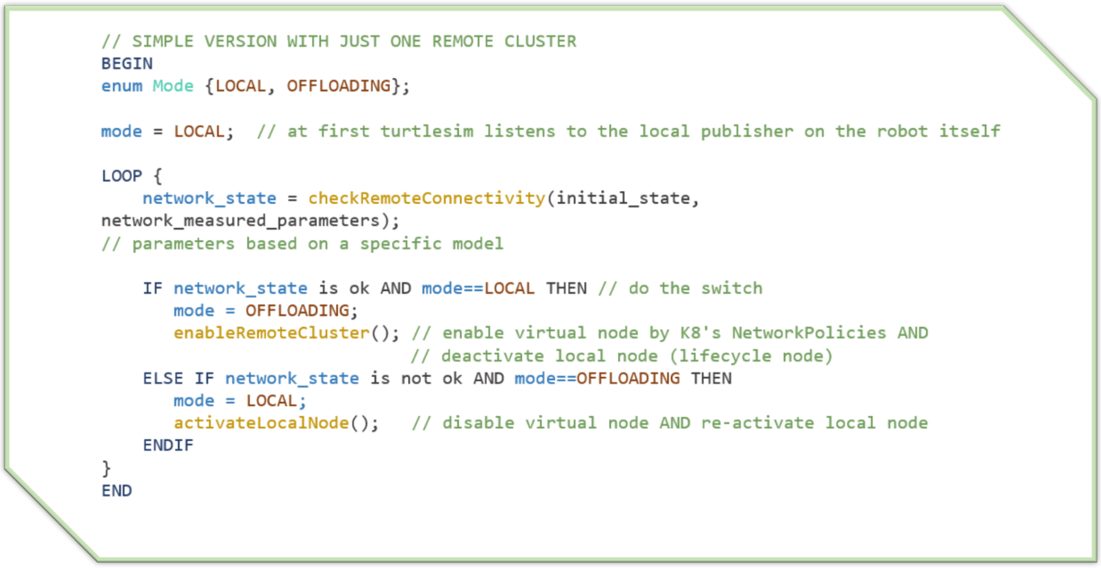
Understanding in depth **how ROS services actually work** proved to be a key point in designing the local Kubernetes cluster. Services in fact differ from the usual publish-subscribe communication paradigm among ROS nodes, implementing a call-and-response model: while topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client. The point now was to verify how service-client and service-server actually behave when they cannot talk directly in multicast, i.e., they are deployed on different pods. ROS discovery traffic is the one sent on the "ros_discovery_info" topic, including topics, nodes and services list, normally needful for those service interaction. By default, the zenoh-bridge does not route DDS discovery traffic to the other remote bridges, but the advertisement can be enabled by adding the "–fwd-discovery flag". However, though in order to contact a service a node must know its name, all the tests done in this direction proved that it is not enough in such Kubernetes environment: with the service name it is possible to send the request but then this fails or even doesn't receive any answer. The reason behind this behavior needs to be further investigated. It is easy to believe that service servers must also be reachable through multicast, and that depends on the internal management by "rclpy" library used undereneath. Those requests in fact are likely to be made in unicast or multicast depending on various factors including the initial configuration of the underlying ROS-middleware (multicast by default): the former would likely provide the use of the same source IP address of the multicast message (the one through which the server advertises its own service name the first time), the latter uses multicast for the services too. In order to explore this specific internal feature, it is possible to dig deep into the python rclpy library and discover first where in the client request (something like a rmw_transport_info structure) the eventual unicast IP of the server is specified, second replacing it with the ClusterIP associated with the target node to be activated (TLP in this case). It is certainly good for this thesis purpose to avoid further modifying the NC code by getting your hands on rclpy library, as that code isn't simple and likely not managed in the same language of the node. On the other hand, solutions for indirectly enabling multicast in the cluster or for managing it in a centralized manner (e.g., DiscoveryServer) won't be further explored in this thesis.

Nodes positioning
By analyzing the relationship between each pair of nodes based on their **connectivity requirement**:

- turtlesim and zenoh-bridge must use multicast for direct communication
- **NC** and turtlesim_lifecycle_publisher must be close to use multicast for ROS service call-and-response model
- turtlesim and its local publisher (**TLP**) need to talk to each other directly when switching to LOCAL_MODE

The conclusion is therefore to place all the nodes of the local cluster in such a way that they are "close" enough for communicating through multicast: from now on NetworkController, turtlesim_lifecycle_publisher, and zenoh-bridge will be considered as containers, all part of the same global pod, working as **sidecars** of the turtlesim main container.

The last module of the Network Controller concerns **network conditions monitoring**. The switching operation must only be enabled under a certain condition and when a set of appropriate network parameters are considered suitable for such operation (see figure 5.6). A custom logic may be implemented for the purpose and in thesis a simple network check is done: every 30s the NC tries to trigger the switching, simulating the moment of the offloading conditions is verified. Then it executes a ping to the remote cluster address (received as an additional parameter for the node) in order to check internet connection is actually up and, if it is true, it allows the transition from LOCAL_MODE to OFFLOADING_MODE, and vice versa. This choice has been done for testing reasons in order to focus on the switching logic rather than on network examination. Further implementations may be certainly be provided, even basing on a Kalman filter fed with an appropriate model for performing some prediction&correction logic, according to what better suits the user's specific use case.



```
// SIMPLE VERSION WITH JUST ONE REMOTE CLUSTER
BEGIN
enum Mode {LOCAL, OFFLOADING};

mode = LOCAL;  // at first turtlesim listens to the local publisher on the robot itself

LOOP {
    network_state = checkRemoteConnectivity(initial_state,
network_measured_parameters);
// parameters based on a specific model

    IF network_state is ok AND mode==LOCAL THEN // do the switch
       mode = OFFLOADING;
       enableRemoteCluster(); // enable virtual node by K8's NetworkPolicies AND
                              // deactivate local node (lifecycle node)
    ELSE IF network_state is not ok AND mode==OFFLOADING THEN
       mode = LOCAL;
       activateLocalNode();   // disable virtual node AND re-activate local node
    ENDIF
}
END
```

Figure 5.6: Network Controller pseudo-code

5.2.2 NetworkPolicies solution

The opposite alternative to the pure lifecycle-activation for a switching algorithm is trying to isolate the turtlesim and the whole local cluster from a network point of view, anytime the LOCAL_MODE it is toggled, and vice versa. This could be done appealing to a typical Kubernetes resource called Network Policy. Such solution will completely exclude the intervention of ROS services, providing a more generic method for the switching algorithm which this way becomes independent from the specific application and makes the NC come with just two of the modules: network monitoring and Kubernetes cluster interaction.

Network Policies are a mechanism for controlling network traffic flow in Kubernetes clusters. They let you to define which of your pods are allowed to exchange network traffic and can be used to prevent apps from reaching each other over the network: this can help limiting the damage if one of the apps is compromised, but more generally they allow to ensure a specific traffic matrix among all pods, even across different namespaces, in order to implement the complex network logic of an application composed by several microservices. Each Network Policy targets a group of pods and can set both the ingress (incoming) and egress (outgoing) network endpoints those pods can communicate with. It is definitely possible to target specific pods but also namespaces or IP address blocks. Once again, the set of rules thus defined will apply to the specific targets through the Kubernetes labels mechanism e.g., pods matching a certain label are allowed to send/receive traffic from other pods matching their own different label. Further details are given in the relative background section.

The point in the choice of the Network Policies (a.k.a. netpol) in this use case is to exploit some firewall rules deciding when the turtlesim should listen only to the local publisher or to the remote one. The selected targets of this new set of rules are therefore the turtlesim pod on the local cluster on one side and the virtual node on the other side (see figure 5.7):

- when in local_mode, a network policy is applied denying all external traffic
- when in offloading_mode, a different network policy allows external traffic but disables the internal communication between the turtlesim and its local publisher

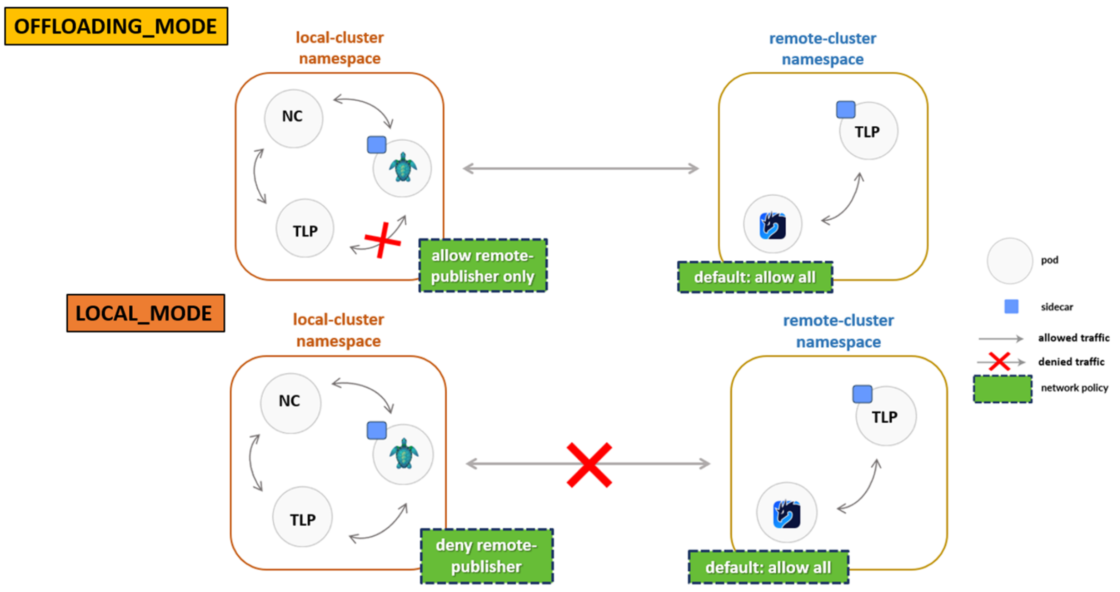


Figure 5.7: network policies solution

Applying the policy to the Liqo node is possible because the environment in which the remote nodes run is represented by the correspondent Liqo namespace previously offloaded: enabling traffic from the virtual node consists in applying a netpol targeting that whole Liqo namespace, and vice versa. The same result would be obtained by reasoning from the local cluster point of view and targeting instead the local cluster namespace, in fact: if A must not communicate with B the result is the same as if B must not communicate with A, because no one can talk and listen to the other when the netpol is applied. This second perspective is actually the only one that would work because of the lack of the **Liqo "reflection"** for this kind of resource. Dealing with network policies in fact is not very intuitive due to their “whitelisting” structure: you can only specify what traffic you want to allow and, once a single netpol is applied, the rest of the traffic will be forbidden. Moreover, referring to default K8s netpols, they are a namespaced resource: this means they must always be specified relating to the namespace in which selected pods are running. The fact Liqo lacks netpol reflection has thus the consequence of preventing the developer to “drop” that resource on the offloaded namespace, forcing to focus on the local cluster perspective. The possible motivations for Liqo not to support reflection of network policy could be potential introduction of unintended security implications, especially when remote clusters are created on public data centers not under the developer’s control. At the same time, reflecting network policies directly could lead to conflicts or unexpected behavior if not carefully managed. Liqo, whose development is ongoing, might therefore be aiming for a more flexible approach where network policies are defined and enforced independently on each cluster, primary focusing on enabling core resource sharing in the safest way.

There are two main possible approaches when dealing with periodical **updates of a Kubernetes resource**, and they are described here.

- Create-then-patch: the resource creation happens only the first time, then it is possible to apply a patch (partial of total) of the existent resource to reconfigure it. This approach is probably the one with the best performance in terms of time spent for contacting the API-server, asking it to update the resource without recreating it. It is although necessary to pay attention to the type of merging of the configuration fields that occurs during the patch
- Create-and-delete: after the first creation of the resource, this will be deleted when necessary, then recreated at the right moment as it was a different new resource. This could take more time, but sometimes it is the only choice, depending of the type of the resource and its purpose

In this use case the approach chosen is create-and-delete. Conceptually there are two different netpols, they are mutually-exclusive so the straightaway solution is that of deleting the first netpol before applying the second one. An attempt was made anyway to find an appropriate transformation patch, but the way Kubernetes normally executes that merging did not lead to the desired result. On the other hand, searching for two appropriate patches respectively to achieve faster operations for both the transitions was not possible as well, since these patches would have been “empty patches” trying to mock the absence of the netpol, and it turned out the Kubernetes client libraries do not accept the creation of such an empty netpol.

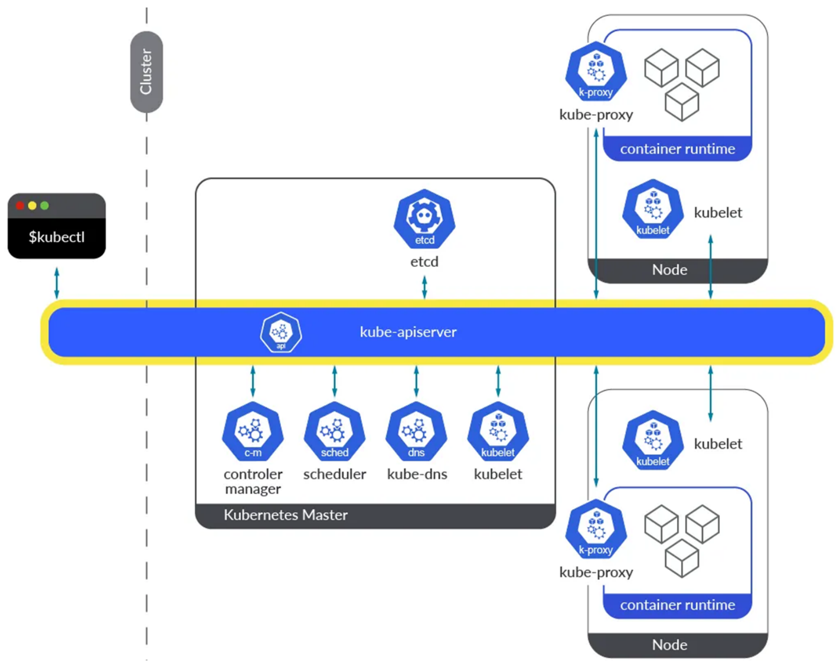


Figure 5.8: K8s API server works as frontend for internal and external request

When **accessing a cluster**, you need to know the location of the API server and have credentials to access it (see figure 5.8). Kubernetes comes with a good support for accessing clusters through its REST APIs, besides the usual one through the kubectl command-line tool. The other two possible methods are:

- direct access: uses kubectl in proxy-mode to locate and verify the API server
- client libraries: it is based on provided libraries for a subset of programming languages which hide the implementation of the REST APIs and automatically handles authentication

The most important question about accessing a cluster is how to do it from inside a pod. When you authenticate to the API server, you identify yourself as a particular user. The methods just described in fact work perfectly if used by the same user that installed Kubernetes, since kubectl automatically looks for the kubeconfig file in the $HOME/.kube directory that was set during the installation. However, this operation needs more technical considerations when a user starts a request from a deployed container. All the processes that run in a Pod in fact have an identity related to the specific service account they are associated to at creation time, which maps to a ServiceAccount object and identifies the user who will be finally available to authenticate. The complete authorization picture includes the concepts of Roles and RoleBindings deeply described in the respective background section. Considering the **NC Python script**, two main attempts have been done to achieve the interaction between the application and the API server, in particular:

1) by means of the python "subprocess" module, it is possible to run a command on a parallel process. In this case the kubectl "apply" command has been used for direct accessing the cluster and Network Policy resource periodic creation and deletion, according to the algorithm previously described. Roles and binding concerns have been appropriately addressed to guarantee this operation success

2) by means of the **Python Client Libraries for Kubernetes**: in this case the NetworkPolicies configuration can be directly incorporated in the NC script and there is no need for first creating new roles bound to the pod service account. It is very important to detect the correct set of library methods which suits the purpose: different resources require different methods, changing if they are namespaced/not-namespaced, custom/default, and basing on the specific operation to be done.

Since these are external libraries, it is necessary to include them within the script. One of the ways to do this is by updating the PYTHONPATH, a special environment variable that provides guidance to the Python interpreter about where to find various libraries and applications. Another way to solve such dependencies is by appending the path of the library using the "sys" module specific method sys.path.append(<k8s_library_path>). However, in this case there was no need to specify the path as the python pip package installer already put them in the right place for the script. The one thing to actually worry about was the **kubeconfig** file: as was said before some information are needed to access a cluster from a pod (i.e., from the NC script running in the pod) and that information is uniquely associated with the cluster. In order to obtain the proper credentials, the Dockerfile of the NC must be modified. It was necessary to: first install client libraries, copy the cluster kubeconfig from its location to the docker image workdir and assign it read/write permissions, then locate the specific file and line of code of the libraries where the attempt of retrieving a kubeconfig file is done, finally modify that path with the one selected on the docker image file system. The main drawback of this method is that you must provide a new appropriate kubeconfig file every time you change the machine you use, since the old cluster configurations, credentials, and API server locations won't work for the new one.

Choice of the CNI

Kubernetes doesn't come with a default networking support: a **Container Network Interface (CNI)** plugin must be specified at installation time to manage the cluster network and security capabilities. Many different CNI plugins exist from many different vendors. You must use a CNI plugin that is compatible with your cluster and that suits your needs, as some of them provide more sophisticated solutions. By default, K3s uses Flannel CNI, but unfortunately that plugin doesn't support network policies. For this work **Cilium** has been chosen because of its versatility and in particular for its wide support to networks policies at layers 3-7 for both ingress and egress (see figure 5.9).

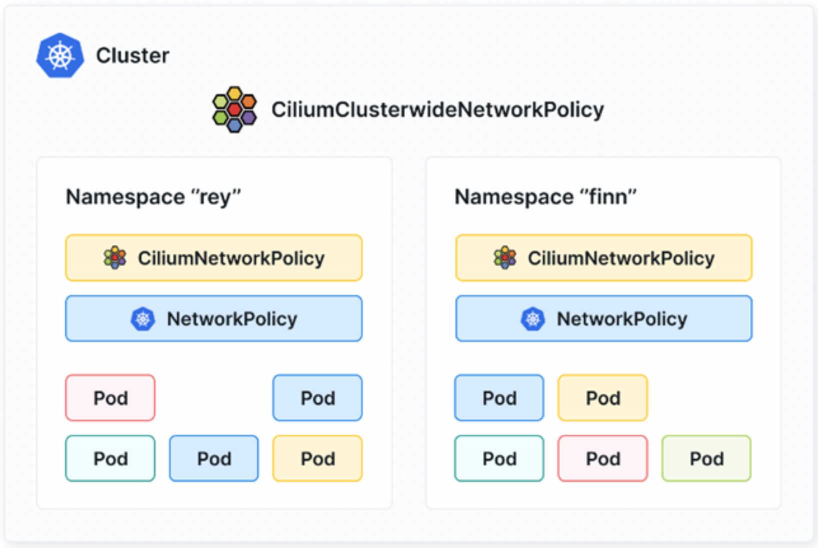


Figure 5.9: cilium network policies hierarchy

All Kubernetes objects including NetworkPolicy and CiliumNetworkPolicy can belong to a particular namespace or not. CiliumNetworkPolicy only allows to bind a policy restricted to a particular namespace. There can be situations where one wants to have a cluster-scoped effect of the policy, which can be done using a CiliumClusterwideNetworkPolicy custom resource.

Note: if you are installing Liqo on a cluster using the Cilium CNI, you must pay attention to the dedicated configuration section to avoid unwanted misconfigurations. In particular Cilium adds a taint to each node where no daemonset is scheduled, like the ones labelled as virtual nodes by Liqo. This taint therefore must be tolerated in order to have Liqo pods to be scheduled on the remote nodes[8].

At the end of the previous chapter, the way in which ROS services work and the connectivity requirements among the nodes brought to the conclusion that the simplest way to guarantee multicast traffic between each pair of nodes is to place all the nodes containers of the local cluster as sidecars for the turtlesim main container, composing a single "fat" pod. Now, with regards to this netpol-only solution, the design constraints for the cluster are of course the same and, despite how much powerful proved to be Cilium netpol resources, unfortunately this finally led to understand that this second solution cannot be implemented either, again considering not to search for any other possible workaround for the intra-cluster multicast problem. The point in fact is that NetworkPolicies rely on Kubernetes labels, but labels are actually assigned to pod, not to sidecars or single containers: it is not possible to define a network policy which targets a single container, since it will instead target the whole pod it is deployed into. To achieve such generic solution, more research and attempts will be needed in future to find a more suitable architecture and try to realize this implementation.

This thesis will now explore a third hybrid approach, combining the best of the first two approaches and testing the related performance.

5.2.3 The hybrid approach

The last implementation proposed for the switching algorithm completes the framework of the Network Controller as the heart of the task-offloading operation: now it incorporates all the modules, including again the one for the Lifecycle activation. The local cluster internal networking will be managed via ROS services, while the periodic creation of a Network Policy will handle traffic towards the remote cluster (see figure 5.10).

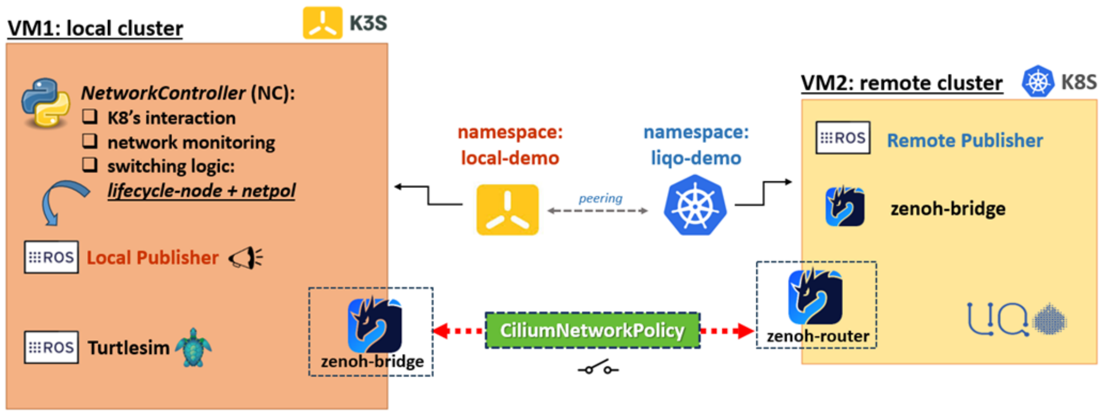


Figure 5.10: hybrid solution

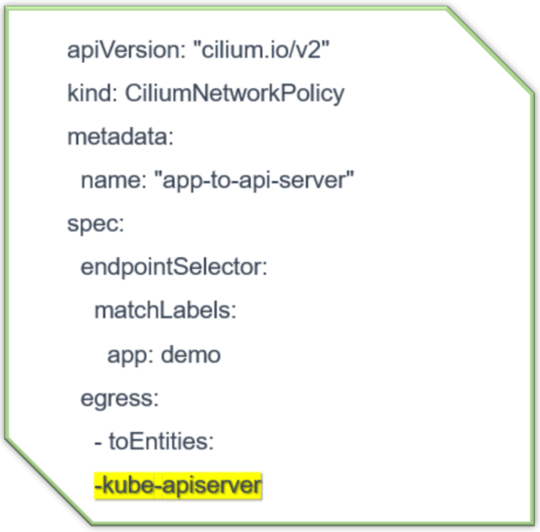
Local mode:

In this solution it is used only one CiliumNetworkPolicy and it targets the local namespace (called "local-demo"). It is applied in LOCAL_MODE to deny all external traffic. Differently from the netpol-only version, this behavior can now actually work, because the containers within a pod, though sharing a single label, can still communicate even when the applied netpol excludes the rest of the egress traffic. At the same time the Network Controller asks the local publisher to activate via a ROS2 lifecycle service call. The combination of the two operations results in the turtlesim receiving clamping messages ending with "from LocalPublisher".

Offloading mode:

The configuration has to be reversed for offloading the task, so: NC sends the deactivation request to the local publisher and, at the same time orders the deletion (from the local-demo namespace) of that CiliumNetworkPolicy previosuly created, with the aim of making the traffic entering again from Liqo namespace. The turtlesim now receives messages ending with “from RemotePublisher”.

The final, comprehensive test of this new scenario revealed a critical issue: once achieving the access to the cluster and after the NetworkController startup, the first netpol creation succeeded, but something consistently interrupted the process each time, from the second attempt onwards. This problem became the driver for effectively adopting a Cilium specific network policy resource, the CiliumNetworkPolicy. It turned out in fact that the default K8s type of Network Policy was actually limiting the connection with the **Kubernetes API server** itself: if a netpol trying to exclude some ingress/egress traffic accidentally involves that server too, there is no way to send any other request or get any response. The one thing missing was thus a particular configuration of the netpol which is able to treat special entities differently, like of course the API server. Such granularity of the firewall rules is not provided from the basic Kubernetes network policies, so it was essential at this point to rely on that particular type of netpol, specific to the Cilium CNI (see figure 5.11).



```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "app-to-api-server"
spec:
  endpointSelector:
    matchLabels:
      app: demo
  egress:
    - toEntities:
    -kube-apiserver
```

Figure 5.11: example netpol allowing all endpoints with the label app=demo to access the API server

The “fromEntities” field is used to describe the entities that *can* access the selected endpoints; similarly, “toEntities” is used to describe the ones that *can be* accessed by the selected endpoints. It is the “kube-apiserver” field that represents of course the Kubernetes API server from the Cilium point of view. This configuration has to be added to the default deny-all rule.

Here follows a chronological recap of the results obtained while deeply understanding how to make a policy as the ones needed for this use case, with multiple targets work simultaneously. A continuous connection to the API server has in fact to be guaranteed concurrently with the application pods excluding connectivity logic. Sometimes it can be confusing to understand the complete set of allowed traffic across multiple policy types. If close attention is not applied this may lead to unintended policy allow behavior. Network policies issues are mainly due to these reasons:

- reasoning at namespace level (e.g., by namespaced netpols like a CiliumNetworkPolicy) would be the simplest thing and could work, but it cannot be done directly in this use case because Liqo does not allow reflection of any netpol, so "dropping" any policies directly into the Liqo namespace is not supported
- at this point it is necessary to think about controlling the local namespace traffic inverting the point of view, but there are other empirical issues to take into account
- netpols in general can only work in terms of whitelisting
- the fact that no netpol can be managed in blacklisting brought to some attempts consisting in using "dummy" labels: if you want someone with label:myapp not to talk to someone else with label:other you can set a policy having label:myapp talking with some label:dummy; then, if no endpoint with “dummy” label actually exist, it will result in your app being isolated as you wanted
- Kubernetes native netpols are not powerful enough
- Cilium netpols (namespaced and not) presented a set of problems found in common with all the tested types of netpols, namely:
  - at the time of writing, all "self-defined" rules, i.e. those referring to your own namespace or labels, have no effect e.g. "a pod can talk only with itself" doesn’t work, and this is due in particular to a misbehavior of the “egress” rules; anytime an egress rule is specified it actually behaves as a

deny-all rule, preventing any possible logical workaround and forcing to rely mostly only on ingress rules

- on the other hand, the alternative is to stop thinking about the namespace level and think instead at the cluster level (via a CiliumClusterWideNetworkPolicy) by playing on individual pods labels with “global” meaning; but despite making all possible attempts it is clear that they present the same problems previously described (in particular it is limiting not to be able to intervene at the egress level)
- finally, the only netpol able to achieve a similar result of enforcing namespace boundaries while still allowing internal pod traffic among containers is the CiliumNetworkPolicy, as long as remembering to ensure traffic to/from the API server entity either. This particular feature is specifically reported in the Cilium documentation [15]

All these tests have been done with a simple local nginx server at first, then with the whole turtlesim deployments and Liqo at a later time. In the two cases the impact of the network policies on connectivity was different and needed different netpol configurations to obtain the desired result, since they have different communication paradigm (client-server vs publish-subscribe), but above all because of the presence of the Liqo virtual node specific network configuration.

The one thing in common is the **time factor.** It was possible to observe and understand that network policies require an arbitrary time for them to become effective in the cluster. This may depend on several factors: virtual machine computational resources and instant load, effective time for the system to translate a high-level network policy into a set of Linux iptables (or eBPF filtering logic in the Cilium case), time required for a new network rule to apply to an active connection, zenoh-bridge and router delays in adapting to the switch, ROS services request/response and node activation slowdowns or even all these factors at the same time. What actually happens at low-level of this architecture must be further explored as a future work. The evaluation section will show now explicit data about these tests.

Chapter 6

Experimental evaluation

In this chapter, the hybrid model implementation will be taken into examination through a series of tests. These tests were performed in the Crownlabs environment, where the local-cluster and the remote one were both implemented as single node clusters using respectively K3s and K8s, running on two different virtual machines. Such setup may not reflect real-world latencies as the VMs can reach each other directly (i.e., not through wireless and/or Internet), but certainly helps, through the collection of meaningful data, to come up with some final considerations about Network Policies and ROS Lifecycle activation. Finally, the chapter analyzes some state-of-the-art benchmarks with regards to the main tools exploited for this thesis, that are of course Liqo and Zenoh.

6.1 Measurements

The tests were performed at a high ROS level, in particular they aimed to observe if and when the messages from the different publishers were received by the turtlesim. It was therefore necessary to modify the NetworkController node where the main logic resides, by adding a Benchmarker class that acted as a listener for the same topic of the exchanged "turtlesim_msgs". At this point, to have an idea of what happened in each of the switching phases, a state machine was defined for such **benchmarks** (see Figure 6.1). It defines the moments in which to obtain timestamps that are significant for the purpose, i.e. at the beginning and at the end of the critical points of the algorithm, each associated with the proper operational mode.

LOCAL_MODE:
a) time it takes to activate a ROS node
b) time it takes for LocalPublisher messages to start after a lifecycle activation
c) time it takes to set a netpol
d) time it takes for RemotePublisher messages to stop after a new netpol
OFFLOADING_MODE:
e) time it takes to deactivate a ROS node
f) time it takes for LocalPublisher messages to stop after a lifecycle deactivation
g) time it takes to remove a netpol
h) time it takes for RemotePublisher messages to restart after the netpol removal

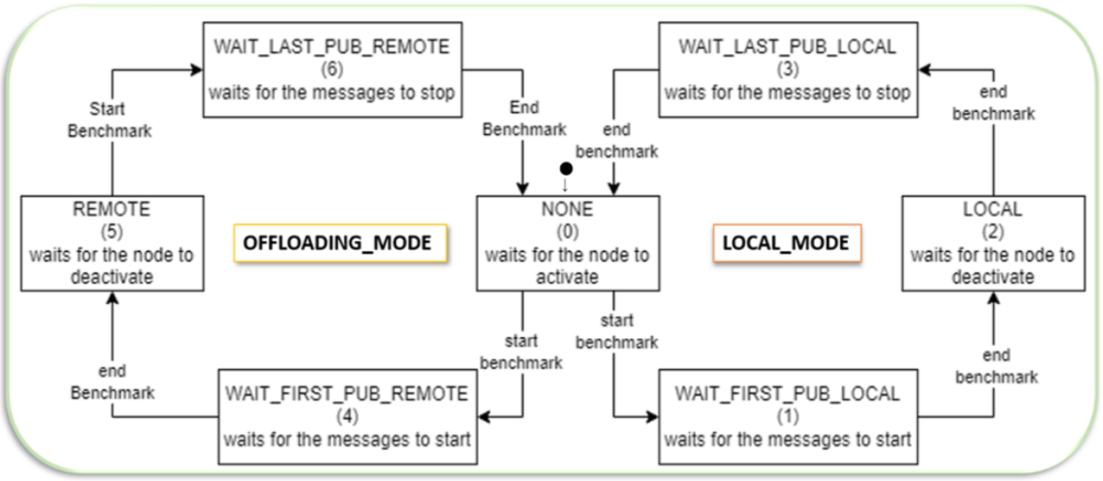


Figure 6.1: benchmark state machine

TLP publication frequency is set by default at 10.0 Hz (a message every 100ms). The "test_controller" (i.e. the NC modified) completes one lap at the frequency of 1Hz (1 cycle per second). However, it is important to understand how the *time*() functions used to get a single timestamp are independent on how fast the NC runs because, in this test case, they are not triggered according to the original network monitoring logic, rather they trigger anytime a message is written on the topic, finally allowing to compute the difference between each pair of time values, placed before and after a single operation.

The following histogram in figure 6.2 (local_mode) shows the amounts of time necessary for the point a) and c), that is comparing the Lifecyle Activation time of a node to the Network Policy time required for it to become effective and thus actually stopping the remote publisher to talk.

The local lifecycle activation, i.e., the time between the service call and the server related response, is very fast; instead, the time needed for the Network Policy to apply can be sometimes be larger.

The diagram reflects how the value remains around 20ms on average for both the operations.

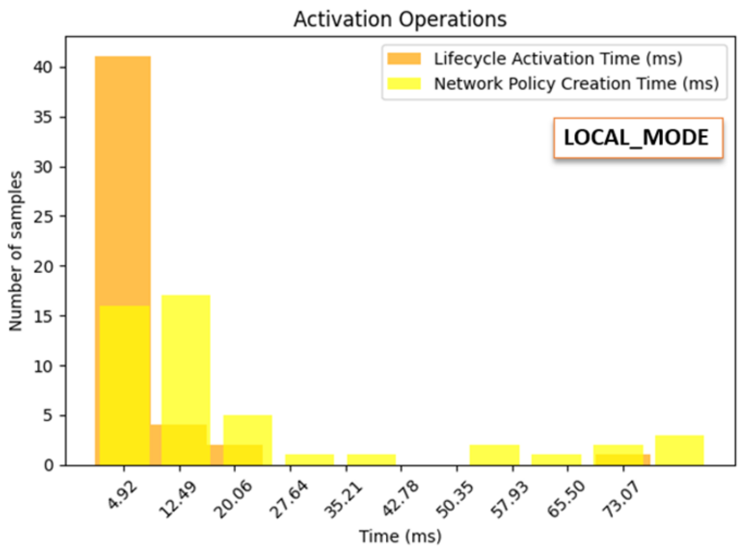


Figure 6.2: comparison of operations a) and c) for local_mode

As far as the next diagram in figure 6.3, it compares operations b) and h): once the ROS service request for activation arrives and the netpol is applied, it takes a certain amount of time for the respective effects to be visible.

[In the left chart]: the reaction time of the LocalPublisher is quite good.

[In the right chart]: this data actually has the most crucial meaning: the RemotePublisher sometimes cause the state machine to hang in this algorithm step, giving the impression that the remote publisher has stopped publishing. On the x axis in fact, this is the only case in which the scale has been set to seconds rather than milliseconds because of the higher average time needed. Actually, the TLP on the remote cluster never stops sending messages: once it has been activated in the first orchestration and setup phase there is no one asking it to turn off. Therefore, the reason of the problem must be sought deeper in the nature of the operation and among the actors involved. At the time of writing, the reason of this spurious behaviors is not yet clear. An attempt has been done forcing the Network Policy “grace period” to 0 i.e., the time between the moment that you issue a K8s resource request for termination and the effective deletion, but it had no different results. At the end of chapter 5, the **time** factor was already mentioned trying to figure out the responsible for such arbitrary delay. The first hypothesis of the machine overload (perhaps even worsen by the ongoing huge amount of collected test logs at those frequencies) no longer seems plausible after an additional try on a more powerful VM. It may be related to how the active connection, once being established by Zenoh, possibly cannot always immediately react to a new network rule: this could be Cilium specific or a more general problem, as it looks like it is independent from the moment in which the NetworkPolicy is applied and observable (its effect may come later). Further and stricter tests need to be done in order to collect other results and evidences.

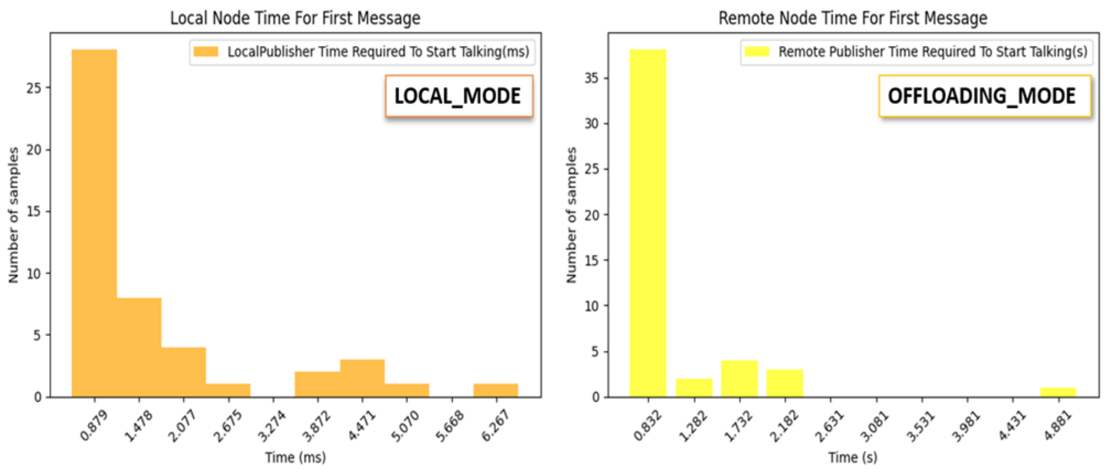


Figure 6.3: restart time required for nodes activity

These tests, as mentioned in the previous chapters, concern the create-and-delete approach for the Network Policies resource lifecycle. It is unknown how much the opposite create-then-patch approach might change the obtained results. This could be further explored, although a significant change is not expected, nor it has been possible, at the time of writing, to find a suitable and working patch for the specific netpol used here, capable of re-enabling the remote publisher by patching the first policy and not by deleting it. Actually, one patch was found, but only in a preliminary test phase when still working directly with the resources yaml files. Unfortunately, the transition ratio from what can be applied manually or by means of the K8s client libraries is not 1:1: the point is in the modifications that have to be applied to the CiliumNetworkPolicy syntax in order to work from inside a Python script, since they must be written differently from the original file (even just the indentation) to be successfully interpreted, and this may lead to an overall distinct behavior. Despite the large number of attempts made, it cannot be excluded the existence of the right combination of a netpol and its patch, especially considering how many different kinds of resource exist (the default K8s NetworkPolicies, the ones provided by Cilium or other CNIs), nor it is certain it would lead to better performances in terms of actual application time.

The third chart in figure 6.4 shows a result similar to the first one: deactivating a node or deleting a netpol does not take much time.

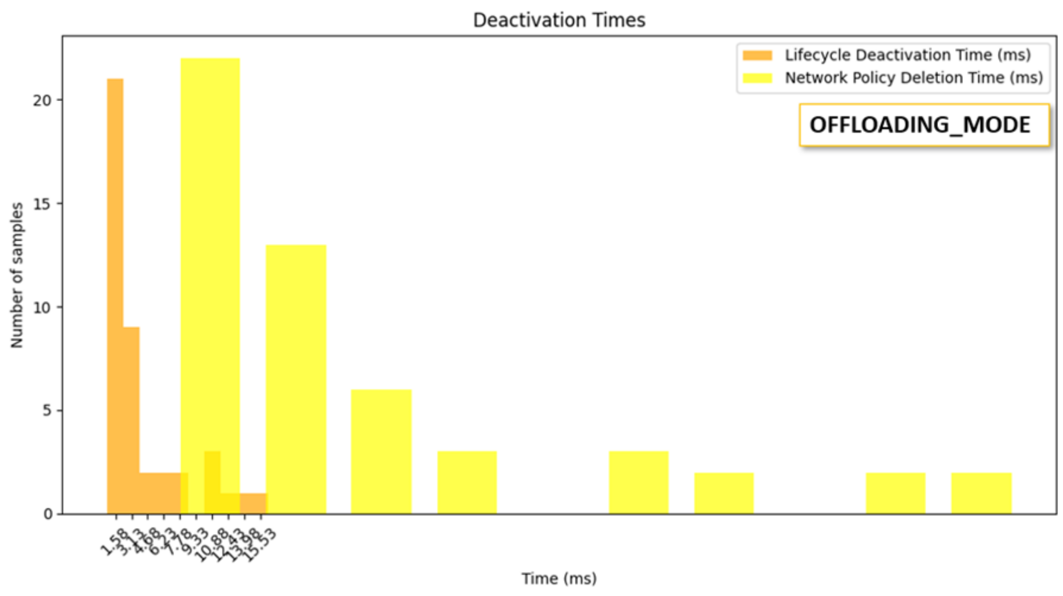


Figure 6.4: comparison of operations e) and g) for offloading_mode

It is interesting to observe how the fourth histogram for the last two operations actually misses. The time taken for a local/remote node to stop talking in fact is null in both cases: when it comes to stopping, the traffic disappears immediately. It was observed many times that the system is more inclined to stop than to restart. This result is actually not bad news, because if it is the transition from local to offloading that requires a greater amount of time, the algorithm should in theory be able to "absorb" this time. The system is in fact intentionally designed to keep working locally in the meantime and then, at some point, switching again, but only when "everything is ready", i.e. when the next netpol is by then applied and remote messages are coming. Thinking of a future different implementation, this kind of “ready-notice” might therefore be carry out by some Kubernetes lifecycle hook for the resources, or even by providing some kind of signal from the Zenoh-bridge informing that messages are finally coming.
In conclusion the final practice demonstration for this hybrid implementation led to satisfactory results and revealed key considerations for future development. More extensive testing and evaluations on physical robots in a real-world environment will provide further data, with the aim of putting this prototype into practice.

6.2 Zenoh and Liqo performances

The tests performed in previous projects and discussed as “related works” in Chapter 2, allowed to make a significant choice by taking Zenoh and the ROS Lifecycle abstraction as two of the main tools for this final implementation. The architecture analyzed in this thesis is certainly different, as from the beginning it aimed to be a more generic solution. However, the current results can finally confirm the expected good performance of such technologies, even with different architectures and implementations.

Regarding the publish-subscribe technology chosen to solve the DDS limitations, **Zenoh** has proven to be a very good choice, especially for being perfectly transparent towards ROS nodes that rely on the bridge/router to advertise their topics and services. It is well integrated with such robotic environment and is able to handle the compression and optimization aspects of the exchanged data strictly necessary for internet-scale applications. For all these reasons, Zenoh has been announced as the official future ROS2 middleware (RMW), meaning that future architectures for this use case will probably not need additional cluster deployments for its components. Official performance comparisons have been done among Zenoh, and other brokers like MQTT, Kafka, and DDS [7]. The available results show that Zenoh consistently outperforms MQTT and Kafka as far as throughput and latency evaluation, thanks to the low overhead design and multiple optimization techniques embedded in its implementation. Besides the performance, Zenoh also come with the simplest API and the shortest learning curve, surely presenting itself as one of the best choices for industrial, IoT, automotive applications, and robots that can seamlessly support the Cloud continuum.

When thinking about the most general multi-cluster architecture for a task-offloading scenario, it is certainly important to analyze also **Liqo** performances. The official documentation shows extensive results about it, focusing on the time required for the peering and namespace offloading main operations [9]. It is reported how the association of a ForeignCluster might require no more than a few seconds, while the Liqo reflection logic performance overhead is very limited, accounting for only a few milliseconds even in the most demanding scenario. In terms of CPU and RAM needed to work in a cluster, it is capable of satisfying also the requirements of resource-constrained devices. Finally, it is clear how Liqo simplifies multi-cluster Kubernetes management, allowing applications to seamlessly leverage resources across remote infrastructures. Despite these promising benchmarks, the particular use case and architecture discussed in this thesis does not actually directly benefit from extremely good Liqo performances. In fact, as explained before, a small set of operations, including peering and namespace offloading, are performed in a preventive phase aiming at preparing the clusters configuration. This means that, although these operations visually prove to take very little time, they cannot influence either the testing phase or the final behavior of this kind of application, since they occur before it is started. The overall system of this prototype is therefore independent from the time needed to put into practice the desired scenario and from the nature of the resources to create, thus not suffering any additional delays even if a single pod has to spawn multiple side containers.

Chapter 7

Conclusion

This thesis has investigated Cloud offloading, a technique that leverages remote Cloud servers to augment the capabilities of mobile robots and, more generally, of Cloud-native services. By offloading some computations, the robots are no longer restricted by their onboard processing power and storage. This clearly helps saving the robots battery and computing power, enabling them to perform more complex tasks. However, some key challenges have to be addressed: sending data back and forth creates delays (latency) which can be a problem for real-time tasks like the ones related to autonomous-driving vehicles. Also, sending potentially sensitive data to the Cloud may expose to security risks. The solution has been about trying to increasingly shorten the distances, exploring new technologies for Edge/Fog computing. By combining nearby datacenters power with local processing on the robot, it is finally possible to get advantages from both the approaches, while minimizing latency and security risks, thus effectively exploiting the benefits of this Cloud Continuum.

While the prototype developed in this work has utilized a simulated environment, the results suggest its applicability to real-world scenarios on a much larger scale. This opens the door to further investigation of the potential of such an approach based on tools like Liqo, leading towards a future of "Cloud-to-anything" computing, where resources can be seamlessly accessed from any device or location.

7.1 Future works

The most challenging and complete scenario to actualize would be the one with **multiple Liqo-peers** of different nature (see figure 7.1). The key technology is Multi-access Edge Computing (MEC) which consists in near real-time processing of large amounts of data produced by edge devices and applications closer to the location of acquisition. In other words, an edge extension of the edge network infrastructure, obtained by placing micro-datacenters near the robot. A radio access network (RAN) is the part of a mobile network that connects end-user devices, like smartphones, to the Cloud. For telecommunications network operators, RANs are crucial connection points that perform intensive and complex processing, today facing the rapid increasing demand of edge and 5G devices emerging as telco customers [10]. MEC implementations make RAN accessible to authorized application developers and content providers, allowing them to use edge computing either at the application level or at the lower level of network functions and information processing [11].

Future implementations could provide the robots with **discover abilities** to find partner servers within the edge of the network, like RAN servers or other nearby robots found via wireless protocols. This can be done by upgrading the switching algorithm and making it become a choice of the best solution among all these available servers where to offload tasks to, effectively reducing delays and reducing the distances. These new functionalities imply equipping the Network Controller node with some more sophisticated logic for identifying the most promising node from time to time, according to its instantaneous free processing power and the correspondent network link quality. A pseudo-code for the complete version has already been designed for this thesis.

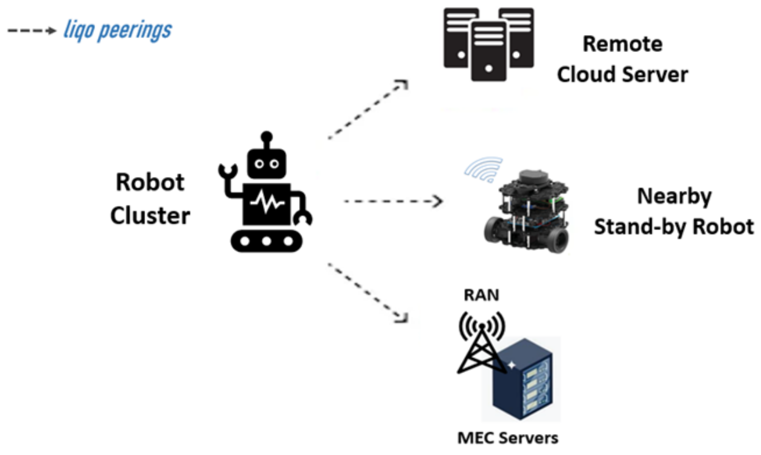


Figure 7.1: "Cloud-to-anything" scenario

It is definitely complicated to understand the extent to which a standby robot can be exploited as a pool of free computational resources, for a moment stopping itself to be the one that exploits it. Usually, the offloading to a different server is executed only after checking it is possible to work on it at higher frequencies. Conversely, it would be necessary to prove how much could a nearby robot actually become the host of an offloaded task, assuming the best scenario in which that robot is totally idle. Furthermore, in this robot-to-robot architecture, wireless coverage, signal quality and bandwidth aspects weigh even more, as well as the question of security (although Liqo internally takes advantages of an ad-hoc VPN actually mitigating that problem by design, at least partially). This study might also positively affect the design phase of robot fleets, helping to create lighter prototypes which are energy-efficient and able to perform more complex tasks. In short, this particular use case has to be deeply explored, highlighting what it means to move from "every robot is a cluster" to **a "cluster of all nearby robots"**.

As far as the NC **network monitoring internal module**, it can surely be as sophisticated as desired. Different applications, depending on how critical they are, may need different tuning of all the related parameters, such as packet-drop rate, SNR, latency and jitter. Recent works proposed an AI-Augmented Kalman filter which, instead of assuming explicit model knowledge for both its state prediction and correction phases, is able to learn such model from direct measurements of the network conditions and construct the filter in a data-driven way [12].

Other useful extra features could be the support to **multiple tasks simultaneous offloading** as well as for a **dynamic peer-list**. At the moment, the remote cluster of the single peer is in fact coupled manually in the configuration phase, hence a dynamic Liqo-based peering mechanism would be smarter and more automated. This could make the most of some Wi-Fi heatmap-software to find idle robots nearby.

Differently from the turtlesim used for testing, real-world devices come with **stateful applications**. One example may be of course autonomous-driving vehicles: they continuously need comprehensive information about the cost map of the surrounding environment they are travelling through. A sort of distributed storage is therefore needed and one of the possible solutions, already implemented in some previous works, is exploiting ROS2 services and specific topics adapted for the storage function. The aim is making those nodes that are deployed across the peers' clusters to quickly synchronize, by obtaining all the information stored up to that moment, or at least a minimum set of configuration data useful for continuing navigation.

In conclusion, several promising technologies need further investigation to achieve the most efficient task offloading strategy, finally leading to significant performance improvements for real-world applications. This study makes significant contributions to a topic actively explored by researchers and industries alike, offering insights from diverse perspectives and application types. Notably, this research has made substantial progress in defining the overall project direction, paving the way for real-world implementations of edge-to-anything architectures that can benefit a vast array of devices.

Chapter 8

Bibliography

[1] S. Moreschini, F. Pecorelli, X. Li, S. Naz, D. Hästbacka and D. Taibi, *Cloud Continuum: The Definition*, in IEEE Access, vol. 10, pp. 131876-131886, 2022, doi: 10.1109/ACCESS.2022.3229185

[2] Dario Paolo Gulotta. *Real time, dynamic cloud offloading for self-driving vehicles with secure and reliable automatic switching between local and edge computing.* Rel. Fulvio Giovanni Ottavio Risso. Politecnico di Torino, Corso di laurea magistrale in Ingegneria Informatica (Computer Engineering), 2023

[3] Jonathan Marsiano. *Enabling an autonomous robot to transparently access local, edge and cloud services.* Rel.Fulvio Giovanni Ottavio Risso. Politecnico di Torino, Corso di laurea magistrale in Ingegneria Informatica (Computer Engineering), 2022

[4] *Kubernetes node affinity: examples & instructions.* 2022 https://blog.kubecost.com/blog/kubernetes-node-affinity/

[5] Kubernetes Documentation. https://kubernetes.io/it/docs/

[6] Zenoh Documentation. https://zenoh.io/docs/getting-started/first-app/

[7] Corsaro, A., Cominardi, L., Baldoni, G., Guimaraes, C., Loudet, J., Hecart, O., Enoch, J., Avital, P., Ilin, M., & Bannov, D. (2023, settembre 30). *Zenoh-Unifying communication, storage, and computation from the cloud to the microcontroller.* 26th EUROMICRO Conference on Digital System Design (DSD), Durres, Albania. https://doi.org/10.5281/zenodo.10635550

[8] Liqo Documentation. https://docs.liqo.io/en/v0.10.2/

[9] *Benchmarking Liqo: Kubernetes Multi-Cluster Performance.*2022 https://medium.com/the-liqo-blog/benchmarking-liqo-kubernetes-multi-cluster-performance-d77942d7f67c

[10] Multi-access Edge Computing. *What is multi-access edge computing (MEC)?.* 2022. https://www.redhat.com/en/topics/edge-computing/what-is-multi-access-edge-computing

[11] Radio Access Network. *What is a radio access network (RAN).* 2021. https://www.redhat.com/en/topics/5g-networks/what-is-radio-access-network

[12] Q. Zhang and S. Pan, "*An AI-Augmented Kalman Filter Approach to Monitoring Network Traffic Matrix*" in IEEE Transactions on Network Science and Engineering, doi: 10.1109/TNSE.2023.3297660.

[13] ROS2 Documentation. https://docs.ros.org/en/humble/index.html

[14] *Publisher-Subscriber Model.* 2024. https://www.baeldung.com/cs/publisher-subscriber-model

[15] Cilium Documentation. https://docs.cilium.io/en/stable/

[16] *Cilium 1.0: Bringing the BPF Revolution to Kubernetes Networking and Securityhttps.* 2018. https://cilium.io/blog/2018/04/24/cilium-10/