POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Automatic deployment of security controls in software networks

Supervisors:

Prof. Cataldo Basile

Engr. Francesco Settanni

Candidate:

Alessandro Zamponi

Academic Year 2022/2023
Torino

Abstract

Over the past few years, companies and organizations with increasingly distributed IT infrastructures have been consolidating the investment trend toward the cloud computing paradigm, given its many benefits: the ability to develop and deliver applications and services dynamically and flexibly through a container-based approach. Concurrently, such service orchestration systems, foremost among them Kubernetes, are receiving increasing interest in effectively managing these resources in the new container architecture. The value proposition of this thesis is related to creating a security function orchestration system in Kubernetes via a software-defined approach, employing the Network Service Mesh (NSM) framework and defining a software network based on service chains. By making specific changes and configurations, chains that reflect the user's intent are created, previously defined through a Command Line Interface (CLI), and managed by a Kubernetes Operator. The latter represents a specific application controller ,which thus extends the basic functionality of Kubernetes, and must ensure the correct configuration of security functions corresponding to the user-specified intent, respecting the specifications received and providing all the necessary resources for implementation: the basic idea is to let the user configure the chain at a level that is as high as possible while giving at the same time a broader and easier to understand view of the network. A Knowledge Base is used to match intent and security functions so that among multiple types of resources, the one best suited to meet the security requirement could be inferred. Unlike what is available in the most common solutions in this area, through this approach, it is possible to operate at a lower level, and more specifically at the network level, since each intermediate pod in the chain makes two network interfaces available, resulting in a fundamental aspect to allow the containers assigned to security functions to monitor or simply apply filters to network traffic. As a result, a cluster is then obtained within which the desired chain is defined to protect a given service. Security functions were identified and chosen to follow a plug-and-play approach to achieve an effective chain configuration, testing them directly within a cluster using different configurations, including filtering rules. Regarding possible future developments, in addition to the extension of the KB with which the mapping is done and its additional configurations, it is possible to add and integrate other software features defined by the NSM framework to extend those already present. The available security features can be extended, respecting the limitation of falling within a single container to avoid possible conflicts with the NSM framework and architecture.

Ringraziamenti

Fino alla fine sono stato dubbioso se scrivere i ringraziamenti o meno, ma credo che a volte sia davvero importante fermarsi per una riflessione ed essere grati per ciò che si ha. Proprio per questo motivo voglio ringraziare mio padre e mia madre, che hanno sempre fatto di tutto affinché non mi mancasse nulla, lasciandomi la giusta libertà nelle scelte e nel pensiero; mia sorella Ludovica, per avermi fatto capire più con i gesti che con le parole, che lei ci sarà sempre per me e come l'amore fraterno rappresenta uno dei legami più forti che possano esistere.

Vorrei ringraziare la mia ragazza Francesca, la persona che più di tutte mi conosce nel profondo, che ha sempre creduto in me ed è sempre stata pronta a rallegrare ogni mia giornata storta, e che più di tutti può comprendere il peso di queste parole.

Un grazie speciale è rivolto anche alla famiglia Grassettini per avermi sempre accolto con amore nella propria casa e che oramai considero la mia seconda famiglia,

Ci tengo a ringraziare anche gli amici di una vita, come Alessandro M. , con cui ho condiviso davvero tanti bei ricordi, Thomas, uno degli amici più fedeli che abbia mai avuto, Riccardo S., l'amico migliore che avessi mai potuto desiderare e Alessandro S., il fratello minore che ho sempre voluto.

Per aver condiviso momenti che porterò sempre nel mio cuore, un grazie va anche a Jacopo, Emanuele, Valerio, Donato, Alberto, Tommaso, Marco, Riccardo C., Alice e Camilla, amici con cui ho avuto la fortuna di passare il periodo più importante della mia vita.

Infine, il ringraziamento più grande ed importante va a mia nonna Tita, la persona che più di tutti mi ha insegnato il significato della parola "amare" e a cui sento davvero di dover attribuire ogni mio traguardo.

Table of Contents

List of Figures

Acronyms

ACL
Access Control List

K8s
Kubernetes

SDN
Software Defined Networking

IBN
Intent Based Networking

NFV
Network Function Virtualization

CRD
Custom Resource Definition

NSM
Network Service Mesh

IDPS
Intrusion detection and Prevention Systema

CLI
Command Line Interface

NSE
Network Service Endpoint

NSC
Network Service Client

Chapter 1

Introduction

In recent years, one of the most significant trends has seen a constant growth in investments by companies and organizations to create cloud infrastructures capable of supporting heavy workloads while managing high dynamism within systems. Considering the process of increasingly virtualizing resources, solutions based on the cloud have gained more ground.

It is no coincidence that the various strengths enabled by these infrastructures are becoming increasingly interesting: dynamism and flexibility are two must-haves for every service offered in this field to compete with what's already present in today's market.

This outlines the characteristics that need to be considered and aimed for. To meet these needs, approaches to application development, such as those based on containers, have achieved notable success, given their characteristics, allowing the development of applications in a way that makes them easy to manage and transport across different environments. Containers indeed represent one of the viable choices for virtualization, differing from virtual machines, which might be more accessible for non-expert users but are one of the best solutions in this context.

However, considering the various user demands, which often vary significantly, and expecting a certain level of service quality, there has been a preference for application development models based on microservices architecture. This decision has paved the way to effectively combine the strengths of this architectural approach with those of containers, significantly aligning with user requirements. Managing each service individually, considering it an independent unit from the others, allows services to be more efficient, scalable, and portable, enabling efforts and work to focus on one unit at a time.

Moreover, as containers are lightweight, offering a certain number of replicas is not so costly, creating container redundancy to achieve greater resilience against possible failures or errors.

Considering this introduction, additional elements are needed in this context to define an approach and a development process that allow all these technologies to collaborate in the best possible way.

For this reason, it is necessary to introduce an orchestration system to manage the containers: given the increasing interest in this specific area, Kubernetes's ideal solution. Kubernetes, in particular, represents the focal point where the work for this thesis was carried out, through the introduction of an additional framework known as Network Service Mesh (NSM).

Temporarily overlooking the more technical details related to Kubernetes, the aim was to define software networks capable of leveraging security functions within the pods of a cluster.

Indeed, the main issue concerning this orchestration system is related to networking, which has limitations, especially when managing a high quantity of pods used for services. Configuring policies within clusters, considering different pods each time, becomes a rather burdensome task, particularly when considering the dynamic nature in which pods are created and destroyed.

To clarify this aspect further, pods are considered ephemeral, meaning they are deleted and subsequently recreated if an error occurs within the cluster. This prevents the utilization of information for routing purposes that might be considered static since, in the event of a pod restart, they would change completely. This is one of the reasons why networking within Kubernetes is extended through plugins or third-party solutions like Service Mesh.

Generally, solutions based on service mesh offer an additional layer used explicitly for networking and communication between different services: in the case of Kubernetes, pods implementing services have a necessary sidecar container to manage communication between these services. The presence of this sidecar container enables the separation of networking from the service's operation, simplifying traffic management.

Typically, solutions in this domain provide a central component to manage traffic routing, apply security criteria, and monitor the network. The core idea behind this work is to enable a user, even someone not very experienced in the field, to define a chain composed of various security functions that can be deployed within a cluster and subsequently utilize to protect a specific service.

With this objective defined, the aim was to provide a way to leverage this work from as high a level as possible, without delving too deeply into low-level details to configure what is desired.

In this sense, it is important to specify that the user wanting to benefit from such a service must outline the functionalities they would like within the cluster. For instance, these functionalities might include monitoring interfaces and traffic or simply applying

rules for filtering. An approach similar to intent-based networking was adopted to achieve this objective. Starting from specific intents defined by the user containing the desired functionalities, a translation is made into specifications that can be understood and used within the Kubernetes environment. What actually happens is the translation of these generally high-level intents into Kubernetes deployment files, encompassing all necessary specifics.

A dedicated Command Line Interface (CLI) was developed to enable users to define these intents. This CLI can be utilized to create a file containing the list of intents, essentially the desired functionalities, to be sent via APIs to an operator. The operator represents another key element in this work because it effectively enables the translated intents into specifications to be executed within a cluster. Once the user completes the definition of intents, they must send them to an API using a specific command. At this point, there will be an initial logical translation using a Knowledge Base of intents into instances of a specific Custom Resource Definition (CRD) for this resource type.

This approach allows the user to have a more precise and broader view of the network than a lower-level view, facilitating its management. Once the CRD instance is obtained, the operator will manage any necessary configurations required for launching the container containing the security function corresponding to the intent. In addition, once configured to monitor a specific type of resource, each operator specifies how to react to certain events, such as the resource's creation, modification, or deletion. Starting from a desired state defined within the CRD instance, the operator enables modifying which actions should be executed to bring the current state to the desired one.

To understand the real added value of this work, it is fundamental to comprehend why the main solutions based on service mesh available in the market did not represent suitable solutions for the goal. In fact, this was not achievable by aiming to work at a lower level, specifically networking, through solutions like Istio or Linker.

Most projects in this domain prioritize connections via the application layer, contrary to what happens through the Network Service Mesh framework, allowing the connection of a workload to multiple network services, each with its unique characteristics. To further the development of the work, it was necessary first to understand some of the features offered by the NSM framework, focusing mainly on those most helpful in building this chain. Subsequently, the approach involved designing a new feature based on one already present in the repository provided by the developers. This feature constructs a chain with different pods, each capable of hosting security functions.

By making some modifications to achieve the desired configuration, pods were created with two network interfaces to manage incoming and outgoing traffic, while simultaneously allowing the use of a third container to implement security functions. For instance, a container was configured with Suricata to enable monitoring of an interface and the associated traffic.

After introducing in general the idea of this work and specifying what technologies will

be used, all the notions necessary to be able to understand the work more technically will be presented in Chapter 2, so that the real potential can also be understood. Next, within the Chapter 3 an overview of the development of the work will be presented, describing what requirements were started from, and then arriving at the work actually done, also discussing the validation carried out. After that, the results obtained are presented in Chapter 4, describing what were the key aspects for the realization of the work, finally leading to Chapter 5, where the conclusions and possible future work related to this project are presented.

Chapter 2

Background

In recent years, the rapid expansion of telecommunications networks and the increasing availability of cloud computing resources have facilitated the emergence of new business opportunities. It is no coincidence that companies are beginning to move into the world of cloud computing, taking advantage of the resources it makes available and enabling the virtualization of most network capabilities by adopting available off-the-shelf (COTS) hardware, which minimizes the costs associated with them[1].

When virtualization and cloud computing are combined together, it opens up numerous possibilities for meeting as efficiently as possible the needs of users, who very often have dynamic demands. Adopting these types of solutions provides flexible architectures that are critical to meeting such requirements. Moreover, moving into the area of cloud computing, it is necessary to introduce a crucial concept known as *decoupling*: a notion that can be extended to both software and hardware, as well as between different software components. Decoupling indicates how much two different elements remain independent of each other. The importance of this notion lies in the opportunity to consolidate resources into a universal pool, so that they can be managed and shared through specific providers in a way that minimizes the effort for the end user. By adopting this concept, several benefits can be achieved, such as

- *High scalability*: to meet the demands, the necessary resources can be scaled up or down more quickly and efficiently. In doing so, only the resources that are truly necessary are effectively utilized, without the need to allocate extra resources.
- *Agility*: it is possible to work more freely on one element without later worrying about how to reflect the changes in the other elements.

However, in this case, possible drawbacks are also outlined as issues may arise regarding how to make the two different elements communicate or, more simply, finding the platform that

best suits the requirements. In a business context, following the decoupling concept allows for cost reduction in service management processes, providing better data management regarding user experience, security, and efficiency.

This underlines the importance of keeping this concept in mind, even from the perspective of future management.

This type of concept is then applied perfectly in the field of cloud computing, where scalability and flexibility are two key terms: what is intended to be achieved is flexible resource management tailored to the needs. Considering resource allocation as well, this idea comes back into play, as resources needed for various components are managed and allocated separately[2]. More generally, cloud computing aims at the distribution of services over the internet in a way that is resilient, scalable, and easy to use. The underlying idea is to use remote storage resources, overcoming potential limitations imposed by local resources. It is therefore necessary to identify the two main actors involved in this context: the *provider*, i.e., the entity that makes available the various resources that will then be used, and the *end user*, i.e., the one who makes a request for a given service and resources. In practical terms, the provider maintains a pool of resources that can be dynamically shared with various users through automated mechanisms, facilitating the release and reuse of resources upon task completion and adjusting them according to the actual load. This approach empowers end users to employ software hosted on remote machines without the need to install or configure anything on their local devices.

Just to give an idea of the potential that cloud computing, and related virtualization of resources, can have, here is a list of the possible types of services that can be offered:

- servers;
- data storage;
- databases;
- networking;
- software;
- analytics;

The objective is to make available all the required resources available to the clients, in a flexible manner that addresses their specific needs.

Another advantage that can be counted with the introduction of cloud services, is that **the focus shifts from delivering capabilities through specific products, to providing services that match the user's requirements.** This change gave users the opportunity to obtain desired resources through services instead of using specific products. In fact, if one goes to examine the traditional IT industry, it is evident that industry

giants such as Microsoft and Intel, responsible for providing operating systems and chips, respectively, dominate the market, conveying to users on the purchases they need to make to achieve their intentions.

2.1 Cloud service models

Having thus generally introduced cloud computing and its main characteristics, as well as the main reasons why it should be used, one can now describe the new types of services emerging. In this sense, through different levels of abstraction related to the resources and how they are offered, different types of services can be defined [3]:

- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)
- Software as a Service (SaaS)
- Hardware as a Service (HaaS)

IaaS

Through this type of cloud service, the provider makes storage or computational resources available based on the user's requests. Therefore, the latter will have virtualized resources such as servers or networks at their disposal, thus without having to purchase and configure them and allowing them to scale resources according to actual usage. It represents one of the most chosen solutions by companies because it allows them to focus only on the implementation part of applications. Another advantage of this model type is related to cost since the user only pays for what they use.

PaaS

Instead, in this case, a development and deployment environment is provided that is complete and inclusive of all needs. A platform is provided on which one can directly work, decreasing the time it takes to develop applications since the underlying infrastructure does not have to be configured. In relation to this last aspect, another advantage is related to the maintenance that is done, which turns out to be automatic.

SaaS

Instead, this type of model allows software to be made accessible to users directly from the Web, without having to install anything on the device. Users make software that is then made available by a cloud provider, making SaaS a rather efficient and flexible solution, simply through an Internet connection.

HaaS

Unlike the other services described above, in this case, providers provide the desired hardware, either through rentals or direct subscriptions. Through this model, the level of abstraction is much lower than through the others, although then the provider bears the hardware upgrade and related maintenance. The main reason this model is useful is that it avoids high upfront costs in terms of purchasing the necessary hardware components.

2.2 Virtualization

Looking now more closely at the context related to the telecommunications industry, however, one of the major aspects to consider is related to resource management, which very often turns out to be quite complex, as well as the configuration of all the equipment and the network.

To meet these types of challenges, one of the fundamental concepts that have been introduced is precisely that of ***virtualization***. In this way, a new solution was made available that would allow developers to build and take advantage of flexible cloud architectures, while also going on to simplify many aspects related to configurations and process management, contributing substantially to making the entire system more efficient. This is referred to as software-defined infrastructure, which allows telco providers to add or change services more quickly and simultaneously respond more readily to changes in network resources that might occur.

In this context, virtualization refers to the *process of translating hardware resources into software-defined definitions.*

Indeed, most telecommunication services require real-time processing, which must be performed with strict network performance requirements in mind. Therefore, the cloud operating system must provide low-latency communications to maximize service responsiveness and take full advantage of the capabilities of the underlying hardware.

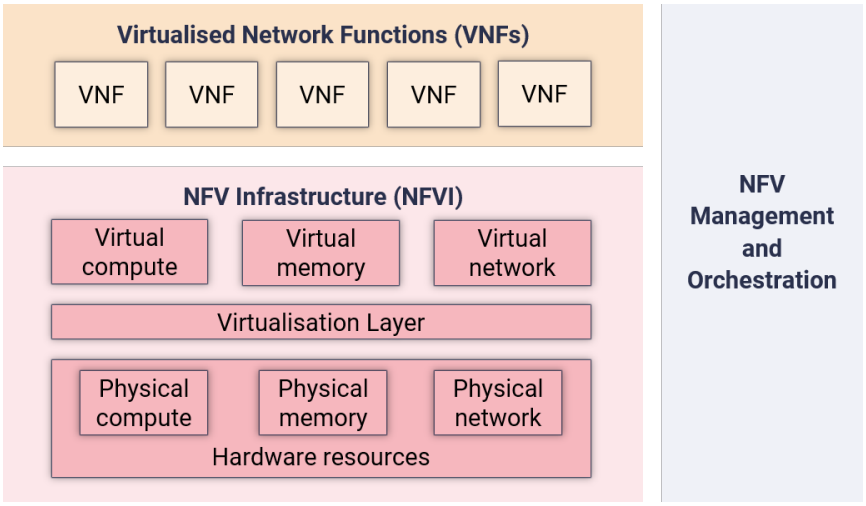

Figure 2.1: NFV schema

2.3 Network Functions Virtualization (NFV)

One approach to achieving virtualization of network services is the use of ***network functions virtualization (NFV)***. NFV allows common network services, such as routers, firewalls or load balancers, which normally require specific, dedicated hardware, to be transformed into virtual machines (VMs). This makes it possible to run the latter on standard COTS hardware, rather than on proprietary systems. Following this kind of approach eliminates the need to have dedicated hardware for each network function, improving system scalability and enabling on-demand delivery of network services and applications. Just to make the potential clear, suffice it to consider a simple example where network cards are involved: in a standard context, a single network card would have to be configured for each user following virtualization, one of the advantages is that multiple virtual network cards can be created to deliver different services, without tying each service to physical hardware As can be seen from the figure, to achieve hardware virtualization, it is necessary to place a layer on top of it to manage the requests and the resulting virtualization. Consequently, corresponding virtualized versions are created in response to these requests and made available for executing the desired function. Therefore, with NFV (Network Function Virtualization), multiple functions can coexist on a single server, significantly reducing the required hardware. Furthermore, it is worth considering the additional advantage of NFV, which is to provide flexibility to run these functions on different servers or be moved as needed, enhancing the adaptability and efficiency of network operations.

2.3.1 Security functions

As already anticipated, through the virtualization of network functions, it is possible to make available what are the most popular services without, however, using dedicated hardware. For this reason, it is important to describe the most widely adopted functions, especially in the area, such as firewalls or intrusion detection systems. of security.

Firewall

Firewalls arise from the specific need to make connections that can be considered secure. Particularly in more modern applications, firewalls are primarily tied to connections made via TCP/IP protocols, even though they have functionality that does not rely on Internet use [4].

In fact, network firewalls are devices or systems that allow the flow of traffic to be controlled by adopting well-defined patterns and thus differentiating between the various types that may be required. Firewalls that thus allow working on multiple layers of the OSI model are the ones that then have the most effect within applications since they offer greater coverage. The most basic type of firewall, and in a sense also the fundamental one, is called a packet filter, which can be seen as a routing device, to which, however, features are added to perform access control functionality. Through defining certain rules, which thus define the policies that are to be applied within the firewall, directives can be specified by which the firewall can block or not block certain types of traffic. Generally, packet filters work on the network layer, although in reality, through the type of traffic, they can also operate on layer 2.

Two features of packet filters that represent strengths are the speed and flexibility with which they can operate, thus facilitating related deployments within network infrastructures. In addition, it is important to consider how this type of firewall, can be considered the first line of defense of an application or system, which in case of possible attacks, can filter out what are the protocols that are not accepted by the protected network and then pass the remaining traffic to another firewall that will, however, work on the higher layers. This last point is important because packet filters do not examine the layers above the network layer, and thus cannot prevent possible attacks involving the application layer.

Intrusion detection/prevention system (IDPS)

One of the peculiarities of cloud environments is related to the various types of attacks that can occur, especially with regard to unauthorized access to user data or services. For this reason, it is also important to mention security features to protect systems from intrusions, while also giving a way to notice possible attacks in order to prevent them later[5]. In this sense, intrusion detection and prevention systems are increasingly being

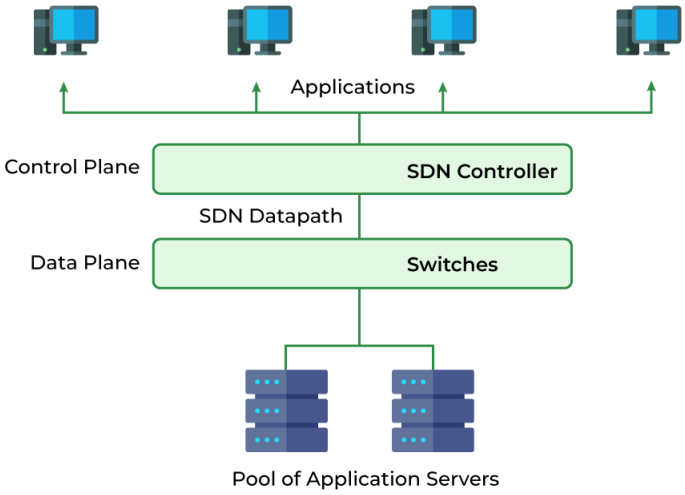

Figure 2.2: SDN architecture

adopted to monitor traffic and possibly block suspicious traffic. In this sense, IDS has the task of monitoring and making ambiguous network behaviour known, while IPS, has to put actions in place to be able to respond to these suspicions. Again, there are different types of IDPS and the two most important ones are:

- *Host-based IDPS*: in this case, it refers to a system working on a specific machine and that's it. It checks and monitors logs, logins, or system configurations so that any changes can be tracked and any attacks notified that otherwise would not be visible by other types of IDPS.

- *Network-based IDPS*: this type of system represents a passive component placed at a given point in the network with which all traffic can be monitored. If it detects suspicious activity, it generates alerts with which mechanisms can be triggered to respond.

2.4 Software-Defined Networking (SDN)

Another main topic to consider is Software-Defined Networking (SDN). Unlike NFV, SDN primarily focuses on ***centralized network management and control:*** thanks to SDN, it is possible to make routing decisions in a highly flexible manner, taking into consideration the overall view of the network. This enables those responsible for managing the network and its operation to have the widest possible perspective, helping them make choices on how to configure the various elements.

The SDN architecture comprises several key components, including a ***control plane***, which plays a central role in determining the routing of network traffic, and a ***data plane***, where actually the packets are. This architecture significantly enhances the overall performance of network systems. Another essential element within this architecture is the SDN controller, which network administrators can leverage without needing to delve into the details of individual components like switches. SDN is now seen as a step forward in managing traditional networks because it offers the following advantages:

- *Enhanced control, speed, and flexibility*: as mentioned earlier, developers do not need to program individual switches but can simply program a software-based controller.
- *Customizable infrastructure*: with this type of network, real-time changes can be made, services can be configured, and virtual resources can be allocated based on needs.
- *Robust security*: Having a broader view of the entire network allows for a better understanding of potential threats or risks within the network.

Furthermore, as SDN is based on software definition rather than hardware, it allows for centralized configuration without the need for additional hardware.

In practical terms, SDN is typically deployed in data centers, where a broader view of things is required, while NFV finds more relevance within the domain of service providers, maximizing the size of the available resource pool.

2.4.1 SDN and Cloud Computing

Opting for cloud computing solutions without Software-Defined Networking (SDN) can pose several challenges, especially from the perspective of service providers. While the end-user experience remains unchanged, as they simply need to request specific resources, functions, or virtual machines (VMs), the provider must deal with additional configurations required to ensure that services are delivered correctly, considering factors like resilience and reliability.

In this context, SDN provides an overarching view that is crucial to having a comprehensive network picture, allowing for effective traffic control and management. As demands increase, the amount of required resources grows, and the complexity of traffic management, such as load balancing, also increases. For these reasons, having a solution that enables rapid resource scalability, as well as the provisioning of additional resources to match dynamic demands, becomes imperative. Given the growing necessity, centralized control is required to coexist various virtual devices and automate the provisioning of functions where possible.

An example that further underscores the importance of adopting cloud and SDN-based solutions is related to security: when considering the addition of a firewall, intrusion detection systems, intrusion prevention systems, or simply an antivirus system, the likelihood of configuration errors increases.

This convergence marks the intersection of cloud computing, SDN, and Network Function Virtualization (NFV), creating a strategic synergy to address these challenges[6]. To provide the clearest possible context, NFV is necessary to enable the virtualization of

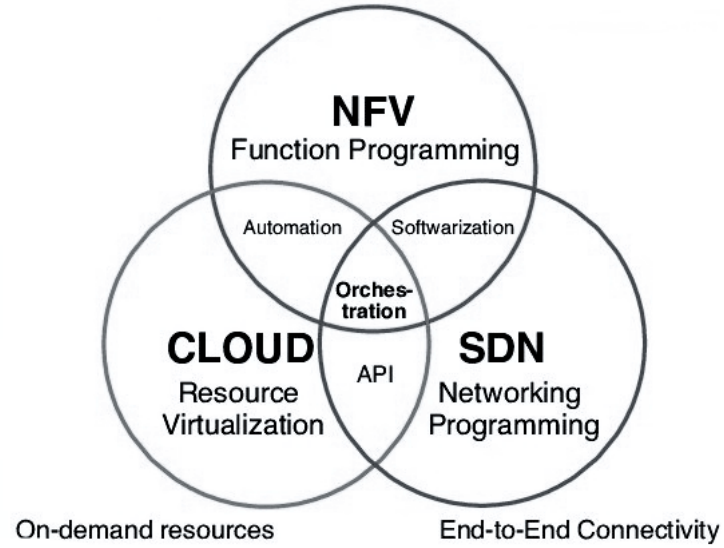

Figure 2.3: Relationship between NFV, Cloud Computing, and SDN

functions for use and delivery to the end user. SDN remains crucial in allowing higher-level network management in the working environment. Combining these two solutions with the benefits offered by cloud computing results in a highly useful solution for developing areas such as the Internet of Things (IoT).

2.5 Microservices

In this context, however, it is necessary to introduce another rather important concept to understand both the state of the art and the direction in which companies are heading today: *Microservices*. Microservices are a solution that increasingly meets and solves the growing needs in terms of scalability and parameters that must be met, for the applications that are provided by organizations. The idea on which the microservice architecture is based is basically to structure an application as a series of services with very specific characteristics:

- microservices can be implemented and deployed independently of the others. Each element could be made in a different language, eliminating possible restrictions related to implementation.
- the services must be as independent as possible, i.e. the malfunction of one service should not affect another service.
- they are designed in relation to the company's capabilities.

In this way, the developer can manage small modules rather than entire applications, introducing new technologies and then adapting the components more easily. Also from a security point of view, by separating each component, data separation can be achieved by linking each module to its own database.

Obviously, also in this case it is necessary to mention the possible problems that could occur, such as:

- first of all, in order for this type of architecture to be effective, it is necessary to have a rather efficient management of requests, coordinating the different modules so that there are no errors
- Having a copious amount of different modules could become problematic for monitoring and configuring services. Similarly, if are required debugging actions, they may take some time because logs of different services or components would have to be checked.
- Even the aspect of automating the entire architecture could take some time.

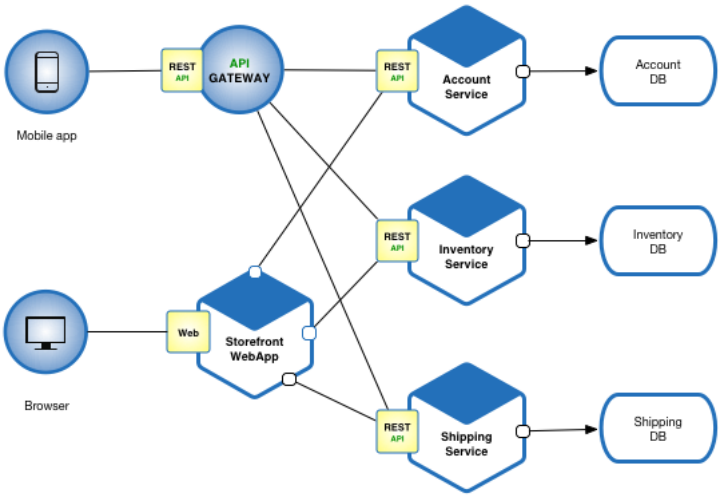

Figure 2.4: Example of Microservice architecture

Without considering this kind of approach, a possible issue lies in the implementation provided for Virtualized Network Functions (VNFs), which are based on Virtual Machines (VMs). However, this approach introduces several inefficiencies, making frequent tasks such as system restarts and updates rather slow. Additionally, VNFs may not offer the ideal scalability, especially when compared to the benefits of orchestration[7]. Imagine a scenario in which a service must operate on a large scale, serving numerous users and allocating a substantial amount of resources; the overhead from virtualization is significant, thus compromising service quality.

A transition to Cloud-Native Network Functions (CNFs) has become essential to these challenges. CNFs are essentially improved versions of VNFs, optimized to operate within virtual cloud environments. They are also based on a micro-services architecture and are designed to be encapsulated within containers, making orchestration easier through systems like Kubernetes. Indeed, very often, cloud computing and micro-services architecture are used simultaneously, optimizing the qualities of both technologies to the fullest. It has already been mentioned how scalability is a fundamental aspect of these technologies: in particular, cloud computing aims to provide the infrastructure to manage the scalability of various services.

Another example of how these two concepts are closely linked is related to cost management, where cloud computing allows paying only for the actual use of resources, while micro-services enable more efficient resource management to facilitate this type of payment.

The micro-services architecture is centered around a collection of individual services, each isolated within its container. This approach offers greater flexibility and efficiency, resolving the issues associated with VNFs and ensuring a smoother and more scalable network function environment. Each microservice can be managed as an independent element, without affecting other services or having a negative impact on customers. Therefore, restarting the relevant container is sufficient without affecting others if updates are required for a specific function or service. Similarly, by requiring fewer resources than virtual machines, new containers can be instantiated, creating a form of redundancy to ensure uninterrupted service.

Furthermore, it is common practice to consolidate multiple micro-services within a single container to meet specific performance requirements. This strategy is particularly useful for resource optimization and the efficient delivery of desired performance levels.

Adopting a cloud-native architecture introduces several advantages, including:

- ***Distributed Microservices:*** Following a microservices-based approach involves exposing APIs to leverage them. These microservices are controlled, versioned, and documented, allowing their use in different contexts. Implementing new technologies or replacing certain components is also not to be underestimated, as this architecture makes it easy to incorporate new components.

- ***Scalability:***Being a key term in this context, leveraging technologies such as containers in addition to those mentioned earlier enables the creation of an architecture capable of independently handling each micro-service.

- ***Operational benefits:*** Considering the aspect of containerized applications, performance can be improved compared to using virtual machines by eliminating the overhead introduced by the need for a hypervisor and promoting container features.

- ***State separation:***: Another aspect to consider is the separation between stateful and stateless services. In the first case, difficulties arise due to the need to save the service state in a specific way and place. Beyond this initial challenge, the service state must be kept consistent, portable, and available.

- ***Lifecycle management:*** Adopting containers and a microservices-based architecture allows for the separation of the lifecycle management of each service. for each service.

- ***Performance Improvements:*** Adopting container technology makes it possible to virtualize application processes, making them lightweight since each container shares the same operating system. This leads to improvements when starting or updating services. operations.

- ***Service discovery:***Considering the possible drawbacks of these solutions, service discovery is a key element as it allows the use of a registry to register all available services. This facilitates dynamic service orchestration, especially when using Kubernetes, which further enhances this aspect.

- ***Availability and resiliency:*** To ensure a certain quality of cloud-native applications, using containers enables faster recovery from potential disasters compared to the time required when using VMs.

2.6 Docker

As mentioned earlier, containers represent another solution in the context of virtualization, leveraging system-level virtualization. This allows the use of a single operating system instance shared among multiple isolated containers. Each container will, of course, have its own file system, providing a distinct environment for applications.

One of the most popular platforms for creating, managing, and distributing containers is **Docker**. Through Docker, it is possible to achieve a solution capable of isolating applications within a container and running them on various infrastructures: it enables the concurrent execution of multiple containers on a single host[8].

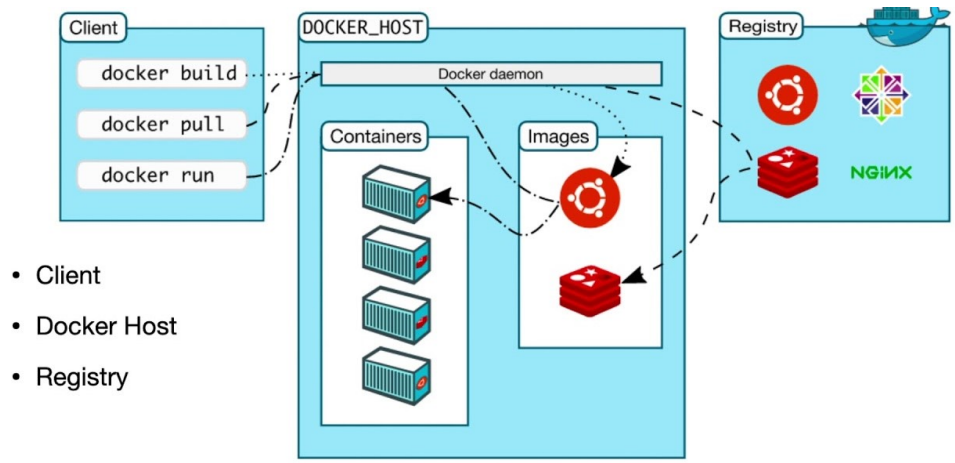

Figure 2.5: Docker architecture

2.6.1 Docker architecture

To better understand how Docker works, it is necessary to introduce the general architecture. There are various elements that collaborate to create containers:

- Docker engine: the engine is considered the crucial component of Docker, as it is the daemon that manages containers. It is responsible for handling interaction with the operating system's kernel. Within the Engine, the following components are present:
  - Dockerd: this is a background process responsible for container management, communicating with the Docker client through a set of APIs.
  - Docker Client: it provides a CLI through which the client can interact with the Daemon to create or manage containers. It utilizes the set of REST APIs to communicate with the Daemon.
- Docker images: they represent another fundamental element for containers. An image is an executable package with everything needed to run an application, including dependencies.
- Containers: they represent a running instance of a Docker image
- Docker registry: they represent registries where it is possible to upload or download images in such a way as to have centralized repositories available. Generally, they are public but can also be configured for private use.

2.6.2 Docker image:

Containers are created from images, which serve as read-only templates containing all the necessary instructions for building a Docker container. Image construction is done using a *Dockerfile*, a text document configured to contain all the commands needed to create the image. It is important to remember that the instructions are executed in the order in which they are written: for this reason, a precise order should be considered. Obviously, this file must have a specific format with dedicated instructions and commands such as:

- FROM: each dockerfile must begin with the FROM statement, which, however, can be placed after reading several directives, variables, or arguments. In a nutshell, the FROM statement specifies the parent image from which the current image is being built.
- ARG: this instruction is used to define global variables. Since the declaration of the variables is thus outside the build stage, characterized by the *FROM* statement, it makes it possible to use them as if they were global variables and integrate these variables within the FROM statement.
- RUN: another fundamental instruction is *RUN* which can be used mainly in two forms:

  ```
  RUN <command to execute>
  RUN ["executable-file", "params"]
  ```

  This instruction executes any command passed to it by creating a new layer on top of the current one and then saves the result of what it gets. At this point this result is used as a starting point for the creation of subsequent layers.
- CMD: an additional rather important command in the context of writing a Dockerfile is the *CMD* command: in general there can be only one such command but if more than one was used, exclusively the last one would take effect as an instruction.
- ENV: next, it is possible to find the *ENV* command with which define environment variables, through key-value pairs, that can be used within the build stage.
- ADD: another widely used instruction is *ADD* with which you go to add new files, instructions or files from a specified source to a destination within the image file system.
- COPY: similarly, there is the *COPY* instruction with which, following the same approach as *ADD* instruction, you go to copy files within the container.
- ENTRYPOINT: using this instruction a container is executed as it were an executable. It specifies the executable to run when the container is started.

Through the use of these commands, it is possible to define a docker file that will then be used to build an image that may or may not be shared in public registries. However, in order for the result to then be valid, one must always keep in mind what additional libraries must be downloaded in order for the program inside to work or, more simply, what packages must be added to the base image to make certain configurations.

2.6.3 Container's isolation

Having mentioned several times among the main features of containers is the isolation that is achieved between containers, it is also necessary to understand how this is made possible. There are two main elements that need to be introduced and explained: *namespaces* and *control groups(cgroups)*.

Namespace

Regarding namespaces, these represent a kind of virtual boundary in an operating system. Namespaces are characteristic of the Linux kernel and there are several types such as PID, Mount, and Network, with which one can isolate respectively the processes of a container, the mounted file system and the network interfaces or routing tables. In fact, each container will have a namespace dedicated to it, where it can run processes in isolation, and configure the filesystem, network parameters and various user profiles. By doing so, everything that happens within a container remains confined, without affecting other containers in any way.

Control groups

On the other hand, as far as cgroups are concerned, the latter represents a kind of delimitator, with which the limits of use of a specific resource are established and controls on it are configured. In this way, potential overloads can be avoided and monitoring activities can be carried out in order to evaluate performance. Another key aspect is that through the use of cgroups, it is possible to have a guarantee that containers will not interfere with shared resources in the system.

These attributes position Docker's containers as fitting for running NFV, accommodating various applications and services. Furthermore, Docker's lightweight nature and swift deployment capabilities align seamlessly with the agility and resource optimization requirements of NFV. So it is possible to virtualize the network functions and run them within a docker container. This adds a new piece to the puzzle where some parts are still missing, such as Kubernetes and its networking.

2.7 Kubernetes

Kubernetes is a portable, extensible, open-source platform designed to simplify and automate the configuration and management of containerized services[9]. As previously mentioned, containers are akin to virtual machines but employ a lighter isolation model, sharing the underlying operating system (OS) among various applications. Additionally, they provide decoupling from the underlying infrastructure, enabling portability across different cloud service providers.

2.7.1 Control Loop Logic

Kubernetes operates using a declarative approach. In most scenarios, the developer aims to achieve specific features within a Kubernetes Deployment, leaving the technical details necessary in an imperative model to Kubernetes. This eliminates the need to specify service delivery's 'where' and 'how', focusing solely on the desired end result.

From a higher-level perspective, developers are primarily concerned with the application in a more general sense, ensuring a continuously functioning instance of the service without causing any interruptions. Following such an approach allows services to possess essential attributes such as elasticity, robustness, and fault tolerance.

But how can the service always be in the desired state as described by the developer? From a more technical standpoint, an intermediate element known as a ReplicaSet is responsible for verifying that the characteristics defined in the deployment are effectively met. A ReplicaSet is defined through various fields, including a selector necessary to specify how to identify the pods it should monitor, the number of replicas to ensure, and a template describing the type of pod to be created. This way, the ReplicaSet can determine what type of pod should be configured when it is necessary to reach the desired number of replicas. In summary, a ReplicaSet ensures that a certain number of pod replicas are running at a given moment. However, Deployments are preferred as they are higher-level and provide declarative updates. Considering instead a higher level perspective, another essential concept is the *Control Loop logic*, a non-terminating loop that observes and responds to specific events, guiding the current state towards the desired state. This type of loop is based on three main steps:

1. Observation: where the state of the desired object is monitored.

2. Checking Differences: all differences between the current and desired states are identified and analyzed.

3. Action: all necessary actions are taken to align the current state with the desired state.

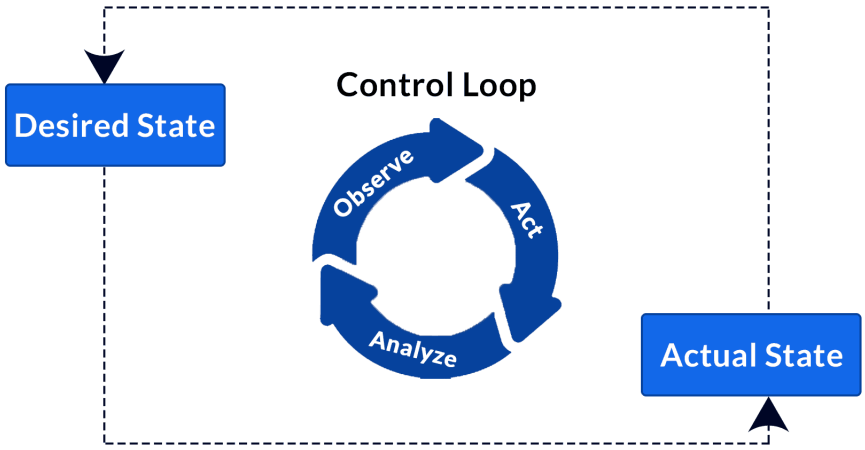

Figure 2.6: Control Loop Schema

Subsequently, this cycle of actions is repeated to ensure that the desired state always matches the current one. Additionally, Kubernetes doesn't adopt a single loop but uses several of them simultaneously, each configured for a specific task.

2.7.2 Kubernetes Components

Moving deeper into Kubernetes, it is important to understand the various components involved and how they work together. Firstly, the main element to deploy and work with in Kubernetes is a ***cluster***: it is composed of a set of nodes, also known as worker machines, where containerized applications are placed. These applications are packaged within a pod, which is the smallest object that can be found inside a cluster. For each node, there should be a control plane in charge of managing all the necessary services and resources that have to be provided to the node to run pods. It is worth mentioning that the majority of the times a pod can be considered equal to a container, but there are cases where more containers can be placed in the same pod. As can be seen in the figure, the main components of the Kubernetes cluster can be dived in ***Control Plane Components*** and ***Node Components***.

Control Plane Components

For what concerns the various elements that make up the Control Plane, have functionalities related to global decisions within a cluster, including planning, event detection, and

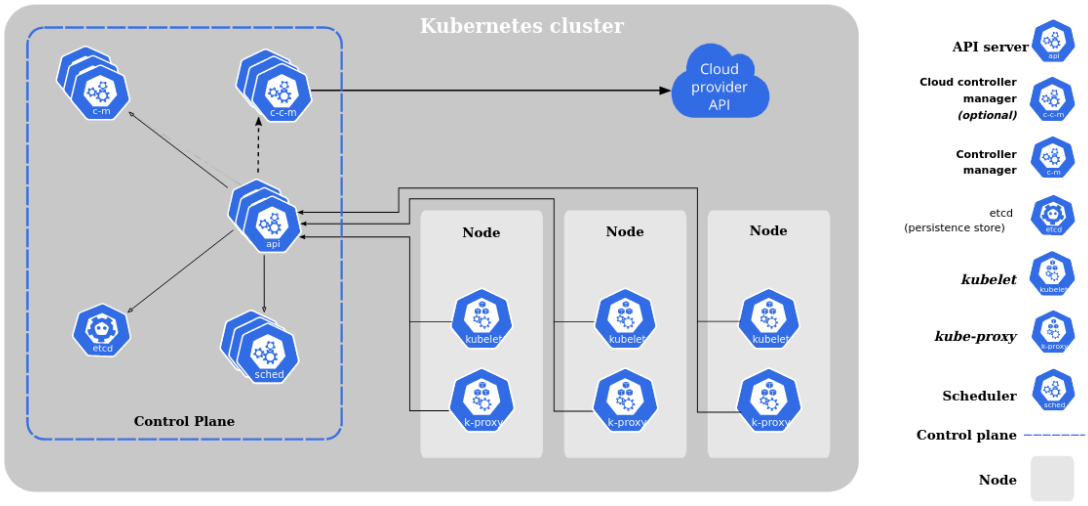

Figure 2.7: Kubernetes Components

response, in order to achieve the desired state. Additionally, these components can be configured to run on any machine within the cluster, but for the sake of simplicity, they are started on the machine without running user containers inside it. Below are described the various components:

- *Kube-apiserver*: it is used to expose the Kubernetes API, representing a sort of front-end for the cluster.

- *etcd*: it is the backing store for all the cluster's data, where a key-value schema is used to store the information.

- *kube-scheduler*: the main function of this component is watching for the newly created Pods and assign to them a node to run on.

- *kube-controller-manager*: this component runs controller processes, and assigning a different process to each controller is important. However, in order to reduce complexity, a single binary is used and run in a single process.

- *cloud-controller-manager*: it is specific for controlling the cloud control logic. It is the component in charge of linking together the cluster and the cloud provider's API. It runs only controllers that are specific to the cloud provider. Also, in this case, the several logically independent control loops are combined together in a single binary and run in a single process.

Node Components

Moving to the elements that characterized the node, each of the following components is running on every node and is needed to maintain the running status of the pods, providing the Kubernetes runtime environment. The components are:

- *kubelet*: it runs in each node in the cluster and ensures that containers are running in a Pod. It takes some specifications thanks to PodSpec field present in the yaml files provided during the deployment of the cluster or the service, and makes sure that the containers described in the section *PodSpecs* are running and healthy.
- *kube-proxy*: it is a component that acts as a network proxy, running in each cluster node. Its main function is to maintain network rules on each node: these rules are important to allow network communication among Pods, from sessions inside or outside the cluster. It uses the operating system packet filtering layer, if there is one available, or forwards the traffic directly.
- *container-runtime*: represents an important component ensuring containers are run effectively. It manages the life cycle and the execution of containers in the Kubernetes environment.

Now that the main components have been introduced, it is possible to move on to describing the various workloads within Kubernetes. Essentially, a workload is an application running within Kubernetes and can be represented by a single component or more, configured to work together.

In any case, a workload must be executed within a series of Pods, which, as mentioned earlier, are running containers.

A clarification in this context needs to be made regarding the life cycle of pods: considering the case in which a node experiences an interruption or an error occurs, the pods inside it will fail. Therefore, a new pod will need to be created to obtain a new functioning pod.

2.7.3 Deployments

In any case, the developer has a set of workload resources available to configure controllers to achieve the desired state.

By default, Kubernetes provides different types of workload resources, such as:

- Deployment and ReplicaSet: represents the best choice for developing stateless applications, where each pod can be replaced without issues.

- StatefulSet: on the other hand, this type of resource is suitable for developing stateful applications where it is important to preserve the state of a pod from time to time. This is where *persistentVolumes* come into play, but they will be discussed in more detail later.
- DaemonSet: represents a useful resource type for specifying the deployment of a pod when a node matches a specific DaemonSet.

In addition to these types of resources, it is possible to define other kinds of deployments using the Custom Resource Definitions, which are needed when the developer wants some specific behaviors or characteristics that are not standard in Kubernetes.

2.7.4 Pods

Since the concept of pods has been mentioned repeatedly up to this point, now more details will be provided on what is the smallest unit that can be deployed within K8s.

As previously mentioned, a pod is represented by one or more containers that share storage and network resources, as well as how the container is executed. Pods are usually created through workload resources rather than being created directly.

Pods can be used in two main ways:

- With a single container: this is the most common model where the pod acts as a wrapper for the container.
- With multiple containers inside that need to cooperate: in this case, the various containers play different roles and tasks, contributing to the realization of a specific service.

Furthermore, a pod can be subject to replication for horizontal application scaling: multiple replicas of the same pod are created to cover multiple instances of the same application. Another important characteristic to remember when talking about pods is that they are ephemeral resources and remain running on the node where they were scheduled until they either finish their execution or encounter events that cause them to fail. For what concerns the description of the pod, a template can be used to specify the requirements that it should have. When this template is modified, a new pod with the new configurations will be created, thanks to the mechanisms provided by the controllers.

Some pod specifications can also be changed at run-time using *patch* or *replace* methods, but there are various limitations, such as the immutability of many fields.

Additionally, it is possible to specify a set of volumes that can be shared, and all containers inside the pod can access the data contained within them.

Regarding networking, each pod is assigned a unique IP address shared by the containers inside it. Therefore, when you want to establish a connection between the containers within the pod and an external entity, it is necessary to correctly configure the usage of network resources. Considering the unique IP address assigned to each Pod, Kubernetes allows for avoiding creating links between various pods to enable communication, significantly reducing complexity. In fact, each pod can communicate with all the other pods within the same node without resorting to Network Address Translation (NAT). Additionally, every agent operating on a node can reach all the other pods within the node.

2.7.5 Service

After explaining how applications can be created within the scope of Kubernetes, it is necessary to introduce how they can be exposed externally, so that they can also be contacted externally by the cluster. For this reason, the services have to be introduced, which have the primary purpose of making the applications running in the pods available outside the cluster. Services are essentially abstractions required to expose the pods where the application is deployed, defining the policies for access and usage. To make a service work, selectors specified by the developer are used to identify which pods should be exposed. Kubernetes offers different types of services, each with its use cases:

- Cluster IP: With this type of solution, the service is exposed within the cluster and is only reachable from within it. If the service type is not specified, this is the default value.
- NodePort: In this case, the service is exposed on every Node IP, specifying a static port known as NodePort.
- LoadBalancer: The service is exposed externally using a load balancer, which needs to be provided because Kubernetes does not implement one.

Considering the case where the Cluster IP services are used, it is possible to make the service available externally through the usage of Ingress or a Gateway.

Ingress essentially involves a solution where services are configured to have URLs that can be reached externally to the cluster. HTTPS or HTTP routes are exposed, connecting the external client to the service. Additionally, rules for the routes can be defined through Ingress resources. It is important to note that an Ingress controller is required for the Ingress to function effectively. The rules specified to configure the Ingress include various information such as a host to which the rules should apply, a backend to which requests are sent, and a series of paths associated with a specific backend.

On the other hand, the Gateway API, being another solution for managing traffic, is conceptually different from Ingress. It represents a centralized way of handling API traffic,

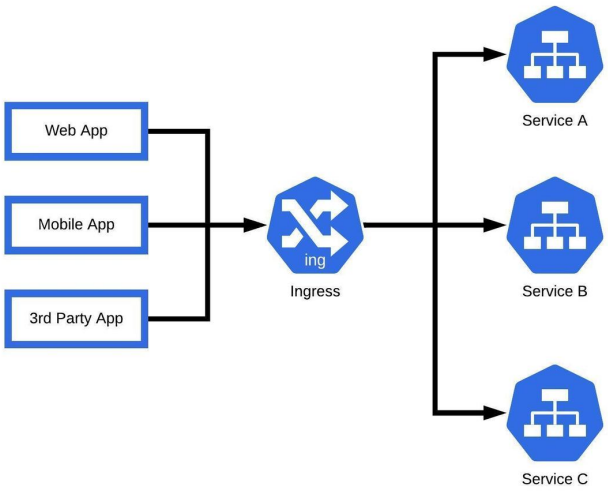

Figure 2.8: Ingress schema

simplifying the management of APIs. The Gateway API is specifically used for managing API-related traffic, offering the ability to configure advanced routing, authentication and authorization techniques, as well as message modifications.

2.7.6 Storage

In the description of the various characteristics of pods, it was mentioned that they are ephemeral and, as such, do not save the state of information once the pod terminates. Another example where this characteristic could be a problem is when is needed to share files within the same pod among multiple containers: managing the setup to access a shared resource can be a problematic task. For these reasons, Kubernetes can use volumes, including different types. There are two main types of volumes: Ephemeral volumes, which have a lifetime equal to that of a pod, and persistent volumes which allow saving the state of a pod even after it is no longer in a running state.

Ephemeral Volumes

Ephemeral volumes are primarily used when applications need additional space, such as when files need to be available for reading configurations useful to the pod. Some types of available ephemeral volumes in Kubernetes are:

- EmptyDir: In this case, the volume is created when a pod is assigned to a node. The

volume is initially empty but can be read and written to by containers within the volume.

- ConfigMap: It is mainly used to make data for configurations available to the pod. In these cases, data is saved within a ConfigMap volume and utilized by the container inside the pod. To exploit this type of volume, the name of the ConfigMap must be specified in the *Volume* section of pod's template. An important annotation is that each ConfigMap is always mounted in read-only mode.

Persisten Volumes

Analyzing persistent volumes (PVs), a portion of the storage within the cluster is made available so that this type of volume has a life cycle not tied to that of the pod. Persistent Volume Claims (PVCs) can be defined, which are user requests that consume resources provided by persistent volumes, specifying the required space and the type of access desired for these resources.

Persistent volumes can be provided in two main ways: dynamically or statically. The first case refers to situations where PVs fail to meet user requests made through PVC. For this reason, volumes can be dynamically provided to users through storage classes. In the second case, a series of PVs are made available that can then be used by various users.

2.7.7 Networking in K8s

However, unlike other platforms, Kubernetes has decided to adopt a flat network model, removing the need to map host ports to container ports. Considering the dynamism present within Kubernetes, solutions that rely on the *Dynamic Host Configuration Protocol (DHCP)* can slow down processes: in fact, it is not possible to consider static IP addresses or ports to perform hard-coded configurations, demonstrating the need for a different kind of solution. Considering this type of model, K8s shows how third-party tools are necessary to implement a suitable network model, that can facilitate the realization of the administrator's intent. Among the most popular there are:

- Flannel: provides an agent to be placed on each host, which will then take care of configuring subnets and managing the data for configuration using the etcd component of K8s.
- Calico: allows the creation of scalable solutions, connecting pods to each other, and ensuring policies that can be used.
- Weave Net: it is used for creating virtual networks, but is proprietary.

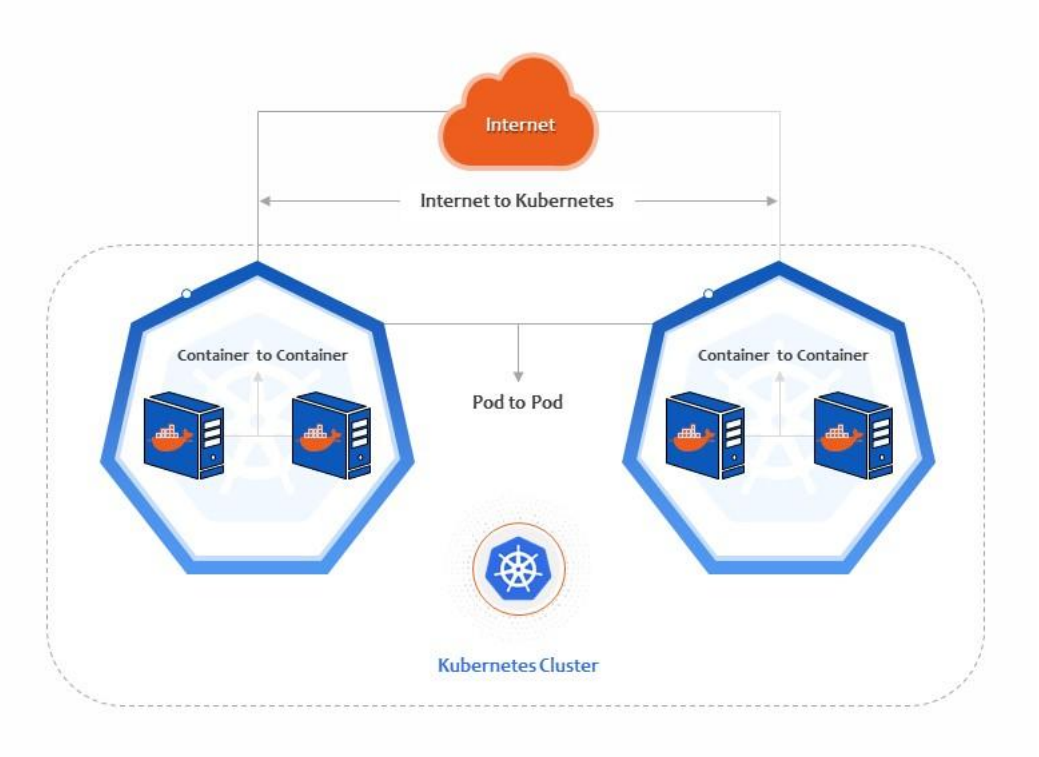

Figure 2.9: Kubernetes Networking Model

2.7.8 How it works

Now that all the major Kubernetes components have been introduced and described, a demonstration of how the entire system works can be done. Starting from the highest level, the main resource the developer goes to work with is the deployment, written through a manifest in *yaml* format. A formal description of the expected outcome can be given through this type of file, such as the number of replicas for a specific pod, how it should be externally exposed, what ports should or can be used, and additional configuration files. Other key fields are those related to the containers that are to be deployed within the pod, as the developer must specify which image to use, details related to the privileges the container must have, and the resources it can consume or needs. As can be seen from figure 2.10, it is also possible to specify how data updates related to the pods should be handled or which ones are affected by the deployment through the use of labels

Once the definition of the deployment is completed, it must be applied within the cluster, through specific commands provided by *Kubectl*. Kubectl is a command-line tool that allows commands to be executed within Kubernetes clusters. Among other features, in addition to applying deployments, pods logs can be inspected through kubectl, as well as visioning and managing resources. Once the deployment is initiated, it is possible to

```yaml
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: nginx-deployment
spec:
  # A deployment's specification really only
  # has a few useful options

  # 1. How many copies of each pod do we want?
  replicas: 3

  # 2. How do want to update the pods?
  strategy: Recreate

  # 3. Which pods are managed by this deployment?
  selector:
    # This must match the labels we set on the pod!
    matchLabels:
      deploy: example

  # This template field is a regular pod configuration
  # nested inside the deployment spec
  template:
    metadata:
      # Set labels on the pod.
      # This is used in the deployment selector.
      labels:
        deploy: example
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
```

Figure 2.10: Example of a deployment yaml file

interact with the pods, again using kubectl, to execute commands inside the containers, allowing further configurations.

2.7.9 Microservices in K8s

The development of rather complex applications can be broken down into the creation of multiple components, each with a specific task, crucial for achieving the final goal.

Generally, there are two main approaches that can be followed for application development:

- Monolithic: In this approach, the development of the application is based on a single unit and is distributed as such. In this case, it is easier to develop, test, and

perform debugging activities. However, at the same time, it results in an less scalable application and may have more complicated maintenance.

- Micro-services-based: In contrast to the previous one solution, the application is developed by decomposing it into multiple autonomous services, each of which manages a specific functionality. In this case, all the advantages and disadvantages related to micro-services architectures are obtained, such as scalability, which is generally the main feature in this context.

Following a microservices-based approach, it is evident how containers represent additional support for implementing an application, given the properties that characterize them. Consequently, Kubernetes, being an orchestrator for containerized deployments, becomes another key element for the realization.

In particular, within Kubernetes, two additional components are made available to enhance this type of distributed application:

1. *API Gateway:* this is a specific component designed to handle all requests coming from the client. This gateway is responsible for forwarding the request to the required microservice, acting as a single entry point.
2. *Services Mesh:* these represent additional network layers with the primary task of managing communication between different services. In doing so the networking aspect is separated from that of the application itself.

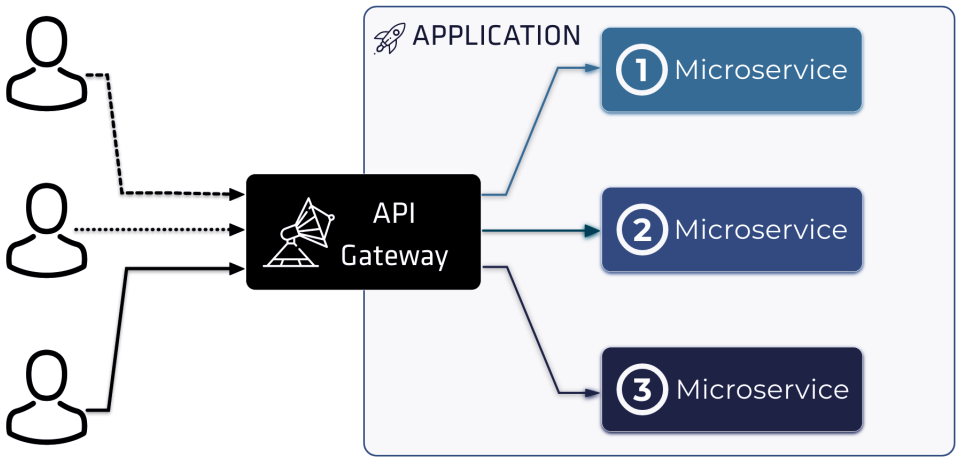

Figure 2.11: API Gateway

Obviously, both solutions have key functionalities. Regarding the first component, it is possible to implement authentication and authorisation systems in addition to the routing functionality provided by the gateway. Moreover, one can aggregate multiple

requests, resulting in a single response, thus improving performance. Considering the second component, instead, implementing a centralized registry can be contemplated. This registry allows the registration of various services over time, controlling traffic flow between services, and monitoring and tracing requests within the network.

2.7.10 Service Mesh

Considering the context of microservices-based architecture in K8s, it is important to consider the scenario with a substantial volume of services to provide. One of the main challenges associated with this type of architecture is precisely the management of communication between services.

Service meshes represent a solution in this sense, offering, as mentioned earlier, a dedicated layer for communication. This type of solution is implemented by adding a sidecar container, placed alongside the main one where the actual service is implemented, acting as a proxy to facilitate communication between services.

If this solution is not considered, developers would need to implement the necessary logic to manage communications. Therefore, the creation of this network involves sidecar containers that communicate with the sidecars of other services.

Adopting this kind of approach makes it possible also to monitor metrics, collect traffic logs, and increase resilience to possible failures. Additionally, security features such as encrypting traffic can also be added. As can be seen from the figure 2.12, in the case

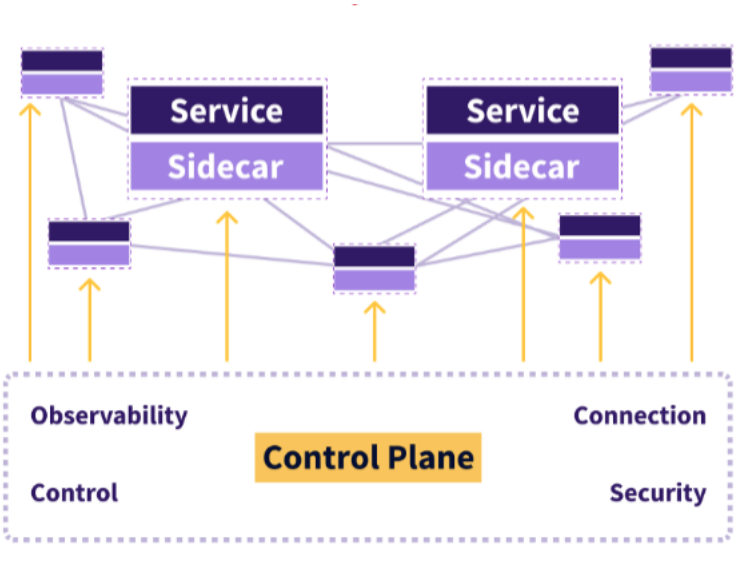

Figure 2.12: Example of how service mesh works

of using service mesh, a sidecar container is created within the pod, which will then be responsible for communicating with other pods, inside the same node and also between different nodes. So it will be this container that will handle communications and networking with the pods.

Among the main open-source solutions for implementing service meshes in Kubernetes there are:

- *Istio*: this platform introduces the functionalities discussed earlier, operating at both network and application layers. It has a centralized control panel for managing policies within the sidecars. To work at the network level, it leverages the Envoy proxy to implement routing between microservices.
- *Linkerd*: This platform is lighter and simpler than Istio. In terms of performance, it has a minimal impact. It provides an automatic load balancer, distributing traffic among various instances of a microservice. In this case as well, monitoring tools are offered. Linkerd primarily operates at the transport and application layers.

However, these solutions focus on the management and communication of the application layer of microservices; for this reason, another kind of solution must now be introduced and will be Network Service Mesh, a project that deals with managing network management and connectivity.

2.8 Operator

Related to the Kubernetes context, another fundamental concept crucial for realising this work is the *operator*. The reason for introducing this notion is associated with the desire to personalize the management of certain automated processes within the cluster by specifying what should be done when certain events occur.

In general, the ability to make certain tasks automatic and repeatable is highly appreciated by users, and for this reason, the use of an operator allows for the extension of functionalities related to Kubernetes automation. Operators leverage the concept of *Custom Resource Definitions (CRD)*, enabling the definition of new types of resources that can then be monitored by the operator itself. The critical point lies in the coding aspect of the actions that need to be executed within the cluster whenever an event related to the CRD occurs. For example, based on the creation of an object of the type specified by the CRD, functions must be registered to execute different deployments to configure, for instance, a namespace or a certain number of pods

So, after creating and defining a new Custom Resource Definition (CRD), an object can be generated containing customizable fields. By default, all unrecognized fields are deleted, hence the need to introduce the field

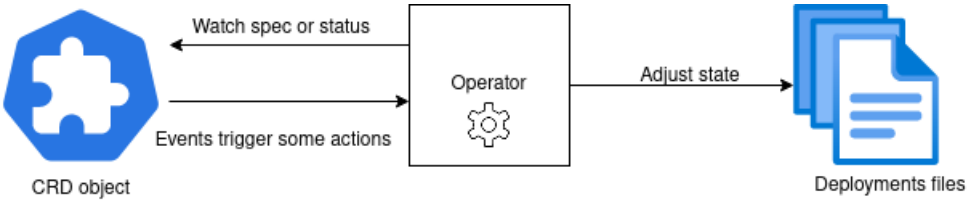

Figure 2.13: Operator schema

```
x-kubernetes-preserve-unknown-fields: true
```

to preserve unknown fields.

In this way, the operator utilises high-level configurations, easily modifiable by the user, implementing a logic close to the one adopted by the control loops. However, it is essential to consider that the specific implementation of the operator can have a more or less significant impact on the quality of the provided applications. Operators can be differentiated based on their capabilities in managing the life cycle of applications, which can be more or less comprehensive. In this sense, a complete operator should be capable of managing all aspects related to pods, such as scheduling, installing, upgrading, scaling, recovering, and so forth [10]. The five macro levels by which operators can be classified, indicating their maturity level, are:

1. Basic Install: The operator must be able to install and configure all the necessary workloads to run the custom resources
2. Seamless Upgrades: Upgrades to various workloads should be possible without data loss.
3. Full Lifecycle: This level refers to the operator's ability to back up and restore the state of workloads.
4. Deep Insights: Monitoring and alert systems are implemented for pod management
5. Auto Pilot: Based on well-defined metrics, the operator can automatically scale resources.

2.9 Network Service Mesh (NSM)

Having introduced all the key elements in this closely related context, we must now define the last important element for understanding the value of this project. Starting with the concept of workloads, this is often executed in a runtime domain, with which a connectivity

domain is associated. In this sense, each workload basically has a connectivity domain to which it must connect. However, very often, workloads that aim to achieve the same application might be running in different runtime domains and for that reason need to connect to each other. Network Service Mesh aims precisely to achieve this goal, that is, where the various workloads, regardless of where they are located, have a way to connect to the required network services, regardless of where they are running [11].

Another key aspect of NSM is that it is runtime domain-independent and is complementary to the previously mentioned common service mesh solutions such as Istio and Linkerd. In fact, despite both being solutions for service mesh, they can indeed be considered complementary, with the possibility of being integrated: considering NSM, this focuses on network management at a lower level than others, while the others emphasize traffic generated at the application level. Obviously, the integration would result in an environment that presents multiple levels of granularity for managing communications between various pods and services. As a direct consequence, an environment with multiple tools and diversified solutions is obtained to meet the possible needs of users.

As already anticipated, these types of solutions provided by the market focus primarily on L7 payloads. Network Service Mesh framework can be leveraged thus a new method for connecting a workload to various network services, "service meshes", which turn out to be unrelated to the cluster in which the workload is running. This type of feature turns out to be an important advantage when different network services from different companies want to connect to a shared network service, thus allowing the ability to cooperate but without exposing the runtime domain. NSM also implements Zero Trust principles where full access is allowed to the minimal amount of resources to ensure that the damage would be contained in the single pod in case of possible damage.

2.9.1 Key concepts

It is now fundamental to introduce what are the most important concepts within NSM:

- Network Services: Here it refers to a set of features including, connectivity, security and observability to which workloads can connect. Through the network service, we go to define the behaviors that the service must have, specifying for example the type of traffic that it wants to handle. By default, the type is IP, but you can also specify *ETHERNET*.
- Network Service Registry (NSR): like other service mesh solutions, NSM offers a registry where both Network services and endpoints are registered.
- Network Service Client (NSC): here, it addresses a client, understood as a workload, that needs to connect to a given network service. The client is authenticated and authorized before making the connection to the service. The connection to the desired service is made through an annotation by which the service is specified, the type of

mechanism desired, such as “memif” or “kernel,” and the name of the corresponding interface that is then created. Be careful not to use excessively long names since a check is made on the length of the name. Another element that can be specified is a label, which is useful for selecting endpoints for the service.

- Network Service Endpoint (NSE): is the pod where the service(s) requested from the client is actually provided. Whenever the endpoint is deployed, it is at the same time logged into a registry where the name of the endpoint is going to be saved, along with the list of what services it provides and what services it needs to connect to instead.

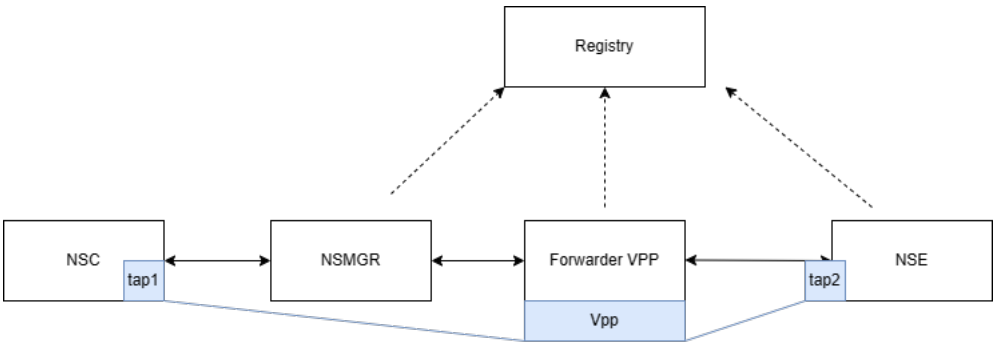

Figure 2.14: Architecture of NSM

As can be seen from the figure 2.14, whenever an NSC requests a particular service it must first go through a network service manager. This component is responsible for authenticating the client and the request in such a way as to implement additional security. In general, there should be a *NSMGR* for each node of the cluster. The *Spiffe(Secure Production Identity Framework for Everyone) and Spire (SPIFFE runtime Environment)* projects are also used to provide an infrastructure for identity management in distributed environments, as in the case of Kubernetes. These two elements work to ensure that communications between different services are safe and secure. Generally, each NSC is provided with a unique identifier, a *SPIFFE ID*, which is necessary to be able to recognize and distinguish the components. In fact, in the cluster configuration phase, specific deployments are recommended in order to have a component that can then handle this type of authentication.

Once the authentication procedure is finished, the *NSMGR* decides to which data plane it forwards the request, which will then be handled by a *vpp forwarder*. VPP stands for Vector Packet Processing, and this type of forwarder is used to achieve high levels of performance by facilitating dynamic packet forwarding. Once the request arrives at the forwarder it must go and register the parameters of the client request within the registry, namely, the requested service and the desired type of mechanism, to which other aspects, such as the name of the interface will be created can then be added, and also selecting what the endpoints associated with the service are Similarly, the endpoint, after registering within the registry with the services it provides, will go on to select the mechanisms it supports. As a final step, both interfaces, the one for NSC and the one for NSE, will

be created and the traffic will be handled directly by the forwarder, creating the tunnel between the two.

2.9.2 Cluster setup

To correctly use NSM, it is necessary to first configure the cluster following certain indications. Generally, apart from prerequisites like Docker, Go, and Kind for creating clusters quickly and easily, a rather specific cluster configuration is required. This configuration entails a node within as a control plane and at least two nodes functioning as workers.

Following this, having already introduced the Spiffe and Spire infrastructure, all requirements need to be installed to ensure these elements work properly within the context of NSM. Only at this point is it possible to configure NSM within the cluster by deploying all the necessary components.

In addition to the components mentioned earlier, an admission webhook is added, which is crucial to allow the registration of an NSC for the requested service. This admission-webhook, once the authentication phase is completed, modifies the client's deployment template by adding an *init container, named "cmd-nsc-init".* This init container is necessary to have configurations within the pod in order to then deploy a sidecar container required to connect the client to the NSM ecosystem.

This type of configuration is specific for testing basic examples; for more complex examples, some modifications might be necessary, involving the addition of further elements. In any case, after checking that all these components have actually been configured, it is possible to test NSM through some examples that are offered by the project developers themselves.

2.9.3 Features

Before presenting some of the features offered by NSM, to facilitate navigation and understanding of the examples, below is the description of the structure they have. Each example presents a file named *kustomization.yaml* in which all the yaml files containing the deployments needed are specified. Among these files are the ones related to the creation of the namespace, the one for creating the network service, and various files for endpoints or additional functionalities. Moreover, considering a standard deployment of the endpoint, each time a *patch-nse.yaml* file is specified in which specific configurations for each example are added. This file is important because, starting from a base template for deploying an NSE, it adds, according to the use case, configurations that enable specific functionality of the pod each time.

Considering more complex or simply more interesting examples in specific use cases,

NSM offers certain features that can be used. For this reason, below are some of the examples that have been considered most useful in the context of this work:

- **Multiple-Services:** This example demonstrates how it is possible to establish a connection between an NSC and multiple services simultaneously. Within the explanation of how NSM works, it has already been mentioned how a client can specify, through a peculiar annotation, which service it wants to connect to, and eventually, the possibility of choosing multiple services at the same time.

  ```
  apiVersion: v1
  kind: Pod
  metadata:
    name: alpine
    labels:
      app: alpine
    annotations:
      networkservicemesh.io: kernel://multiple-services-1/nsm-1,
                                  kernel://multiple-services-2/nsm-2
  ```

  Considering this extract from a YAML file for the client configuration, within *metadata.annotations.networkservicemesh.io*, it can be noticed that two distinct services are requested, namely 'multiple-services-1' and 'multiple-services-2.' In this case, the names 'nsm-1' and 'nsm-2' are also specified for the kernel-type interfaces that will be created to connect to the services

- **Policy-based routing**: In this case, a ConfigMap is created containing various policies for managing routing. Afterwards, within the yaml file for the endpoint configuration, this volume containing the policies is mounted so that the NSE can apply them. By following the example, it is possible to test how these policies are effectively applied through pings from different IP addresses.

- **Scale from zero:** this example aims to demonstrate how endpoints can be created on-demand based on client requests, thus enabling an effective endpoint scaling system. The requested endpoint is deployed on the client's node for better connectivity. What happens is that once the NSC is removed, consequently, the endpoint located on the same node is also removed. To make this type of mechanism work, a third endpoint intervenes, tasked with providing the endpoint when requested and removing it once the client disconnects. In this regard, deploying a component named "nse-supplier-k8s" is carried out. Inside this component, a volume containing a ConfigMap with the endpoint template to be deployed on the client's node is mounted. Just for clarification, this pod, serving as a supplier, is equipped with specific roles that allow it to create or remove pods within the cluster.

  Another fundamental aspect in this example, which is actually quite important within NSM in general, is related to the definition of the Network service.

```
        ---
apiVersion: networkservicemesh.io/v1
kind: NetworkService
metadata:
  name: scale-from-zero
spec:
  payload: ETHERNET
  matches:
    - source_selector:
      fallthrough: true
      routes:
        - destination_selector:
            app: nse-icmp-responder
            nodeName: "{{.nodeName}}"
    - source_selector:
      routes:
        - destination_selector:
            app: icmp-responder-supplier
```

Through this specific configuration, it becomes possible for the NSC to attempt to connect to the requested endpoint on its own node. If the NSE is absent, the request will be handled by the second route, forwarding it to the supplier. This will result in deploying the NSE effectively on the client's node, returning an error to trigger a process of endpoint re-selection, ensuring the connection between the NSC and the requested NSE.

- **NSE composition:** This example illustrates the necessary configuration for creating a more complex network service than standard ones. Specifically, a chain is created consisting of an NSC, a firewall to create and use an Access Control List, three different NSEs acting as pass-throughs, and the corresponding requested NSE.

```
apiVersion: networkservicemesh.io/v1
kind: NetworkService
metadata:
  name: nse-composition
spec:
  payload: ETHERNET
  matches:
    - source_selector:
        app: firewall
      routes:
        - destination_selector:
            app: passthrough-1
    - source_selector:
```

```
      app: passthrough-1
    routes:
      - destination_selector:
          app: passthrough-2
  - source_selector:
      app: passthrough-2
    routes:
      - destination_selector:
          app: passthrough-3
  - source_selector:
      app: passthrough-3
    routes:
      - destination_selector:
          app: gateway
  - routes:
      - destination_selector:
          app: firewall
```

Again, by analyzing the yaml file related to the network service creation, it is possible to notice how the chain is created, respecting a given order. It also specifies a file containing the necessary configuration for the firewall, creating a ConfigMap to mount inside the firewall's container. What happens is that the client request, in order to reach the specified endpoint, at the end of the chain, must pass through an initial firewall that will precisely block or forward requests according to the specified rules, and then this request will be processed by the various pass-throughs. The latter are based on the same image as the firewall, but they do not use any kind of configMap to adopt rules. For this very reason, traffic coming in, via *memif interfaces* and the *vpp forwarder*, is mapped directly to the corresponding outbound interface to the next pod. By running commands provided by the example to test this configuration, it is possible to observe how the firewall actually performs checks, following what is specified in the configuration file.

2.9.4 Labels

In order to fully understand how the NSM configuration processes work and how to use them to configure certain aspects, attention must also be paid to the use of environment variables. In fact, by analyzing the various deployment files, whether for NSC or NSE, it is possible to notice a 'env' section dedicated precisely to this type of variable within the container configuration. This section specifies details like the name, the image to use, volumes, and other aspects.

This step is important because within the NSM ecosystem, containers with specific images are used, capable of interpreting the variables passed to them.

Below is a list of the most commonly used variables in the examples:

- NSM_LOG_LEVEL: indicates the type of logs level that will be printed inside the container. Different types are provided,as *trace* or *debug*, according to the specific use case where activity of debugging is required to monitor what is happening inside the pod.
- NSM_CONNECT_TO: specifies the URL to which the endpoint should connect. This is the variable that allows specifying which manager the corresponding pod should authenticate to and interface with.
- NSM_LABELS: it is used to associate a particular label to identify the pod. This label can then be used within the network service definition, for example, defining how a chain should be composed.
- NSM_SERVICE_NAMES: indicates which services the endpoint should be associated with. This helps in identifying endpoints for various network services.
- NSM_CIRD_PREFIX: it is a list of CIRD prefixes used to assign IP addresses to the created interfaces. Generally, prefixes like '172.16.0.0/31' are used to avoid ambiguity in address assignment; there are only two available addresses: 172.16.0.1 for the client interface and 172.16.0.0 for the corresponding endpoint interface.
- NSM_REGISTER_SERVICE: indicates the endpoint's ability to register the corresponding network service. By default, it is set to false because, in various examples, the service is implemented through a specific file, usually named *"netsvc.yaml"*

Chapter 3

Design and implementation

After introducing all the fundamental concepts to understand the state of the art related to these technologies, it is now possible to introduce the work carried out to achieve the initially set goal.

3.1 Requirements

Before we begin with the actual discussion of the work developments, it is necessary to present the main requirements in some detail. Since we have to follow an approach based on defining software for networks, in which the user has a clear and simplified view of how a network is composed, two main questions have to be answered:

- How can the user be allowed to define desired functionalities?
- How can these features then be translated into deployments?

It is important to note that the user must also be able to change the network configuration without going into too many technical details. In addition, another key aspect to keep in mind is that the user must be able to define functionality using as high a level of abstraction as possible: as an example, to define a firewall that works as a packet filter, he or she would simply have to express it through general concepts, such as “filtering” or “monitoring”. This shows how intent-based networking, the functionality desired by a specific user, can be considered a further development of the SDN approach.

Finally, reasoning more from a technical point of view, to be able to implement a chain of security functions to be placed before the pod in which a service is made available, one

must go to work on layers lower than the application layer to thus allow controls to be applied sequentially.

3.2 Design

In an attempt to give as clear an explanation as possible, the starting idea for this work was the implementation of an SDN within Kubernetes, which would allow controls to be applied or security features to be exploited to protect certain services, in a dynamic and compliant manner with user requests. These controls would then have to be placed in series to form a chain through which traffic would flow.

Considering high-level development, different methods were sought to create the Operator: Kubernetes, within its documentation, provides a list of possible frameworks for building operators, including Java Operator SDK, Operator Framework, and Kopf. The choice fell on the latter due to the ability to program everything needed using Python. In this case, the Operator will take the translated intents and then apply them within a cluster, providing all the fundamental configurations for operation. Next, to realize the starting idea, it was necessary to implement the chain composed of different security functions working, however, at a lower level, especially in the context of Kubernetes. After evaluating the various options of the most commonly used plugins or frameworks to exploit service meshes, we made considerations to discard them as they could not be exploited for the realization of the goals defined at the start. The final choice then fell on the open-source Network Service Mesh project, with which the network layer could be worked for the realization of communications between the various elements of the chain. This is where most of the work was focused since before using the framework, it was necessary to understand the basic mechanisms to move on to the implementation part. Finally, as a last step, the whole user communication part was realized, thus defining the command line interface and the related API, and consequently, a system to realize intent-based networking, so as to meet the initial requirements. In this way, the user can be allowed to define the intents with which to configure the chain.

3.3 Developments

The Kubernetes Operator Pythonic Framework (KOPF)[12] is a library and framework capable of supporting various types of resources within K8s, such as CRDs, Pods, and Namespaces. It also offers different types of handlers, both high and low levels: the former involves detecting changes like resource creation or modification, while the latter monitors events related to K8s APIs. Moreover, it provides all the necessary primitives to configure a level 5 operator, requiring only two files: a Dockerfile and a Python file.

At this point, to comprehend what logic should be implemented to make the Operator

work to manage the resources of the NSM framework, a study was done on the operation of *"nsm-operator"* [13]. In this sense, the main point of this research was to understand how this Operator managed and interfaced with NSM and then replicate a similar strategy through Kopf, adapting it to the use case. In fact, within the *MAKEFILE* of *"nsm-operator"* certain commands are defined, such as *"deploy-spire"* and *"deploy-nsm-operator"*, which are functional to create the resources necessary for the operation of NSM and the deployment of a pod that will then act as the actual Operator. In this way, the network service mesh ecosystem within the cluster was being made available.

After identifying the preferred framework for building the Operator, the focus shifted to how to create the chain effectively. As explained earlier, the core idea was to leverage service mesh for developing a system capable of individually managing each security function within the pods. One of the issues encountered during the development process was understanding how to work via the service mesh at the network level, which was more suitable for adopting security functions. After several searches, the decision was made to use the Network Service Mesh framework, which, unlike other common solutions, focuses on the network level of workloads rather than the application one.

This decision led to significant challenges because the lack of detailed documentation hindered following a programmable workflow. Therefore, an experimental approach was necessary, attempting to understand the framework's internal workings and then modifying to achieve the desired outcome.

Following an initial phase where basic examples like *"kernel to kernel connection"* or "memif to memif Connection" were studied and tested within a cluster, some potential paths to follow began to be defined[14].

Regarding the *memif interfaces* or connections of this kind, reference is made to memory interfaces primarily utilized in virtual network contexts and orchestrations. This type of mechanism allows communication without traversing the entire network stack of the system, operating within shared memory and avoiding potential overhead. As a result, significantly improved metrics are achieved regarding data exchange speed.

On the other hand, kernel interfaces can be found, where access to the operating system's kernel is possible to utilize the services offered by the system. The interface can then be used by other functions or services to monitor traffic or apply specific policies. The analysis of the YAML files from these examples revealed essential details beyond understanding the structure used for maintaining a certain organization. Typically, as previously described, two fundamental concepts emerge within these examples: the network service client (NSC) and the network service endpoint (NSE) are always present.

Each of these files includes a different Golang-written image for the containers. These images also vary between examples, whether related to kernel connections or Memif connections. Therefore, to understand how this framework worked, some of the images used in the examples were analyzed to find a possible solution.

3.4 cmd-nse-icmp-responder image

This is the image used within the NSE (Network Service Endpoint). To properly understand how it worked, further investigation was carried out on the code used for its creation.

After an initial section related to imports and the creation of a struct containing all the necessary configurations, the main phases with which the image is developed are introduced:

- Phase 1: The code retrieves all configurations from the environment. By default, it uses the values present in the struct, which can, however, be replaced by the environment variables passed through the YAML file.
- Phase 2: The Spiffe SVID (Verifiable Identity Document), necessary for authentication, is retrieved. After obtaining the source from which to retrieve the x.509 certificates, an attempt is made to retrieve the certificate associated with the specific SVID.
- Phase 3: The ICMP nse server is created, where the endpoint functionalities are effectively implemented. Navigating the imported libraries made it possible to recreate the characteristics offered by the endpoint. In detail, additional functionalities are added after handling standard configurations, such as the configuration of the name and authorization. Particularly interesting for this project was the definition of supported mechanisms through:

  ```
  mechanisms.NewServer(map[string]networkservice.NetworkServiceServer{
         kernelmech.MECHANISM: kernel.NewServer(),
          noop.MECHANISM:       null.NewServer(),
  })
  ```

  This is the code used to define a mechanism supporting the creation of the kernel interface.
- Phase 4: The gRPC (Remote Procedure Call) server is created, and the NSE (Network Service Endpoint) is mounted.
- Phase 5: The NSE is registered within the registry containing all the other endpoints. Technically, this registration should not take place if the “REGISTER_SERVICE“ environment variable passed through the YAML configuration file is set to false.

  ```
  if config.RegisterService {
          ..
          }
  ```

This *if statement* within the code checks the value of the variable previously assigned "RegisterService", and as already explained, if this variable is set to false, it indicates not to register the endpoint for the corresponding service.

Suppose that a developer aims to create an endpoint entirely from scratch; it is essential to consider that some of the described phases are still necessary and, in a sense, can be considered as a starting point. Indeed, assuming the context involves working with NSM as provided in the documentation, it is possible to focus mainly on modifying phase number 3, where the endpoint's functionalities are described.

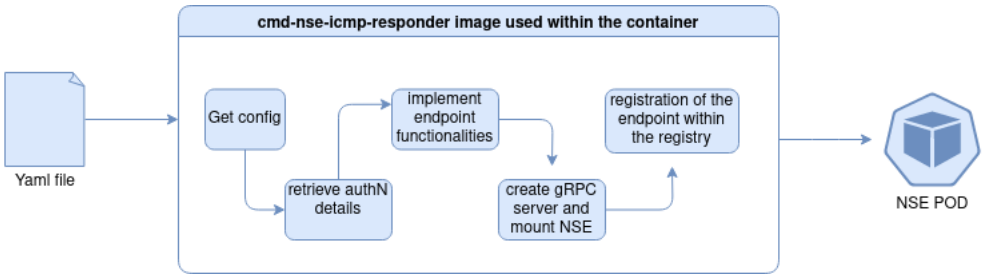

Figure 3.1: Workflow of the YAML file and the corresponding image

Starting from the YAML file used to configure the pod, it is possible to specify specific environment variables so that the image reads their values and uses them to make certain configurations. By executing the five phases just described, what you obtain is the corresponding pod with the functionalities specified in phase 3.

3.5 Attempted approaches

Having reached these conclusions, which indicated that this specific image allowed the creation of a kernel interface, an attempt was made to modify the code to obtain another interface of the same type. To understand how to achieve this goal, by studying one of the offered features, namely ***"nse-composition"***, another image was analyzed to understand its functioning: *cmd-nse-firewall-vpp*.

In this case, as well, the structure and composition of the various phases remain quite similar to those of "cmd-nse-icmp-responder", except for the addition of some details related to VPP connections and for the firewall configuration. The aim was to achieve a mechanism similar to that implemented within the firewall image, where a mapping occurred between an interface that gathered incoming traffic and a second interface to forward traffic to a subsequent endpoint.

However, this time, the image doesn't provide an implementation for the kernel mechanism, as in the previous case, but exclusively supports the *memif mechanism* as can be

seen by this piece of code extracted from the image's implementation:

```
mechanisms.NewServer(map[string]networkservice.NetworkServiceServer{
    memif.MECHANISM: chain.NewNetworkServiceServer(
                                    memif.NewServer(ctx, vppConn)),
})
```

As mentioned earlier, all images related to *VPP connections* need to declare a variable and use it precisely to manage this type of connection.

A clarification concerns the first two approaches as they involved modifying the code within the used images. For this reason, every time, to test their functionality, new images containing the modifications were built and subsequently tested within a cluster where the NSM framework was installed. At this point, different approaches were attempted are explained.

3.5.1 First Approach

Following an approach quite similar to what happens in the *cmd-nse-icmp-responder image*, efforts were made to add functions for creating and supporting the kernel mechanism. Needing two interfaces, to implement the second one, following the logic within the firewall image, a third image and its implementation were considered: "cmd-nsc". This image is used by the NSC to interface with the NSM framework and ensure the creation of a kernel interface.

```
responderEndpoint := endpoint.NewServer(ctx,
    spiffejwt.TokenGeneratorFunc(source, config.MaxTokenLifetime),
    endpoint.WithName(config.Name),
    endpoint.WithAuthorizeServer(authorize.NewServer()),
    endpoint.WithAdditionalFunctionality(
         onidle.NewServer(ctx, cancel, config.IdleTimeout),
         groupipam.NewServer(config.CidrPrefix),
         policyroute.NewServer(newPolicyRoutesGetter(ctx,
                config.PBRConfigPath).Get),
         mechanisms.NewServer(map[string]networkservice.NetworkServiceServer{
            kernelmech.MECHANISM: kernel.NewServer(),
            noop.MECHANISM:       null.NewServer(),
                 }),
         tokenServer,
         dnscontext.NewServer(config.DNSConfigs...),
         sendfd.NewServer(),
         ),
)
```

This extract was taken from the image implementing the endpoint: among the various functionalities added within *endpoint.WithAdditionalFunctionality* it is possible to find the one related to the configuration of IP addresses, through *grouipam.NewServer(config.CidrPrefix)*, which precisely leverages the corresponding variable passed to it to carry out this configuration.

Subsequently, the method related to policies is made available, specifically to retrieve the file where certain rules for traffic management are configured and then applied. In the *policy-based-routing* feature, the same image is used for the realization of the endpoint, which, however, accesses the present policy file for utilization and then is managed by this specific method.

Regarding instead the *cmd-nsc* image where this piece of code implements the corresponding functionalities for the client

```
nsmClient := client.NewClient(ctx,
client.WithClientURL(&c.ConnectTo),
client.WithName(c.Name),
client.WithAuthorizeClient(authorize.NewClient()),
client.WithHealClient(heal.NewClient(ctx, healOptions...)),
client.WithAdditionalFunctionality(
    clientinfo.NewClient(),
    upstreamrefresh.NewClient(ctx),
    sriovtoken.NewClient(),
    mechanisms.NewClient(map[string]networkservice.NetworkServiceClient{
        vfiomech.MECHANISM:
            chain.NewNetworkServiceClient(vfio.NewClient()),
        kernelmech.MECHANISM:
            chain.NewNetworkServiceClient(kernel.NewClient()),
        }),
    sendfd.NewClient(),
    dnsClient,
        ...
          ),
         ..
)
```

The main difference noticeable between this code and the previous one is related to the concept of client and server.

Within NSM, it is possible to reason in terms of pairs, meaning there must be one side of communication representing the client that initiates a request, and the other acting as an endpoint/server that receives such a request. This justifies the consistent usage of the terms *NewClient()* or *NewServer()*, depending on the image.

However, this approach proved to be problematic since, despite following a similar logic to that within the firewall, NSM does not provide an actual kernel forwarder. This detail is crucial because, instead of having a VPP forwarder, a kernel forwarder allows managing the two interfaces and mapping them to create a tunnel between them to facilitate traffic flow.

3.5.2 Second Approach

The second attempted approach was based on considerations and suggestions provided in an **issue**, where it was recommended to use an approach based on memif interfaces, capable of being managed by the VPP forwarder, and subsequently map these interfaces to kernel interfaces using an L2 bridge domain. What was suggested was to start from the implementation of the *cmd-nse-icmp-responder-vpp* image, specifically linked to memif interfaces. From there, it would have been necessary to create a *tap interface* to map the memif interface into a kernel using an L2 connection. While always keeping in mind the implementation of the firewall, this mapping process would have had to be done for both interfaces.

Due to the lack of documentation to follow to understand how to implement it or simply what logic should be followed, it was not possible to achieve a conclusive result.

3.5.3 Third Approach

The last and definitive attempt relied on a different pod configuration. With NSM developers' assistance, a different configuration of the YAML file was used to achieve a pod containing two kernel interfaces. It specified two containers based on the images "cmd-nse-icmp-responder" and "cmd-nsc" respectively. Through the definition of a precise network service and the utilization of an environment variable for the container based on cmd-nsc, the initial goal was successfully achieved. As can be seen from the Image 3.2, setting aside for the moment the client and endpoint, which maintain a standard configuration, a YAML file is used to specify three different containers. Two of these containers are implemented in the same way as the NSC and the NSE.

The third container will instead serve to position the security function within the pod. By doing this, the pod in the middle of the chain has 2 kernel interfaces that can then be used for subsequent purposes.

Through this type of configuration, it is possible to define a chain of several intermediate pods, each of which has a different security function. In this regard, attention must be paid to two particular details so that the chain is properly configured:

- the value of the *CIRD_PREFIX* variable with which you then assign IP addresses

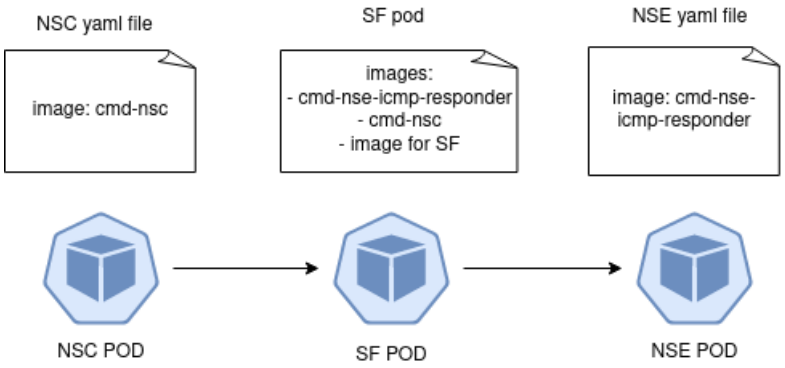

Figure 3.2: Deployment schema

to the interfaces. This is why you must take into account that each NSE will have a different value for this variable and they must be configured so that they do not overlap.

- the value of the *NSM_LABELS* variable needed to identify the different endpoints and thus implement a network service that allows traffic to be handled in a certain way, considering a specific order.

3.6 Further Developments

At this point, having obtained the necessary configuration and recognizing the absence of a forwarder capable of managing kernel mechanisms, it was time to understand how to create a sort of routing inside the pod between these two containers, allowing incoming traffic on one interface to be passed to the second interface. Hence, the approach was to comprehend how to achieve traffic forwarding through *iptables*. To ensure the forwarding worked as expected and to validate this solution and its configuration, a third container was added, allowing the use of *"tcpdump"* within the pod. This validated that the incoming traffic on the final endpoint indeed originated from the client. This type of approach was used throughout the development, that is, constant testing and validation of results so that we could ensure that the results obtained did not exhibit unknown undesirable behaviors. In fact, the network service mesh ecosystem has several behaviors that could create malfunctions within the examples, giving rise to the need to test the various configurations from time to time. For example, in the case of using the "nse-composition" feature, a client-related problem arises: If a cluster with more than one worker node inside is used, the NSC through the variable *NSM_REGISTRY_TO*, does not know which manager to connect to, since, having two, likewise there are two different nsmgr. Actually, this type of configuration leaves the NSC always in the "init" state since it fails to forward the request to the manager.

Having resolved this step, the next challenge was to integrate security functions within these pods. Introducing a third container within the pod did not seem problematic in its interaction with the NSM framework. However, many containerized security functions entail the use of multiple containers, each dedicated to handling a specific aspect of the function. For this reason, images were sought that would allow the implementation of the security function through a single container, minimizing as much as possible the potential errors due to wrong configurations or lack of necessary resources. It is also essential to consider another kind of limitation due to the use of these images within Kubernetes, as, unlike other use cases, it might require additional configurations to ensure that everything functions properly.

After preparing the design and testing it to ensure that the various configurations work as intended, the focus shifted to the actual implementation of the Operator and command-line interface.

As mentioned earlier, aiming to provide users with a rather comprehensive view of the network within this context in order to manage it more effectively, an attempt was made to offer a way to interact with Kubernetes deployments that eliminate as many details as possible.

In this regard, efforts were made to implement what is referred to as 'Intent-Based Networking' [15]. Through IBN, it is possible to follow a declarative approach to the requirements, describing exclusively the desired state. In particular, the greatest benefit of this approach is linked to the ease with which a developer can request a specific network setup, leaving all the configuration behind to the service that implements IBN. The idea is to leverage this kind of definition for networking in order to be able to create a given chain, following what are the user's requirements, allowing one to approach the implementation of an SDN in the context of Kubernetes. What the user sees is precisely a high-level representation of what is the chain, and therefore the network, within the cluster.

By considering certain intents, which in the specific context of this work represent the desired functionalities within the user's chain, corresponding configurations are created. There must be a form of translation that allows the interpretation of the intent, written in a specific language with its own semantics, into specifics that can be associated with practical configurations.

For this reason, a Command Line Interface (CLI) was developed to compile a series of intents that are then sent to APIs on the Operator's side. At this stage, these intents are translated into more detailed specifications that can be useful for configuring various deployments in the Kubernetes context.

In detail, a system had to be created that would read the received intents, make a kind of dictionary needed to do the translation, and finally configure everything to create objects that could then be exploited by the Operator.

The implementation of the Operator is based on the usage of Custom Resource Definitions (CRDs) and the objects defined within them. This way, whenever a resource of the specified type in the new CRD is created, the Operator, being in a listening mode to manage changes related to that resource, gets activated. Methods for handling CRD-related events were then developed and configured to provide the user with a ready cluster with everything needed to run the service.

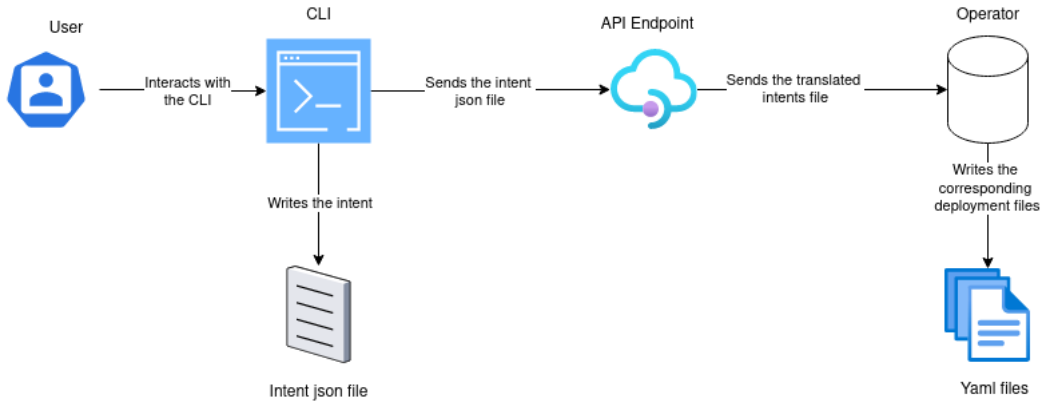

Figure 3.3: Interactions among components

Therefore, considering an overall picture that encompasses the entire work, the creation of a CLI to manage user-side requirements, written within a file that is then sent to specific APIs, was eventually achieved. From there, after the translation of the requirements into specifications suitable for the K8s context is done, the Operator is queried, with which these resources must be managed to have an effective deployment.

Chapter 4

Results and validation

In terms of the results achieved, the premises with which this work started have been largely fulfilled, as it has come up with a Software Defined Networking within Kubernetes that can be actually exploited. In addition to offering more detailed documentation than Network service mesh, this work, has made possible the realization of a new feature that can then be considered in the framework. Indeed, the composition of endpoints, where each provides two kernel interfaces, each of which can be exploited by other containers within the pod, represents further achievement. Wanting now to describe the results achieved in the order in which the various components interact, we start with the CLI implemented.

4.1 CLI and intents mapping

As explained earlier, this CLI has the key role of allowing the user to interact with the operator, allowing the user to define what requirements are desired and received, and as a result, a cluster within which they are implemented. To make this possible, the command line interface offers several commands, such as:

- add: this is the command by which you add an intent, or requirement, that is, within a specially created JSON file in the directory where you are working. You can also specify parameters such as “src” or “p” to have a set of IPs that you can use or ports where you listen for traffic.

```
sfc add <name> [--src <src>] [--p <p>]
```

- modify: through this command, you can modify a given requirement, specifying a new value that will replace the present one.

```
sfc modify <name> --field <field> --newValue <new_value>
```

- get: this command returns the contents of the JSON file and therefore the list of requirements.

```
sfc get [--name <name>]
```

- clear: this is the command that takes care of clearing the contents of the file.

```
sfc clear
```

- rm: removes a given requirement from the file, through the name passed to it.

```
sfc rm <name>
```

- setchain: this command contacts an API through which the intents are translated and the CRD and related objects are created.

```
sfc setchain <url>
```

- deploychain: this is the command used to make the actual deployment of the chain happen, also running the operator.

```
sfc deploychain <url>
```

So what happens is the compilation of the JSON file with all the intents inside, which will appear to have this structure.

```
{
‘‘requirement1‘‘: {
    ‘‘name‘‘: ‘‘Firewall‘‘
    },
‘‘requirement2‘‘: {
    ‘‘name‘‘: ‘‘packetfilter‘‘
    }
}
```

Then this file will be sent to the operator via APIs, with which a new CRD type and the object related to it will be created.

As already anticipated, a translation of these intents must take place at this point, and through the use of Knowledge Base, a translation of what the user has requested into something that can then be used for the definition of the YAML file can be made. This represents the way in which an attempt was made to implement a specific IBN for this work. The key point is to start from a more abstract definition and then arrive at something specific to implement networking. Moreover, since there is not yet a defined standard that can be adopted, one must, in a sense, implement a customized version based on use cases[16]. Therefore, the logic used for the development of this part, first considers the name of the requirement in the JSON file, which will be searched within a dictionary, where a certain name is categorized under a label.

```
{
    ‘‘firewall‘‘: [‘‘firewalling‘‘, ‘‘accesscontrollist‘‘,
                        ‘‘protection‘‘, ‘‘firewall‘‘],
    ‘‘packetFilter‘‘: [‘‘filtering‘‘, ‘‘packetfiltering‘‘,
                        ‘‘filter‘‘, ‘‘packetfilter‘‘],
    ‘‘monitoring‘‘: [‘‘monitoring‘‘, ‘‘suricata‘‘, ‘‘ids‘‘,
                        ‘‘intrusiondetectionsistem‘‘]
}
```

By doing so, the requirement should be expressed according to one of the values within the dictionary and then, they will be translated using the corresponding label. Just to make this clearer, if the user defines an intent using the name *“ids”*, this intent, on the operator side, will be classified with the label *“monitoring”*.

Through this label then, all necessary information is retrieved for the YAML file, such as the image or additional parameters needed for proper operation. Another file is then used where the mapping between the label and the corresponding configurations can be specified.

```
{
‘‘firewall‘‘: {
    ‘‘image‘‘: ‘‘corfr/tcpdump‘‘,
    ‘‘src‘‘: ‘‘172.16.1.0/31‘‘,
    ‘‘port‘‘: 8080
    },
‘‘packetFilter‘‘:{
    ‘‘image‘‘: ‘‘ubuntu/squid:edge‘‘,
    ‘‘port‘‘: 3128
    },
‘‘monitoring‘‘:{
```

```
        ‘‘image‘‘: ‘‘jasonish/suricata‘‘
        }
    }
```

As a final step with regard to this translation, a folder is created with the date in which a random string is made, so as to keep track of what has been done. Within this folder, the files containing the custom resource definition and the object bound to it, are then created. The definition of the CRD is done in this way:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
    name: securitychains
spec:
    group: kopf.dev
    names:
        kind: SecurityChain
        plural: securitychains
        shortNames:
        - scs
        - sc
        singular: securitychain
    scope: Namespaced
    versions:
    - name: v1
      schema:
        openAPIV3Schema:
            properties:
                spec:
                  type: object
                  x-kubernetes-preserve-unknown-fields: true
                status:
                  type: object
                  x-kubernetes-preserve-unknown-fields: true
            type: object
        served: true
        storage: true
```

The corresponding object will have this type of structure:

```
apiVersion: kopf.dev/v1
kind: SecurityChain
metadata:
```

```
    name: security-chain
spec:
  securityFunctions:
  - image: jasonish/suricata
    name: monitoring
    port: ''
    src: ''
```

So it can be seen that starting with the definition of the requirement made by the user, this is translated into specifications that can then be exploited by the operator. In fact, it will be the latter that will use these last two files to ensure that the desired configurations take place.

4.2 Operator

Regarding the Operator, it was mentioned how the choice fell on Kopf and how in general, operators work through the use of CRDs. At this point, several handlers were defined to handle various events such as object creation, modification, and deletion. Within these must then be defined the functions needed to configure the Kubernetes environment, creating and configuring all the necessary files. Therefore, starting from the YAML file describing the previously defined CRD object, the various deployment files are created, respecting the structure and organization used within the NSM examples, where the *kustomization file* is adopted. Next, a specific configuration is used for the various security functions to be included within the pod. In detail, the functions offered by these jobs are two:

- *Suricata:* via the *"jasonish/suricata"* image, it is possible to obtain a container that implements several functionalities as an intrusion detection system or general traffic monitoring on a given interface. By default, Suricata uses a set of rules, which can also be updated time by time, that covers most cases of suspicious traffic activity or possible well-known attacks. Wanting to leave it up to the user to configure the rules set specific to their use case, the ability to write or edit a file that is then used to apply the rules has been added. For example, here there is the content of the file *"suricata.rules"*, where some customs rules have been added:

  ```
  alert icmp $HOME_NET any -> $HOME_NET any
      (msg:``ICMP dangerous traffic has been detected``; sid:1;)
  alert icmp any any -> any any
      (msg:``Violation of a suricata rule has been detected  ``; sid:2;)
  ```

  Care must be taken when writing the rules, as they must have a very specific format, which can still be found in the Suricata documentation. In any case, it is needed

to specify the type of event, which in this case is an alert, and the type of traffic affected by the rule, both in terms of source and destination, and the ports involved.

This file is then mounted within the container is a specific path via a *ConfigMap*, which the kustomization file creates. After the desired rules have been configured by the user, and the interface to monitor has been detected, it is possible to run Suricata within the container, using a specific command. The logs that can be generated from the possible event are saved within the container and can be accessed by the user.

- *Squid:* another functionality offered by this work is related to traffic filtering, where a container setup can be done in order to leverage a squid-based image. Doing this, through the definition of a *"squid.conf"* configuration file, necessary to run the container, the filtering criteria can be applied to a given interface.

```
acl special_ips src  172.16.0.0/12
http_access deny special_ips
```

Considering this small portion of the configuration file, a range of IP addresses is defined, named *"special_ips"*, and in the next line, they are prevented from accessing it. In this way, it was tested how by making an HTTP request for any website, from any of the IPs that fall in that range, this request is actually blocked and a notification message is returned by Squid. Again, the process is similar to what happened in Suricata's case, where the configuration file is created through a *ConfigMap* and mounted within the container.

What is different, however, from the previous use case, is that now other components such as *StorageClass* and *PersistentVolumeClaim* are required, to allow Squid to work properly and make the caching mechanism, to save persistently the data.

There would then be two other functionalities offered but which are actually quite similar: *Zeek* and *Tcpdump*. Considering Zeek, it was the security function identified before opting for Suricata: the reason for this decision is related to the fact that it was not possible to find a configuration in such a way that the container could be started and then run as an intrusion detection system. In contrast, Tcpdump, is a feature that can be added if necessary: the corresponding image *"corfr/tcpdump"*, in a container was used to allow verification that the traffic was correctly routed by the forwarding rules necessary to then implement the chain. However, it was considered a simplified version of monitoring, and for that reason, it is simply offered as support for development.

The last step in making this chain work was to implement routing within the pods. In fact, the various tests performed showed how it was not possible to ping directly from the NSC to the NSE of the corresponding service, but passing through the pods in the middle of the chain, as was precisely the case with the endpoint composition example. The main problem was due to the lack of a forwarder acting at the kernel level, capable of mapping incoming traffic on one interface to outgoing traffic on another, which happens instead in

the presence of the VPP forwarder. Therefore, a manual routing had to be configured, capable of mapping and forwarding incoming traffic to the interfaces of the other pods, so as to reach the final endpoint.

```
/bin/sh
-c
|
 apk add --no-cache iptables
 INTERFACE=$(ip -o -4 addr list | grep ‘‘172.16.0.0/32‘‘ | awk ’{print $2}’)
 iptables -t nat -A PREROUTING -i $INTERFACE -j DNAT --to 172.16.1.0
 iptables -A FORWARD -i $INTERFACE -o kernelint2 -j ACCEPT
 iptables -A FORWARD -i kernelint2 -o $INTERFACE -j ACCEPT
```

By executing these commands within the various pods where containers are configured for security functions, it was possible to resolve this shortcoming. In particular, the name of the interface with that specific IP address is obtained, then it is the one created by the *cmd-nse-icmp-responder image*, and each time, within the name, it has a different final string. So, by getting that variable it goes to specify that incoming traffic on that interface should be forwarded to the other IP address, corresponding to the kernel interface of the next pod. In this way, it is like changing the destination IP address of the packet sent by the NSC. In addition, two forwarding rules are set for both kernel interfaces within the pod. The name of the second interface is well known since it is specified within the YAML file with which the deployment is done.

4.3 Custom NSE composition

The most important goal achieved was to figure out how we could modify the “nse-composition” example offered by the NSM developers in such a way that it met the needs of this specific use case. Specifically, after describing the various approaches that were attempted, the final solution was based on a particular configuration of YAML files. Making a brief summary, what one must keep in mind is that within the framework in question one must think in terms of client-server pairs. Furthermore, after understanding the purpose of the variables defined within the deployment file and how they are used within the images it was possible to define this new endpoint composition where kernel interfaces are made available instead of memif. To understand how the whole mechanism works, it is possible to start with the definition of network service:

```
apiVersion: networkservicemesh.io/v1
kind: NetworkService
metadata:
  name: security-chain
```

```
spec:
  payload: ETHERNET
  matches:
    - routes:
        - destination_selector:
            app: monitoring
    - source_selector:
        app: monitoring
      routes:
        - destination_selector:
            app: packetFilter
    - source_selector:
        app: packetFilter
      routes:
        - destination_selector:
            app: gateway
```

It can be seen how the service is identified by the field name "metadata.name" and then how the chain and its order are characterized through the set of matches. In particular, the first route identifies what is the first endpoint to be contacted in the chain. By default, all requests will fall into this match. From here, "source" and "destination" selectors are defined from time to time with relative values. These represent the labels associated with the endpoints, necessary to allow a new interface to be created between pods.

Turning instead to the definition of the NSC, it is necessary to use the annotation provided by the framework to specifically request a service.

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    networkservicemesh.io: kernel://security-chain/nsm-1
  labels:
    app: alpine
  name: nsc
```

It is possible to notice how the request within the client YAML file, goes to request the service defined earlier, also specifying the desired name for the interface that will be created within the pod.

Moving then to the various pod deployment files containing the security functions, these have a similar structure. For simplicity, only the parts related to the container definition will be shown, so as to focus on what are the main details.

```
containers:
 - name: nse
   image: ghcr.io/networkservicemesh/ci/cmd-nse-icmp-responder
   imagePullPolicy: IfNotPresent
   securityContext:
     privileged: true
   env:
   - name: SPIFFE_ENDPOINT_SOCKET
     value: unix:///run/spire/sockets/agent.sock
   - name: NSM_LABELS
     value: app:monitoring
   - name: NSM_SERVICE_NAMES
     value: security-chain
   - name: NSM_CIDR_PREFIX
     value: 172.16.0.0/31
   - name: NSM_NAME
     valueFrom:
       fieldRef:
         fieldPath: metadata.name
   - name: NSM_CONNECT_TO
     value: unix:///var/lib/networkservicemesh/nsm.io.sock
   - name: NSM_REGISTER_SERVICE
     value: 'false'
```

Regarding the container that implements the endpoint, it can be seen how the variable *NSM_LABELS* has *"monitoring"* as its value, which is exactly the one present within the network service definition. Next, this endpoint must be configured for the specific *"security-chain"* thanks to the *NSM_SERIVCE_NAMES* variable and how *NSM_CIDR_PREFIX* has value *172.16.0.0/31.* According to the latter, the interfaces that will be created will have IP address 172.16.0.0 in the pod in question, where the *nse* container is defined, and 172.16.0.1 in the NSC pod, respectively.

Considering instead the container that exploits the client image, namely *cmd-nsc*, what should be noted is the type of annotation used, where, following the specification of the desired interface name, *"?app=monitoring"* is added.

```
- name: cmd-nsc
   image: ghcr.io/networkservicemesh/ci/cmd-nsc:d105640
   imagePullPolicy: IfNotPresent
   env:
   - name: SPIFFE_ENDPOINT_SOCKET
     value: unix:///run/spire/sockets/agent.sock
   - name: NSM_REGISTER_SERVICE
     value: 'false'
```

```
    - name: NSM_NETWORK_SERVICES
      value: kernel://security-chain/kernelint?app=monitoring
    - name: NSM_LOG_LEVEL
      value: TRACE
    - name: NSM_CONNECT_TO
      value: unix:///var/lib/networkservicemesh/nsm.io.sock
```

This aspect turns out to be critical because it is what allows the matching to be performed within the network service, thus looking for the one that will have *"packetFilter"* as the label value as the destination endpoint. Doing so again adopts the client-endpoint mechanism, by which the second kernel interface is then made available within the pod.

Finally, regarding the third container, which is then used for the security function, in addition to the basic configurations, a path is specified within which the volume with the name *"suricata-rules"* created from the *ConfigMap* with the same name must be mounted.

```
  - name: monitoring
    image: jasonish/suricata
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - name: suricata-rules
      mountPath: /additional/rules/local
      readOnly: 'false'
volumes:
  - name: spire-agent-socket
    hostPath:
      path: /run/spire/sockets
      type: Directory
  - name: nsm-socket
    hostPath:
      path: /var/lib/networkservicemesh
      type: DirectoryOrCreate
  - name: suricata-rules
    configMap:
      name: suricata-rules
```

It is not strictly necessary that the names must be the same, but in this case they are for simplicity.

Considering the figure 4.1, it can be possible to have a sharper understanding of what is the final scheme for representing the chain, where precisely a client goes to request a specific network service with which the deployment of the chain in question is done. The image also shows the various kernel interfaces, with their IP addresses, in order to illustrate the mechanism by which they are created, within the NSM framework. Basically, what

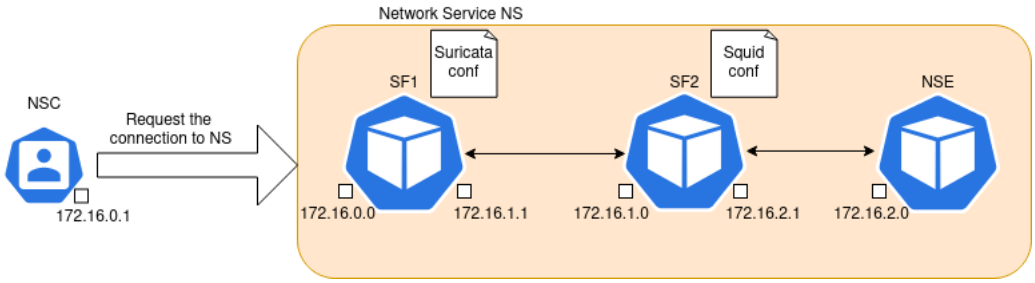

Figure 4.1: Example of a chain

happens is that the client, through its kernel interface with IP 172.16.0.1, will contact the first kernel interface, which is 172.16.0.0. From there on, the request will be passed to the endpoints until it reaches the actual NSE where the requested service is available.

Chapter 5

Future developments and conclusions

Having discussed in detail what has been accomplished within the development of this work, it is now possible to focus on what may be possible insights for future work that can start here. There have already been some hints about what aspects could be improved or deepened, but here, they will be grouped together for convenience and clarity in this section. Three macro areas can be identified, corresponding precisely to the three main aspects developed: CLI, Operator and NSM.

5.1 Future developments

Wanting to leave clear reasons for the possible improvements or additions that could be made, below are explanations of what aspects to work on and what functionality they could bring, thus justifying the value of possible research.

5.1.1 CLI and IBN possible developments

As far as the CLI is concerned, one of the possible improvements that can be made concerns the definition of the language, where through the creation of a richer and more detailed vocabulary, an even higher level of abstraction can be achieved than at present. This would give the user the ability to define requirements in a more general way. In addition to the changes needed to provide the user with a CLI capable of expressing functionality, the language translation system would involve the most work. As explained within this work, starting with the file containing user-defined intents, two other files are used with which

categorization and then mapping with actual values can be done. For this reason, there should be a system to translate the intents similarly, eventually arriving at configurations that NSM can then exploit. An interesting aspect would be implementing some sort of manager to expose APIs to receive intents, process them, and configure files useful to the Operator. The reason for this development lies in the fact that IBN can be seen as a step forward from SDN, allowing the widest and simplest possible network configuration.

5.1.2 Operator improvements

Reasoning instead about the Operator, as explained earlier, there are several levels by which it can be classified, depending on the level of automation and management it offers. Considering what has been obtained from this work, what could really be done is the

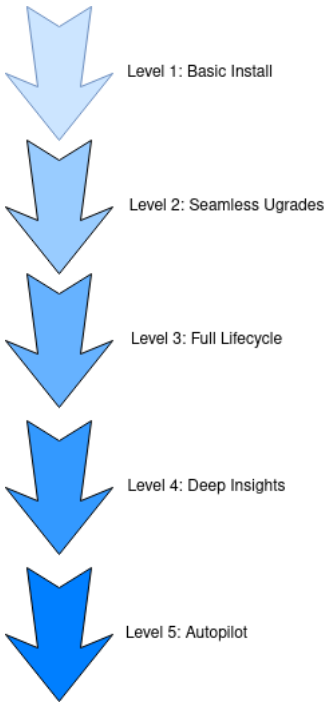

Figure 5.1: Possible levels of the Operator

extension of the functionality offered by the Operator so that it can manage the resources on different levels, in addition to additional instances for the management of events that might happen. An example, clearer and more specific in this context, could be related to the creation of a kind of dynamic chain: assuming that the pod within which there is *suricata logs* suspicious traffic that is followed by a generation of specific logs, the Operator could be programmed to inspect these logs and act accordingly, configuring more restrictive policies within the pod with which Squid is configured. In this way, a chain could be created that reacts in real-time to critical issues that might occur within the

system. Another aspect on which improvements could be made concerns implementing more security functions after having identified all the necessary files for the configurations. As such, it needed first to find a Docker image that allows the function to run within the pod, verify that it does not conflict with the NSM framework, and then implement custom management within the Operator to configure it. Obviously, there has to be management via the Knowledge Base in the beginning, with which the intent can be translated. Other possible developments inherent to the Operator actually concern configurations versus what can be done through NSM and, in this sense, depend more on figuring out how to integrate other examples or take advantage of particular features offered by NSM.

5.1.3 Exploration of NSM

The NSM framework has provided numerous features that can be used that could be quite useful for specific use cases. Browsing through the examples in the repository, in fact, there are possible ideas for implementing new features. The primary difficulty, however, lies in the lack of documentation that is useful to understand how the various mechanisms work so that they can be modified or made more specific as needed. However, those that might be the most interesting and related to the work being done are k8s-monolith, policy-based-routing, scale-from-zero and observability.

K8s-monolith

This particular feature is inherent in the ability to decompose existing monolithic applications, through the use of pods to enclose services. In this sense, what is done is precisely the creation of a docker container external to the cluster, which is configured to interface with the service within the cluster. Within it, instructions are defined to define both a client and an external endpoint. An attempt was made to implement this feature within this project as well. Still, unfortunately, the external docker could not connect properly with the service, so the interfaces were created to be able to communicate with the service itself.

To accomplish this type of communication between an out-of-cluster element and the services defined inside it instead, a LoadBalancer must be defined. As mentioned earlier, K8s does not provide a default LoadBalancer that can be used, making it necessary to implement one. The LoadBalancer used within this example is MetalLB which must be configured via the guide provided by Kind, as the one in the repository is an older version and no longer supported. As a next step, the external container representing the service client is started: again, the annotation specified earlier is used to request the desired service.

Being able to exploit this type of feature would allow approaching a use case very close to reality, where the client requesting the service is not arranged within the cluster where

Figure 5.2: Example of external NSC

the latter is defined. It would then allow for a more widespread use of this work, projecting it toward a version capable of wider distribution.

Policy-based-routing

As had already been anticipated, one of the examples provided by NSM that could be quite useful in this context is related precisely to a routing based on policies defined in a file mounted and then through a ConfigMap. In fact, through this type of file, specific policies are defined and then applied with which traffic can be blocked:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: policies-config-file
data:
  config.yaml: |
    - from: 172.16.2.201/24
      proto: 6
      dstport: 6666
      routes:
        - prefix: 172.16.3.0/24
          nexthop: 172.16.2.200
    - from: 172.16.2.201/24
      proto: 6
      srcport: 5555
    - proto: 17
      dstport: 6666
```

```
- proto: 17
  dstport: 6667-6670
- from: 2004::3/120
  proto: 17
  dstport: 5555
  routes:
    - prefix: 2004::5/120
      nexthop: 2004::6
```

In fact, by performing the various steps defined within the guide, one can see how, through pings, the rules are actually set. This approach is specific to endpoints based on the "cmd-nse-icmp-responder" image, which then creates a kernel interface within the pod. This approach could then be adapted within this work as an alternative solution to Squid with which to filter traffic or set additional policies. This configuration is actually enabled and used within the endpoint through this function:

```
endpoint.WithAdditionalFunctionality(
...
     policyroute.NewServer(newPolicyRoutesGetter
                         (ctx, config.PBRConfigPath).Get),
            ...
        )
```

with which the use of policy-based routing is specified. In fact, always within the image code, a function *"newPolicyRoutesGetter()"* is defined that monitors the file by which policies are defined and applies changes whenever changes are made.

Scale-from-zero

The description of the operation of this example was covered previously. We want to explain why it represents a possible future development and an aspect that could be explored further. What happens through "scale-from-zero" is that the endpoint requested by the NSC is sent running directly on the node where the client is located through the use of "supplier" that allows this operation. Then, once the NSC disconnects from the network, the previously requested endpoint is also deleted. In this way, making the system more scalable is possible, freeing up resources that can then be reallocated.

Observability

Through this example, the study, which has not been as in-depth as in the other cases, it is possible to implement a system to collect logs with which to do analyses related to the

performance or various behaviours of the system. It leverages a setup that includes an Open Telemetry Collector, Jaeger, and Prometheus to collect all the data it needs. Being able to implement this example within the project or simply understand the logic of it and then replicate something similar through the Operator would allow additional functionality to be defined that could come in handy for mechanisms with which to scale pods within Kubernetes.

Endpoint customization

Not strictly related to the examples within the repository, one could go into more detail on how to customize the creation of endpoints. In this sense, since there is no real guide to creating an endpoint from scratch that one can then leverage, one would need to better understand the steps necessary for an image to be then the basis for making an NSE. If this step were reached, one could then analyze in more detail the "cmd-nse-firewall-vpp" image, with which a firewall is precisely created but works with memif interfaces. If one were to make a custom implementation to achieve the same result but with kernel interfaces, one would have an additional security feature to exploit.

5.1.4 Kernel forwarder

One of the last possible ideas that could be implemented is the creation of a forwarder kernel: either related to the context of NSM or, more generally, to the context of routing between containers, it could facilitate the management of traffic forwarding. In this sense, one of the NSM developers within an issue suggested following the approach defined within "cmd-forwarder-kernel" [17], in which, however, there is no documentation whatsoever.

5.2 Conclusions

Considering the premises at the outset, it can be said that the results achieved were more than satisfactory. In fact, the main objective to be achieved was the implementation of SDN within the Kubernetes context, with which to extend the basic functionality. Also, related to this specific objective, security features were identified and subsequently configured so that the network definition software would allow the user or developer to specify some functionality to protect some services offered within the cluster. However, the initial vision proved to have some challenges within it: specifically, the implementation of these security functions had been modeled through the use of service meshes, thus seeing them as microservices that could communicate with each other, which presented some perplexities during the feasibility study. In fact, the solutions on the market to exploit the concept of service mesh were designed for application-level communications, whereas in this specific work, we wanted to work at the network level to exploit as many security

features as possible. For this reason, the Network Service Mesh framework was studied, with which implementing a chain of services dedicated to security was possible.

To comply with the principles and especially the idea behind software-defined networking, such as precisely the ease of management and understanding of the network, a command line interface was offered to the user with which it can be possible to declare intentions. The conception of intent is related to what one would like within the cluster, but without specifying the necessary technical details for configuration and operation in practice. Thus, one starts from as abstract a definition as possible, all the way to the actual implementation, so that the work is accessible to more or less experienced users. Finally, having to manage the deployment of various files within Kubernetes, an Operator was created, allowing a link between the user-defined intent and the actual deployment, taking care of managing resources and various files.

Indeed, this work represents a rather considerable initial cue, given the many future developments that could be conducted to realise a high-quality SDN for Kubernetes. In addition, what was intended was a prototype that could then be exploited in concrete cases as well and not just theoretical ones, demonstrating how through a decidedly new solution related to Intent-Based Networking and cloud-based solutions, a first version or starting point for a level product could be achieved. However, the most significant difficulty lies in using the NSM framework, which needs to be understood more thoroughly to proceed with modifications. However, not having appropriate documentation makes this task rather challenging.

Appendix A

User's manual

The following is a description on how to use the work done so that it can be exploited locally and new features can be tested.

A.1 Setup

First of all, the correct setup must be defined in order to make everything work, and for this reason the requirements are listed:

- In addition to the basic configuration to be able to use Kubernetes, you need to install Kind, Kubectl, Python and Golang, following the respective guides available online.
- Clone the corresponding *"NSM-SecurityChainingService" and "sfc-cli"* repositories, enter inside both and run

  ```
  pip install -r requirements.txt
  ```

  So that all the necessary dependencies are installed.
- Enter the *sfc-cli* folder of the corresponding repositories. Assuming you have cloned the repository to the desktop, you should place yourself at this level:

  ```
  Desktop/sfc-cli/sfc_cli
  ```

  and at this point run the command:

```
pip install -e .
```

This will install the CLI inside the machine and make it available to be used wherever it is desired.

- Enter into *"NSM-SecurityChainingService"* folder and create a cluster using the following command:

```
kind create cluster -name nsm -config 2nodes.yaml
```

A.2 Usage

Having done the setup, it is possible now see how to take advantage of what has been produced. First it is needed to head inside the API folder of the *"NSM-SecurityChainingService"* project and run the command:

```
python3 api.py
```

This will start the server in order to listen for requests that will be made by the CLI.

After that it is possible to go to any other folder and execute the various CLI commands. For instance:

```
sfc add monitoring
```

This command will create a JSON file in the folder where the command is run. Obviously, not every type of requirement can be specified; for testing purposes, it is suggested that the implemented dictionary be consulted first to understand what terms can be used. Next it will be necessary to execute the commands:

```
sfc setchain http://localhost:28900/api/upload_json
```

o send the file to the server, where it will be managed and translated, and then

```
sfc deploychain http://localhost:28900/api/deploy
```

So as to start the creation of all the necessary resources. In fact, through this command the setup of NSM within the cluster is executed.

At this point one has to wait until the configuration of NSM is completed. Opening a new terminal, it is then necessary to locate within the API folder and then inside the newly generated outputs folder of this kind:

```
outputs_2023-11-29_7b25d733-b08d-442a-9f3b-d748e1dc64ff
```

and run the command :

```
kubectl apply -f obj.yaml
```

A corresponding folder, which has a name of this kind

```
nse-composition-2023-11-28_11406f67-ee7d-4ee7-881c-9c8515fbe345
```

is created. Within this folder all the deployments files are placed. As a final step, to apply the deployments, you must again place yourself within the "NSM" repository and use the command:

```
kubectl apply -k nse-composition-2023-11-28_11406f67-ee7d-4ee7
```

it is important to note that one must always consider the last folder created. The alphanumeric string corresponds to that of the outputs folder, within API.

Before continuing with the final part of the configuration, it is important to note a bug present when writing YAML files. In fact, there are some fields that are not written correctly, meaning that they do not maintain the use of double quotes when specified. Specifically within spec.template.metadata.labels the field

```
spiffe.io/spiffe-id: 'true'
```

should actually be

```
"spiffe.io/spiffe-id" : "true"
```

Similarly, the environmental variables "NSM_REGISTER_SERVICE" and "NSM_SERVICE_NAME" present as values *'false'* and *security-chain* when it should be *"false"* and *"security-chain"*. By then making these changes in the various pods files, thus *nsc, nse, patch-nse, and those of the security functions*, the deployment will occur correctly.

The last step for the chain to work concerns the routing configuration that must be done in the intermediate pods of the chain. In particular, this type of configuration of IP tables must be done:

```
apk add --no-cache iptables
INTERFACE=$(ip -o -4 addr list | grep "172.16.0.0/32" | awk '{print $2}')
iptables -t nat -A PREROUTING -i $INTERFACE -j DNAT --to 172.16.1.0
iptables -A FORWARD -i $INTERFACE -o kernelint2 -j ACCEPT
iptables -A FORWARD -i kernelint2 -o $INTERFACE -j ACCEPT
```

Therefore, we need to see the IP address directly from inside the container and then replace it to *172.16.0.0/32*. Similarly, one has to consider the ip of the next pod: so it might be faster to consider the variable "NSM_CIDR_PREFIX" to do this.

At this point, in case you are using Suricata, you have to enter the container, through the command *kubectl exec -it pod_name -n namespace -c container – /bin/sh*, and execute the command:

```
suricata -S additional/rules/local/local.rules -i interface_name
```

Doing so will consider the rules in the file defined and loaded inside the container to start suricata.

While, regarding the use of Squid, after defining the rules within the squid.conf configuration file, you can test whether or not the rules are enforced by:

```
curl -x http://172.16.0.0:3128 http://www.google.com
```

run from the client container.

Bibliography

[1] Xu Zhiqun, Chen Duan, Hu Zhiyuan, and Sun Qunying. «Emerging of Telco Cloud». In: *China Communications* 10.6 (2013), pp. 79–85. DOI: 10.1109/CC.2013.6549261.

[2] Ch V Raghavendran, G Naga Satish, P Suresh Varma, and G Jose Moses. «A study on cloud computing services». In: *International Journal of Engineering Research & Technology (IJERT)* 4.34 (2016), pp. 1–6.

[3] SK Sowmya, P Deepika, and J Naren. «Layers of cloud–IaaS, PaaS and SaaS: a survey». In: *International Journal of Computer Science and Information Technologies* 5.3 (2014), pp. 4477–4480.

[4] John Wack, Ken Cutler, and Jamie Pole. «Guidelines on firewalls and firewall policy». In: *NIST special publication* 800 (2002), p. 41.

[5] K. Shanthi and R. Maruthi. «A Comparative Study of Intrusion Detection and Prevention Systems for Cloud Environment». In: *2023 4th International Conference on Electronics and Sustainable Communication Systems (ICESC)*. 2023, pp. 493–496. DOI: 10.1109/ICESC57686.2023.10193694.

[6] Robert Botez, Jose Costa-Requena, Iustin-Alexandru Ivanciu, Vlad Strautiu, and Virgil Dobrota. «SDN-Based Network Slicing Mechanism for a Scalable 4G/5G Core Network: A Kubernetes Approach». In: *Sensors* 21.11 (2021). ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/21/11/3773.

[7] Nathan Sousa, Danny Perez, Raphael Rosa, Mateus Santos, and Christian Esteve Rothenberg. «Network Service Orchestration: A Survey». In: *Computer Communications* 142 (Mar. 2018). DOI: 10.1016/j.comcom.2019.04.008.

[8] *Docker Doc.* https://www.docker.com/.

[9] *Kubernetes Doc.* https://kubernetes.io/.

[10] Boris Lublinsky, Elise Jennings, and Viktória Spišaková. «A Kubernetes 'Bridge' Operator between Cloud and External Resources». In: *2023 8th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*. 2023, pp. 263–269. DOI: 10.1109/ICCCBDA56900.2023.10154770.

[11] *Network Service Mesh.* https://networkservicemesh.io/.

[12] *Kopf perator.* https://kopf.readthedocs.io/en/stable/.

[13] *Nsm-operator*. https://github.com/networkservicemesh/nsm-operator.

[14] *networkservicemesh/deployment-k8s*. https://github.com/networkservicemesh/deployments-k8s/tree/main.

[15] Ákos Leiter, István Kispál, Attila Hegyi, Péter Fazekas, Nándor Galambosi, Péter Hegyi, Péter Kulics, and József Bíró. «Intent-based 5G UPF configuration via Kubernetes Operators in the Edge». In: *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*. 2022, pp. 186–189. DOI: 10.1109/ICUFN55119.2022.9829576.

[16] Lei Pang, Chungang Yang, Danyang Chen, Yanbo Song, and Mohsen Guizani. «A Survey on Intent-Driven Networks». In: *IEEE Access* 8 (2020), pp. 22862–22873. DOI: 10.1109/ACCESS.2020.2969208.

[17] *cmd-kernel-forwarder*. https://github.com/kubeslice/cmd-forwarder-kernel/tree/master.