POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Transparent access to local/edge/cloud services: the autonomous driving use case

Supervisors	Candidate
Prof Fulvio RISSO	Mario GIORDANO
Dr Marco IORIO	

April 2021

Acknowledgements

To my family, my girlfriend and my friends, who supported me throughout this journey.

"All we have to decide is what to do with the time that is given us."
J.R.R. Tolkien

Abstract

Nowadays, there is a lot of talk about autonomous driving and all the possibilities it could offer in the future and those that it already offers, alongside with all the challenges that robotic automation brings. This kind of application requires a huge computational effort, best carried out to small data centers at the edge of the network or in the cloud. For resiliency reasons (e.g., network outages), the same service must be also offered locally. Monitoring the network status and deciding what instance of the service has to be used, should not be taken care of by the autonomous vehicle itself. All things considered, this thesis aims at investigating the problem of providing transparent access to the services mentioned above, by implementing a network layer responsible for network monitoring and of the forwarding of the data to the appropriate service instance. To achieve the goal, this thesis deeply investigated Kubernetes and ROS2. An image recognition application (e.g., identifying the other vehicles and the pedestrians) has been considered as a use-case. The ROS2 topology is composed of an image publisher, simulating a camera feed, a local and a remote YOLO node, the local and remote object recognition nodes and an image viewer node, used to view the images images elaborated by YOLO. The above-mentioned network layer has been developed by leveraging Kubernetes functionalities. Two Kubernetes clusters have been deployed: one remote (e.g., at the edge), into which only the remote YOLO instance is running, and one local (e.g on the robot), for all the other nodes. The virtualization of the autonomous driving service has been obtained by making use of the concept of Kubernetes Service, which exposes either the local or the remote YOLO node. The image publisher node will contact the YOLO node by using the Kubernetes Service IP address, then Kubernetes itself will forward the traffic to the right destination, based on the Selector field of that service. By changing that field, we can decide what the actual destination will be. In this way, the ROS2 nodes will not have to care about the actual traffic destination. The selector has to change dynamically, therefore a Kubernetes Operator is deployed into the cluster; it monitors the status of the network by means of "Quality" and "Signal Strength" indicators. As soon as one of those two metrics does no longer meet the requirements, the Service Selector is changed to local. Hence, all computations start to be performed locally (with lighter algorithms) and the robot continues to operate correctly. All in all, the investigated approach has shown to be promising, allowing to save local computational resources by leveraging the ones available at the edge of the network. Yet, it automatically fallbacks to local services as well in case the network connectivity is no longer available, to avoid the occurrence of possibly safety critical issues.

Table of Contents

Chapter 1

Introduction

Autonomous driving and robotics have been, in these last years, a **very popular topic** among professionals and even non professionals. Big companies like Google with Waymo (currently owned by Alphabet, a Google parent company), Amazon with Zoox and Tesla with Autopilot, are all participating in the challenges offered by the developement of a fully-autonomous driving system.

Such systems must be able to deal with **huge amounts of data** coming from external sensors like RGB cameras, depth cameras, LiDARs and others.

At every instant, the system has to perform object detection on the RGB camera feed, combine it with depth information collected from depth cameras and/or LiDARs, keep track of what eventual other nearby vehicles are doing by means of the data sent among themselves and many other things. Everything has to be done respecting a **huge time constraint** because all that information is no longer meaningful if it takes one second or even more to be elaborated and sent back to the system since, especially at higher velocities, everything could change in the blink of an eye.

It is evident, at this point, that for a system to be able to perform such things it is necessary a **huge computational effort**. Old autonomous system prototypes used to perform all of these tasks by means of an onboard computer system, which therefore was not only responsible of collecting the data, but also elaborating it and sending it back to the vehicle control systems. These onboard computers were huge machines, which left basically no space for eventual passengers. That made sense from a developement perspective, but not from a more commercial point of view.

Nowadays, with the advent of technologies such as 5G, which enables users to take advantage of enormous upload and download speeds even wirelessly, the computational effort mentioned above could be more suitable to be carried on **data centers** at the edge of the network, or even in the cloud. This would relieve the system itself of the majority of the workload.

In order to face possible **network outages**, the vehicle must still be able to carry on those elaborations (maybe with less precision, therefore requiring less resources), thus **two different instances of the data elaboration service** have to be made available: one remote (e.g. on the edge) and one local (e.g. onboard). The autonomous system, though, **should not have to care about what instance of it is using** and neither how and when it should switch betweem them.

Working on a real autonoumous system, though, requires a huge amount of work and, most importantly, a huge economic effort. That is why, for the scope of this thesis, we tried to find a way to approximate a real system with something less expensive and more suited for this work. The solution has been found in the service robotic world, and that is why this thesis is done in collaboration with the PIC4SeR, the Politecnico Interdepartemental Center For Service Robotics. They introduced and helped us approaching a world that was unknowkn for the most part. Even though a robotic system complexity is definitely lower than the one of a real autonomous system, it was enough given the goals of this work are not focused on the vehicle itself but on the networking side.

Considering all of the above, this thesis **objective** has been developing a **network layer that takes care of**:

- **Virtualizing** the service so that the vehicle does **not have to worry about what destination** it is sending the data to, but only about actually sending it.
- **Transparently modifying** the service requests and **redirecting them** to the correct service instance.
- **Deciding**, at any given time, **what instance of the service is the best one**, by monitoring network status and possibly predict malfunctionings.

1.1 Related Work

Computational offloading to the cloud/edge for mobile devices (e.g., smartphones, Unmanned Aerial Vehicles, Autonomous Vehicles) is a topic that nowadays has become more and more relevant, considering, for example, the amount of data that an autonomous vehicle has to analyze at each moment and the time constraints that it must respect. Offloading this computational effort will not only bring advantages in terms of speed, but also in terms of **energy consumption**, which is a very important factor to keep in mind when it comes to mobile devices.

Many of the works treating this argument are focused mainly on the **automotive sector**, and most of them heavily rely on the **5G infrastructure** because of the need to have a huge bandwidth and very low latencies. The work described in [1] aims at providing an introduction and an overview of the physical layer design

that enables **Ultra-Reliable Low-Latency Communication** (URLLC) on the 5G techonology. In particular, [2] discusses and elaborates how this technology can enable the cooperation between edge and cloud offloading, until now almost impossible due to techonological limitations.

Another recurring paradigm is the **Multi-access Edge Computing** (MEC), which has been introduced and analyzed in [3]; it is based on the idea that running applications and performing tasks at the edge of the network, therefore closer to the mobile device that is requesting the service, brings improved performances and reduces network congestion.

An important aspect to keep in consideration is the fact that, especially in the automotive field, the mobile devices that need to offload to the edge/cloud **move very fast**, thus the distance between them and server they are trying to contact may **vary greatly** in a short amount of time: by leveraging the capabilities offered by 5G, [4] investigates on how to maintain a **low latency** between the roaming vehicle and the service hosted on the MEC server.

Aside from the latency problem, it is also fundamental to remember that servers, while having a great computational power, **do not have an infinite amount of available resources**. Therefore, a vehicle not only has to choose the server which offers the lowest latency as possible, but it has also to take in consideration **the load** that the particular server is undergoing at that moment and also predict how the load will change in the near future by exchanging data with nearby vehicles. This concept is analyzed end elaborated in [5].

All the works mentioned above do not take in consideration the possibility of **containerizing the autonomous system** services; this approach has been analyzed and elaborated by both [6] and [7].

1.2 Structure

This thesis is structured as follows: the **next two chapters** are focused on giving an introduction to the two main technologies used in this work, **Kubernetes and ROS**, by presenting both their basics and a little bit of history. **Chapter five** is concentrated on an **initial investigation** regarding hardware specifications of the robotic system that has been dealt with and latency constraints that the system has to respect while **chapter six** puts the spotlight on what has been done in order to make a **ROS system properly run inside a Kubernetes cluster**. **Chapter seven** focuses on analyzing **traffic routing** solutions offered by Kubernetes and **chapter eight** presents the **network monitoring** solution as well as the demo that has been prepared to show the achieved results. The last chapter talks about conclusions and gives suggestions about future works.

Chapter 2

Kubernetes

In these last years, a huge migration has been seen: developers and companies started to prefer **containers** over standard virtual machines due to the improvements they bring. At the beginning, every container had to be **managed manually**, but, especially as applications dimensions began to increase, doing so had become a very bothersome and **time consuming** task.

That is when ***Kubernetes*** [8], a containerized application orchestrator, started to gain a lot of popularity. Kubernetes allowed developers to easily manage and keep track of the health of their containerized software. After a first configuration, most of the work can be done **automatically** by the Kubernetes system itself, requiring human intervention only in case of problems that cannot be solved automatically.

This chapter presents the Kubernetes **architecture**, a bit of its history and also an introduction to virtualization, in particular the concept of *container*, fundamental in understading Kubernetes.

2.1 Kubernetes History

Around the year 2004, Google presented *Borg*, a small project that aimed to create an internal large-scale cluster management system. Fast forward to 2014, Kubernetes was introduced as an open source version of Borg and, in June of the same year, the first GitHub commit was done. The following year, Kubernetes v1.0 gets released and Google signs a partnership with the Linux Foundation. This collaboration gave birth to the *Cloud Native Computing Foundation*, which aims to building a sustainable ecosystem for cloud applications to grow. In the same year, the v1.1 update was released, bringing major performance improvements and new features.

The successive year, 2016, though, is the year that will give Kubernetes popularity. Along with the release of *Helm*, the Kubernetes packet manager, and the

v1.5 release, which brought Windows Server compatibility, 2016 was the year of the biggest Kubernetes deployment to date: Pokemon GO. Due to the massive and unexpected user base, it pushed the Kubernetes system like never before, therefore allowing the developers to discover and solve several problems. In 2017, enterprises like Docker, Amazon and Microsoft started to adopt and support Kubernetes and v1.9 was released, bringing new features like beta Windows support along with general improvements.

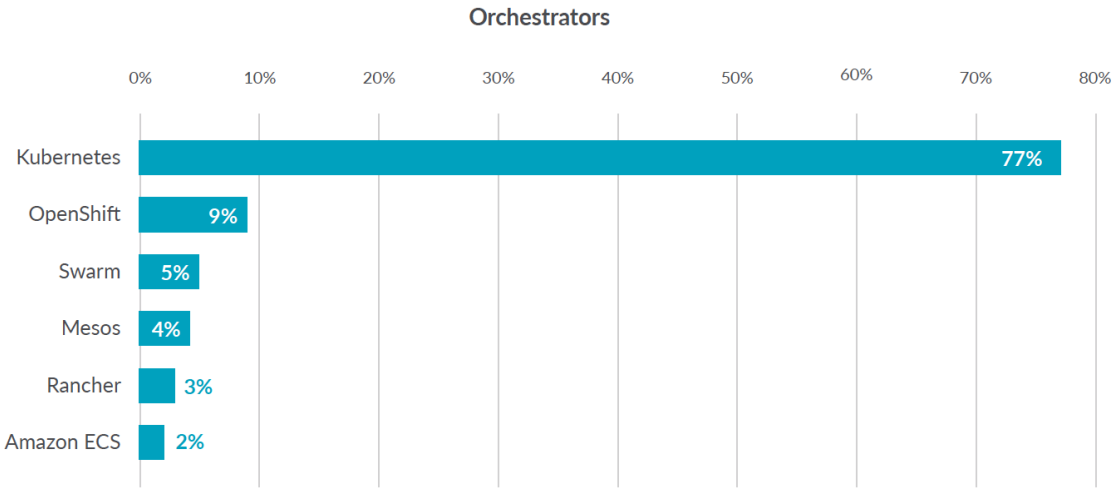

Figure 2.1: Container orchestrators usage as reported in [9]

By now, Kubernetes has reached v1.20 and, despite the fact that other container orchestration solutions exist, **more than 75%** of the companies included in the CNCF [9] report declared that Kubernetes is the go-to choice.

2.2 Containers

In order to properly understand what Kubernetes does and how it does it, it is fundamental to have at least a basic grasp of what a container is.

Before containerization and virtualization were even a thing, applications ran on **physical servers**. With this approach, though, there was **no separation** between different applications, so, for example, a problem in one application could influence another one, or an application could use all the resources available, leaving the others starving. A solution was to run each application in a different physical machine, but clearly this is not acceptable, since, aside from the economic cost of owning and managing several physical machines, it would lead to a possibly enormous waste of resources.

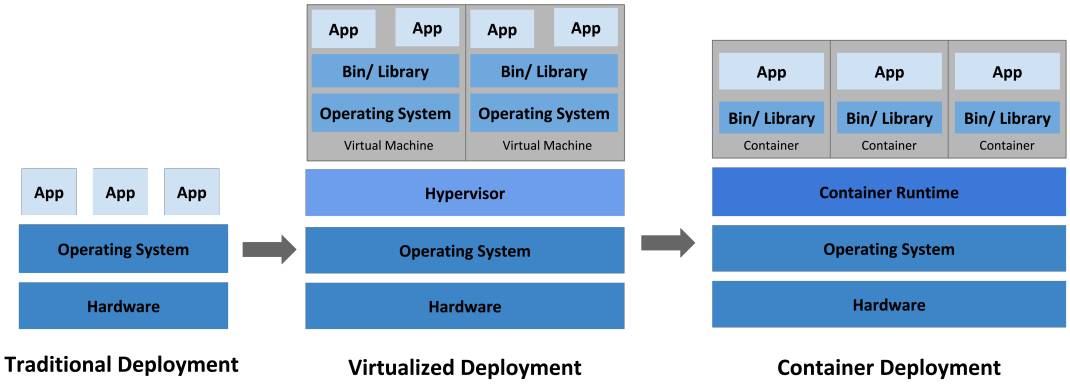

Figure 2.2: Evolution of applications deployment

A **proper solution** did not arrive until the birth of the concept of **virtualization**. By means of a **Hypervisor** running on top of the host machine operating system, virtualization allows to create **many** Virtual Machines(VM) on a **single** physical machine. Each VM is an **exact copy** of the underlying hardware, thus allowing the user to run a different operating system on each one of them. Thanks to this, applications running in a VM are completely separated from the applications running on another VM, therefore increasing the reliability of the whole system. In this way, physical servers' resources can be used in a much better way, thus reducing the cost of having and managing several physical machines. VMs have a remarkable problem: since each single VM is a complete copy of the underlying hardware and runs its own operating system, they come with a **considerable overhead**.

In more recent years, a new solution proposal was born: containers. A container can be seen as a **lightweight** VM. Lightweight because, unlike traditional VMs, each container contains only a copy of the application it has to run and the necessary libraries, thus leading to having a dimension of the **order of Megabytes**, instead of Gigabytes, an uptime of the **order of seconds**, instead of minutes. The hypervisor of the container world is called **Container Runtime** (for example, *Docker Engine*), and it runs on top of the host operating system. Another important feature of containers, is their **portability**: being they decoupled from the underlying structure, they can easily be deployed on different OS and cloud distributions.

2.3 Containerize An Application

As it has just been introduced, in order to run an application inside a Kubernetes cluster it needs first to be containerized. To do so all that is needed is to create a so

called **Dockerfile**, which contains all the information (such as required environment variables, commands to be executed, etc...) about the container image that it is going to be generated. After the image has been built, it can be either started directly by interacting with the Docker CLI, or, like in this case, into a container orchestrator, for instance, Kubernetes.

Every Docker image has an **entrypoint**; an entrypoint is the first command that the container will execute as soon as it starts running and its whole **lifecycle depends on it**. Most of the times, the entrypoint is the command that will start the main application of the container and as soon as its execution ends, whether it was because of an error, or just because it rightfully terminated, the container terminates as well.

```
1  ############################
2  # STEP 1 build executable binary
3  ############################
4  FROM golang:alpine AS builder
5  # Install git.
6  # Git is required for fetching the dependencies.
7  RUN apk update && apk add --no-cache git
8  WORKDIR $GOPATH/src/client/
9  COPY . .
10 # Fetch dependencies.
11 # Using go get.
12 RUN go get -d -v
13 # Build the binary.
14 RUN go build -o /go/bin/client
15 ############################
16 # STEP 2 build a small image
17 ############################
18 FROM alpine
19 # Copy our static executable.
20 COPY --from=builder /go/bin/client /go/bin/client
21 # Run the server binary.
22 ENTRYPOINT ["/go/bin/client"]
```

Figure 2.3: Example Dockerfile to Build and Run a Go Program

Online repositories like **Dockerhub**, contain a huge amount of **container images ready to be pulled and used**. These pre-built images can, not only be run right away, but they can also be used as a starting point for custom Dockerfiles, for example, the Ubuntu Docker image can be used as a foundation for containerizing an application that needs a Linux environment in order to be executed.

2.4 Kubernetes Features

As said previously, when the number of containers we need to manage increases, it becomes more and more difficult to keep track of everything that is going on and even doing a simple operation means that the developer has to take in considerations many containers at once.

That is when **container orchestrators** come to the rescue. After being configured based on the needs of the developer, they can automatically manage many tasks, from replacing crashed containers, to automatizing a canary release.

Here are some of the features Kubernetes provides the user with:

- **Self-healing:** whenever a container fails or does not respond to user defined health-checks, Kubernetes will automatically restart it or replace it. The new containers will not be advertised until they are ready to serve requests.
- **Automatic scheduling:** the user provides the Kubernetes system with a cluster of nodes it can use to run tasks. After telling the system what kind of resources a specific container needs, Kubernetes will automatically deploy it to make the most out of the resources it has.
- **Automated rollouts and rollbacks:** the user can describe the desired state for its deployment and the Kubernetes system will gradually change the current state until it will reach the desired one. If something goes wrong during the rollout goes wrong, Kubernetes will automatically rollback to the latest stable state.
- **Service discovery and load balacing:** containers can be exposed by either using a DNS name or their own IP address. Kubernetes will also do load balancing and distribute the traffic in order to keep the deployment stable.
- **Storage orchestration:** users are allowed to automatically mount a storage system of choice, such as local storages, public cloud provider and more.
- **Secret and configuration management:** Kubernetes lets the user store and manage sensitive information, like passwords and SSH keys. Those secrets can be deployed and updated withouth having to rebuild container images, and without exposing them in the stack configuration.

2.5 Kubernetes Components

Whenever a cluster is created, two kind of worker machines, called nodes, are created: **worker and control plane**.

Each cluster must have **at least one** control plane node, even though usually multiple are created to provide resiliency and high availability. The goal of the control plane is to **manage** worker nodes and Pods across the cluster.

On the other hand, worker nodes **host Pods**, where a Pod is a set of running containers in the cluster, and there must be at least one worker node per cluster.

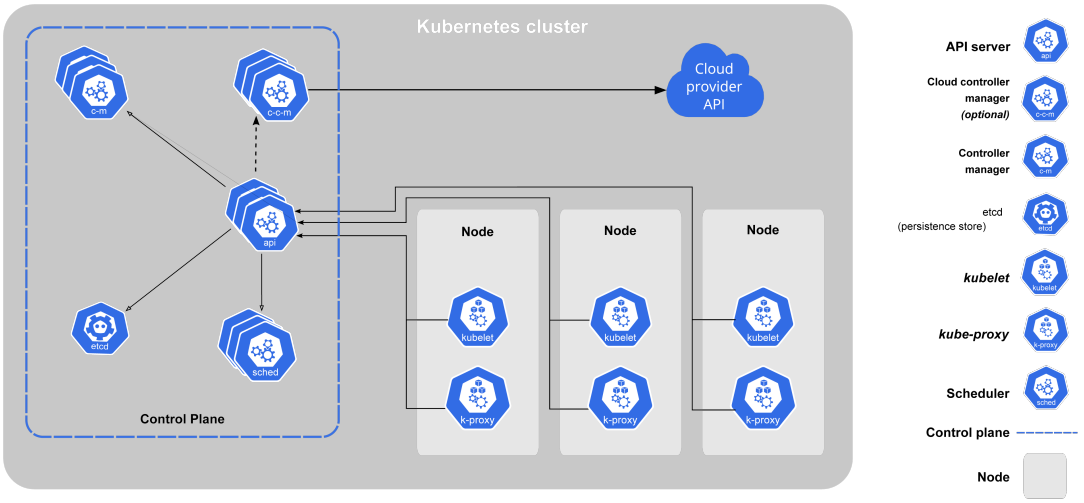

Figure 2.4: Kubernetes Components

2.5.1 Control Plane Components

Control plane components make **global decisions** about the cluster and detect and respond to **cluster events**. These components can be run on any machine in the cluster, but usually, for simplicity, they are scheduled on the same machine. This machine will **not** be used to run user containers.

Here is a list of these components:

- **kube-apiserver:** the API server is a component of the Kubernetes control plane which exposes the Kubernetes API. It acts as the front-end of the Kubernetes control plane. It intercepts API requests, validates and processes them. The main implementation of the Kubernetes API server is *kube-apiserver*. It can scale by increasing the number of replicas and traffic can be balanced between those istances.

- **etcd:** key-value store used as Kubernetes' backing store for all cluster data. Distributed, consistent and higly-available. To ensure data consistency across all nodes in the cluster, etcd uses the Raft Consensus algorithm [10].

- **kube-scheduler:** it watches for newly created Pods and select a node for them to run on. In order to make a good scheduling decision, it takes in consideration factor such as resource requirements, user constraints, data locality and more.

- **kube-controller-manager:** it runs the controller processes. A controller is a control-loop which keeps track of the current state and, in case of any problems, it tries to get the current state back to the desired one. At a logical level, each one of these controller processes is separated from the others, but to simplify things, they are all compiled into a single binary and run in a single process.

  Some of these controllers are:

  - **Node controller:** notices and responds when nodes go down;
  - **Replication controller:** maintains the correct number of pods for every replication controller object in the system;
  - **Endpoints controller:** populates the Endpoints object. It basically joins Services and Pods;
  - **Service Account and Token controllers:** create default accounts and API access token for new namespaces.

- **cloud-controller-manager:** this component embeds all the cloud-specific control logic. It lets the user link its cluster into the cloud provider's API, and separates the components that interact only with the cluster from those which interact with the cloud platform. Even in this case, multiple controller processes are compiled into a single executable in order to reduce complexity.

  The controllers that can have cloud dependencies are:

  - **Node controller:** to check whether a node in the cloud has been deleted after it stops responding;
  - **Route controller:** to set up routes in the cloud infrastructure;
  - **Service controller:** to create, update and delete cloud provider load balancers.

2.5.2 Worker Node Components

These components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

They are:

- **kubelet:** it runs on each node in the cluster and checks whether containers in a Pod are running. The kubelet takes a set of PodSpecs and makes sure that the containers describe in those PodSpecs are healthy and running.

  The kubelet only manages containers created by Kubernetes.

- **kube-proxy:** it is a network proxy that runs on each node in the cluster, implementing a part of the Kubernetes service concept. It allows Pods to communicate to destinations both inside and outside the cluster, by maintaning network rules on nodes.

  kube-proxy forwards the traffic itself, if there is no available packet filter on the underlying operating system.

- **Container runtime:** it is the software responsible for running containers. Multiple container runtimes are available, such as: Docker, containerd, and any implementation of the Kubernetes Container Runtime Interface.

2.5.3 Addons

By means of already existing Kubernetes resources, addons implement **new cluster features**. Since these provide cluster-level features, such as DNS (the only addons which every cluster should have) or a Dashboard, resources for addons belong to the kube-system namespace.

2.6 Kubernetes Objects

Kubernetes objects are persistent entities that represent the cluster **desired state**. So, whenever an object is created, the user is telling the Kubernetes system what it wants the cluster's workload to look like.

In order to interact with a Kubernetes object — by creating, updating or deleting it — the user must use the *Kubernetes API*. To use the Kubernetes API, the user can either use a CLI tools, such as *kubectl*, which automatically makes the necessary API calls, or interact with them directly in its own program by means of one of the **Client Libraries**.

The fields necessary to describe an object are:

- **apiVersion:** which version of the API is being used for the project
- **kind:** what kind of object it is being described
- **metadata:** data that helps uniquely identifying the object
- **spec:** the object desired state

- **status:** the object current state, as supplied by the Kubernetes system itself

In order to deploy an object into a Kubernetes cluster, the most common way to do so is to describe it in a **.yaml file** containing all the necessary information. An example of such a file is provided in figure 2.5, which shows the description of a Kubernetes Deployment resource.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Figure 2.5: An example .yaml file that describes a Deployment object

Usually these files are then provided to *kubectl*, which will take care of **converting** all the data contained in them into *JSON* format and make the proper requests to the API Server.

The Kubernetes API allows the user to do four main operations on Kubernetes resources:

- **Create:** creates the resource in the storage backend. After its creation, the desired state is applied
- **Update:** comes in two forms:

  - **Replace:** replaces the existing resource specifications with the provided ones.
  - **Patch:** applies a change to a specific field.

- **Read:** comes in three forms:
  - **Get:** retrieves a specific object by its name
  - **List:** retrieves all objects of a specific type within a namespace
  - **Watch:** streams the results for one or more objects as they get updated
- **Delete:** deletes a specific resource

More info on this can be found in the Kubernetes API documentation [11]. Next, some of the **fundamental objects** will be briefly presented.

2.6.1 Namespaces

Namespaces are nothing more than **virtual clusters** backed by the same physical cluster. Namespaces are intended to be used when the cluster's resources need to be divided between multiple users or projects.

A Kubernetes cluster has four default namespaces:

- **default:** default namespace for objects with no other namespace
- **kube-system:** namespace for objects created by the Kubernetes system
- **kube-public:** namespace readable by all users, even non-authenticated ones, used to make some resources visible and readable publicly throughout the whole cluster
- **kube-node-lease:** namespace for the lease objects associated with each node, used to improve performances of the node heartbeats as the cluster scales

2.6.2 Labels and Selectors

Labels are key/value pairs attached to objects, intended to be used to specify **identifying attributes** that are meaningful to the user, but do not have an implication to the core system.

Labels can be attached at the object creation time and added/deleted/modified at any subsequent time. Each object can have **multiple labels**, the only constraint is that each key must be unique for that object.

By means of Selectors, labels can be used for **grouping together** multiple objects that have some particular characteristic in common.

2.6.3 Annotations

Annotations, like Labels, are key/value pairs attached to a specific object. Unlike Labels, though, Annotations are used to add information **not useful for identifying** the object. They allow also characters not allowed by Labels.

The informations described by Annotations can be retrieved and used by external tools and libraries.

2.6.4 Pods

Pods are the **smallest deployable computing unit** that can be created and managed in Kubernetes. A single Pod contains one or more containers, with shared storage/network resources, with specifications for how to run them.

The most common use case is having a **single container** per Pod. In this case, the Pod can be seen as a wrapper for the container. Kubernetes will then manage the Pod, instead of directly managing the container itself.

Another way of using a pod, even though it is less common than the previous one, is running **multiple containers** in the same pod. These containers are tightly coupled and need to share resouces in order to properly work.

In practice, the user will **rarely directly create** a Pod resource. Pods are ephemeral resources, so they cannot, for example, handle replication and self-healing by themselves.

2.6.5 ReplicaSet

A ReplicaSet is in charge of **maintaining a stable number of replica Pods**. By means of a selector, it can identify what Pods it can acquire.

Using the value of the fields in its configuration file, a ReplicaSet knows **how many replicas** of the Pods it manages should be running in the cluster, and how to create a new Pod in order to meet the required replicas amount. ReplicaSet are usually **not directly** created and deployed, instead they are deployed by means of the higher-level concept of Deployments, which automatically create them, along with providing declarative updates to Pods.

2.6.6 Deployments

A Deployment **manages** the creation, the update and the deletion of Pods. It does so by creating a ReplicaSet, that will then create the Pods themselves. That is why, usually, applications are managed and deployed by creating or modifying a Deployment, instead of manually creating ReplicaSets and/or a Pods. An example configuration file for a deployment can be seen in **figure 2.5**.

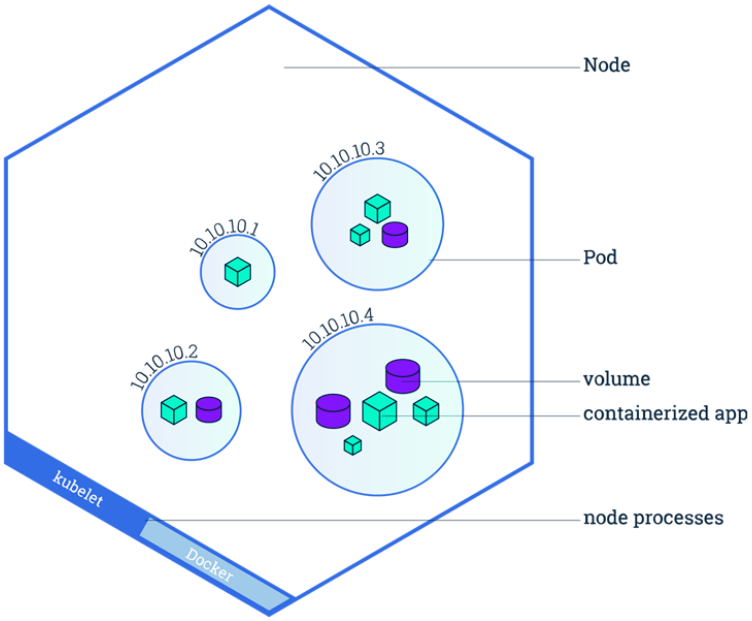

Figure 2.6: Pods in a Node

2.6.7 DaemonSet

By means of a DaemonSet, the user can ensure that all, or some, Nodes, run a copy of a Pod. If a node is added to the cluster the DaemonSet will make sure to add a Pod to it. If a node gets deleted, all the Pods are garbage collected.

By deleting a DaemonSet, all the Pods created by it, will be cleaned up.

2.6.8 Services

Services give the user the possibility of **exposing an application** running on a set of Pods as a network service. There is no need of modifying the application in such a way as to use a particular or unfamiliar discovery mechanism, Kubernetes will **automatically assign** to each Pod an IP address and a single DNS name for a set of Pods. Traffic is also **automatically load balanced** between them.

Services are particularly useful in order to avoid having to deal with Pods' **non-static IP** addresses. Since Pods are often destroyed and created to match the desired state of the cluster they are running in, if we imagine having a set of Pods offering a service to other Pods, the latter will have a difficult time keeping track of which IP to connect to. Exposing a set of Pods via a Service, will enable other Pods to connect to those by using the IP address or the DNS name of the

Service itself.

There are different Service types:

- **ClusterIP:** the Service is exposed on a cluster-internal IP, thus making it only reachable from within the cluster. This is the default option;
- **NodePort:** the Service is exposed on each Node's IP at a static port;
- **LoadBalancer:** exposes the Service externally using a cloud-provided load balancer;
- **ExternalName:** the Service is mapped to the DNS entry defined in the field *externalName*.

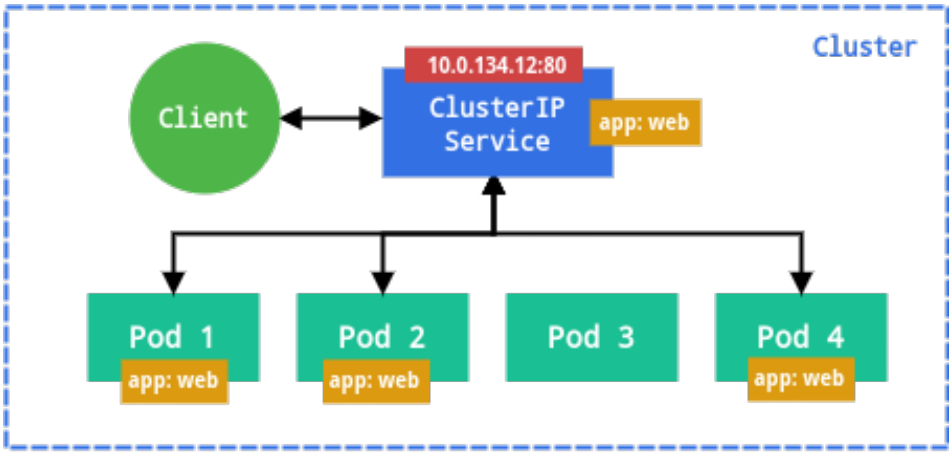

Figure 2.7: ClusterIP Service

2.6.9 Configmaps

Configmaps allow to store **non-confidential** key-value pairs. They can easily by consumed by Pods, and enable the user to decouple the application code from configuration data.

In this way, applications are made even more portable, since they do not depend on environment-specific configurations anymore.

2.7 Networking

The Kubernetes network model follows the so called *IP-per-pod* model. Each Pod receives a **unique** IP address in the cluster, that other Pods can use to interact with the containers running inside it.

Containers running inside the same Pod, will **share** the same IP and MAC address and can communicate with each other by reaching their ports on **localhost**. Since they share the same network namespace, the usage of ports in a Pod must be coordinated.

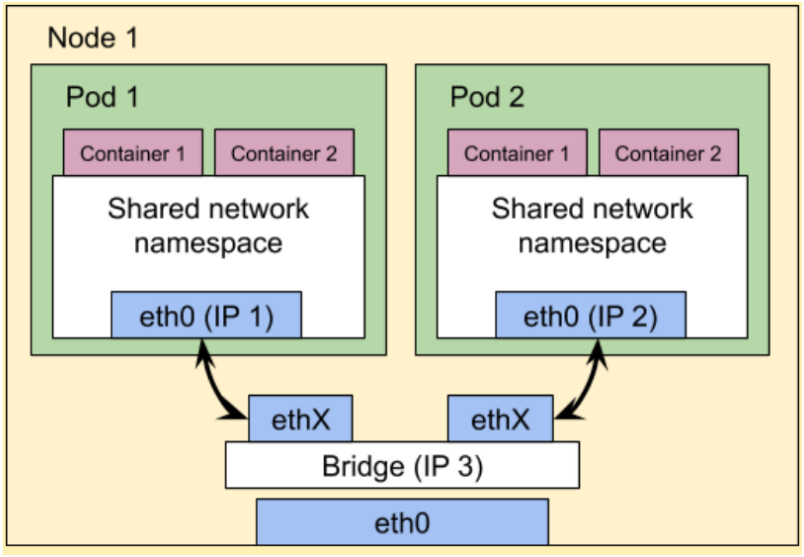

Figure 2.8: Pod to Pod communication

Each Pod network capabilities can be **restricted** by using *Network Policies*. They act as a **whitelist**, allowing only the specified ingress or egress traffic on a specific set of Pods.

Kubernetes imposes some fundamental requirements on the network implementations:

- Pods on a Node can communicate with all Pods without a NAT
- Agents on a Node, such as system deamons or kubelet, can communicate with all Pods on that Node
- Pods in the host network can communicate with all the Pods on all Nodes without NAT

This model and these requirements were devised in order to allow a **transition** from traditional VMs to containers as smooth and simple as possible.

A CNI, acronym for "Container Network Interface", is a CNCF project that consists of a specification and libraries for writing plugins used to configure network interfaces in Linux containers. A CNI is only concerned by containers' network connectivity and removing allocated resources when containers are deleted. It will

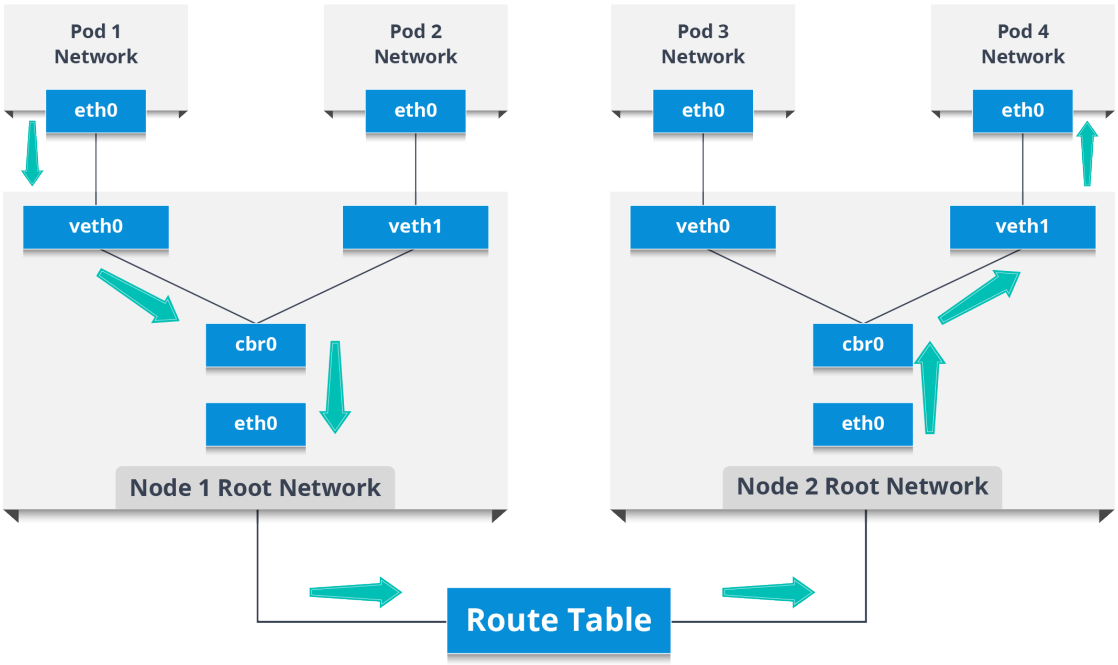

Figure 2.9: Node to Node communication

assign to each Pod a virtual ethernet pair and an IP address and new routes will be added to the routing table.

There are **several implementations** of this network model available to be used on Kubernetes, following there is a brief introduction to the one that has been used in this thesis.

2.7.1 Calico

Calico [12] is an open-source network solution for containers and VMs. It is an implementation of the interface defined by the CNI. Calico is not only supported by Kubernetes, but also by other platforms like OpenShift, OpenStack and Docker EE.

The main components of Calico, depicted in figure 2.10, are:

- **etcd:** all necessary information about the cluster and other components are stored here. This is the entrypoint for *calicoctl*, the command line tool to manage calico network and policies

- **BIRD:** a BGP client. BIRD will establish connections between nodes in order to share their router and enable inter-node communication.

- **FELIX:** by using the data stored in etcd, this is the component responsible of enforcing policies.
- **IPAM:** stands for IP Address Management, and it is used during the network creation process. It keeps tracks of what IP are currently assigned and in-use.

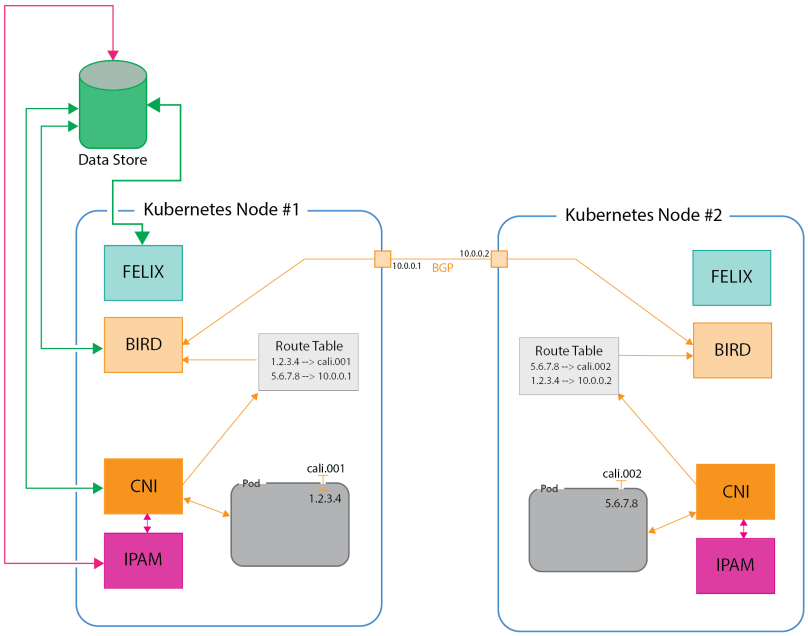

Figure 2.10: Calico Architecture

Chapter 3

ROS

At the beginning, developing complex and robust robot behavior was not only a **difficult and time consuming task**, but it had to be done explicitly for one particular kind of robot platform, since the software had to be written **taking very strongly in consideration the hardware** it was going to run onto. Thus, it was rarely, if not never, possible to re-use a piece of already existing code without having to adapt it to the new platform. Needless to say, this was an **enormous effort to take on**, especially for smaller companies or laboratories. It is clear that a **new approach was desperately needed**. That is where **ROS comes to aid**. ROS stands for ***Robot Operating System*** and, despite what the name suggests, it is not a real operating system; it is a **collection of libraries, tools and conventions** especially devised to enable developers to share and re-use robot behavior independently of the platform it was initially thought for. This also **allows different teams to cooperate**: one team might have a lot of expertise when it comes to robot movements, while another might be the best in developing software for sensors data elaboration; while this is still a complex job, with the help of ROS those two team can easily share their work and therefore as a result obtaining a way better robot behavior than if they did everything by themselves. Before delving into ROS specifications, a little bit of history.

3.1 ROS History

The **first ideas** of what would later become ROS, came together around 2007, when its creators, Eric Berger and Keenan Wyrobek, at the time PhD students at Stanford, noticed that many of their colleagues **struggled because of the multiple sides of robotics**: an expert in computer vision may not be an expert when it comes to hardware, and so on. They consequentely started developing software and received some funding to support the creation of the PR1 hardware

prototype, the one the initial pieces of software was thought for. Later that year, the ROS project migrated to **Willow Garage**, a technology incubator, and started to receive contributions even from outside the incubator. It was then already clear that ROS was **destined to be a multi-robot platform**, not only related to the PR1 and the PR2 (an upgraded PR1 developed while inside at Willow Garage). By the end of 2008, the team reached a couple of **important goals**: a PR2 managed to not only navigate continuosly for two days straight, but also to move around the office, open doors and plug itself into the power outlet; soon after that an early version of ROS was released along with the first bits of documentation. The successive big goal was reached in 2010 when Willow Garage managed to gift eleven PR2 to different institutions around the world, thus boosting ROS popularity; this led, after the official **release of ROS v1.0**, to the first drone and autonomous car running on ROS. In 2012, Willow Garage gave birth to the **Open Source Robotics Foundation** (OSRF) which, after immediately being given a software contract by DARPA, became in 2013 the primary software maintainer for ROS and absorbed Willow Garage itself. Since then, a new version of ROS has been released on a per year basis, gaining more and more popularity everyday, to the point that NASA and tech giants like Microsoft and Amazon took an interest in it, with NASA announcing the first robot running ROS in space in 2014. Lastly, the biggest recent step has been the announcement and the **release of ROS2**, which offers significant changes to the ROS API and support to more recent technologies.

3.2 ROS Working Principles

A ROS system is generally composed of **multiple not tightly coupled nodes**, where a node is a process responsible for, usually, a single task. Each one of those, in order to achieve a more complex robotic behavior, must communicate with the other components of the system and that can be done by means of **messages**. Those are **exchanged on channels called topics**, and each single node can **subscribe to and publish on multiple topics** at the same time. ROS has an **integrated node discovery system**, therefore, as soon as a node enters the system, its presence will automatically be announced to all the other nodes, so that they can be aware of what topics the new node is publishing on and which ones it is subscribing to. Figure 3.1 depicts an example ROS system in which multiple ROS nodes publish and subscribe to one or more particular ROS topics. In the following sub-sections there will be a deeper presentation of the main ROS concepts.

3.2.1 Nodes

As mentioned above, a node is a running process that, usually, carries out a **single, specific task** (e.g. retrieving a camera feed, controlling the robot wheels). A

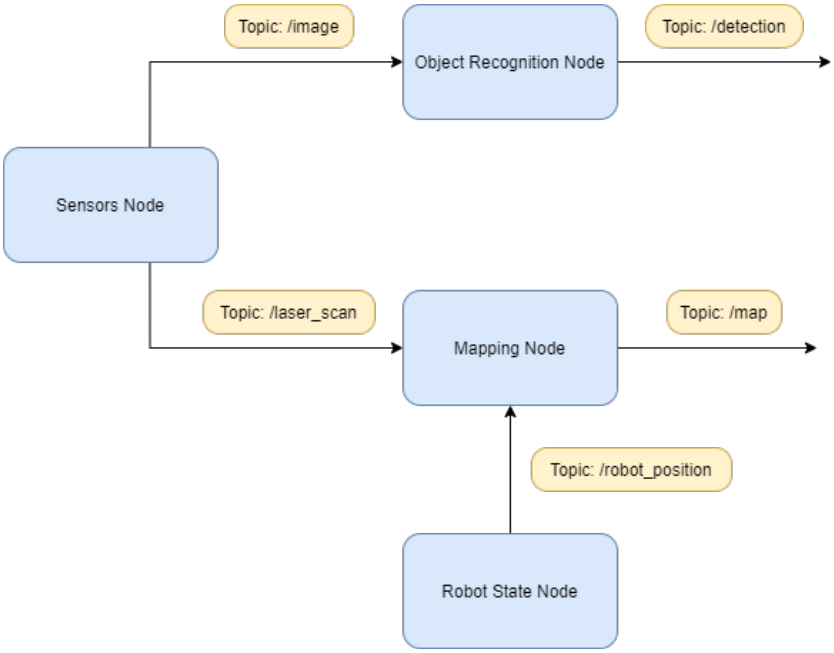

Figure 3.1: An example of ROS System

complete ROS system is **composed of many nodes**, sharing data with each other in order to achieve the determined robotic behavior.

This leads to **many benefits**, such as additional fault tolerance, since the failure of a single node does not spread to the others, reduced code complexity with respect to monolithic systems and makes it easier to use alternative implementation of a node, even if written in a language different from the one the node was originally developed with. In order to be **uniquely identifiable**, every single node has a **Graph Resource Name**, which is the identifier of the resource in the hierarchical naming structure of ROS, as well as a **type**, a combination of the package the resource belongs to and, in the node case, its executable name. A node can be written by using the **ROS Client Library**.

3.2.2 Messages

As anticipated earlier in this chapter, nodes can communicate with each others by publishing messages on specific topics. Messages are nothing more than a **data structure** that describes what kind of data that particular message will be used to deliver. Most primitive types (e.g. boolean, integer, floating point) are available, and it is also possible to define primitive types arrays as well as nested structures, similar to C structs. A **message definition** is stored in a **text file with *msg* extension** which is then kept in the msg subfolder of the related package. Messages,

exactly as nodes do, **follow the ROS naming conventions**, so in order to use a particular message we will have to refer to *nameOfThePackage/nameofTheMsgFile*. Additionaly, messages go through a **versioning based on a MD5 sum** of the msg file itself; two nodes can exchange a particular message only if both are using the same message type and the MD5 sum matches. Each message definition can include a special type called ***Header***, which includes metadata about the message like, for example, a timestamp. Some of these fields can even be automatically set by ROS itself, so developers are exhorted to use them. Figure 3.2 shows the definition of the LaserScan message, which main field is the *ranges* array; it contains all the data measured by a 2D LiDAR sensor.

```
1  std_msgs/Header header
2  float32 angle_min
3  float32 angle_max
4  float32 angle_increment
5  float32 time_increment
6  float32 scan_time
7  float32 range_min
8  float32 range_max
9  float32[] ranges
10 float32[] intensities
```

Figure 3.2: The definition of the LaserScan Message

3.2.3 Topics

Topics are **named buses** over which nodes can communicate, and were thought for **unidirectional streams of data**. Each topic is **strongly connected to the type of the message** that it is used to transmit. Nodes can either publish on or subscribe to a particular topic, in this way they do not have to care about who is they are communicating with and only focus on the data they receive and transmit. A single topic can have **multiple subscribers and multiple publishers** and a single node can publish on and subscribe to several topics at the same time. As long as the right message type is being used, everyone can subscribe and publish to a topic. The **default communication protocol is TCP**, but nodes can communicate even over UDP. The negotiation regarding the transport protocol is done by nodes themselves at runtime.

3.2.4 Services

Sometimes, though, a many-to-many unidirectional communication is not what one might need, especially when a **request/reply interaction** is needed (e.g. a node wants to know what the state of sensor is). That is when services come to aid. Services are **defined in files with *srv* extensions**, by default kept into the srv subfolder, in which two messages are defined: one for the request and one for the reply. A client can use a service by simply sending to another node a request message, and waiting for the reply. Services, exactly like topics, are tightly coupled with the messages they are used to exchange and follow the same naming schema. Services are **subjected to versioning** (as for messages, MD5 sum of the srv file) and therefore, client and server nodes can communicate only if the MD5 sums match.

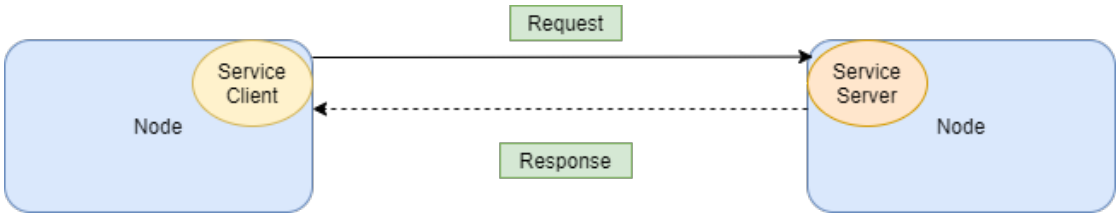

Figure 3.3: ROS Service

3.2.5 Actions

On occasions, it may happen that a service requested by a client takes quite some time to be completed and, by using ROS services, the client **cannot either cancel the request nor know the state of it**. Actions were thought exactly to address this problem: a client requesting an action can interrupt it whenever it wants and it will constantly receive feedbacks regarding its completion percentage. An action can be easily **described into a file with *action* extension**, into which a few messages are defined: **goal, feedback and result**. The **goal message** is the one initially sent by the requesting client to the server, and it **describes the final state** that it wants the system to reach; the **feedback** message is sent periodically from the server to the client, containing **updates on the requested action state**; lastly, the **result** message is the final message to be sent and it contains the **final state** of the system. An example use of a ROS action can be related to robot movements: the user may want to make the robot move from a point A to a point B, so, after issuing the movement command, it will receive updates on the state of the robot until it reaches the destination. Needless to say, actions follow the same naming convention as all other ROS resources presented so far.

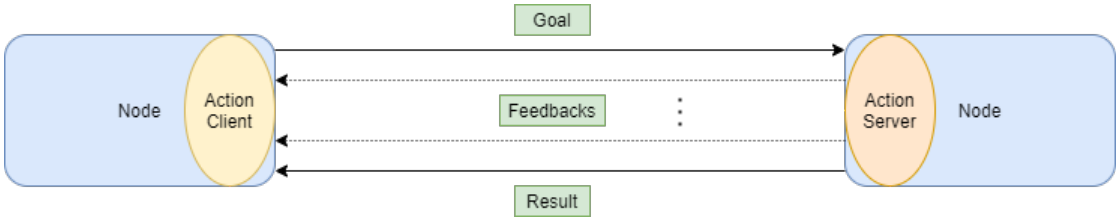

Figure 3.4: ROS Action

3.3 ROS1 and ROS2

After many years developing ROS, the team behind it gained a lot of experience and learnt what kind of important features were missing or had to be improved, and what had to be removed. To apply all that knowledge onto ROS would have meant making **massive changes** in the code and consequentely introducing the possibility of causing **important unstability issues**. That is why the team decided to start from scratch and **develop a brand new ROS, ROS2**[13]. In this thesis, ROS2 has been used since ROS1 is reaching its end of life and no new functionalities will ever be added. The only updates the last version of ROS1 (ROS Noetic) will get is the support for Python3, which ROS2 uses by default.

The followings are some of the **main differences** between the two:

- **ROS API:** ROS1 API supported only **C++ and Python** and each one of them had **completely independent libraries** built from the ground up, respectively *roscpp* and *rospy*. That means that some features could be added to one library and not the other and also that the API was not necessarily the same in the two cases. ROS2 tries to solve this problem by **adding a layer**: *rcl*. rcl is a **base library** written completely in C that contains all the ROS2 basic features. Other languages libraries are **built on top of that**. In case a **new feature** needs to be added, the majority of the work will be done **only on rcl**, while the other languages just need to **add the binding** to that new feature. This leads to having similar API for the supported languages (currently C++, rclcpp and Python3, rclpy), an easier time creating other languages libraries and new features will spread much faster in all the supported languages.

- **Cpp and Python Compatibility:** for most of its life ROS1 supported **only Python2 and C++ 98**. In its latest version it added support to Python3 and C++ 11/14, even though the latter is not guaranteed to work properly without breaking some dependencies. On the other hand, ROS2 supports by default **Python3 and C++ 11/14**, and support for **C++ 17** is on the way.

- **Node Writing Convention:** ROS1 **did not have specific rules on how**

to write nodes, so that made really difficult for developers not only to write clean and organized code, but it made it almost impossible to share it, since the same node implementation could be completely different if written by two different developers. ROS2 solves that by **introducing a convention** that must be followed.

- **ROS Master:** in order for a ROS1 system to work, a **ROS Master** node was needed. It basically acted as a **DNS server** for all the other nodes, allowing them to discover each other and thus communicate. ROS2 eliminates the need of that: every single node can automatically discover other nodes using **broadcast traffic**, and this led to the possibility of creating fully distributed systems.

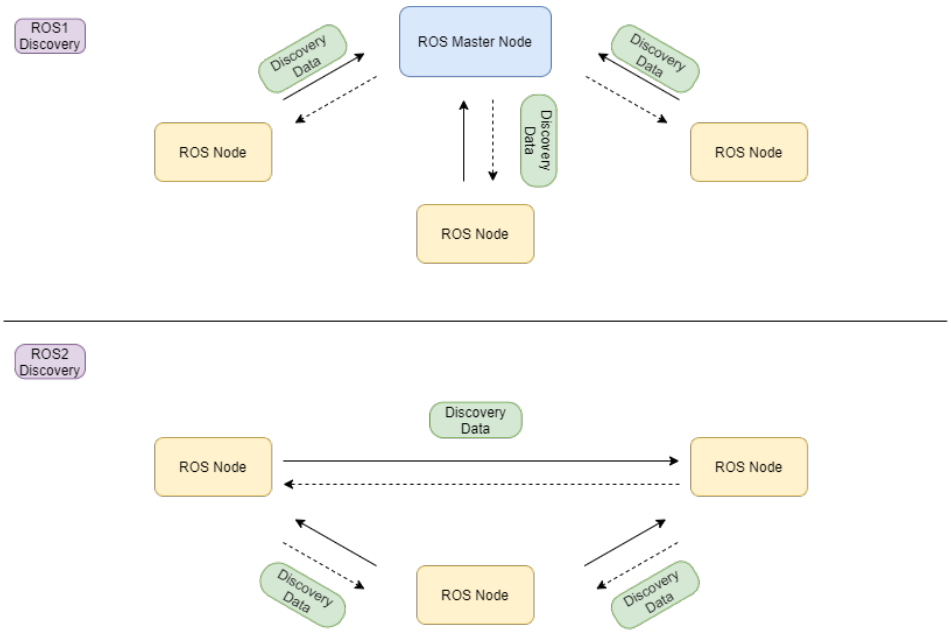

Figure 3.5: Discovery Mechanism Comparison

- **Services:** ROS2 services are **asynchronous**, that means that a callback function can be added in the node that will be triggered as soon as the server sends the reply back, thus allowing the main thread **not to be stuck on waiting** for it.

- **Actions:** in ROS1 actions were **not a core functionality**, they were just added to solve some of the problems encountered with services being synchronous and not having a feedback cancellation mechanism. ROS2 **added them to its core**, and they use a combination of topics and services.

- **Quality of Service:** by means of QoS settings, ROS2 allows the developer to define how nodes should **handle interactions with other nodes** (e.g. is it okay to lose a message?). Nodes that talk with each other, need to have compatibile QoS settings in order to properly work.
- **OS Compatibility:** ROS1 is compatible **exclusively with Ubuntu**, while ROS2 added support to **MacOS and Windows 10**, making it more easily usable in many more environments.
- **Networking:** while ROS1 uses a custom made network middleware, ROS2 is based on already existing solutions, specifically FastDDS an implementation developed by eProsima of DDS (Data Distribution Service, more is told about it later).

Chapter 4

Initial Investigation and Requirements Analysis

The first step toward the final goal of this thesis was gathering information about the **hardware**, with special focus on the CPU and the sensors, that is nowadays available and equipped on robotic systems and the **time constraints** that an autonomous system (therefore, real-time) should respect. In particular, a **TurtleBot 3 Waffle Pi** was kept in consideration, since this was the robotic platform available in the **PIC4SeR laboratory**, which this thesis was written in collaboration with.

The PIC4SeR, acronym for **PoliTO Interdepartemental Center For Service Robotics**, is, as the name suggest, a service robotics research center born from the collaboration of **five different departments of the Politecnico di Torino**: **DET** (Electronic and Telecommunication Department), **DAUIN** (AUtomatic and INformatics Department), **DIMEAS** (Mechanical and AeroSpatial Engineering Department), **DIATI** (Environment, Land and Infrastructure Department) and **DAD** (Design and Architecture Department).

By definition, a **service robot** is a robot that **helps humans to perform useful tasks**; tasks that can space from simple but repetitive house chores, to helping mountain rescue teams to localize and save people buried under an avalanche. The PIC4SeR goal is to **support the service robotics market**, which currently is in a rapidly expanding phase, by providing it with **innovative solutions** achievable only in research centers where a real **multidisciplinary approach** is possible and therefore help it develop faster.

The sensors, particularly the cameras, that were considered in this phase are **real sensors** used on board of either robotic platforms, or autonoumous driving vehicle (Tesla, specifically) in order to get as close to a **real application** as possible.

In this chapter, the Turtlebot platform will be briefly presented, as well as the LiDAR and the Depth Camera technology.

Finally, the collected data and the consequent constraints will be introduced.

4.1 Turtlebot

Turtlebots are **low-cost, entry level robotic platforms running on open-source software**. The first Turtlebot was developed at Willow Garage (the same technology incubator in which ROS was initially developed in) in late 2010. They consist of a mobile base, distance sensors (either 2D and 3D) and a Single Board Computer. They are specifically designed to be **easy to build and to use**, without the need to utilize very particular tools. Being their core technology SLAM (Simultaneous Localization And Mapping) and navigation, they are particularly **suitable for home service robots**: they can do things like autonomously mapping and driving around a room, or follow someone as they walk around the house. There is also the possibility to use an **arm attachment**, specifically thought and designed for enabling object manipulation.

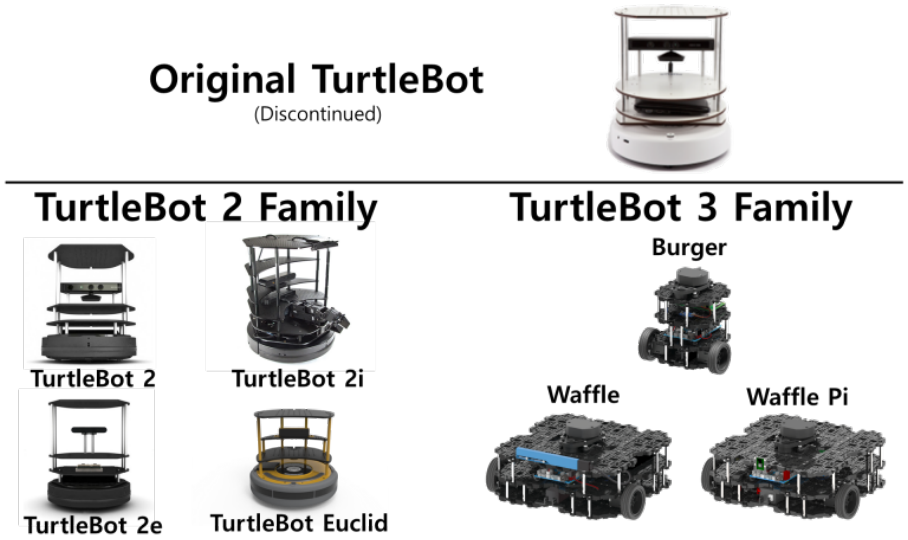

Figure 4.1: All Turtlebot iterations

Throughout time, **three different versions** of Turtlebots came out, each one in **different flavors** (except for the original Turtlebot):

- **Turtlebot:** the original Turltebot developed at Willow Garage. Especially thought for ROS deployment, it was built on top of Create, the iRobot's research robot. It mounted an Asus 1215N dual core laptop, a 3000 mAh battery, a Kinect sensor and a Turtlebot power board with gyroscope. It could easily be assembled by means of single allen wrench.

- **Turtlebot 2:** it came out in two different editions, TurtleBot 2e and TurtleBot 2i. They both were created on top of a YUJIN Kobuki with addition of a 2200 mAh battery pack, a Kinect sensor, an Asus 1215N dual core laptop and a fast charger. In addition, the 2i edition had native support for robotic arms. Exactly as in the case of its predecessor, the Turtlebot 2 was very easily assemblable by using a single allen wrench.

- **Turtlebot 3:** the most recent Turltebot iteration, it includes three edition: Burger, Waffle and Waffle Pi. As their predecessors, their hardware is built on top of a common starting point, but it differentiates, among other things, in what are, for the scope of this work, key aspects: onboard CPU and available sensors. Talking about the Single Board Computers, Burger and Waffle Pi both use a **Raspberry Pi 3 B/B+** (quad-core, 64-bit working at, respectively, 1.2GHz and 1.4GHz with 1Gb of RAM), while the Turtlebot 3 Waffle operates on a **Intel Joule 570x** (quad-core, 64-bit working at 1.7GHz with 4Gb or RAM). When it comes to the installed sensors, all three of them are equipped with the same **2D LiDAR** sensor (specifically, the LDS-01) and, while the Turltebot 3 Burger has no camera sensor, the other two do: Turtlebot 3 Waffle is provided with an **Intel Realsense R200**, a RGB Depth camera, while the Waffle Pi edition uses the **Raspberry Pi v2.1 camera** sensor, which is a simple RGB camera.

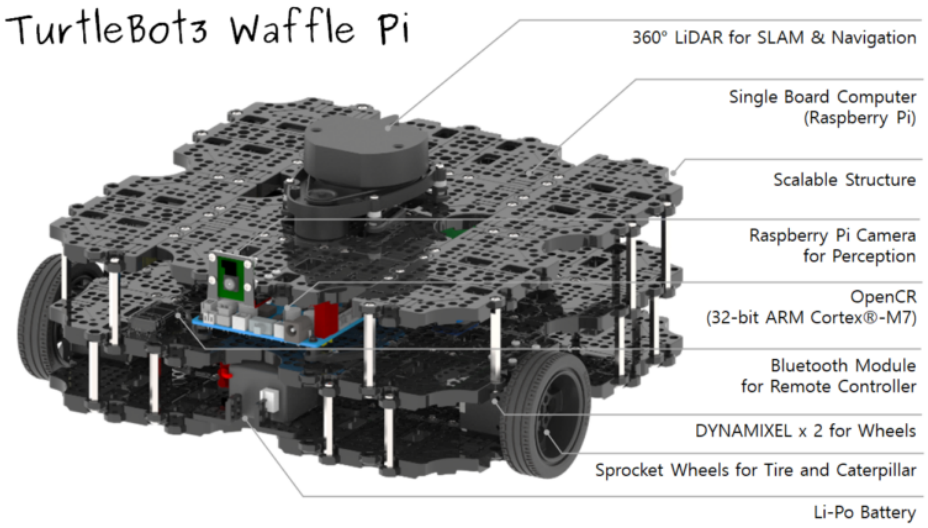

Figure 4.2: Close-up on the Turltebot 3 Waffle Pi

4.2 LiDAR

LiDAR, acronym for **Light Detection And Ranging**, is a method used mainly for detecting the presence of obstacles and measuring distances. Its working principles are basically the same as a **traditional radar**, but, instead of using radio waves to do measurements, it uses **ultraviolet, visibile or near infrared laser light** (the wavelenght is chosen depending on the target of that particular sensor): the LiDAR sensor illuminates its surrounding and by measuring the time that the reflection (more specifically, diffuse reflection, as in contrast to specular reflection) takes to get back to the source it can measure distances between the sensor and the measured object and even make a 3D representation of it.

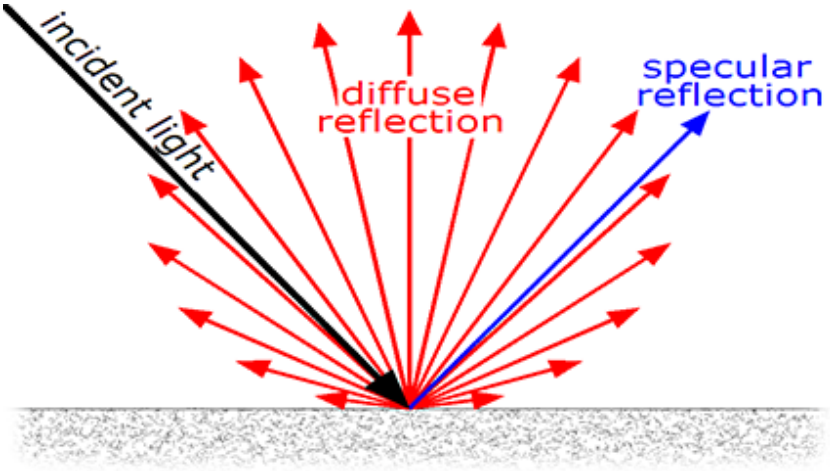

Figure 4.3: Diffuse vs Specular Reflection

The LiDAR techonology finds application not only in robotics and autonomous driving, but in a **wide variety of fields** like astronomy (where, for example, is used, with the help of reflectors put on the Moon's surface, to measure with a millimiter precision the distance between the Earth and its satellite), archaeology, geology and many more.

LiDAR sensors can come in three different variants, as illustrated in the following list:

- **LiDAR 1D:** one-dimensional LiDAR are the simplest form of LiDAR sensors available and their only actual use is as **distance measurement** device. They are composed of a single fixated laser beam which, when pointed toward a reflective surface, allows the sensor, by means of the time it takes the light to go from the source to the surface and back, to accurately measure how distant the object in question is.

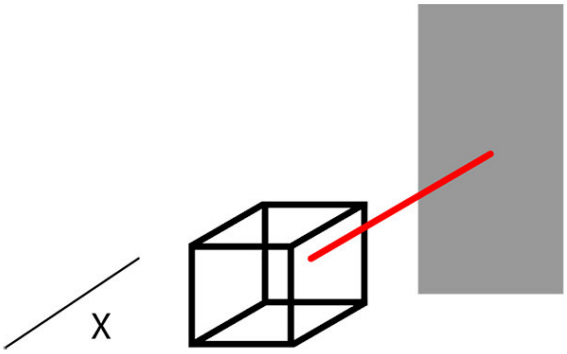

Figure 4.4: 1D LiDAR

- **LiDAR 2D:** two-dimensional LiDAR are nothing more than a one-dimensional LiDAR in which the laser beam and the sensor are **rotating around one axis**. This allows the device to measure not only the distance from reflective surfaces, but an **angle** as well. Measurements are taken at a fixated rate, for example, the LiDAR available on board of Turtlebot 3 platforms, takes a measurement **once every one degree angle**, so to scan the whole surrouding environment it needs 360 measurements. This kind of LiDAR can be used, not only to measure distances, but also to perform simple 2D mapping.

  It is important to notice, though, that this variant is only able to detect obstacles that are put at the **same height** as the sensor, everything above and below it will be lost. Therefore, a single two-dimensional LiDAR is **not enough** for an autonomous driving system, even at very low speeds like the ones achievable by Turtlebots.

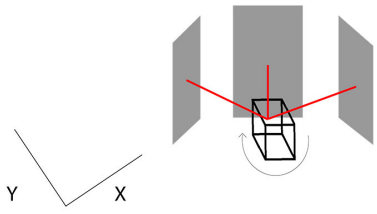

Figure 4.5: 2D LiDAR

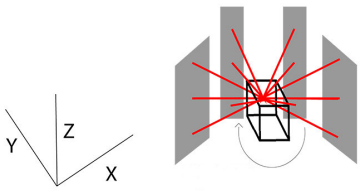

Figure 4.6: 3D LiDAR

- **LiDAR 3D:** the last LiDAR variant is a further evolution of the previous

one. Multiple laser beams and sensors are put at **different heights** in order to be able to obtain data related to the third dimension, previously impossible to collect because of the reason explained above. This kind of sensor by itself offers already enough data for a basic autonomous driving system.

4.3 Depth Camera

Digital cameras, like the ones nowadays available in every smartphone, output 2D images composed of millions of pixels; each one of those pixels contains some data that, usually, is associated with and **RGB value**, which describes the amount of red, green and blue (thus RGB) present in that particular pixel. Values can oscillate between 0 and 255, so, for example, RGB(255, 255, 255) is pure white and RGB(0, 0, 0) is pure black. The same goes for depth cameras, but, whereas standard digital cameras pixels hold the previously mentioned RGB values, their pixels hold another kind of data, called **depth**. This value measures the **distance between the camera and the photographed objects**. Depth information is very useful to the vehicle; thanks to it, it can determine how distant nearby ostacles actually are and not only, by means of object recognition algorithms, what those ostacles are. Some depth cameras also have an RGB sensor, so their pixels hold RGB-Depth (RGBD) data.

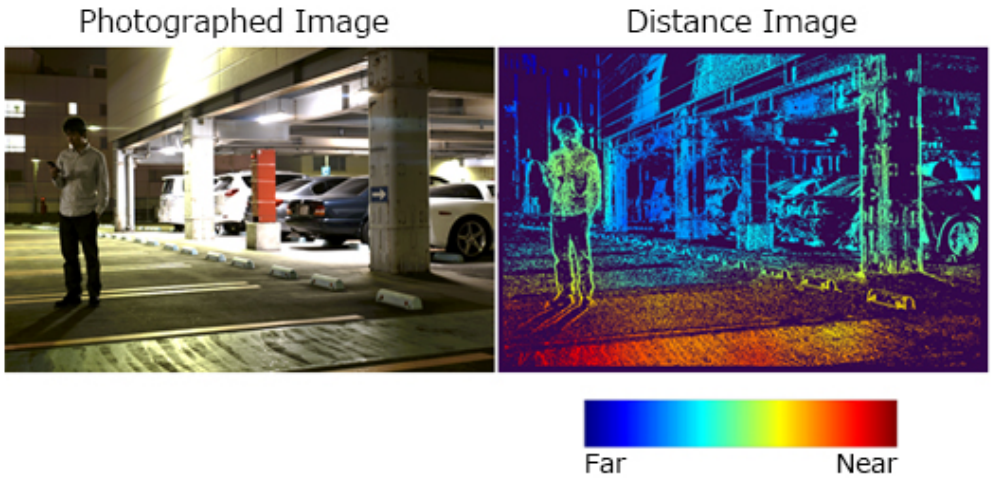

Figure 4.7: RGB vs Depth Image

The depth information can be measured in different ways, therefore **multiple**

types of depth camera exist, each one with its own advantages and disadvantages; some of those will be now briefly introduced:

- **Structured Light Cameras:** this type of depth cameras works by emitting light, usually infrared, in a **specific visual or time pattern** (sometimes, a combination of the two). Since the emitted pattern is known, the changes in the perceived reflected pattern allow the camera to compute the distance value between itself and the illuminated things for every pixel. Due to how they work, this kind of cameras works better in **close environments and at close distances**. Also, since they use light in the infrared spectrum, they can be disturbed by other cameras or infrared emitting devices.

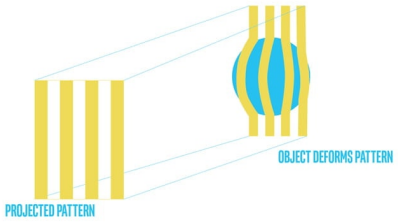

Figure 4.8: Structured Light

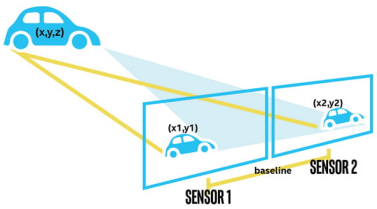

Figure 4.9: Stereo

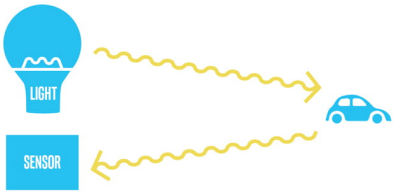

Figure 4.10: Time of Flight

- **Stereo Depth Cameras:** those cameras basically **mimic how human eyes work**: two sensors placed a small distance apart take two images and compare them; since the distance between the sensor is known, the depth value can be calculated. The measurable distance is **directly proportional to how much the two sensors are distant from each other**: the more they are far apart, the bigger it is. These cameras can work with **any kind of light**, but they are usually equipped with an infrared light emitter in order to both increase the precision of the measurement and also to help them in case of scarce illumination. Contrarily to what happens with structured light cameras, they are not bothered by infrared noise and multiple cameras can be used in the same space withouth any interference. They are suitable for **outdoors applications**.

- **Time of Flight Cameras:** this last type of cameras works by measuring how long an emitted beam of light takes to get back to the source. By sweeping this light beam all over their surroundings they can easily collect depth data. Previously introduced LiDAR sensors **work in the same way**, as they are, as a matter of fact, a particular kind of time of flight device, which uses laser light. The measurable distance depends on the **power of the light emitting source and the emitted light wavelenght**. The biggest disadvantage of these cameras is that they are **very susceptible to other light sources**, since every light that hits the sensor but it is not emitted by the camera itself, will degrade the quality of the final measurement.

4.4 Specifications and Constraints Analysis

As anticipated in the beginning of the chapter, in this phase I collected some data related to cameras used in the robotic and autonomous driving field in order to get a better general idea on especially the order of **magnitude of the throughput** the network should deal with: at each moment the network must be able to transfer the data collected by the sensors to the data center and back and it has to do so in the shortest amount of time possible. Taking in consideration that the data must also be elaborated, it is evident that the network has to be able to deal with quite an amount of stress. Lastly, I did a brief estimation of possible **maximum latency** times that a system constituted of a 2D LiDAR and/or a depth camera should concern itself with.

4.4.1 Camera Specifications

The estimations about the throughput have been calculated by assuming that all the cameras are working at their **maximum performances**, even though this is not necessarily true in real applications (usually, lower resolutions are used, especially when it comes to RGB only images), since the goal was obtaining worst case scenarios. The considered cameras are:

- **Intel Realsense R200:** this is the camera available on the Turltebot 3 Waffle. It is a **RGB Stereo-Depth camera**, with the depth sensor working at a maximum resolution of **480x360p@60FPS**, and the RGB sensor working at **1920x1080p@30FPS**.

- **Intel Realsense R435**: an **updated model** of the previous camera. The depth sensor now operates at a maximum resolution of **1280x720p@90FPS**, while the RGB sensors still operates at the same resolution and framerate as on the other model.

- **AR0132AT**: an **RGB only** camera. It has been considered since it is the sensor availble on **Tesla vehicles**. It works at a resolution of **1280x960p@45FPS**.
- **Raspberry Pi Camera Module v2.1**: the **RGB camera** equipped on the Turtlebot 3 Waffle Pi. It runs at a maximum resolution of **1920x1080p@30FPS**.

In table 4.1, further data can be found. Follows a brief explanation of its content:

- **Model:** the model of the camera sensor and the platform it can be found equipped on.
- **Max Resolution:** the maximum operating resolution of the available sensors. In particular, the notation RAW10/12 means that every pixel captured by that sensor contains 10/12 bits of data.
- **Max Framerate:** how many frames per second the sensor can capture.
- **Viewing Angle:** the maximum Vertical and Horizontal viewing angle of the camera.
- **Range:** related only to depth sensors, it represents the minimum and maximum distance it can measure.
- **Type:** the types of the sensors available on the device.
- **Frame Dimension:** approximate dimensions of a single frame captured by that particular sensor. It is calculated by multiplying the bit depth and the maximum resolution.
- **Bit Depth:** amount of data stored in each pixel
- **Throughput:** an estimated throughput, calculated by multiplying the framerate with the frame dimension.

Model	Max Resolution	Max Framerate	Viewing Angle	Range	Type	Frame Dimension	Bit Depth	Throughput
Intel Realsense R200	480x360 Depth	60	V46° ± 5° H59° ± 5°	0.5 - 10m	Stereo Depth and RGB	~337KB	16bits	~158Mbps
	1920x1080 RAW10	30	V43° ± 2° H70° ± 2°			~2.5MB	10bits	~593Mbps
Intel Realsense R435	1280x720 Depth	90	V90° ± 3° H63° ± 3°	0.2 - 10m	Stereo Depth and RGB	~1MB	10bits	~790Mbps
	1920x1080 RAW10	30	V69.4° ± 3° H42.5° ± 3°			~2.5MB	10bits	~593Mbps
AR0132AT	1280x960 RAW12	45			RGB	~1.8MB	12bits	~633Mbps
Raspberry Pi Camera Module v2.1	1920x1080 RAW10	30	V62.2° H48.8°		RGB	~2.5MB	10bits	~593Mbps

Table 4.1: Cameras Specifications

By looking at the throughput values (the lowest one is approximately **158Mbps**), it was immediately clear that using those cameras at their native resolutions and framerates was not feasible and thus some sort of **compression** of the camera feed was mandatory.

4.4.2 Latency Constraints

By definition, for a **real-time signal elaboration application** to be rightfully called so, it must respect a fundamental constraint: **the time it takes for a single signal sample to be analyzed and elaborated must be lower than the sampling period** (the time the sensor takes to produce a single sample of the signal).

Taking in consideration a depth camera (particularly the R200) at a resolution of 320x240p and a framerate of either 60FPS or 30 FPS, and a LiDAR 2D, an initial time constraint has been calculated, with the sampling period computed as the inverse of the frequency at which the sensor operates.

The result can be seen in the following table:

Sensor	**Resolution**	**Sampling Period**	**Frame Size**
Depth Camera	320x240@30FPS	∼33ms	∼150KB
	320x240@60FPS	∼17ms	
LiDAR 2D	1 measurement per degree (360 per sampling period)	∼200ms	∼3KB

Table 4.2: Latency Constraints

It is evident that the **hardest contraints** are set by the camera sensors that at 30FPS and 60FPS have respectively a sampling period of ~33ms and ~17ms: in that amount of time the system must transmit 150Kb of data, elaborate it and send it back to the autonomous platform.

By taking in consideration the time constraint imposed by the depth camera working at 30FPS (~33ms), table 4.3 illustrates how that influences the system from a network point of view. It is clearly noticeable that, even when the system has a gigabit connection available, the time left to the data center to elaborate the received information and send it back to the system is very short.

In table 4.4 it can be seen what those two time constraints mean for the autonomous system by taking in consideration Italy's current speed limits.

Data Center Location	**Average RTT From Data Center**	**Connection Speed**	**Frame Size**	**Transfer Time**	**Time Left for Elaboration and Response**
Edge	∼4ms	100Mbps	150KB	∼12ms	∼17ms
		1000Mbps		∼1ms	∼28ms
Cloud	∼10ms	100Mbps	150KB	∼12ms	∼11ms
		1000Mbps		∼1ms	∼22ms

Table 4.3: Real Time Constraints Applied to the Network

Speed	**Distance Traveled in ∼33ms**	**Distance Traveled in ∼17ms**
50 Km/h	∼0.46m	∼0.24m
90 Km/h	∼0.83m	∼0.45m
110 Km/h	∼1.00m	∼0.52m
130 Km/h	∼1.20m	∼0.61m

Table 4.4: Latency applied to Italy's Speed Limits

Chapter 5

Running ROS2 onto Kubernetes

Aside from both ROS2 and Kubernetes being the go-to choice in their respective fields, there is another reason why it has been decided to make ROS2 work on top of Kubernetes: the concept of **Kubernetes everywhere**. Into a Kubernetes cluster we are able to run many different applications, even very different in nature, at the same time (e.g., a ROS2 system and a web server) and make them interact with each other. By means of the interaction among applications in the same cluster, it is possible to build more complex applications than the ones that are deployed in each single Pod. The goal is basically to use Kubernetes as if it were an **operating system**, even though it is not.

Many applications are already easily deployable and usable in a Kubernetes environment, but this is not the case of ROS2. A ROS2 system is usually deployed on a **single LAN** and, more often than not, on a **single machine**, therefore, while being composed of many independent nodes, it is not really thought for being used in a distributed environment such the one Kubernetes offers.

In this chapter, it is going to be illustrated the process by which a ROS2 system was made runnable onto a Kubernetes cluster, in particular how the incompatibility between the ROS2 discovery mechanism and Kubernetes was solved.

5.1 Containerizing ROS2

ROS2 is, by design, a **very easily containerizable application**. Since a ROS2 system is nothing else than a **collection of communicating nodes** running independent processes, it is simple enough to imagine each one of these running in a **separate Kubernetes Pod**, therefore in a different container. In order to containerize a ROS2 node, there are two main alternatives: write a Dockerfile

starting from a Docker Ubuntu image, or write one starting from one of the already available ROS2 Docker images.

Starting from the Ubuntu Docker image will give the user more freedom, since it allows to choose exactly what **ROS2 version** has to be installed and with what packages/tools. Installing ROS2 is actually quite a simple task; by just following the installation guide on the ROS2 website (available for several operating systems), it is possible to either install it via Debian packages, or by dowloading the sources and building it ourselves. In a matter of minutes, ROS2 will be installed and almost ready to operate; almost beacuse one last step is needed before being able to run a node: **sourcing the ROS2 workspace**. By doing so, the current terminal session is populated with the **right settings and environment variables**, allowing the user to invoke ROS2 tools directly from there. Now, by executing the command *ros2 run <package_name> <executable_name>* a ROS2 node can be started.

New packages can be added to the system; they can either be **installed as pre-built packages** (mainly ROS2 official packages), or built using the **colcon tool**. By means of a *package.xml* file available in the root folder of every ROS2 package, it can automatically take care of the task of building the required package along with the **needed dependencies** (if available in the system). Another very useful tool to build packages is *rosdep* (stands for ROS Dependencies); *rosdep* uses the same *package.xml* file as colcon in order to **gather information** about all the required package dependencies and then will proceed to install them into the current ROS2 workspace, if not already present. After building a node from source, it is necessary to source the new workspace in order to be able to use it.

If, on the other hand, the ROS2 Docker image alternative is chosen, the user will not have to care about installing ROS2 itself, since it already comes preinstalled in the image itself, but only about **building and installing necessary packages** and start the node. In both cases, though, the endpoint field of the Dockerfile should be populated with the command that **spins the ROS2 node**. Doing so allows, as explained above, the node to start as soon as the container is up and running.

5.2 ROS2 Discovery Problem

As explained above, containerizing ROS2 nodes is quite a simple task, and the same goes for running those Docker images into Kubernetes: all that is needed is to create a YAML file to describe either a Kubernetes pod or deployment, and specify the Docker image's name in the image field of the configuration file. Kubernetes will then automatically pull the requested image, create the desired resource and run the specified image.

Despite this, ROS2 is **not completely compatible with Kubernetes** just as

it is, the problem residing in its **discovery mechanism**: ROS2 takes advantage, as explained in chapter 3, of **multicast traffic** in order to allow nodes to discover each other; broadcast traffic, though, is **not compatible with all Kubernetes CNIs**. Since it is not realistically possible to assume that every Kubernetes cluster that ROS2 has to be deployed onto will be under full control by part of the user, it is mandatory to find a way to force the discovery protocol to use **unicast traffic**, in this way making it fully compatible with whatever CNI may be running on the available Kubernetes cluster.

5.2.1 FastDDS

The whole discovery mechanism of ROS2 is based on the ***Data Distribution Service*** framework and its wire interoperability protocol, ***Real Time Publish Subscribe*** (RTPS), in particular on the eProsima FastDDS implementation[14]. The Data Distribution Service (DDS) is a connectivity framework defined and maintained by the Object Managenent Group, a computer industry standards consortium which main goal is to provide specifications of portable and interoperable object models; DDS aims at providing a dependable, high-performance, interoperable, real-time, scalable data exchange by using a publish-subscribe pattern. DDS itself takes care of everything related to the data transfer, because of this applications using it do not have to care about having information regarding other participants. On top of that, RTPS ensures that information published on a topic can be consumed by one or more subscribers running on either the same DDS implementation or a different one. It is designed to be able to run over multicast, best-effort, connectionless transports, like UDP. Some main features are:

- **Communication Policies:** configurable best-effort and reliable publish-subscribe communication policies for real-time applications.
- **Plug and Play:** new applications are automatically discovered as soon as they enter the network and removed when leaving it.
- **Type Safety:** to prevent programming errors from compromising other nodes.
- **Modularity:** to allow basic devices to implement just a part of the protocol, but still be able to interact with the network.
- **Fault Tolerance:** it enables a distributed discovery mechanism, therefore the network will not have a single point of failure .

Taking all this in consideration, it is clear that the solution to the encountered compatibility problem has to be searched for in the DDS world.

5.2.2 Discovery Server

The solution was provided by eProsima itself (i.e., the same developers of the DDS implementation used by ROS2), for situations in which the basic multicast-based mechanism was not suitable: the **Discovery Server** [15]. They extended their DDS implementation with a **client-server mechanism**, much like the one used by ROS1 (it used a master node for metatraffic); the Discovery Server will **receive all the metatraffic** coming from the network participants and **distribute** it to each one of them. Multiple instances of a Discovery Server can be run on the same network, in order to provide some resilience since the server is a single point of failure. It is also possible to multiple Discovery Servers, each one responsible for a different client subset, and have them exchange the collected metatraffic so that every client has knowledge about topics even if they are not published and subscribed by nodes in their subset. Figure 5.1 briefly compares a ROS2 topology that uses the standard multicast discovery mechanism, with one the uses the Discovery Server one.

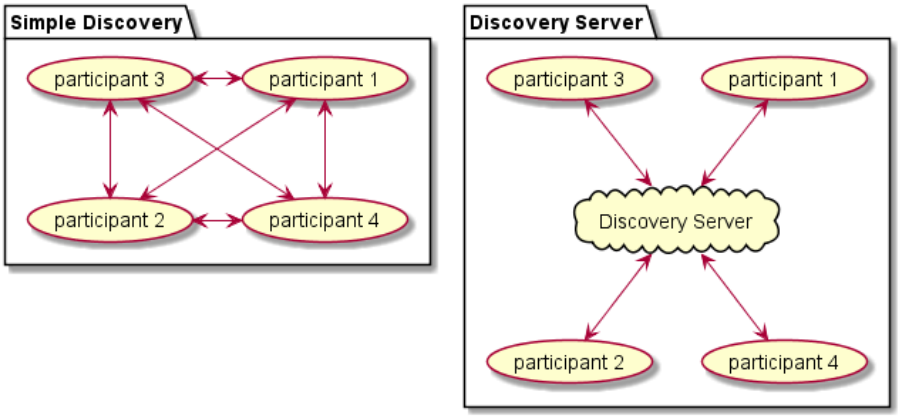

Figure 5.1: Default Discovery and Discovery Server Mechanisms

The Discovery Server **does not come pre-installed** with FastDDS (planned for future FastDDS updates), so anyone who wants to use its functionalities has to build it on their computer. The procedure is fairly easy and well explained in the related GitHub repository page, the only requirement is that the application that needs to take advantage of the Discovery Server has to be running on top of **FastDDS 2.0.2**, or newer. Summarizing, the procedure consists in cloning the Discovery Server GitHub repository (*https://github.com/eProsima/Discovery-Server*) along with all its dependencies and running the colcon tool in the root folder; after that, the workspace has to be sourced by means of the appropriate script (in case bash is being used, the script will be the one named *setup.bash*)

generated by the colcon building process. By doing so, the current terminal session will be populated with the right environment variables and settings so that the user is able to run the Discovery Server directly from the terminal itself.

5.2.3 Configuration Files

Since this is not the default discovery mechanism, all the applications that want to take part in the network **must be configured** in such a way that forces the underlying FastDDS to use the new client-server mechanism. A configuration is needed for the Discovery Server as well, to set things such as the **IP address and port** on which it will listen for metatraffic and its unique identifier. In order to make the configuration an easier process and especially not bound to hardcoded settings, FastDDS uses **XML configuration files**; as soon as a FastDDS instance starts running, it will look in the running directory for a file called *DEFAULT_FASTRTPS_PROFILES.xml*, if there is no such file it will try to use the file pointed to by the environment variable *FASTRTPS_DEFAULT_PROFILES_FILE*, if any; in case neither of those two alternatives is a valid option, it will start with default settings.

In the following pieces of code, a configuration file for a Discovery Server and a client will be shown and briefly explained.

The main fields in the Discovery Server configuration file are:

- **Prefix:** an unique ID used to identify the server in the network. Usually FastDDS automatically provides those, but it is necessary to explicitly set it in the case of the Discovery Server, since it has to be used into the clients configuration files.
- **Address:** the IP address on which the server will listen to for incoming metatraffic. It is set to 0.0.0.0 in order to make the server listen on all interfaces.
- **Port:** the port onto which the will listen for incoming connections. The same port has to be set into the clients configuration file.
- **Discovery Protocol**: set to SERVER. Tells the underlying FastDDS that the application running on top of it will act as a Discovery Server

```
<?xml version="1.0" encoding="UTF-8" ?>
<DS>
  <servers>
    <server prefix="4D.49.47.55.45.4c.5f.42.41.52.52.4f" persist="SERVER" profile_name="UDP
    ↪  server" name="server">
      <ListeningPorts>
        <locator>
          <udpv4>
            <port>6000</port>
          </udpv4>
        </locator>
      </ListeningPorts>
    </server>
  </servers>
  <profiles>
    <participant profile_name="UDP server">
      <rtps>
        <prefix>
          4D.49.47.55.45.4c.5f.42.41.52.52.4f
        </prefix>
        <builtin>
          <discovery_config>
            <discoveryProtocol>SERVER</discoveryProtocol>
          </discovery_config>
          <metatrafficUnicastLocatorList>
            <locator>
              <udpv4>
                <address>0.0.0.0</address>
                <port>6000</port>
              </udpv4>
            </locator>
          </metatrafficUnicastLocatorList>
        </builtin>
      </rtps>
    </participant>
  </profiles>
</DS>
```

Figure 5.2: Configuration File for a UDP Discovery Server

When it comes to clients, the most notable fields in the configuration file are:

- **Prefix:** the unique ID of the Discovery Server that the client will have to contact to exchange metatraffic.
- **Address:** actual IP address of the Discovery Server. That is where the metatraffic has to be sent to.
- **Port:** the port onto which the client has to contact the Discovery Server.
- **Discovery Protocol:** set to CLIENT. Tells the underlying FastDDS that the application running on top of it will act as a client.

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles>
  <participant profile_name="UDP client" is_default_profile="true" name="client">
    <rtps>
    <builtin>
      <discovery_config>
        <discoveryProtocol>CLIENT</discoveryProtocol>
        <discoveryServersList>
          <RemoteServer prefix="4D.49.47.55.45.4c.5f.42.41.52.52.4f">
            <metatrafficUnicastLocatorList>
              <locator>
                <udpv4>
                  <address>10.109.90.228</address>
                  <port>6000</port>
                </udpv4>
              </locator>
            </metatrafficUnicastLocatorList>
          </RemoteServer>
        </discoveryServersList>
      </discovery_config>
    </builtin>
    </rtps>
  </participant>
</profiles>
```

Figure 5.3: Configuration File for a UDP Client

5.3 Discovery Server Test Demo

The next step was to **integrate this discovery mechanism into Kubernetes** and test if it behaved properly. In order to do so, the Discovery Server had to run inside the same Kubernetes Cluster as the ROS2 application; a Docker image of it has been built, by simply creating a Dockerfile that, starting from a Ubuntu Docker image, installs FastDDS and builds the Discovery Server as described previously. As discussed above, every FastDDS application which has to use the Discovery Server mechanism needs a configuration file; passing a file to a Kubernetes pod is easily done by means of a **Configmap** and by combining that with the previously mentioned *FASTRTPS_DEFAULT_PROFILES_FILE* environment variable, ROS2 nodes running inside a Kubernetes Pod can access the needed configuration file and thus use the new discovery mechanism.

The actual test has been conducted by using an already existing ROS2 demo deployment, the dummy robot. It consists of four different ROS2 nodes, initially designed to be deployed onto the same machine:

- **dummy_map_server:** it constantly publishes an empty map with a periodic update

- **dummy_laser:** continuously publishes fake laser scans.
- **dummy_joint_state:** publishes fake joint state data.
- **robot_state_publisher:** after parsing a URDF (**Universal Robot Definition Format**; used to describe a robot and its components in an XML fashion) file, it listens to the incoming joint states published by the *dummy_joint_state node*. It elaborates them and publishes transforms data for the example robot that can be visualized by means of *Rviz*, a GUI ROS2 topic visualizer.

The first thing I did was creating a separate Docker image for each one of these nodes, in order to be able to run them separately into the Kubernetes cluster; after that the Discovery Server was deployed, along with a Kubernetes Service to expose it, in order to have a **static IP** to which the other nodes can send traffic to (the same IP which has to be inserted in the address field of the client configuration file). At this point, all the other nodes can be started. As anticpated, by means of a Rviz instance, a GUI topic visualizer tool that comes installed with ROS2, running inside a Pod in the same cluster (which is also hosting a noVNC session, to provide a graphic environment to a Pod), all the **published topics can be visualized** exactly as showed by the original dummy robot demo deployment guide, thus meaning that the discovery server **works as planned**. Two failure scenarios have also been tested:

- **Failure of the Discovery Server Pod:** if the Discovery Server should for any reasons crash, its Pod will **automatically be deleted and recreated** by Kubernetes. This does not lead to any discovery problem, since, after an initial transition period, all the discovery data will be obtained again thanks to heartbeat messages sent by clients to the server.
- **Failure of one or more clients:** if one or more client nodes should fail, the server will remove their entries after a short period of time. As soon as the failed Pods are recreated, the **discovery phase will be repeated** making the nodes effectively part of the network again.

In figures 5.4 and 5.5, an overlook on how a Discovery Server based ROS2 system topology is seen from both the point of view of ROS and Kubernetes.

It is fundamental to pay attention to some things:

- Kubernetes treats all the Pods (shortened as KP in the image) in the **same way**, no matter whether they are running a Discovery Server, or a client.
- The only data exchanged between the clients and the Discovery Server is the **discovery metatraffic**. User traffic is exchanged directly among nodes.

- It is not important on which node Kubernetes schedules the Pods, since Kubernetes assures that every Pod in the same cluster is always able to directly contact others.

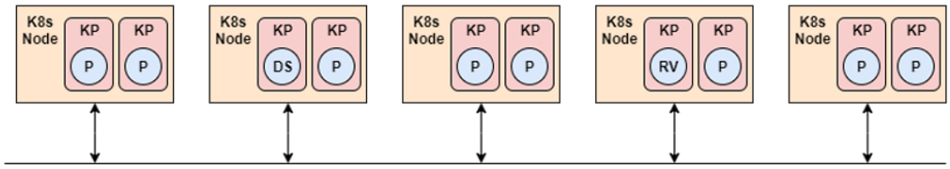

Figure 5.4: Topology from Kubernetes POV

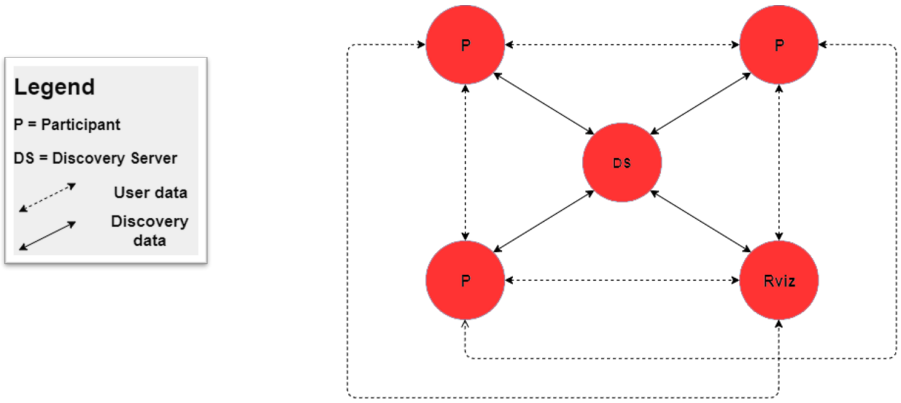

Figure 5.5: Topology from ROS2 POV

Now a ROS2 system is **completely compatible** with any Kubernetes cluster independently of what CNI is running onto it.

In figures 5.6 and 5.7, we can respectively see what is the impact on **CPU and memory usage** of plain Kubernetes (in particular a K3s cluster, where K3s is a very lightweight version of Kubernetes, designed specifically to be used in low resources environments), plain ROS2 (running up to a 32 nodes system) and ROS2 on Kubernetes (up to 32 ROS2 nodes, plus the discovery server). The tests have been executed on a Intel i7-7700HQ CPU (quad-core at 2.8GHz base frequency boostable at 3.8GHz, hyperthreading enabled) with 16GB of RAM, running plain, GUI-less Ubuntu 20.04. It is clearly visible that the **biggest impact** on resource usage (for both CPU and memory) has been caused by K3s, while ROS2, at least when running two basic nodes, had an almost unnoticeable one. Considering the specifications of a Raspberry Pi 3 B+ (the onboard computer of the Turtlebot3) it is evident that the amount of necessary resources is huge, especially when it comes to memory consuption, which already exceedes the 1GB available on the Raspberry

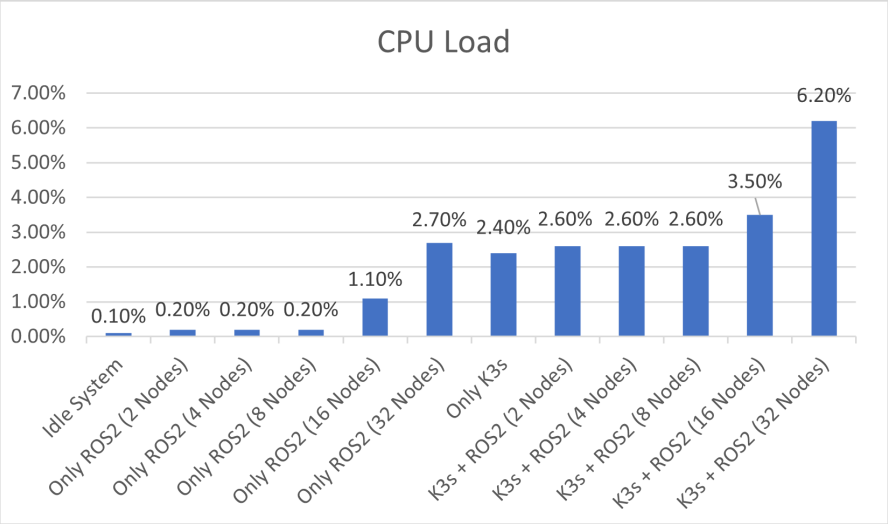

Figure 5.6: Impact on CPU

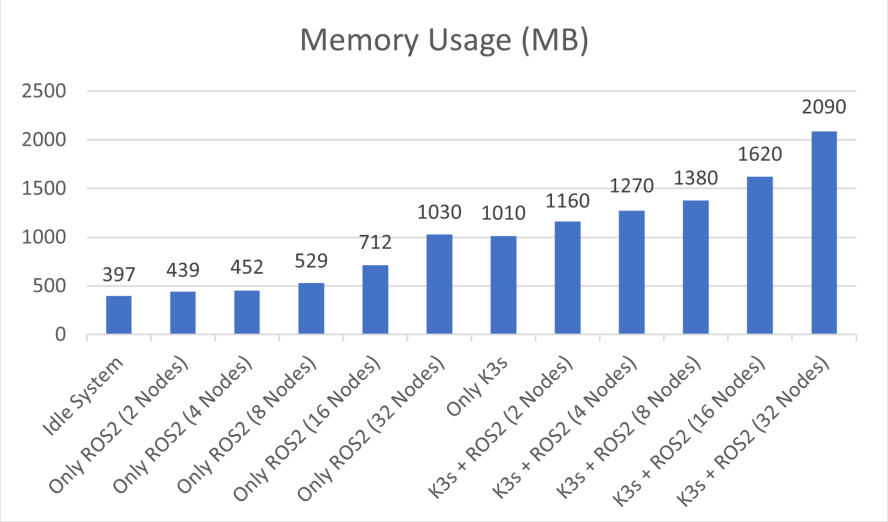

Figure 5.7: Impact on Memory

Pi 3 when only an empty K3s cluster is running. As expected, increasing the number of running nodes **increases** the amount of requested resources: 32 nodes require, in both cases (plain ROS2 and ROS2 on K3s), a huge amount of memory, respectively 1.03GB and 2.09GB, which is definetely too much for a Raspberry Pi 3 to handle; the CPU average load is quite high as well, touching peaks of 6.2% in the case of ROS2 working on top of K3s. Aside from the evident difference in magnitude, we cannot see any big differences on the rate with which the resource usage grows in ROS2 vanilla and in ROS2 on K3s.

The next step is analyzing different alternatives for forwarding the traffic to specific destinations in Kubernetes; the next chapter will be focused on that.

Chapter 6

Traffic Routing in Kubernetes

One of the objectives of this thesis is to allow the autonomous system to use a service **without having to care about what actual instance** of it (either remote or local) is actually using; it is therefore necessary to find a mechanism that allows us to do. This mechanism has to be responsible of **redirecting the incoming traffic** to the most appropriate service instance without the autonomous system noticing any difference or having to modify the requests it is sending. In figure 6.1, an example system can be seen: traffic exiting the autonomous system will be received and forwarded to the right service instance by an intermediate participant which implements the routing mechanism. Since Kubernetes is responsible for the actual network connectivity, the traffic routing feature has to be provided by either a Kubernetes native functionality, or by a third party solution. Many possible alternatives have been evaluated, and some of them have also been tested, not always succesfully, because of either incompatibility with ROS2 (more precisely with FastDDS), or just because they did not actually provide the required feature, but only a similar one. In this chapter, all the considered ideas, as well as the one that has eventually been chosen, are briefly introduced, giving an explanation on how they work and their pro and cons.

6.1 Native Features

The alternatives that are now going to be presented are based on already tested and working features offered by Kubernetes. Every single one of the solutions presented in this section allows us to achieve the desired goal: redirecting the traffic to specific destinations. All these options will now be introduced with their pros and cons.

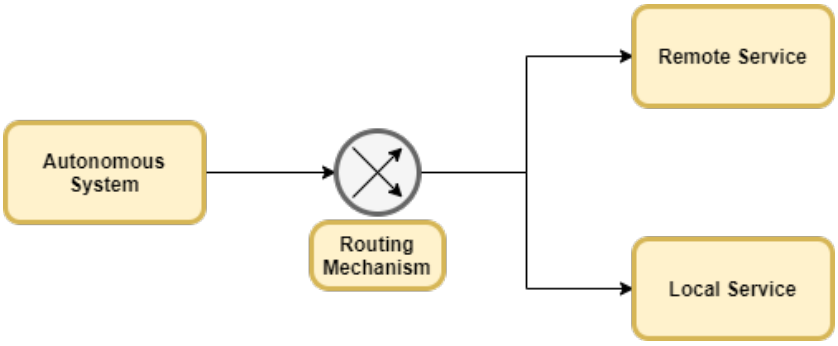

Figure 6.1: Traffic Redirecting Mechanism

6.1.1 Kube-proxy Modification

This first possibility consists in **manually modifying the Kubernetes source code**, in particular *kube-proxy* (the Kubernetes component responsible of maintaining network rules on nodes), and adding the endpoint selection functionality by ourselves and thus making it perfectly compatible with all our needs. While this is the solution that allows the **best performances**, since the functionality will be directly integrated in the source code, it is definetely the one that **requires the most work** by the user, both the first time it is applied and in the long run. Being this a custom addition to Kubernetes, it will **not be carried over to new Kubernetes versions** when they are released; this either forces the user to keep using a potentially outdated version of Kubernetes, which is not recommended since every new Kubernetes version introduces new features, bug fixes and general improvements, or to rewrite and re-add the feature at every update, which is quite a time consuming task. In addition, given that in the majority of cases Kubernetes users want to keep traffic in a local zone in order to reduce costs and improve network performances, this modification, which aims at doing the exact opposite, is very unlikely to be accepted as a KEP and successively added in future releases.

6.1.2 Service Selector

This option, which is the one that **has eventually been chosen** for the scope of this work, relies on using Kubernetes Services and more specifically leverage the Selector field in their specifications. Indeed, the Service specifications include a **Selector field which explicitly determines what Pods are going to be exposed by it** (the Pods to which the incoming traffic will be forwarded to); an Endpoint resource is then created by kube-proxy to collect all the references to the Pods that match the Selector. If that field is changed, the Endpoint resource will be populated by the references to the Pods that match the new Selector and

therefore the traffic will be redirected toward them. This solution not only does not introduce overhead but it is also a native Kubernetes feature, **very fast and easy to implement**. In order to apply this solution to our case, we need to create a service which, in turn, exposes either our remote, or our local instance of the service. We can do that by assigning to each one of these endpoints a different label, the same label that has to be matched by the Selector the exposing Service.

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    labels:
5      name: local-listener-service
6    name: local-listener-service
7  spec:
8    #Exposes all the Pods that have both "app" and "version" labels with those values
9    selector:
10     app: listener
11     version: local
```

Figure 6.2: Definition of a Service with a Selector

6.1.3 Network Policy

A Network Policy is a construct by which Kubernetes allows network **traffic at IP address or port level to be controlled**. They behave as **whitelists**, allowing to specify what ingress or egress traffic the selected Pod should be able to receive and send. By default a Pod is **non isolated**, meaning that it is allowed to receive and send traffic to whichever destination it wants; as soon as a Network Policy is applied to it, it becomes **isolated**. Allowed traffic destinations and sources can either be single endpoints identified by their IP address, whole namespaces identified by labels or a Pod/set of Pods also identified by labels. A universal selector is also available; it can be used to allow all the ingress and/or the egress traffic no matter what the source/destination is. If this universal selector is applied to both ingress and egress traffic, the behavior will be the same as Pods without Network Policies. In figure 6.3, we can see the definition of a Network Policy which regulates the traffic for the Pods with label *name* and value *local-pod*. This Network Policy will influence both ingress and egress traffic, allowing all egress traffic, and only the ingress traffic coming from Pods with label *name* with value *discovery-server*.

In order to use this feature, it is important to remember how ROS2 nodes communicate with each other: each node can publish data on a topic and one or more nodes can subscribe to that and receive the data. Keeping that in mind, if

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-local
spec:
  podSelector:
    matchLabels:
      name: local-pod
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          name: discovery-server
    ports:
  egress:
  - {}
```

Figure 6.3: Definition of a Network Policy

the ingress and egress traffic for a specific ROS2 node is blocked (allowing only the discovery traffic, since it is important that other nodes are aware of its existance), traffic will be **received and elaborated only by the non-blocked node** and this is basically the same as explicitly selecting a specific destination. In our case, this translates in allowing the endpoint that generates the traffic to send it to both remote and local instances at the same time; by means of a Network Policy, ingress traffic of the local instance will be blocked when the remote service is reachable, and allowed when it is not. The only traffic that will be always allowed to every endpoint is the discovery traffic, in order to allow a quicker switch since we will not have to wait for the discovery to happen. While this solution works, does not introduce any overhead and is easy to implement, it is **not advisable to block all the traffic** for a specific Pod, since there is always the risk of mistakenly blocking necessary traffic. Also, it is important to notice that the source endpoint will send twice as much traffic: the source is not aware of the applied Network Policy, thus it will try to communicate with both the endpoints at the same time.

6.1.4 Manually Created Endpoints

This solution is based on the use of Services defined **without selectors**. While for standard Services Kubernetes automatically takes care of creating and populating an Endpoint resource with all the references to the exposed Pods, the same does not happen for Services without selectors. Since the selector field does not exist,

Kubernetes does not know how to populate the Endpoint resource and therefore leaves the task of creating and populating it to the user. By modifying the Pods to which the Endpoint resource refers to, it is possible to decide to what destinations the Service should forward traffic to. This option achieves the **same results as the Service Selector solution** introduced above (manually creating Endpoint resources gives the user more freedom, but that is not needed for the scope of this thesis), but it introduces **more work on the developer side**, that is why the other solution was preferred to this.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Figure 6.4: Service Without Selectors

```yaml
apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 192.0.2.42
    ports:
      - port: 9376
```

Figure 6.5: Definition of an Endpoint Resource

6.2 Kubernetes Enhancement Proposals

All these ideas are **Kubernetes Enhancement Proposals** (often referred to as KEPs) that aim at providing Kubernetes with a way of doing traffic routing based on the **topology of the cluster**. They all take advantage of the concept of Kubernetes Service; this Service will expose a set of Pods of which we want to choose the most appropriate at every instant. By default, Kubernetes Service select a Pod in a round robin fashion; the solutions that are going to be introduced in this section all work by altering this behavior. To apply this solutions to our case, it was simply necessary to expose both the remote and the local Pods with a Kubernetes Service and set in its specifications the right fields defined by each of these alternatives. This topology-based traffic routing feature right now is not available in plain Kubernetes and it corresponds exactly to what was needed: a means by which telling Kubernetes what is our preferred traffic destination based on the **node location in the cluster**, in this case a remote node, or a local one. Being them, as a matter of fact, only enhancement proposals, they were either still not available in a standard Kubernetes installation, or needed to be enabled (disabled by default, because in alpha state) by means of Kubernetes *Feature Gates*,

key-value pairs that allow the user to enable or disable Kubernetes features. The considered KEPs are, in particular, *Service Topology (KEP 536), Topology Aware Subsetting (KEP 2004), Service Internal Traffic Policy (KEP 2086)*, and will be introduced in the following subsections. Before doing so, though, it is needed to introduce three other Kubernetes features that are used by the KEPs mentioned above in order to have a better understanding of them:

- **Standard Topology Labels:** defined in KEP 1659 [16], those labels (*topology.kubernetes.io/region and topology.kubernetes.io/zone*) allow to define two possible topology groups in a cluster: regions and zones. A zone is a subset of a region, a region can contain multiple zones and each zone can have multiple nodes. The already existing label *kubernetes.io/hostname*, used to identify a precise node, has been proposed to be integrated in this KEP by changing its name to "topology.kubernetes.io/node".

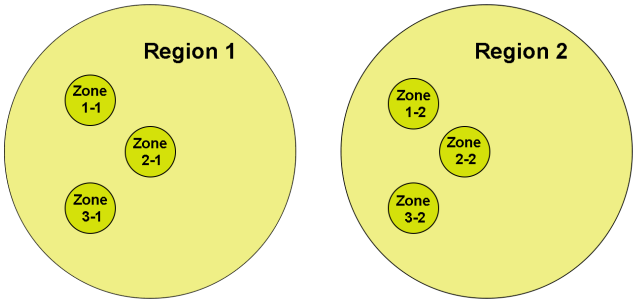

Figure 6.6: Zones and Regions

- **EndpointSlices:** released in Kubernetes v1.17 as beta feature, they offer a more **scalable and fast alternative to standard Endpoint resources**. Traditionally, when a Service is deployed an Endpoint resource is automatically created by Kubernetes in order to keep track of all the Pods that are exposed through that service. When the number of exposed Pods becomes very large, though, the respective Endpoint resource does so as well, introducing a lot of **network and processing overhead** when it has to be modified by either adding or removing Pods. EndpointSlices have been designed especially to avoid that: they behave in the exact same way as standard Enpoints do, but they have a **customizable limit** of Pods they can manage: if the number of Pods grows larget than that value, another EndpointSlice is created in order to keep track of newcoming Pods.
- **EndpointSlices Subsetting:** defined in KEP 2030 [17] and possibly released as an alpha feature in Kubernetes v1.21 (the current version is 1.20). Two labels

(*endpointslice.Kubernetes.io/forZone and endpointslice.Kubernetes.io/forRegion*) are introduced by means of which the user will be allowed to subset an EndpointSlice resource based on the zone and the region the endpoints (i.e. backend Pods, not Endpoint resource) belong to. This gives the possibility of routing traffic to **specific zones/regions**.

6.2.1 Service Topology (KEP 536)

A newly introduced field in the Service specifications called *topologyKeys* allows the user, by means of the previously mentioned *Standard Topology Labels* along with the *kubernetes.io/hostname* label, to decide where incoming traffic should be preferably forwarded to. More than one of those labels can be inserted in the topologyKeys field and will be evaluated in the same order as they are written in service definition; therefore, it is important, in order to have a proper behavior, to write them in order of specificity, starting from the **less specific label**. A universal selector (*'*'*) is also available; it **matches with every Pod**, without caring about topology and is usually written in the last position to be used as a **fallback** in case the previous labels did not match with any Pods. It is important to notice that, in case the universal selector is not specified, if no Pod is found that matches the defined labels, the traffic will be **dropped**. Under the hood, Service Topology works by comparing the values of the specified labels of both the node from which the traffic is coming from and the nodes to which the Pods exposed by the Service belong to; the first of those nodes that matches, will be selected.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
    - "kubernetes.io/hostname"
    - "topology.kubernetes.io/zone"
    - "topology.kubernetes.io/region"
    - "*"
```

Figure 6.7: Example Service Using TopologyKeys

There has been a proposition to introduce the use of **custom labels** in future

versions of this KEP (which is exactly what was needed for this particular use-case, considering that the remote destination is to be preferred over the local one), but that will not see the light of the day since this feature will be **dismissed** as of Kubernetes v1.20 and replaced by the combination of the two KEPs that will be presented in the next subsections. The decision to dismiss the Service Topology feature has been made for sake of simplicity; the user had to do too much work in order to obtain a behavior which is, in the majority of cases, the only one really needed: keeping traffic **as close as possible to the source** in order to reduce costs of traffic traversing different zones.

6.2.2 Service Internal Traffic Policy (KEP 2086)

This feature will be made available in Kubernetes v1.21 as an alpha feature. It introduces a new field called *internalTrafficPolicy* [18] in Service specifications, with three possible different values:

- **Cluster:** the Service will behave exactly as if no *internalTrafficPolicy* value was defined, forwarding traffic to the first available Pod. If *Topology Aware Subsetting* (more on this in the next subsection) is configured, it will fallback to that.
- **PreferLocal:** the Service will try to keep the traffic into the **local node**; if no Pod is available in it, it will fallback to the Cluster behavior.
- **Local:** very similar to PreferLocal, but in this case, if no available Pod is found, traffic will be **dropped**.

This KEP works by comparing the value of the label *kubernetes.io/hostname* of the node traffic is originating from and the nodes to which the available Pods belong to. The similarities with the previously presented Service Topology are quite evident, but in this case the user will not have to worry about selecting the right topology labels and putting them in the right order, it will just have to specify the wanted behavior.

6.2.3 Topology Aware Subsetting (KEP 2004)

This KEP [19] is thought as of a **hint for the EndpointSlice controller**; by setting the value of *endpointslice.kubernetes.io/subsetting* to Auto, the controller will try to keep all the traffic in the local zone (as for the other KEPs, it does so by comparing nodes' labels) as long as it has a sufficient number of endpoint (as in network endpoints, Pods). If there are not enough endpoints, traffic will be routed to other zones as well. This proposal bases its whole logic upon two main assumptions:

- Incoming traffic is **directly proportional** to the number of allocatable **CPU cores** in that zone. If the number of CPU cores is not retrievable, it will fallback to the number of **available nodes**.

- Service capacity is proportional to the **number of endpoints** in a zone, assuming that every endpoint has the **same computational power** as the others.

This solution works by splitting EndpointSlices into *EndpointSlice Groups* (subsets of an EndpointSlice resource), which are only consumed by one zone and receive traffic only from that zone. The expected number of Pods belonging to each group is calculated by doing $total\_endpoint \times nodes\_in\_zone/total\_nodes$. Each Group is filled as much as possible by following this algorithm:

1. Iterate through all the existing zones filling each Group with the correspondant local Pods. By the end of this part a 7situation like the one depicted in figure 6.4 will be reached.

2. If the current zone has enough endpoints to achieve a traffic load below the **overload threshold** (maximum traffic overload for each endpoint, by default is 50%), it will be put into a *endpoints_available* pool. On the contrary, if the current zone has not enough endpoints, it will be put into a *endpoints_required* pool. Both pool are implemented as **priority lists**, in which the zone with most extra endpoints (or with the most insufficient number of them) will be put first.

3. Iterate through both pools; endpoints are removed from zones belonging to the first pool and assigned to zones in the second one. It is important to notice that endpoints are **not actually moved** from one zone to another, but they are just **logically assigned** to another EndpointSlice Group and therefore will from now on receive traffic and serve requests only from the zone to which the Group belongs to. If the zone from which the endpoint has been taken from has still enough endpoints to achieve overload below the threshold, it will be put again in the *endpoints_available* pool; the same goes for the receiving zone in case it still has not enough endpoints. This will go on until either **one of the two pools is empty**. In figure 6.5 it can be seen what would happen if a zone (in this case, Zone C) had not enough endpoints to acheve a traffic load below the maximum overload threshold, and a zone (in particular Zone A) had more than enough endpoints to achieve that same goal; at this particular point of the algorithm, a Pod in Zone A would be logically moved into the EndpointSlice Group of Zone C, so that it could be used by the receiving zone in order to decrease the traffic overload.

4. If the available pool is empty and the required one is not, the behavior will fallback to a random approach, delivering traffic to endpoints **without caring about topology**.

5. Otherwise, iterate through the available pool again assigning endpoints to zone which have a number of endpoints less than the expected one. This will be repeated until either the available pool is empty, or every zone has at least the expected number of endpoints.

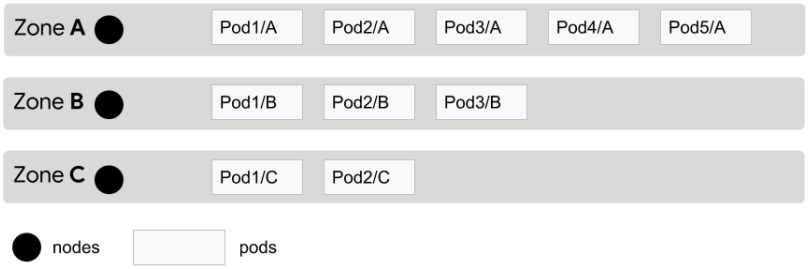

Figure 6.8: A Three Zones Cluster After the First Step of the Algorithm

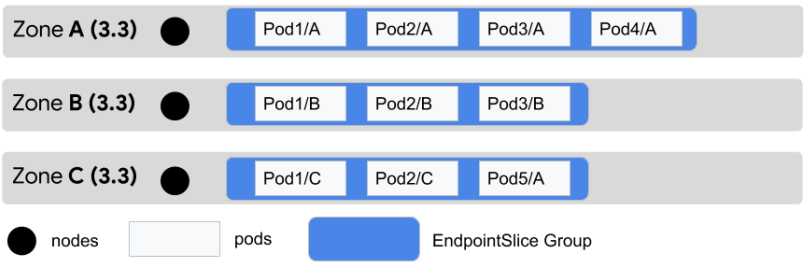

Figure 6.9: A Three Zones Cluster After the Conclusion of the Algorithm

It has been tested that applying this algorithm with too few endpoints leads to a high probabilty of having an **imbalaced traffic load and EndpointSlice churn** (endpoints migrating too frequently from a Group to another), that is why Topology Aware Subsetting will start functioning only when the number of endpoints is greater than a **starting threshold plus a padding**. By default the starting threshold is set to three times the number of zones, but it can be customized; the padding is used in order to **defer the mechanism change** in case of too frequent endpoints addition or removal (e.g, the number of endpoints is exactly the threshold value, an endpoint is added and then removed; withouth a padding, Topology Aware Subsetting would start and stop in a very short time).

In conclusion, the three alternatives that have just been introduced had to be **discarded**, not only because they were going to be discontinued (KEP 536), or not ready for testing, but mainly because, while they aim at providing the possibility of doing traffic routing based on the topology of the cluster, those solutions main goal is to keep traffic as close to its source as possible, in other words they try to **keep traffic into a local zone** and this is exactly the opposite of what is needed for the goals of this thesis.

6.3 Service Mesh Based Solutions

The concept of Service Mesh initially appeared when microservices became more popular: microservices belonging to the same application were often developed in different languages, and this introduced problems when it came to service discovery and communication. Service Meshes take care of solving those problems.

Services Meshes are responsible for managing the network traffic between service by means of proxies called *sidecars* that are deployed alongside each service; they are generally used to make communication between services **safe and reliable**. These proxies operate at Layer 7 of the OSI network stack, meaning that they can use information contained into HTTP headers or other application level metadata. The service mesh pattern focuses mainly on handling the traffic that is generated and that travels inside a data center (or a Kubernetes cluster). Aside from offering reliability features, Service Meshes are often used in testing environments and when particular update rollout patterns are followed.

A Service Mesh is generally constituted by two components:

- **Data Plane: lightweight proxies** distributed as sidecars; in the case of Kubernetes, they are processes that run side by side with applications inside each Pod.
- **Control Plane:** provides the configuration for the proxies, issues TLS certificates and contains policy managers.

There are **many different option** when it comes to choosing a Service Mesh implementation, but two of the most popular ones (and also the ones that have been considered in this work) are **Istio and Linkerd**. Each one of them offers more or less the same set of basic functionalities, but if more complex functionalities are needed the user must be ready to do some work in order to set them up properly.

Both the considered Service Mesh alternatives offer a very similar solution that allows a Service to **split the incoming traffic** (by means of weights defined into its specification) between two or more different istances of the Pods it exposes (e.g., a canary release, which consists in the gradual redirection of a portion of the traffic toward a new version of a certain service, while the rest of it is still forwarded

toward the previous working version of it. This avoids, in case of problems in the updated version, a complete outage of the service): the concept of **Virtual Service** for Istio and the **Traffic Split** on Linkerd.

6.3.1 Linkerd

The solution proposed by Linkerd [20] is, as mentioned above, called Traffic Split. Each set of Pods we want to split traffic among has to be exposed by a Service (e.g., a Service exposing remote Pods and a Service exposing local Pods); all these Services must in turn be exposed by a **TrafficSplit resource**, a Linkerd custom resource. This, by means of a **weight value** assigned to each one of the exposed Services, will **split traffic among them** and consequentely among the Pods exposed by those. By properly setting weights, we can easily decide to send all the traffic toward a specific Service. In order to take advantage of this feature, though, the Pods must contact the TrafficSplit resource and not the exposed services. Moreover, it needs to be contacted by means of its **domain name** and not by directly using its IP address.

```
1  apiVersion: split.smi-spec.io/v1alpha1
2  kind: TrafficSplit
3  metadata:
4    name: traffic-split
5    namespace: web-app
6  spec:
7    service: web-server
8    backends:
9    - service: web-server-v1
10     weight: 100
11   - service: web-server-v2
12     weight: 0
```

Figure 6.10: TrafficSplit Example

6.3.2 Istio

The option proposed by Istio [21] is called **Virtual Service**. A VirtualService is a custom resource belonging to the Istio system and it allows to **expose two or more sets of Pods**, splitting traffic among them using a user-defined weight, very much likely Linkerd does; the difference here is that using Istio there is **no need of exposing those sets of Pods with Kubernetes Services** because they can be directly referenced by the VirtualService itself. Same as for Istio, the VirtualService must be referred to by using its domain name.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-service
spec:
  hosts:
    - web-server
  http:
    route:
    - destination:
        host: web-server
        subset: v1
      weight: 50
    - destination:
        host: web-server
        subset: v2
      weight: 50
```

Figure 6.11: VirtualService Example

6.3.3 Solutions Evaluation

Both solutions, while in principle they can do exactly what it is needed, **cannot be used in this particular case** because neither the TrafficSplit nor the VirtualService can be addressed by ROS2 by using their domain name since **FastDDS cannot be made to contact a destination identified by its domain name**, but only by using its IP address. Also these two solutions are, compared to all the others mentioned so far, the only ones which introduce an **additional overhead** since they are not part of basic Kubernetes and therefore add an additional layer on top of the standard architecture (the traffic directed to each Pod must now also traverse a proxy before being actually received by the actual destination). A **basic performance comparison** has been conducted by running a simple test application not based on ROS using a plain Kubernetes cluster, a cluster with Linkerd installed and one with Istio installed; in table 6.1 the results of this comparison are presented.

It is important to notice that performances have been evaluated on a much more powerful machine than a Single Board Computer like the one used on-board of a Turtlebot, for which the increased resource usage could be more relevant. It can be noticed how the memory usage increases of more or less the same amount in both Linkerd and Istio cases, while the CPU usage is higher when Linkerd is used, and stays very similar to the plain Kubernetes with Istio. The latency has been measured from a Pod to another in the same local (that is why the measured latencies are very small) cluster exposed by the Traffic Split feature; if we take in consideration the already strict time constraints introduced in chapter 4, this

latency, while not greatly increased, could still present a problem for the system.

Cluster Type	CPU Usage	Memory Usage (MiB)	Latency (min/avg/max)
Plain Kubernetes	28%	872	0.034/0.055/0.130ms
Linkerd	33%	1363	0.034/0.074/0.183ms
Istio	29%	1378	0.045/0.078/0.232ms

Table 6.1: Perfomance Comparison

6.4 Conclusions

As anticipated, the selected alternative is the one based on the use of Service Selectors, since it is the simplest and easiest to implement working solution which, not only does not introduce any computational overhead, but it has been a core feature of Kubernetes since early versions, so by now it is has been thoroughly tested in a wide variety of cases and it is guaranteed to keep existing even in future Kubernetes versions.

Some tests have been conducted to **evaluate the performance** of this switching mechanism. Both TCP and UDP connections have been tested, by deploying into Kubernetes a simple **client-server system** (both TCP and UDP versions have been written in Go), where the client sends a packet once every 1ms toward a Kubernetes Service the exposes at every instant only one of the two servers available, based on its Selector. Inside the packets sent by the client there is a *packet number* field, that will be printed by the server in order to check whether there was packet loss or not. At every packet sent (the client) and at every packet received (the server) a timestamp will be printed as well, to evaluate the behavior of the system when a switching occurs.

The TCP system, as expected, **does not work very well** with this mechanism since in this case the client and the server instaurate a stateful connection. What happens is that when the switching occurs, the client keeps sending traffic to the server it did the handshake with, thus it will **not be influenced by the switching**. In order to make it change destination, it is necessary to kill the client Pod and wait for Kubernetes **create another one.** As soon as the new Pod is up and running, it will perform the handshake with the new destination exposed by the Service and will start communicate.

On the other hand, the UDP system **works as predicted** with the client sending the traffic to the actual destination exposed by the Kubernetes Service, being UDP is a stateless protocol. There is **no packet loss** when migrating from one server to another and the switching happens in **less than a millisecond** (the

client sends one packet every millisecond and the last packet received by the initial server is the one that immediately precedes the first packet received by the final server).

The last thing that is needed in order to achieve all the goals of this thesis is a way to **monitor the network status and switch to the most appropriate Service instance** (at this point, this just means changing the Service selector) consequently. This will be done by means of Kubernetes operator deployed into the cluster. More on that, along with the description of the final demo, in the next chapter.

Chapter 7

Network Status Monitoring

The last step towards the goal of this thesis is to define a way of monitoring the **status of the network** the autonomous system is connected to in order to be able to select the most appropriate instance of the service that has to be used: if the quality of the network is good enough, the remote instance will be preferred and used by the vehicle, while, if the quality is deemed not to be good enough or if the network connectivity is dropped completely, the local instance will be contacted. The component responsible of doing so, is also responsible of interacting with the Kubernetes cluster (by means of the **Kubernetes API**) since it has to change the Selector field of the Kubernetes Service exposing our local or remote Pods (the switching mechanism has been explained in the previous chapter) based on the quality of the network connectivity. This chapter is focused on explaining how this component works and will also introduce and describe the demo that has been prepared to demonstrate the final results achieved by this thesis.

7.1 Kubernetes Operator Pattern

A **Kubernetes Operator** is a software extensions to Kubernetes used to manage applications and their components. It follows the Kubernetes principles, in particular the **Control Loop**, an infinite loop that keeps track of the status of the Kubernetes cluster and, as soon as the state is no longer equal to the one defined or requested by the user, it will make the necessary changes in order to make it so. In a single cluster there can be **multiple controllers** that implement that logic, each of them responsible of keeping track of at least one Kubernetes resource. A controller can either carry out the needed actions by itself or, as it happens in most of the cases, by interacting with the **Kubernetes API server** (the Kubernetes component responsible of elaborating incoming Kubernetes API requests).

Since the developer knows how the system should behave in standard conditions

and how it should respond in case of problems, he can write an Operator in order to **automatically take care** of the tasks that it normally should manually take care of, beyond what plain Kubernetes offers.

An Operator is, at the end of the day, simply a program, so in principle it could be written in **any existing programming language**, even though it is preferable to use one which offers a good Kubernetes client library and is simple to containerize and run into a Kubernetes cluster.

The go-to choice for most operators is **Go language** [22]; Kubernetes itself is written in Go so this guarantees a very smooth interaction between it and the operator. In order for a Go program to interact with a Kubernetes cluster, it needs to use the *client-go* library.

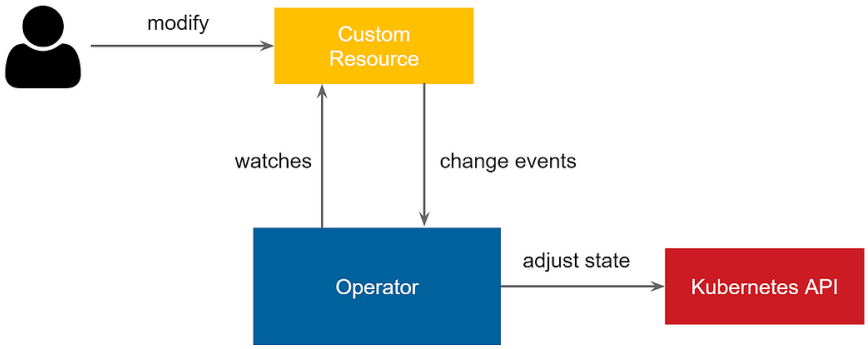

Figure 7.1: The Operator Pattern

7.1.1 The Network Operator

As it can be noticed in figure 7.1, usually an operator works by keeping track of the status of a Custom Resource Definition (CRD) and, as soon as a change arises, it starts trying to adjust the observed cluster status in order to make it converge as the user desires it to be. That is not the case of the operator written for this thesis. This operator does not monitor a Kubernetes resource, but it monitors the status of the network which is completely **unrelated to the cluster** and something that no Kubernetes feature natively allows. The algorithm behind the operator is quite a **simple one**, given that the goal of this work was not developing a refined, precise and cutting-edge network monitoring application; it is based on the usage of the Linux command line tool *iwconfig* (an example output of which can be seen in figure 7.2), in particular on the values of the fields *Link Quality* and *Signal Level*. Now, an explanation of the algorithm:

1. An infinite cycle starts and once every second the *iwconfig* [23] command is

executed.

2. The *Link Quality* and *Signal Level* values are parsed. If an error during parsing arises, that means that *iwconfig* has not returned the right values. That could happen in two cases: there is no WiFi network card onboard (this can be excluded since we now beforehand that the system has one available), or the WiFi network card is not connected to any network.

3. If the network card has completely lost connection to the network, the operator will immediately switch to the local instance of the service the vehicle is using.

4. If, on the other hand, the values have been successfully parsed, the algorithm takes them in consideration: if either one of them is **below a certain threshold**, the operator will switch to the local service. As soon as both of them are above the same threshold, it will switch back to the remote one.

5. An **hysteresis time** has been introduced to avoid too frequent instance switching in case the above-mentioned values are oscillating near the threshold. A switch cannot happen if another one happened too recently.

```
1 wlan0         IEEE 802.11bgn  ESSID:"NETGEAR64"
2         Mode:Managed  Frequency:2.452GHz  Access Point: C0:FF:D4:91:49:DF
3         Bit Rate=57.8 Mb/s  Tx-Power=20dBm
4         Retry  long limit:7  RTS thr:off  Fragment thr:off
5         Power Management:on
6         Link Quality:47/70  Signal Level=-63dBm
7         Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
8         Tx excessive retries:0  Invalid misc:8  Missed beacon:0
```

Figure 7.2: An Example Output of the *iwconfig* Tool

Both the hysteresis time and the name of the interface the tool has to run onto can be defined by means of environmental variables into the description file of the Deployment. In order to use this operator, it is simply needed to containerize it and write a definition for a Kubernetes Deployment that uses its Docker image. It is important to notice that containers make use of **virtual network interfaces**, so, without an explicit configuration, it will **not be able to** detect and interact with the network cards belonging to the host machine. Kubernetes allows a container running into a Pod to use the network namespace of the host by setting, into the specifications file, the field *hostNetwork* to true. In this way, the Pod can access all

the network interfaces present on the host. In the specific case of our operator, it can now execute *iwconfig* on the correct wireless interface.

7.2 Demo

In order to present in a more practical way the achievements of this thesis, a demo has been designed. It has been created by keeping in mind a realistic service that an autonomous vehicle may actually use: **object detection**. The object detection has been performed by **YOLO** (acronym for You Only Look Once), an object detection algorithm written and designed by Joseph Redmon and Ali Farhadi that works on top of the Darknet neural network (designed and developed by Joseph Redmon). Since the demo has not been run on a robotic device, but on home PCs, we needed a way to **simulate a camera feed** for the object detection algorithm to work on and a way to **watch the results** of that elaboration. On top of that, for it to work properly, a Discovery Server was necessary, in order to allow ROS2 to execute the node discovery. Aside from the actual working components of the demo, it is important to notice that in a realistic scenario the local instance of the service and the remote one may not, and most probably they will not, **belong to the same Kubernetes cluster**. Plain Kubernetes does not offer a way of doing **service discovery among different cluster**, especially if we consider that the remote cluster may not be fully under our control, but belonging to the owners of the data center it is run into. Given all of that has just been said, a way of merging two different clusters and making them behave as if they were a single one was needed; this, in the Kubernetes world, is called Cluster Federation. Federated clusters share pieces of their configuration, which is then usually managed by a cluster called *host cluster*. Any resource configured to take advantage of the federation, will treat all the member clusters like if they were a single distributed cluster. A common use-case for Federation is when an application has to be scaled across multiple data centers, which is very similar to our goals. Different alternatives exist to enable cluster federation, like:

- **Kubefed:** standing for Kubernetes Cluster Federation [24], is a Kubernetes feature, currently alpha, that provides the necessary mechanisms to allow the federation of multiple clusters. It is implemented as an extension of the Kubernetes API which basically redefines all the standard Kubernetes resources that will now be Federated objects (e.g., FederatedDeployment) and defines some now federation-specific objects. The main drawbacks are: the federated clusters do not exchange information among themselves, thus they are not aware of the federation; any modification applied directly to one of those clusters, is not propagated to the others, therefore creating a divergence; also, this approach builds a federated ecosystem on top of a non-federated one:

it not only re-implements a lot of the already existing features to adapt them to a federated system, but it also creates new kinds of objects not compatible with a standard cluster.

- **Submariner:** a third-party solution [25] to achieve federation between multiple clusters at L3 via encrypted VPN tunnels. It addresses the problematics presented in the previous alternative: this approach does not require the definition of a new API, it just relies on some new Custom Resource Definitions that store the metadata necessary to enable the communication among clusters. Submariner successfully manages to transparently share Pods and Services amoung different clusters, thus making it preferable to Kubefed. This alternative really emphasizes the importance of information sharing among clusters, but it may encounter some scalability problems when the amount of shared resources increases; it is therefore important to keep local objects into the local cluster as much as possible.

- **Liqo:** a framework developed at the Politecnico of Torino. It allows to seamlessly federate multiple clusters. It works in a similar way to Submariner, but, to my experience, has proven way easier to deploy, especially for clusters belonging to the same LAN and that is the main reason why it ended up being chosen for this thesis. More about it will be told in a later section.

Before delving into a deeper explanation on the topology of the demo and its components, a brief introduction will be given about YOLO and Liqo, being them fundamental technologies for the scope of this thesis and, especially Liqo, for the scope of a real application.

7.2.1 YOLO

YOLO (You Only Look Once) [26] is a state-of-the-art, real-time **object detection system**. It is very accurate and fast, designed to be built and run using the **CUDA** (Compute Unified Device Architecture) API on top of CUDA enabled Nvidia graphic cards. It offers the possibility of being built and run on top of a **CPU instead of a GPU** as well, at the price of **significantly worse performances** (e.g., running it on a Pascal Titan X GPU makes it work at 30FPS with a mean Average Precision of 57.9% on a COCO dataset, while running it with the same configuration on a i7-7700HQ CPU, while the mAP remains the same, it reduces the perfomances to ~4 FPS).

YOLO works in a **different way** from previous detection systems: to do the detection, they divide the image in multiple pieces and apply to each one of them the detection model; zones with high scores are considered detections. YOLO, on the other hand, applies a single neural network to the whole image; the neural

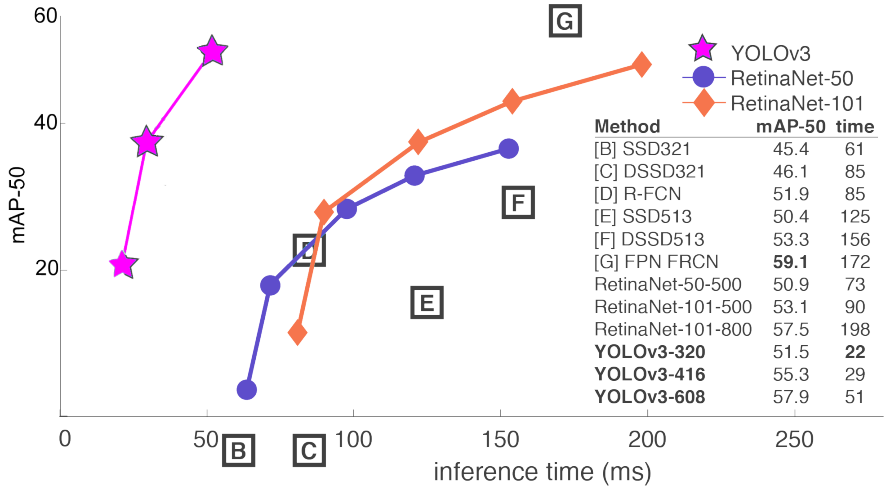

Figure 7.3: Comparison of YOLO and Other Detection Systems

network will divide the image into smaller regions and do the detection. This brings advantages such as the fact that, since the neural network is fed the whole image and not just a subsection of it at a time, the predictions are informed by the global context in the image, and that the prediction is made with a single network evaluation, instead of what happens with other systems that require thousands evaluations for a single-image. A **brief performance comparison** between YOLO and other object detection systems can be seen in figure 7.3 [26].

By default, YOLO will do the detection on single images and save the results in image files for the user to use. Needless to say, this is not suitable for a real-time application. In order to make YOLO show the results in real-time and correctly take as input the feed of a camera, it is needed to compile the underlying Darknet using **OpenCV**, an open source library mainly aimed at real-time computer vision (on the YOLO website it is recommended to compile it using CUDA as well, because of the performance issues mentioned above). For the scope of this work, YOLO has been used with a **very lightweight configuration** file (based on the Pascal Visual Object Classes dataset), in order to counterweight it being executed on a CPU instead of a GPU, and thus improve performance.

7.2.2 Liqo

As mentioned above, we can imagine as a realistic situation that the local and remote instance of the service do not run in the same Kubernetes cluster, since the remote one will run in a data center that will likely not belong to us, thus we cannot simply add that machine to our cluster as a node. It is therefore needed a way that allows us to logically treat those two cluster like a **single**

entity, without them actually being so. Liqo [27], a framework developed at the Politecnico of Torino, allows us to **seamlessly and securely share resources and services** among any cluster on which Liqo is installed. It aims at extending the Pod-to-Pod and Pod-to-Service communication (guaranteed by Kubernetes when it comes to resources belonging to the same cluster) to multiple clusters. It allows Pods belonging to different clusters to communicate without the need of a NAT (exactly like it happens in a traditional Kubernetes system). In case there is some **overlapping** between Pods and Services CIDRs (the IP pools of Pods and Service) of two peering clusters, Liqo will use a NAT to automatically take care of that. Liqo can be installed and used no matter what Kubernetes distribution or CNI it has been adopted. Also, Liqo does not require the user to do **any modification** to the cluster in order for it to be installed and work. This makes it especially suitable for cases like the one of this thesis. Liqo enables the user to run tasks on any connected cluster, without it having to behave differently from what it is accustomed to do in a standard Kubernetes environment.

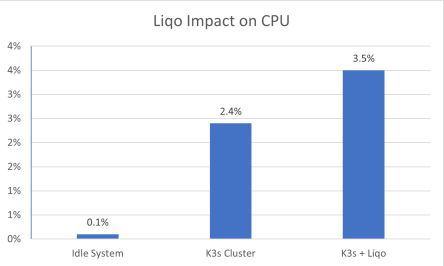

Figure 7.4: Impact on CPU

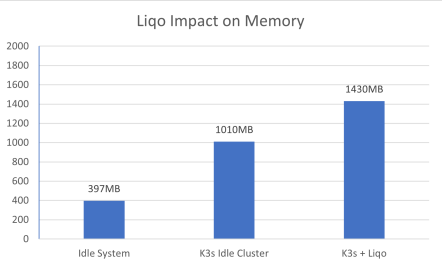

Figure 7.5: Impact on Memory

From the user perspective, a connected cluster will appear as a **node belonging to its cluster**, while, as a matter of fact, it is not. This local virtual node pretends to have available resources (the resources of the whole remote cluster it represents) and to actually handling all the Pods that are scheduled onto it, but it actually acts as a proxy toward the remote cluster. In figures 7.4 and 7.5, we can see what is the impact of Liqo on CPU and memory usage when it is used to federate two clusters deployed in the same LAN. The test have been conducted on an Intel i7-7700HQ CPU with 16GB of available RAM. While not as big as K3s, it can be seen how Liqo has quite an important impact both on CPU and on memory usage. It is clear then that some optimizations are needed in order to make it suitable for mobile systems with limited available resources, such as a Turtlebot. When the Liqo enabled clusters are deployed into the same LAN, Liqo will **automatically discover** them and make their resources available for all the others to use. If, on the other hand, they do not belong to the same local network, the user will have to manually create and apply a ForeignCluster custom resource, which defines how Liqo is supposed to reach the remote cluster. An example of such a defintion is shown in figure 7.6.

Advertisement messages are sent **periodically** among connected clusters to keep updated the information about the available resources.

```
apiVersion: discovery.liqo.io/v1alpha1
kind: ForeignCluster
metadata:
  name: remote-cluster
spec:
  join: true
  authUrl: https://liqo-auth.9b0ea54d4b59499e9331.switzerlandnorth.aksapp.io:443
```

Figure 7.6: A ForeignCluster Resource Example

7.2.3 Topology and Components

Going back to the demo, it is now going to be explained how it works and what are the parts that it is composed of. As it can be seen by figure 7.7, there are not a lot of components and the topology is quite simple, but it still needs some explanation in order to understand how it works. Let us start by examining each single part:

- **Image Publisher:** it is the ROS2 node responsible of simulating the camera feed. It receives as input a video file that will be successively published onto the ROS2 topic *image_raw*.
- **Image Viewer:** it is the ROS2 node responsible of visualizing the images elaborated by the object detection node. It listens to the ROS2 topic *detection_image* published by the YOLO nodes. Alongside it, runs an instance of *noVNC* in order to allow us to have a graphical user interface, necessary for the Image Viewer node to properly work and for us to actually see the detection results.
- **YOLO Local**: ROS2 node deployed into the local cluster, responsible of doing object detection. It subscribes to the *image_raw* topic and publishes the results onto the *detection_image* one. It runs on a ROS2 porting of YOLOv3. It is a less sofisticate and performant instance of YOLO, suitable for system with limited resources.
- **YOLO Remote:** it works exactly as its local version, but, as the name suggests, it is deployed onto the remote cluster. A more complex and performant version of YOLO that leverages the huge amount of resources offered by the data center.

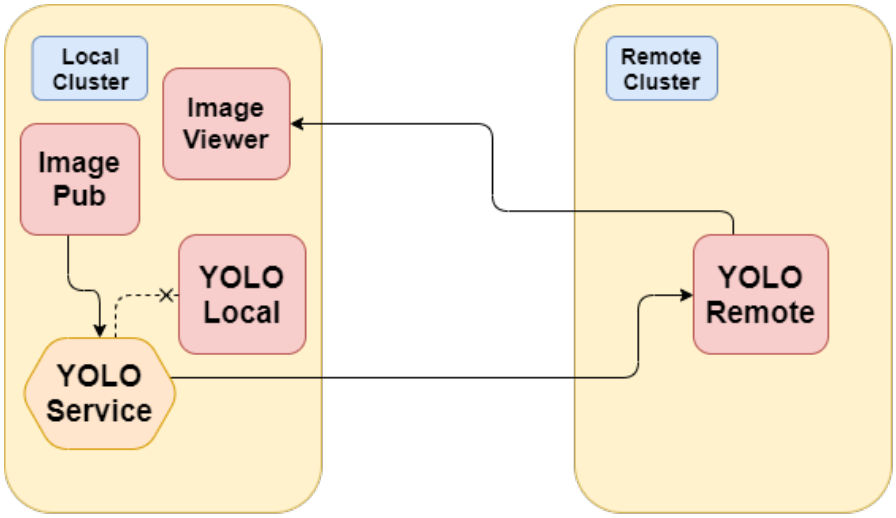

Figure 7.7: Demo Topology

As figure 7.7 shows, the demo is deployed onto **two clusters**, one local, acting as the autonomous vehicle, and one remote, acting as the data center. Thanks to Liqo, we are able to make these two cluster behave as if they were one, and thus deploying the remote instance of YOLO into the *remote node*. Even though it is not explicitly shown in the figure, a Discovery Server is present in order to guarantee that ROS2 is able to properly execute the node discovery.

The **data flow** is structured as follow:

1. The *Image Publisher* starts transmitting a video file toward the Kubernetes Service named *YOLO_service*.

2. *YOLO_service* will take care of forwarding the traffic to the actual destination, defined at each moment by its Selector field.

3. Only one between YOLO Remote and YOLO Local will receive the image feed and therefore execute the detection. Only the YOLO node that has **actually done** the object recognition will publish the data on the relative topic.

4. The Image Viewer will receive the traffic from the YOLO instance that performed the actual work and it will show the results.

It is, after all, a quite simple data flow, but, in order for it to work, it was necessary to apply some **preliminary configuration** to the ROS2 nodes, in particular to the underlying FastDDS implementation. By default, each ROS2 node announces itself to the Discovery Server (or to other nodes, in case the traditional multicast discovery is used) using **its own IP address**. This, though, would

destroy, not only the demo, but the traffic routing itself, since the data coming from the Image Publisher would be sent **towards the actual Pods** instead of the Kubernetes Service that exposes them and does the traffic routing. FastDDS can be configured to overcome such an obstacle: by means of the same XML configuration file used in order to make ROS2 work with a Discovery Server, the user can define with which IP address each participant should **announce itself with**. In this case, we make both YOLO nodes announce themselves with the IP address of the **YOLO service**, in this way, when the Image Publisher sends data, it will be sent toward the service, therefore achieving the desired behavior. As introduced earlier in this chapter, the automatic switching based on the network status is taken care of by an Operator; it is not explicitly shown in figure 7.7, but it must be deployed into the local cluster since it has to run the *iwconfig* tool on the wireless interface of the autonomous system.

In figure 7.8, an example detection performed by a complete YOLO version (not the lightweight one) using the COCO dataset. The quality of the detection **changes as intended**, based on the status of the network connectivity: by moving the device away and closer to the network (and even completely disconnetting it from the network), it can be seen how the device **never stops** elaborating the data feed, but just automatically uses the most appropriate YOLO instance (immediately noticeable by looking at the labels of the detections).

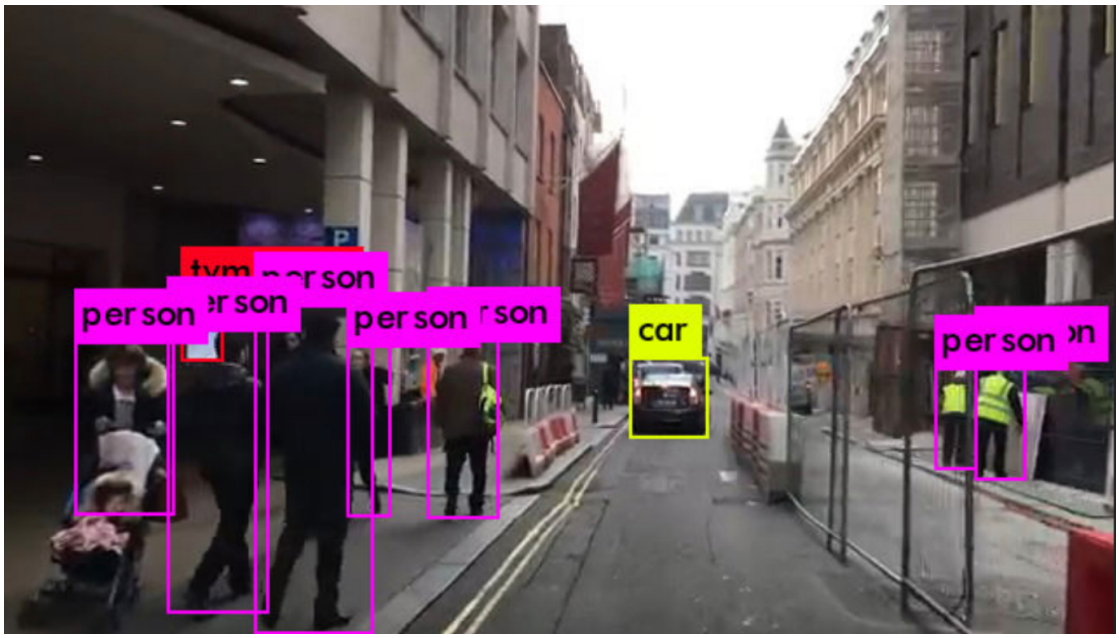

Figure 7.8: Example YOLO Detection

Chapter 8

Conclusions and Future Work

The topic of autonomous vehicles currently is one of the **most discussed ones**, with big companies such as Google, Amazon and Tesla participating in the race for the development of a fully autonomous system.

Such systems must, at every moment, deal with an **enormous amount of data** coming from a variety of sensors; this data has to be elaborated in a very **short amount of time**, since otherwise it will no longer be meaningful for the vehicle. This requires a **huge computational effort** that might be best offloaded to a data centers at the edge of the network, or in the cloud. With the advent of the 5G technology, which grants an **enormous bandwidth** to mobile devices, this possibility has become more and more reachable.

This thesis main goal has been developing a **network layer** responsible of virtualizing an autonomous system and providing it with a way of **offloading the computational effort** on a remote machine by taking in consideration, at each moment, the status of the network connectivity. In particular, this objective has been achieved by means of a combination of ROS2, responsible for simulating a robotic platform (the easiest way of approximating a real autonomous vehicle), and Kubernetes, leader in the field of containerized applications orchestration.

The idea of offloading to the edge/cloud of the network has been already treated in many other works (see Related Works subsection in chapter 1), but **almost none** of those does so by taking advantage of **containers** and container orchestrators, that is why this work focuses on this new approach. Another motivation has been the idea of **integrating the autonomous system world**, in this particular case the ROS2 ecosystem, into the concept of *Kubernetes everywhere*, in which Kubernetes behaves similarly to an operating system, allowing a variety of applications of **different nature** to be deployed into the same environment and **seamlessly**

communicate with each other.

The solution is based on the concept of **Kubernetes Service**: by means of a Selector, it exposes all the Pods that have a label matching its value. If the Selector changes, the set of network endpoints changes. In this case, we imagined creating a set of Pods representing the local instance of the services needed by the vehicle, and a set of Pods representing the remote ones. An operator is then responsible for keeping track of the **network status**, by means of the *iwconfig* tool: by parsing and evaluating its output, it will **automatically change** the value of the Selector to expose either the remote or local instances.

The proposed solution **proved to be working**, offering a switching time of the order of the millisecond, but it is limited **only** to the UDP world, since it is a stateless communication protocol. TCP introduces the necessity of **sharing the state of the connection** between the local and remote instances; without doing so, the system does not behave as desired and no switching happens, in the worst case leaving the vehicle without the possibility of analyzing the incoming data.

Another outcome of this work has been the **difficult coexistence** of ROS2 and Kubernetes, mainly when it comes to ROS2 discovery system. It has been possible to make them cooperate by using a Discovery Server which forces the underlying DDS to make use of **unicast traffic**, instead of the standard multicast one (not compatible with most Kubernetes network implementations). In this way, though, the improvements introduced by a distributed discovery mechanism are lost.

When it comes to **performance**, the data collected when running a K3s cluster (a very lightweight version of Kubernetes) and a very simple ROS2 system shows that they add quite an **significant overhead**, especially in terms of memory consumption (K3s and ROS2 together require more than 1GB of RAM), thus, while they still **might be suitable** for modern and more performant mobile devices (e.g., a Raspberry Pi 4 B), they definetely are not for a platform such as the Turtlebot3.

Future work may be focused on:

- **Improving the Network Monitoring Algorithm:** the current algorithm is very simple, since it was not the main focus of this work. More precise and refined approaches could be considered, even a very lightweight machine learning based solution.
- **TCP Compatibility:** as of now, the switching mechanism does not behave properly when TCP is used. Adding TCP compatibility might be useful to make the solution usable in more situations.
- **ROS2 Alternative:** as explained, ROS2 is not a very good choice when it comes to integrating it into Kubernetes. It might be useful to search for an alternative to ROS2 and apply this solution to it.

Bibliography

[1] T. -K. Le, U. Salim, and F. Kaltenberger. «An Overview of Physical Layer Design for Ultra-Reliable Low-Latency Communications in 3GPP Releases 15, 16, and 17». In: *IEEE Access* 9 (2021), pp. 433–444. DOI: 10.1109/ACCESS.2020.3046773 (cit. on p. 2).

[2] E. Coronado, G. Cebrian-Marquez, and R. Riggio. «Enabling Computation Offloading for Autonomous and Assisted Driving in 5G Networks». In: *2019 IEEE Global Communications Conference (GLOBECOM)*. 2019, pp. 1–6. DOI: 10.1109/GLOBECOM38437.2019.9013490 (cit. on p. 3).

[3] R. Soua, I. Turcanu, F. Adamsky, D. Führer, and T. Engel. «Multi-Access Edge Computing for Vehicular Networks: A Position Paper». In: *2018 IEEE Globecom Workshops (GC Wkshps)*. 2018, pp. 1–6. DOI: 10.1109/GLOCOMW.2018.8644392 (cit. on p. 3).

[4] K. Gilly, A. Mishev, S. Filiposka, and S. Alcaraz. «Offloading Edge Vehicular Services in Realistic Urban Environments». In: *IEEE Access* 8 (2020), pp. 11491–11502. DOI: 10.1109/ACCESS.2020.2965258 (cit. on p. 3).

[5] L. Li, H. Zhou, S. X. Xiong, J. Yang, and Y. Mao. «Compound Model of Task Arrivals and Load-Aware Offloading for Vehicular Mobile Edge Computing Networks». In: *IEEE Access* 7 (2019), pp. 26631–26640. DOI: 10.1109/ACCESS.2019.2901280 (cit. on p. 3).

[6] Christian Berger, Bjornborg Nguyen, and Ola Benderius. «Containerized Development and Microservices for Self-Driving Vehicles: Experiences & Best Practices». In: Apr. 2017, pp. 7–12. DOI: 10.1109/ICSAW.2017.56 (cit. on p. 3).

[7] Y. Wang and Q. Bao. «Adapting a Container Infrastructure for Autonomous Vehicle Development». In: *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*. 2020, pp. 0182–0187. DOI: 10.1109/CCWC47524.2020.9031129 (cit. on p. 3).

[8] *Kubernetes Documentation*. URL: https://kubernetes.io/docs/home/ (cit. on p. 4).

[9] Kim McMahon. *2019 CNCF Survey*. Blog Post. 2020. URL: https://www.cncf.io/blog/2020/03/04/2019-cncf-survey-results-are-here-deployments-are-growing-in-size-and-speed-as-cloud-native-adoption-becomes-mainstream/ (cit. on p. 5).

[10] *Raft Consensus Algorithm*. URL: https://raft.github.io/ (cit. on p. 9).

[11] *Kubernetes API Doc*. URL: https://kubernetes.io/docs/reference/ (cit. on p. 13).

[12] *Calico Documentation*. URL: https://docs.projectcalico.org/about/about-calico (cit. on p. 18).

[13] *ROS2 Documentation*. URL: https://docs.ros.org/en/foxy/index.html (cit. on p. 25).

[14] *FastDDS Repository*. URL: https://github.com/eProsima/Fast-DDS (cit. on p. 41).

[15] *Discovery Server Repository*. URL: https://github.com/eProsima/Discovery-Server (cit. on p. 42).

[16] *Topology Labels*. URL: https://github.com/kubernetes/enhancements/tree/master/keps/sig-architecture/1659-standard-topology-labels (cit. on p. 55).

[17] *Endpoint Slice Subsetting*. URL: https://github.com/kubernetes/enhancements/tree/master/keps/sig-network/2030-endpointslice-subsetting (cit. on p. 55).

[18] *Service Internal Traffic Policy*. URL: https://github.com/kubernetes/enhancements/tree/master/keps/sig-network/2086-service-internal-traffic-policy (cit. on p. 57).

[19] *Topology Aware Subsetting*. URL: https://github.com/kubernetes/enhancements/tree/master/keps/sig-network/2004-topology-aware-subsetting (cit. on p. 57).

[20] *Linkerd Doc*. URL: https://linkerd.io/2.10/overview/ (cit. on p. 61).

[21] *Istio Docs*. URL: https://istio.io/latest/docs/ (cit. on p. 61).

[22] *Go Docs*. URL: https://golang.org/doc/ (cit. on p. 66).

[23] *iwconfig manual*. URL: https://linux.die.net/man/8/iwconfig (cit. on p. 66).

[24] *KubeFed Repository*. URL: https://github.com/kubernetes-sigs/kubefed (cit. on p. 68).

[25] *Submariner Docs*. URL: https://submariner.io/ (cit. on p. 69).

[26] Joseph Redmon and Ali Farhadi. «YOLOv3: An Incremental Improvement». In: *arXiv* (2018) (cit. on pp. 69, 70).

[27] *Liqo Docs*. URL: `https://doc.liqo.io/` (cit. on p. 71).