POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Degree Thesis

Security orchestration in Kubernetes with Verefoo

Supervisors

Prof. Riccardo SISTO

Prof. Fulvio VALENZA

Prof. Jaloliddin YUSUPOV

Candidate

Gaetano BUSCEMA

2020-2021

Portami con te via
Finché sento che c'è sintonia
Diretti verso altri lidi

Table of Contents

List of Figures

Summary

With reference to the conventional networking communication mechanism, nowadays the virtualized networks are introducing some changes in the way the networks are built and modelled. These networks have the purpose of incrementing the flexibility of the overall configuration by exploiting a totally new software approach: instead of some dedicated hardware components, it is possible to decouple the network functions from physical devices and run them inside VMs (Virtual Network Functions - NFVs).

These functions can be inter-connected to create a chain of services that must be launched to perform a specific sequence of operations: this concept is called Service Function Chain (SFC) which found a concrete implementation in the Service Graph (the concatenation of services established by the network designer). One of the main problems in the creation of the Service Graph (which is, usually, a manual task) is that is prone to human errors and not flexible to network changes (especially if new security constraints will be put inside the network itself).

In this scenario, the VEREFOO tool proposes a way to solve all these problems through a Security Automation approach. The framework's main goal is to automatically allocate the Network Security Functions (NSFs), such as firewalls or anti-spam filters, inside the Service Graph received by the user. The NSFs allocation will be carried out considering:

- the optimal arrangement of the NSFs in the graph
- the compliance with the Network Security Requirements that were received with the graph itself

On the other side, during the last years, we are facing an increasingly migration of the IT services on the cloud, more and more by using containers. The containerization (the mechanism that let the developer to encapsulate the applications inside autonomous environment) owes its success to the lightness of its approach, in fact it has various advantages over the traditional virtualization approach:

- containers' resources will be isolated, in the kernel level, without the need of using the hypervisor technology
- containers are more efficient and light-weight than a traditional Virtual Machines
- major deployment agility, scalability and development environment portability

With the containerization another problem arises because, by enlarging the system and increasing the number of containers in it, there is the need of managing and controlling the applications life-cycle in a proper way. Here come into play container-orchestration systems like Kubernetes, an open-source project powered by Google, which main goal is to manage medium-large sized application cluster. An orchestration framework has the goal of automating the applications deployment and automatically scaling the system (for example, if some topology changes took place).

One of the main aspect that this paper aims to cover, and discuss, is the reflection of the network rules, produced by the Verefoo tool, inside Kubernetes and, when it won't be possible to do so, understand the limitations and the problematic that were encountered during the work.

Inside the perimeter of this paper, a micro-services architecture was developed to reflect the network rules produced by Verefoo over Kubernetes and to update the Verefoo graph if some topology changes took place inside the cluster.

Chapter 1

Introduction

1.1 Thesis Objective

In relation to the traditional network system, the users connect to each other and to different services by using **communication networks**: large and complex infrastructures, continuously evolving, that are made of multiple nodes components and links. A network has a strong dependence on hardware, that's to say on the physical equipment: this characteristic makes it a very rigid and expensive infrastructure to operate with in fact, making changes to it or launching new products and services are difficult and long-time consuming activities. The inclusion of new functionalities is delayed by the need for economies of scale to manufacture the specific network hardware. As well as by the long standardization processes between the vendors and operators, what's more the very complexity of the network makes it hard to introduce changes from the perspectives of management, configuration and control. In this situation, it is needed to transform the network to make it a much more flexible infrastructure: the network must be able to adapt in a better way to the quick pace imposed by the evolution of the digital world. To that end we have **Network Virtualization** as a significant lever for transformation. Network virtualization is composed of two different technologies:

- **NFV**: virtualize the network consists of separating software from hardware and network equipment, so that network functionalities become independent of the physical equipment supporting them. As all functionalities are found in software and mounting the physical machines, the same server can be used for several purposes depending on the software installed on it. Now the functionalities reside exclusively in software, and for this reason it is possible to package each network function in one or more virtual machines and to decide where to execute them. This is what's known as network function virtualization (NFV)

- **SDN**: Hardware resources must be allocated with great care in order to achieve high performances, so that functions can work at the same speed as the traditional network functions. At the same time, those network functions must be interconnected with each other in a coherent way, in order to provide network services. These interconnections can be managed from a centralized point: the centralization of the control plane is what is called software-defined networks (SDN)

All these concepts are inherited by the **Service Function Chaining** (SFC) mechanism: this is the ability to route the network packet flows through the network via path other than the basic destination-based forwarding (which is typical of IP based networks). This mechanism can be used to enforce policies, perform security functions and other broad range of features for selective packet handling in the network itself. Service functions such as packet filtering on the firewalls, load balancing and others, have been widely utilized in the delivery of services to the end-users. With the introduction of virtualization ,the deliveries of such services is undergoing of significant change. SFC can be used in the context of network function virtualization (NFV) where virtual network functions (VNF) replace the traditional physical network devices like firewalls and load balancers to offer a chain of end-to-end services. The main aim is to **classify the traffic** (classification mechanism) to route the packets through a chain of specific services, in order to be processed correctly through the security service functions needed by the packets themselves. The software defined network infrastructure provides **service function forwarders** that are responsible for forwarding traffic to one or more of the connected service functions as well as handling traffic process by service functions.

In the perimeter of the previous assumptions, **VEREFOO**(VErified REFinement and Optimized Orchestration) is a framework [1] which purpose is to automatically allocate on a **Service Graph** (a generalization of the SFC provided by the user) the network service functions (NSFs) that are needed to cover all the security constraints/requirements given to the tool itself. In this scenario, by adopting a formal and rigorous approach based on the **MaxSMT** problem, which make the solution more robust, the Verefoo tool proposes a new way to automatically place, in a correct and optimal way, the firewalls inside a network environment provided by the user: the network-designer will provide to Verefoo a service graph, containing some specific security constraints, and the tool will process the firewall positions in it.

On the other side, with the cloudification, the functionalities introduced by the virtualized network were brought into a new context: the network functions will be

deployed as **containers** in some specific virtual machine.

Differently from the generic virtualization, one of the strongest aspect introduced by the container approach was the **resource consumption** [2]: because the memory and CPU consumed by a container was lower than instantiating a specific Virtual Machine, the machine that is used to run the containers won't need a lot of resources to keep everything up and running.

The idea of running complete services as they were operating system's processes was revolutionary but, on the other side, because of the limited resources consumption, many different machines were able to run, potentially, an high number of containers. Of course this aspect was an important advantage, but it has its drawbacks: by increasing the number of containers inside the architecture, the overall design could be confusing and error-prone. To overcome all these problems, it is necessary to use an **orchestrator**: it is an entity that can manage the containers lifecycle and coordinate the communication between them. Here comes into play technologies like Kuberentes, proposed to orchestrate and manage easily the containers-lifecycle.

The thesis objective has been to understand how we can reflect the security constraints and rules, produced by the Verefoo tool, inside a Kubernetes cluster and, when it wasn't possible, understand the limitations and the problematic that were encountered during the work. In order to do that, some activities were performed inside this paper:

- **Verefoo functionalities classification**: study and classification of the Verefoo's functionalities of interest that must be reflected above the cluster (such as the Neighbour or the Forwarding Mechanism supported by Verefoo)
- **Technology studies**: in order to understand how to reflect the Verefoo functionalities inside Kuberentes, some technologies were discovered and studied (such as the Istio service mesh or the Calico CNI). Inside this phase, were involved also some design activities in order to understand how these technologies can be used to reflect the Verefoo functionalities
- **Technology Limitations studies**: when some Verefoo's functionalities couldn't be reflected (such as the Verefoo Forwarding Mechanism), will be discussed the reasons why this behaviour couldn't be covered by the involved technologies
- **Use-case appliance activities**: if a specific technology can reflect a Verefoo functionality (like the Neighbour Mechanism), different scenarios were created to understand, potentially, the limitations (like in the inter-cluster use case)

- **Software development activities**: a small micro-service architecture was designed and developed to make the Verefoo tool communicate with the Kubernetes cluster. The application evolved parallelly to the thesis progressions

1.2 Thesis Description

After the introduction chapter, the thesis will follow the structure described below:

- **Chapter 2 - Background**: This chapter represents the thesis background section in which will be covered, briefly, all the technologies and concepts that must be known to understand the whole work
- **Chapter 3 - Topology changes: reflect cluster changes inside Verefoo**. Inside this chapter will be described the activity that begun the thesis work. The purpose of the task was to reflect the cluster topology changes (regarding the pod's life-cycle) inside the Verefoo graph
- **Chapter 4 - Network changes: reflect Verefoo changes inside Kubernetes**. Inside this chapter it was studied a way to apply the Verefoo Neighbour and Forwarding Mechanisms inside Verefoo. As it will be described later on, only the Neighbour Mechanism could be reflected in an appropriate way above the cluster
- **Chapter 5 - Neighbour Mechanism: Inter-Cluster use case**. This chapter aimed to apply the Verefoo neighbour mechanism between a chain of clusters. As we will see later on in more detail, because of the Kubernetes ingress and egress traffic policies, the use case couldn't be reflected correctly
- **Chapter 6 - Neighbour Mechanism: Intra-Cluster use case**. This chapter comes back to the scenario viewed inside the Chapter 3. Its aim was to reflect the neighbour mechanism only inside a specific cluster, between a group of pods. Differently from the previous chapter, here will be described a demo in order to understand how the mechanism works
- **Chapter 7 - Conclusions and Future Works**. This chapter summarize briefly all the topics covered by the thesis and the uncovered aspects that could let to some future research works

Chapter 2

Background

In order to have a better comprehension of the context of work of this thesis, this chapter provides a brief description of the main technologies and tools that were used during the whole work.

2.1 Verefoo

VEREFOO (VErified REFinement and Optimized Orchestrator) [3] is a framework designed to provide an automatic way to allocate packet filters – the most common and traditional firewall technology – in a **Service Graph** defined by the service designer and an auto-configuration technique to create firewall rules with respect to the specified security requirements.

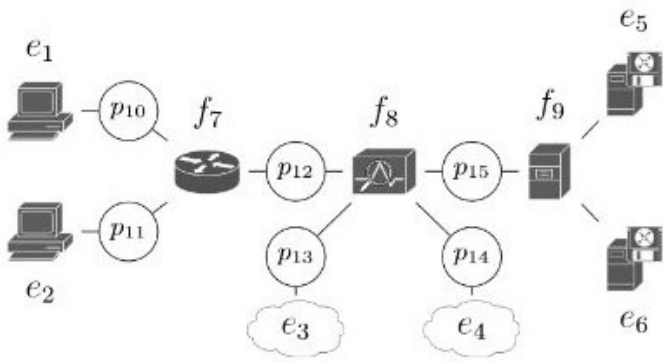

Figure 2.1: Service Graph example

Software-Defined Networking (SDN) and **Network Functions Virtualization** (NFV) are new technologies, designed to introduce flexibility in networking.

The **SDN** let to define, in runtime, the paths that traffic flows must follow. The **NFV** enables virtualized network functions, installed on general-purpose servers in the cloud. The service designer, by using these two features, will define the intended network services through a **Service Graph** (SG), a component that will collect the involved service functions and their interconnections. Inside a virtualized environment, the security automation is becoming more and more important because of the complexity of the environment itself: if we take into account all the defenses and constraints that we must respect inside our architecture, setting the security or building up the system defenses will become harder with the introduction of new components.

Because of the repeating nature of the tasks that must be done, all these tasks could become automatic. In order to simplify the human effort, a common security task could be easily done by using some software-based controls: the user will simply define some Network Security Functions (NSFs) in order to cover the security constraints that the network must respect, called Network Security Requirements (NSRs), without define manually the security rules. For example, if we need to isolate the system or some components after an external attack (DoS or resource exploitation), we can enforce this behaviour by configuring some NSFs that will enforce this mechanism. If we consider to perform this task manually, it could become difficult and sometimes error-prone: we don't have any guarantee that the user will always define a correct configuration or consider some other depending constraints.

As we can easily understand, by adding even more business logic the probability of creating an incorrect setup will increase drastically: for this reason the security automation controls can be used to perform some correctness check during the creation of the rules. Finally, to guarantee an optimal configuration setup, these controls must consider also the resource consumption in order to improve the overall performances.

By considering all the previous aspects, the principal aim of the Verefoo [4] tool is to define, automatically, the optimal allocation placement of the firewalls inside our architecture by refining the Service Graph, received in input, in order to respect the constraints and requirements defined before.

To sum up, Verefoo will focus its functionalities and mechanism all around the **packet filters** for two reasons: first of all it is the most exploited security defense in an architecture and then it represents the most used firewall technology.

2.2 Kubernetes

Kubernetes is a container orchestrator or, in other terms, it is a system for managing containerized applications across a cluster of nodes.

Gordon Haff, Red Hat technology evangelist, will describe Kubernetes in his book, *"From Pots and Vats to Programs and Apps"* [5]:

"Kubernetes, or k8s, is an open source platform that automates Linux container operations. It eliminates many of the manual processes involved in deploying and scaling containerized applications," and continues *"In other words, you can cluster together groups of hosts running Linux containers, and Kubernetes helps you easily and efficiently manage those clusters."*

To summarize his thought, Kubernetes is a system that will help the systemist to organize, manage and deploy applications like virtual machines (group of VMs) or application that runs in container (e.g. Dockerized applications). Kubernetes will also give to the user the possibility to manage and control the networking between each component inside the cluster.

By deploying Kubernetes on a machine, a cluster will initially be created: it will be the **case** that wrap the overall infrastructure and components. The Kubernetes cluster is composed by two main kind of entities: Master and Nodes [6]. The Master represents the **control plane**. It will be composed by some components that will be used, from the outside (the user), to interact with the cluster. These components represent the main contact point between the user and the cluster itself. On the other side, Nodes are the place where the **business logic** will run (the containerized user's apps). To summarize, the containerized applications will run inside each node and will be controlled through the master-node.

Both master and nodes are composed by many different components that need a little overview to understand as best as possible their role and functionalities.

2.2.1 Master Components

As already mentioned, the Master Node is composed by elements which operate as the main management contact point for users, and consists of the following components:

- **Etcd**: is a consistent and highly-available key-value map store used as K8S backing store for all the data used by the cluster. In other terms, it is

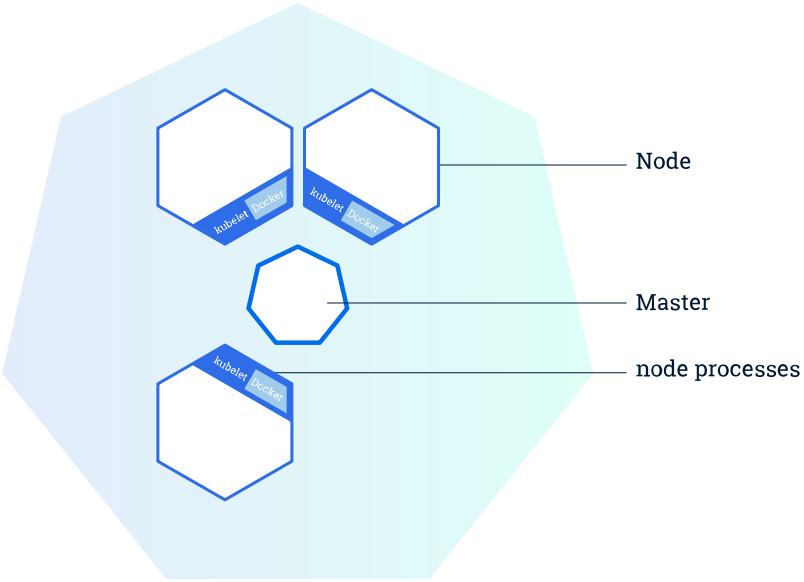

Figure 2.2: Kubernetes Cluster

a database that will be inside the cluster, used to reflect the state of the kubernetes cluster itself.

- **API Server**: is what exposes all the APIs developed by the Kubernetes comunity. These APIs represent the methods offered to the user to interact with the cluster itself. In simple terms, it is the management core of the entire cluster. It is a sort of bridge between the input commands and the various components that compose the cluster. It is the facade of the Kubernetes control pane.

- **Controller Manager**: it will manage the state of the cluster and regulate its life cycle. It is also responsible of performing routine tasks. For example, if we have defined a service with a certain number of replicas, the replication controller will try to understand if the replicas matches the number currently deployed on the cluster.

- **Scheduler Service**: is what manage the nodes workloads. To summarize, it is responsible of placing the workload on an acceptable node(s): it will try to determine which pods have to be placed inside each node, according to the scheduling queue.

2.2.2 Node Components

As we said, the Node is the place in which the containerized applications will run. It is composed by the following elements:

- **Docker**: is the virtualization engine used to run all the applications.

- **Kubelet**: is the main contact point between the node and the control pane services (the ones managed by the master-node).

- **Proxy**: it is used for reflecting the networking rules defined in the cluster and performing connection forwarding.

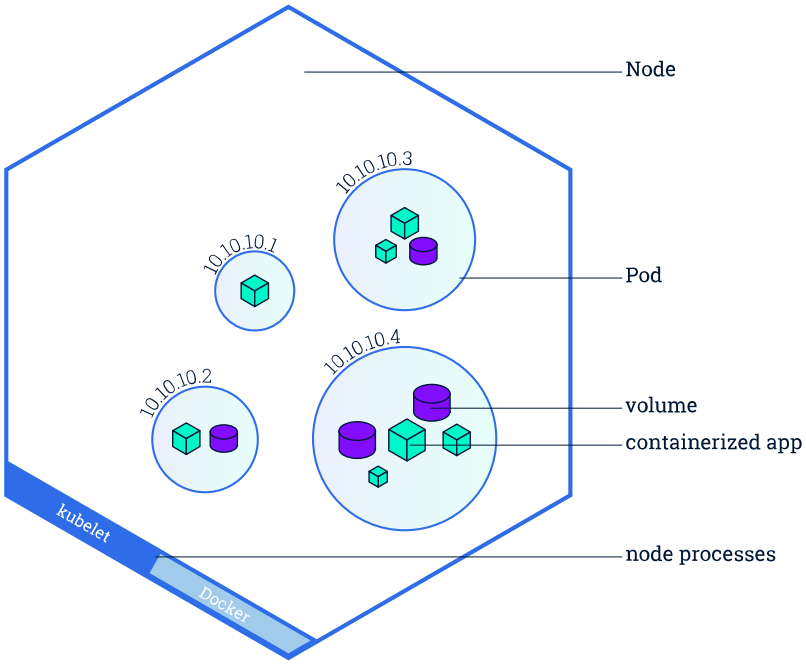

Figure 2.3: Kubernetes Node

2.2.3 Kubernetes Work Units

The end user will never directly interact with the components showed before. The Kubernetes API will give to the user the following interaction methods, to let him interact with the infrastructure:

- **Pod**: it is the lower level unit in Kubernetes. The Pod represents the place in which the business logic will run. It could be composed by a single container or a group of inter-dependent containers that share some resources or need to interact each other. In simple terms, it is a service or a collection of tightly coupled applications. The Pod will contain a storage unit and it will be represented by a unique network IP address.
- **Service**: The Service is a component that stands in front of a group of pods. Usually it will be used to expose a pod to the outside environment or to perform some balancing operations to distribute the incoming network traffic (load balancer).
- **Label**: The Label is a key-value tag pair. It is used to recognize a specific group of pods and it will be used by the service to group different pods.
- **Deployment**: it is a mechanism to update create and balance the pods instance. It will monitor a pod state and will try to allign the state itself to

the desired one. For example, the developer can tell kubernetes how many pods the cluster has to mantain: if the number of pods will be below the desired one, the cluster will proceed by creating the missing ones.

- **Ingress**: it will manage all the incoming traffic to a specific cluster. It will provide an SSL termination and path-based routing mechanism.

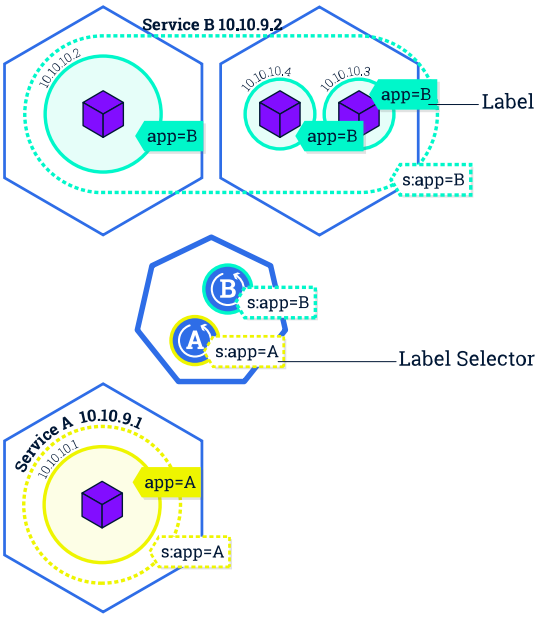

Figure 2.4: Kubernetes Work Units

2.3 Publish-Subscriber pattern

As mentioned by Matthew O'Riordan in the article titled "Everything You Need To Know About Publish/Subscribe" [7]:

"The Publish/Subscribe pattern, also known as pub/sub, is an architectural design pattern that provides a framework for exchanging messages between publishers and subscribers. This pattern involves the publisher and the subscriber relying on a message broker that relays messages from the publisher to the subscribers. The host publishes messages to a channel that subscribers can then sign up to."

In other terms, the publish/subscribe is a distributed system pattern that will need the presence of three main kind of components: the broker, the publisher and the subscriber.

The broker will be the engine (server side) that manages the topics and the availability of the resources: it is usually supported by a zookeeper service that will try to maintain the resources up and running. The topic is a lightweight mechanism to send messages and event notifications through the system. It is a little bit different from a classic message queue, because on the same topic there could be, potentially, many different subscribers interested in the same message: as we will see later on, there are many different policies that can be used to specify the way the subscribers will consume from the topic.

The publisher is the component that will produce some messages on a specific topic and the subscriber will be the entity that listen to a specific topic in order to read messages from it.

Publisher and Subscriber (we could have many different publishers and subscribers) don't know anything of each other's identity: this represents the strongest and main feature of this pattern because it allows to broadcast a group of messages to different parts of the system asynchronously. As we can easily understand, the publisher and the consumer will be totally independent to each other's.

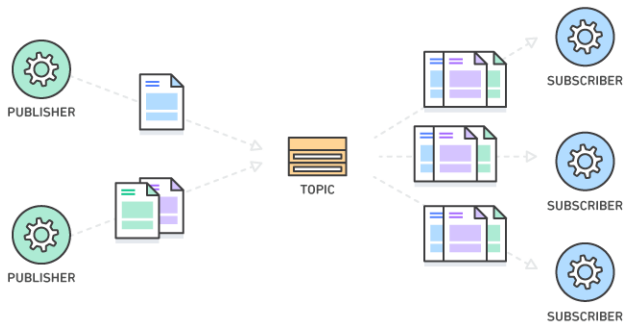

Figure 2.5: Publish Subscriber Architecture

As we already mentioned, the publish/subscribe is a generic pattern: it will simply offer some guidelines to implement and use it. By exploring this design pattern, we will discover many different frameworks which implement the topic-base publishing architecture. Below, some of the most used frameworks:

- **Apache Kafka**: One of the most famous and used framework (because of its robustness and popularity). It implements the pub/sub feature by using a log system: the messages will be saved inside a log file. This behaviour will let the user to read again some messages that were already consumed.

- **RabbitMQ**: RabbitMQ, instead of strictly following the publish subscribe pattern, better fit the message log mechanisms. In any case, it will offer some configuration flexibilities to configure it for a fanout or direct communication between the system components.

- **PushPin**: It is an open source project. It is designed to become a real-time messaging system, meant to be used at the edge of the distributed system. It will work over the WebSocket standard protocol to push messages. It will better work with low throughput environments.

2.3.1 Apache Kafka

Kafka is designed to be a distributed system [8]: it will be composed by many different server and client that continuously will communicate through the TCP network protocol. Thanks to its resources consumption, it can be easily deployed and run inside a Virtual Machine or Container.

One of the main feature offered by the Kafka framework, is the overload distribution. As mentioned, in Kafka we have two kind of components, the server and the client.

- **Servers**: Kafka is a cluster of servers that will be distributed in the whole system. Some of these servers will compose the "storage layer" by saving the data needed by the system; this collection of servers is called "broker". One of the mission hold by Kafka is the fault-tolerance aspect: the cluster has to be scalable and robust in order to avoid data loose if some of the servers, that compose the system, will shut down or stop unexpectedly.

- **Clients**: the client will be implemented by the end-user of the distributed system. The client will act as a publisher or subscriber and will strictly interact with the broker through the topic mechanism: by selecting a specific topic, in which publish/subscribe, the edges of the system will communicate through the event-stream. The event streams can work both in parallel and in a fault-tolerant way so that, if some network problems will occur, no data will be lost.

2.3.2 Apache Kafka main concepts

The event or message is the unit of known in a Kafka distribute system [8]. The event will carry inside some particular and specific information that the user or the publisher wants to transfer through the system.

The publisher is the edge-user which is writing something on a specific topic, in order to be read by the subscriber, and the end-user is the entity interested on a specific kind of information (topic) that will read/consume from a topic. One of the main feature of this distributed system is that publisher and subscriber are fully decoupled: they don't have to know each other, but simply rely on the topic to communicate. This design element is fundamental to guarantee the full scalability of the system.

Events will be stored and memorized inside the topic. The topic can be thought as a single unity of storage in which many events or messages will be collected. One of the strongest feature of topics is that are designed to be multi-producer and multi-subscriber: the topic can have zero, one or many producers and subscribers that will work in parallel.

There are many consuming policies that can be adopted to process the events from the topics: for example, the messages can be processed exactly-once or consumed by distributing the load to the available consumers (following some policies, like the Round Robin one). In any case, the messages are not deleted after the consumption (here there is one of the strongest differences between a topic oriented and a message queue system) but the user can define how long the broker should retain the data inside the topic, after which they will be deleted from it.

Topics are divided into **sub-topics** called partitions: we can think the partition as a sort of **bucket** into which the broker will put some specific events. The publisher can write a new message by pairing it with a key: the key will represent the value used by the broker to assign this message to a specific partition. Messages with the same key will be collected inside the same partition, otherwise they will be wrote inside an undefined partition. The Kafka broker will guarantee that, messages with the same key will be published inside the partition in exactly the same order of production: this represents an important feature because the consumer will be able to read these messages in sequence.

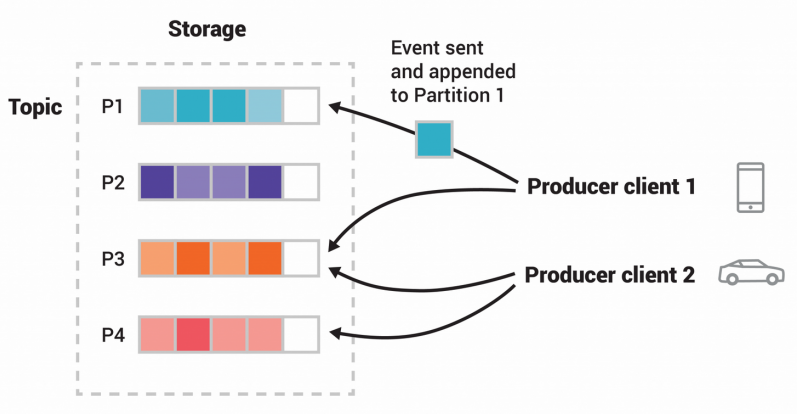

Figure 2.6: Topic-Partition Architecture

In order to make the whole system fault-tolerant, the user can replicate each topic across the distributed system: by this way, the messages will be available on multiple broker. If something goes wrong with the data, each broker will have its copy of the data, that can be recovered easily with no data loose.

2.4 NFV

Many service providers go beyond simply by providing some network connectivity for their enterprise customers. They also offer additional services and network functions like Network address translations (NAT), firewall, encryption, domain name service (DNS), caching and others.

In the past, when the service providers need to distribute some network functions, proprietary hardware was used to run the functionality needed and then deployed on premise, at the customer edge [9]. This approach provides additional revenue, but deploying multiple proprietary devices is costly and makes upgrades difficult: every time a new network function is added to a service a truck roll is required to install the dedicated hardware.

Service providers began exploring ways to reduce cost and accelerate deployments through network function virtualization (NFV). The aim of this technology is to decouple functions like firewall or encryption from dedicated hardware and move them to virtual servers. Rather then using proprietary hardware (sometimes also expensive), service providers can build and setup inexpensive servers to allocate virtual machines that perform some specific network functions: the network function will be represented by the software installed over the hardware (VM, K8S, OpenStack, ecc...). Because of the fact that this technology is in its early stages, it

is constantly evolving and changing: as evidenced by the studies carried out in the researches titled *"A novel approach for security function graph configuration and deployment"* [10], *"Adding Support for Automatic Enforcement of Security Policies in NFV Networks"* [11] and *"Towards an Efficient Management and Orchestration Framework for Virtual Network Security Functions"* [12], many researches were made to improve the overall security and configuration mechanism.

Another task, that this technology aims to solve, is the one of detecting errors and anomalies in the data plane to prevent some future failures: if we consider the dynamism of the networks, this operation must be performed within quite strict times. Also this issue has been extensively studied nowadays, like in the work done in the thesis entitled *"Improving the Formal Verification of Reachability Policies in Virtualized Networks"* [13], in which it was tried to improve the performances in verifying the reachability of the packets by solving a Satisfiability Modulo Theories problem.

As it was anticipated, the NFV groups many different functions into a single physical server (hardware) reducing the cost and minimizing the technician support on the customer's location. If a customer wants to extend the network functionalities, by adding a new network function, the service provider can easily instantiate a new VM in which will be installed and run the needed network function (firewall functionalities, proxy, packet filtering, white/black lists, ecc...).

A generic NFV architecture will mainly consists of the following components [14]:

- **Virtualized network functions (VNFs)**: are network functions that are deployed as software on a specific hardware (VM, container ec...) such as NAT, DNS, encryption ecc...
- **Network functions virtualization infrastructure (NFVi)**: it is the collection of the components that build the overall infrastructure used to run the VNFs.
- **Management, automation and network orchestration (MANO)**: it is the framework that will manage the platform (NFVi) and provide the virtual network functions.

To summarize, NFV refers to the strategy of virtualizing network functions moving from separate proprietary pieces of hardware to software running on virtual servers using standard hardware.

Chapter 3

Topology changes: reflect cluster changes inside Verefoo

As we anticipated, Verefoo is a service that can perform a provably correct and optimized automatic configuration and orchestration of network security functions (e.g., packet filtering firewalls or channel protection systems) in virtualized networks, by adopting a policy refinement mechanism.

To better understand the purpose of this chapter, we have to remember the thesis' final goal: we searched for a mechanism to reflect the network rules, produced by Verefoo, inside Kubernetes. As it is understandable, we need to react also to the topology changes inside the K8S cluster because any change could influence the networking itself: for this reason, the Verefoo graph must always reflect the topology inside the Kubernetes cluster and have as many nodes as the pods up and running. For example, if some pod will die or will be allocated the corresponding Verefoo node will be deleted/created.

Kubernetes emits events whenever some changes occur in any of the resources that it is managing. These events help understand what happened inside the cluster when a specific entity will change its state. For example, each time a pod will start, stop or change its configuration a new event will be produced by the k8s controller and emitted to all the listeners: sometimes these events are useful to understand the status of the cluster itself or get warned when some errors occurred unexpectedly.

As already mentioned, the first step, for the thesis progression, was the development of a service that aimed to stream Kubernetes events into Kafka for

observability and topology reaction from Verefoo. This step was fundamental: by passing the service graph configuration inside verefoo (represented by the kubernetes cluster configuration), it will be able to recompute the network infrastructure, if some changes took place, in order to be communicated to the Kubernetes cluster.

In particular, in the following chapter, we considered a simple kubernetes configuration composed by:

- one cluster
- many different pods (each one running some specific business logic)
- each pod will run onlu one container
- inside the cluster there is no networking restrictions: each pod is able to communicate with all the others

3.1 Verefoo-Kubernetes Relationship

First of all, we had to understand how the cluster topology could be reflected inside Verefoo: in order to do this we had to define a relationship between the Verefoo and the K8S main components.

The Verefoo Graph collects a list of different Nodes, each one identified by unique string and connected with different neighbours and containing different tags.
On the other side, the Kubernetes cluster is composed of different Pods and each one could run many different containers.

As we can easily understand, there is a "natural" relationship between the Verefoo and the Kubernetes components:

- the Verefoo Graph can easily correspond to the Kubernetes Cluster
- the Verefoo Node (inside the graph) corresponds to the Kubernetes Pod (inside the cluster)

Initially, it was decided to follow the previous relationships between the Verefoo and the Kubernetes environments but, as we will see later on, we had to change this relationship to adapt Verefoo to the different scenarios that will be covered in the next chapters.

3.2 Developed Architecture

In order to keep the environments (Kubernetes cluster and Verefoo graph) aligned and coherent, it was needed to develop a micro-services application: the architecture, that will be proposed inside this chapter, represents the starting point of this paper and its structure will change with the evolution of the thesis scope.

The architecture proposed in this chapter will focus on three main aspects:

- Kubernetes events generation
- Event transmission through Kafka
- Verefoo event reaction

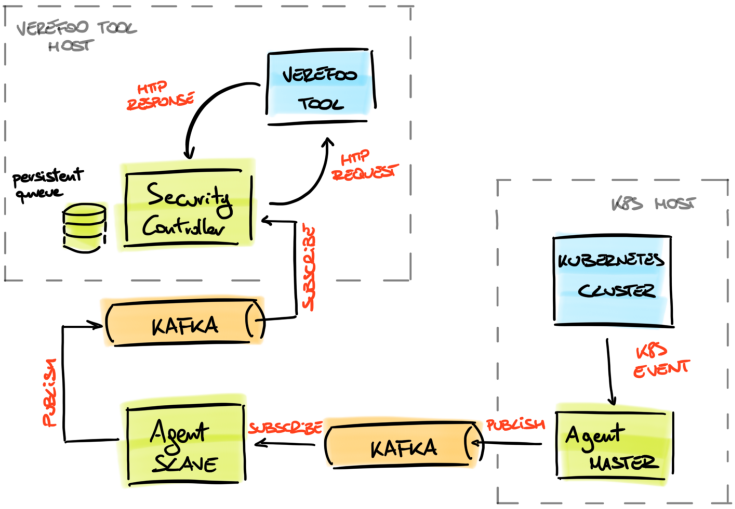

Figure 3.1: Topology changes: micro-services application initial architecture

From an high-level point of view, the designed structure relies on four different micro-services:

- **Agent Master**: this component, on one side, will communicate with the Kubernetes Local server and, on the other side, will communicate with the Kafka Message Broker. The role of this micro-service is that of collecting all the events coming from the Kubernetes cluster and redirect them on the Kafka Queue. In summary, the Agent Master represents both a Kubernetes Client and a Kafka Producer

- **Agent Slave**: the role of this micro-service is that of collecting all the events, coming from the Kafka Queue, and redirect them to the whole list of Consumers that was configured. It represents also a Kafka Consumer because it will subscribe to the same queue used, by the Master, to push the events
- **Kafka Message Broker**: in order to let the Slave and the Master communicate each other, we'll need a running message broker to transport all the upcoming messages
- **Security Controller**: this micro-service will receive all the events from the Agent Slave and modify (according to the events) the Verefoo graph thanks to the Verefoo REST API

In the following chapters we'll describe more in detail each component.

3.2.1 Agent Master

The Agent Master is the service that will connect to the local Kubernetes cluster and listen to the events generated from it, in order to transmit them through Kafka: on one side, will communicate with the Kubernetes server and, on the other side, with the Kafka Message Broker.

The most important goal of this micro-service is that of collecting all the events coming from the Kubernetes Server and filter them, in order to consider the most valuable.

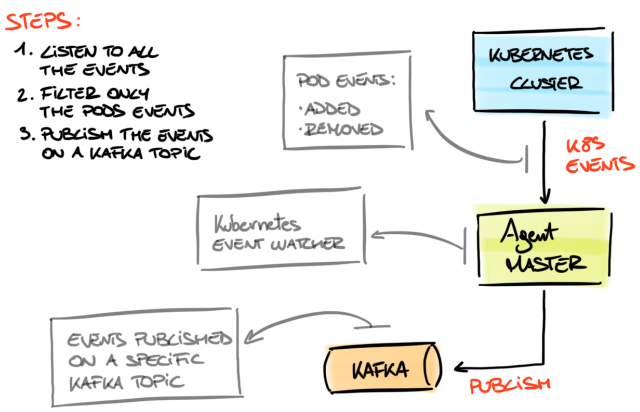

Figure 3.2: Topology changes: Agent Master architecture

Once the Master receive one event, it will redirect a new message through the Kafka Message Broker.

The presence of the Agent Master gives also the possibility to configure different topics: for example, if there is the need to create a queue for different purposes or needs, it is possible to configure one Agent Master (in order to publish, on the correct topic, the upcoming messages) and different slaves (one for each topic) in order to work with a specific topic. In summary it could be possible an 1:N or an 1:1 Master-Slave(s) configuration: as we will see later on, the 1:N relation between the slave and the master will be covered in the Inter-Cluster chapter, when we aimed to enforce the networking rules between many different clusters.

Later on, in this paper, we will "switch" the slave and master nomenclatures because of some changes in the relationships between the Kubernetes and Verefoo environments.

Fabric8io Library

As it was anticipated, the Agent Master need to listen to the Kubernetes events. In order to do that, some libraries were developed and offered from the community as open source projects.

By making some searches through the developed interfaces, two main libraries emerged as the main repositories [15] :

- fabric8io/kubernetes-client
- kubernetes-client/java

Fabric8 Kubernetes Client project started in 2015, and started in parallel with Kubernetes; the first idea was that of serving it as a Java framework to interact with the cluster. From an usage point of view, the Fabri8io project was provided as an abstraction layer over the Kubernetes REST API.

Official Kubernetes Java Client project started two years later in the 2017, together with other several clients written in many different languages like JS, Python, Perl etc. All these clients were generated from a common OpenAPI generator script: because of the OpenAPI skeleton, the usage of all the clients is similar.

Here in the following, there are some stats based on the GitHub repositories that were considered for this work:

GitHub Group	# Repositories	# Contributors	# Commits	# Releases
Fabric8io Group	190	192	3683	626
Official Kubernetes Group	13	95	1580	31

Figure 3.3: GitHub K8S libraries Statistics

we can notice, from an high level point of view, that the Fabric8io Group library is a way more supported from the development community; if we look also at the number of users, the Fabric8io library is way ahead.

From a coding point of view, the Fabric8io library seems to be a little bit less verbose and simpler; here are some examples of code usage [15]. The following chapters aim to show the differences, in the usage, between the two libraries.

Libraries comparison: List pods in Namespace

```
//Official Kubernetes Client

ApiClient client = Config.defaultClient();
Configuration.setDefaultApiClient(client);

CoreV1Api api = new CoreV1Api();
V1PodList list =
        api.listPodForAllNamespaces(null,
                null,
                null,
                null,
                100,
                null,
                null,
                null,
                null,
                null);

list.getItems().stream()
        .map(V1Pod::getMetadata)
        .map(V1ObjectMeta::getName)
        .forEach(System.out::println);
```

```
//Fabric8 Kubernetes Client

try (KubernetesClient client = new DefaultKubernetesClient()) {
    client.pods().inAnyNamespace().list()
            .getItems()
            .stream()
            .map(Pod::getMetadata)
            .map(ObjectMeta::getName)
            .forEach(System.out::println);
}
```

Libraries comparison: Creating a simple Service

```
//Official Kubernetes Client

ApiClient client = Config.defaultClient();
Configuration.setDefaultApiClient(client);

// Load Service YAML manifest into object
File file = new
    File(LoadAndCreateService.class.getResource("/test-svc.yaml").getPath());
V1Service yamlSvc = (V1Service) Yaml.load(file);

// Apply Service to Kubernetes API
CoreV1Api api = new CoreV1Api();
V1Service createResult = api.createNamespacedService("default",
    yamlSvc, null, null, null);
```

```
//Fabric8 Kubernetes Client

try (KubernetesClient client = new DefaultKubernetesClient()) {
    Service svc = client.services()

    .load(LoadAndCreateService.class.getResourceAsStream("/test-svc.yaml"))
            .get();

    client.services().inNamespace("default").createOrReplace(svc);
```

```
}
```

Libraries comparison: Creating a Deployment using Builders

```
//Official Kubernetes Client

ApiClient client = Config.defaultClient();
Configuration.setDefaultApiClient(client);

V1Deployment v1Deployment = new V1DeploymentBuilder()

    .withNewMetadata().withName("nginx-deployment").addToLabels("app",
    "nginx").endMetadata()
         .withNewSpec()
         .withReplicas(3)
         .withNewSelector()
         .withMatchLabels(Collections.singletonMap("app", "nginx"))
         .endSelector()
         .withNewTemplate()
         .withNewMetadata().addToLabels("app", "nginx").endMetadata()
         .withNewSpec()
         .addNewContainer()
         .withName("nginx")
         .withImage("nginx:1.7.9")
         .addNewPort().withContainerPort(80).endPort()
         .endContainer()
         .endSpec()
         .endTemplate()
         .endSpec()
         .build();

AppsV1Api appsV1Api = new AppsV1Api();
v1Deployment = appsV1Api.createNamespacedDeployment("default",
    v1Deployment, null, null, null);
```

```
//Fabric8 Kubernetes Client

try (KubernetesClient client = new DefaultKubernetesClient()) {
    Deployment deployment = new DeploymentBuilder()
```

```
    .withNewMetadata().withName("nginx-deployment").addToLabels("app",
    "nginx").endMetadata()
             .withNewSpec()
             .withReplicas(3)
             .withNewSelector()
             .withMatchLabels(Collections.singletonMap("app", "nginx"))
             .endSelector()
             .withNewTemplate()
             .withNewMetadata().addToLabels("app", "nginx").endMetadata()
             .withNewSpec()
             .addNewContainer()
             .withName("nginx")
             .withImage("nginx:1.7.9")
             .addNewPort().withContainerPort(80).endPort()
             .endContainer()
             .endSpec()
             .endTemplate()
             .endSpec()
             .build();


    client.apps().deployments().inNamespace("default").createOrReplace(deployment);
}
```

Libraries comparison: Final considerations

At the end, taking into consideration:

- Experience (year of launch)
- Github Repositories statistics
- Java Library Simplicity

the Fabric8io Kubernetes-Client is better in terms of usage. Because of its simplicity it can provide a good Java developing experience: for this reason it was chosen to implement the event listening mechanism inside the micro-services application.

Kubernetes Events

One of the most important feature exploited from the Fabric8io library was the Event Watcher: it is an API component that will listen to all the upcoming events

generated from the Kubernetes Cluster.

As we already know, Kubernetes offers a lot of APIs to be used and extended to interact with the cluster itself (for example, if there is the need to add new functionalities or simply interact with the K8S controller): this mechanism can be performed thanks to the K8S API which is composed by a group of loosely coupled entities/components that, potentially, interact together. Because of the complexity of this architecture, it is needed a mechanism to let each component interact with each others and communicate its status coherently with its life-cycle: the event management is the mechanism offered by Kuberentes to overcome the challenge described before [16].

Kubernetes events are entities and objects used for the cluster observability: they will transport some status changes informations, to better understand what is happening to each component inside the cluster. Both the native and the extended components are able to create Kubernetes events through the API Server component.

In particular, the Kubernetes event [17] is a resource type that is automatically created when other resources change their status, or some errors occur, or other messages, that should be broadcast to the system, are received; this detail represents the starting point for the development of the micro-service application described above: in order to reflect the Kubernetes topology changes, into the Verefoo graph, we have choosen to listen to some specific event types in order to understand what is happening inside the cluster itself. For the purpose of this thesis, as it will be described later on, we don't need to listen to the whole set of Kubernetes events, but simply to a small subset of them.

To summarize, for this work, the Kubernetes event represents the object to be used to understand what changes took place inside the cluster, in order to react to them and communicate the changes to the Verefoo tool. In particular, the Agent Master service will connect to the local Kubernetes cluster and listen to the events generated by the Pods inside the cluster.

For this initial step, the Agent Master will only manage the following event types coming from the Pod resources:

- ADDED
- DELETED

by this way, it will be aware of the pods creation and destruction and can

communicate this knowledge to the Verefoo graph in order to change its topology accordingly to the Kubernetes cluster life-cycle.

3.2.2 Kafka Broker

The Kafka Broker represents the contact point between the Agent Slave and the Master: these services don't have to know each other but simply the topic on which to produce/consume. By creating two different endpoints, there is also the possibility to run the services on different machines in order to distribute the services on different machines.

Kafka

As we know, Apache Kafka is a publish-subscribe system [18] and a based durable messaging system. A messaging system it is an entity/component that will send messages between servers, applications or processes.

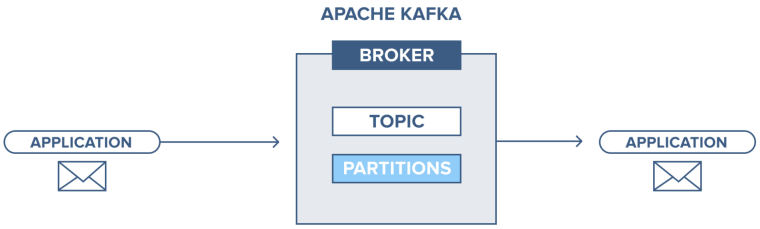

Figure 3.4: Publish-Subscribe: high level broker architecture

Apache Kafka is a distributed system, topic based: the topic is the place in which the client applications will process and reprocess the records (sometimes the topic is used to group the same types of messages all together).

Edge applications (producers and consumers) will connect to thy system and communicate by publishing/consuming new records transferred through the topic. One topic-record can collect and transport any kind of information: for example if a website is producing some specific events (maybe, happened on a page) this event can be transferred through Kafka, and later trigger another event etc. Any applications can connect (through some specific clients) to the broker and consume/re-consume records from a topic. The data sent will be retained until a specified period has passed by.

The Kafka mechanism is able to use different serializer/deserializer to work with many different records (in any format). A record is composed by four different

attributes: the value attribute is mandatory and the key, timestamp and headers attributes are optional. The value can be anything is needed by the infrastructure (a simple string, a complex object like XML or JSON etc.). One of the strongest features of Kafka is that it doesn't rely on a specific object type.

At the end, the Kafka broker component will be composed by two main services:

- **Broker**: is the main core component. It will handle all the requests from client components (producer, consumer, and metadata) and retains the data replicated inside the Kafka topic. Kafka is able to generate a cluster of brokers distributed in cloud.
- **Zookeeper**: is the component that keeps the cluster's state (brokers, topics, users).

For this thesis work, both the Zookeeper and Kafka Broker will run on the same machine for simplicity.

Kafka Broker

Kafka was born as a distributed system [18]: we are able to instantiate many different servers, each one running Kafka, as Kafka broker. We can also run only one single Kafka broker, but this configuration doesn't give to us all the benefits that Kafka could give to us with a cluster configuration, such as the data replication.

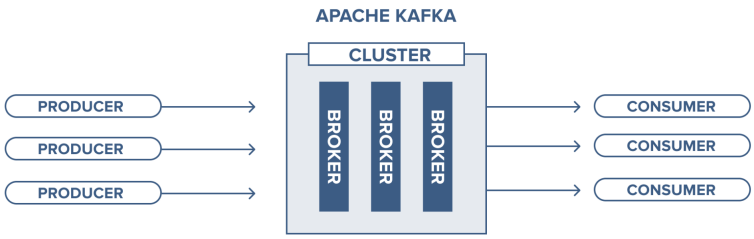

Figure 3.5: Publish-Subscribe: high level broker cluster architecture

The brokers management is performed by a Zookeeper server. As the Kafka Broker, there can be many different zookeepers running inside the distributed system.

For this thesis work, the data replication functionality wasn't needed. Inside the designed architecture we used only one Broker (with the respective zookeeper server).

Kafka Topic

The Topic is the category record name to which all the messages are published inside the broker. The topic is a mandatory information because is required to publish some specific messages: it represents the unity of storage from which consume all the existing messages. The message will live inside the topic until the retention time has passed by.

Kafka will manage the events and messages inside the log files: these files let the consumers to read messages in a specific position (offset). For this reason, the consumer is the component responsible to advance/set the offset on a specific position. By controlling the record position, the consumer is able to consume messages in any order and, potentially, is able to process over and over some previous messages.

Once the topic is created, it must be defined the number of partitions to be assigned to it [18]: the partition represents a sub-topic in which the messages published will maintain their order and are identified by an unique offset. The number of multiple consumers depends on the number of partitions the topic is composed by: we can have, at most, as much consumers as partitions for a specific topic.

In Kafka, the partition is the core of the redundancy mechanism: the redundant unit is called replica. Each partition can have one or more replicas: this means that each partitions can collect messages that are duplicated over the Kafka brokers instantiated inside the cluster. Each partition is leaded by a server acting as a Master and all the other servers will act as followers. The master replica will manage all the read and write operations for that partition. The followers servers will simply replicate the operations performed by the master: this mechanism is important to guarantee a fault-tolerance behaviour because, if the leader server will shut down or fail, one of the other servers will become the new master. When some producers will create and publish a new message it will be processed and managed by the leader server. The leader will put the message to its commit log file and update the offset. The Kafka system will expose the record to a consumer only after it was committed.

The producer, before sending a new message, has to know on which partition it must write to: it cannot be decided by the broker. The producer can choose a specific key to be attached to the record. All the messages with the same key will be committed to the same partition. By default, the producer will calculate the hash of the key in order to understand on which partition it will write to (some other mechanism are possible by overwriting the default behaviour). The producer must always write to the partition leader. In order to do that, before sending a

specific record, the producer must request also the metadata about the cluster: the metadata will contain information regarding the server leader for each partitions. Commonly, each publisher will produce records with no key or with the same key: this will result on an unbalanced topic because all the messages will end up in the same partition.

For this thesis work, it was configured only one partition per topic and we simply managed messages with no key (we didn't need to balance the load between the consumers or to differentiate the message type).

Kafka Groups

Kafka let two types of consumers to be created:

- **low-level consumer**: these consumers must specify topic, partition and offset from which read records. This mechanism can be difficult to be used and let the consumer to read a record no more than once. For this reason, Kafka added another (easiest) way of consuming messages.
- **high-level consumer**: it is known as consumer group and it is a collection/-group of consumers. The Kafka group is deployed by setting the property "group.id" and assigning it to a consumer. Many consumers can share the same group (by this way the consumer will join a specific group and load the messages consumption with the other consumers).

By configuring a consumer on a specific topic, it will commit the group topic after having consumed the message: all the consumers, that belong to a specific group, will commit on the same offset.

Each time a new consumer is added or removed from an existing group the record consumption is re-balanced inside the group: all the consumers are stopped and the broker will assign again the topic partitions to each consumer.

As mentioned before, the consumers will pull messages from one or more topic partitions: inside a group, different consumers will belong to different partitions. Kafka is able to support many different consumers: if the consumers belong to the same group, they can work in parallel on the same topic because they will read different messages from different partitions (the topic's partition will be distributed among the group's consumers). This mechanism will increase drastically the overall throughput. Of course we could have, at most, as many consumers as the number of partitions inside the topic: if the group is composed by too much consumers, some of them won't receive messages from the partitions (Kafka isn't able to assign

any partitions on some consumers because they were already assigned).

Kafka, because of its design, will never push the messages to the consumers: this will let the system to follow a "backpressure" [19] strategy, by equalizing and balancing the system from message overload. By this way, all the consumers will ask for data only when they are ready to manage them. The messages that are not consumed already, will be retained inside Kafka until a consumer will consume them.

For this thesis work, each Kafka Consumer (Agent Slave and Security controller) will belong to a different Kafka Group and consume messages from totally different topics.

Apache Kafka Library

In order to dequeue/enqueue messages from/into the Kafka Topic, it will be used the official Java Client offered by the Apache Software Foundation.

The Apache Software group collects more than 2000 repositories, thousands of collaborators and its Kafka Client Repository is continuously updated. In summary, the Apache Kafka project will be used to send and receive all the events (generated from the Kubernetes Cluster) through the Kafka topics, thanks to the Publisher and Subscriber micro-services developed inside the architecture.

Apache Kafka Java Consumer

The Java consumer is constructed with a standard Properties file [20].

```
Properties config = new Properties();
config.put("client.id", InetAddress.getLocalHost().getHostName());
config.put("group.id", "foo");
config.put("bootstrap.servers", "host1:9092,host2:9092");
new KafkaConsumer<K, V>(config);
```

The client-consumer API will focus on the poll() method: it represents the core of the consumer business logic and it is used to consume records from a specific topic. The topic that must be used, to fetch the data from, is specified in the subscribe() method. Typically, after calling the subscribe method, the poll method will be invoked inside a loop.

In the consumer example showed below, the loop is performed inside a Runnable class, in order to be executed by a specific thread.

```
public abstract class BasicConsumeLoop implements Runnable {
  private final KafkaConsumer<K, V> consumer;
  private final List<String> topics;
  private final AtomicBoolean shutdown;
  private final CountDownLatch shutdownLatch;

  public BasicConsumeLoop(Properties config, List<String> topics) {
    this.consumer = new KafkaConsumer<>(config);
    this.topics = topics;
    this.shutdown = new AtomicBoolean(false);
    this.shutdownLatch = new CountDownLatch(1);
  }

  public abstract void process(ConsumerRecord<K, V> record);

  public void run() {
    try {
      consumer.subscribe(topics);

      while (!shutdown.get()) {
        ConsumerRecords<K, V> records = consumer.poll(500);
        records.forEach(record -> process(record));
      }
    } finally {
      consumer.close();
      shutdownLatch.countDown();
    }
  }

  public void shutdown() throws InterruptedException {
    shutdown.set(true);
    shutdownLatch.await();
  }
}
```

Apache Kafka Java Producer

In the same way as the Kafka consumer, the producer is constructed with a standard Properties file.

```
Properties config = new Properties();
```

```
config.put("client.id", InetAddress.getLocalHost().getHostName());
config.put("bootstrap.servers", "host1:9092,host2:9092");
config.put("acks", "all");
new KafkaProducer<K, V>(config);
```

If some configuration errors will be detected, a "KafkaException" will be raised from the KafkaProducer constructor. The Java Kafka Producer will include a method to publish records on a specific topic: the send() method. This API will return a future that can be processed and polled to retrieve the result from it.

```
final ProducerRecord<K, V> record = new ProducerRecord<>(topic, key,
    value);
Future<RecordMetadata> future = producer.send(record);
```

Usually some format checks are performed over the message that was consumed by Kafka, like in the example below shows what to do after invoking the send() method.

```
final ProducerRecord<K, V> record = new ProducerRecord<>(topic, key,
    value);
producer.send(record, new Callback() {
  public void onCompletion(RecordMetadata metadata, Exception e) {
    if (e != null)
      log.debug("Send failed for record {}", record, e);
  }
});
```

3.2.3 Agent Slave

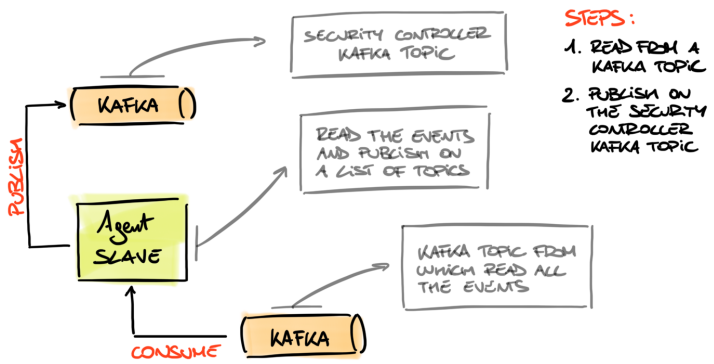

Figure 3.6: Topology changes: Agent Slave Architecture

The Agent Slave is a software component that will subscribe to one specific Kafka Topic: it will collect and process all the incoming events from the Kubernetes cluster.

Once it receives the events, it will redirect all of them on the known Topics related to the different Controllers: the idea is that of implementing different controllers that will perform different operations even if, for now, we implemented only one controller. Each controller will listen to a different Topic and receive some events from the Agent Slave, that will know the list of consumers to which send the messages.

What are the reasons for this adoption?

- It will be possible to add "extra controllers" to achieve different purposes (for now we'll simply use the Security Controller but, in the future, it could be useful to create different controllers such as the one responsible to save the data into a Database).
- With different Kafka Topics (one per each controller) it could be possible to consume all the events independently from other consumers/services: many different controllers can work in parallel without influencing their life-cycle.

The Slave will simply collect the events and send them to all the topics that were configured for the slave itself: we'll use a simple configuration file to let the slave know what the are active topics (and so the active controllers) to which redirect the consumed messages.

Finally, the designed architecture, potentially, is able to deploy in future other Agent Slaves: the only reason to increase the number of Agent Slaves is to improve the overall throughput. This can be easily achieved thanks to the use of the Kafka Topics and the Kafka Groups: by creating many different Slaves belonging to the same Kafka group, and consuming from the same topic, they can manage different overcoming messages without consuming the messages that were already de-queued.

3.2.4 Security Controller - SeC

The controller is a micro-service that perform a specific role or business logic. The idea is that of using an abstract Class to let all the controllers inherit some common behaviours but, each of them, will perform a specific operation, like the Verefoo graph alignment.

Each controller will be linked to a specific Kafka Topic, to receive all the events it is interested in: the controller needs a different topic because, if a new one will be developed, it could be interested on a different portion of the events. In any case is fundamental to separate the controller topics to avoid conflicts in the messages consumption.

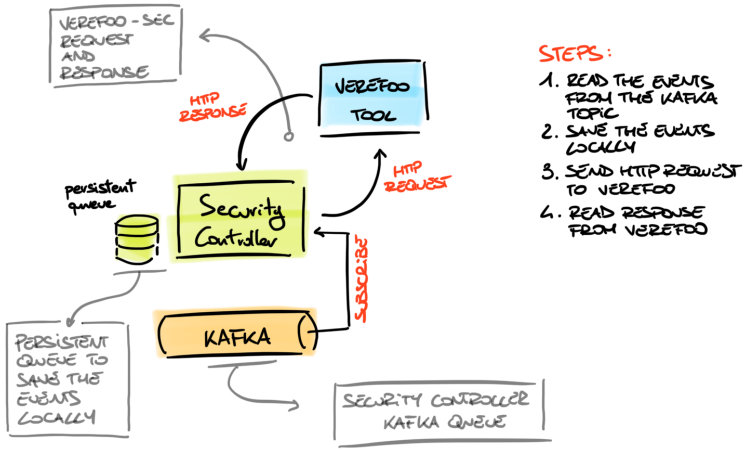

Figure 3.7: Topology changes: Security Controller Architecture

In particular, the Security Controller (SeC) represents the component that will modify the Verefoo Graph when an event is received from Kubernetes: the main idea is that of keeping the Verefoo graph topology and the Kubernetes cluster one aligned; in order to do that the Security Controller will receive only the events produced by the Kubernetes pods (the events generated by all the other resources

won't be significant for the purpose of this work).

The SeC will receive the following pod-event types:

- **ADDED**: check if the corresponding Verefoo Node exists yet, if not it will try to create it through the Verefoo REST API
- **DELETED**: check if the corresponding Verefoo Node was deleted yet, if not it will try to delete it through the Verefoo REST API

When the Security Controller receives a group of events, it must consume all them from the Kafka Topic:

- Once the event is consumed from the kafka topic, it isn't available anymore so, at the next restart, it won't be able to re-consume it another time (the kafka-topic offset increased after reading from the topic itself)
- If some errors will occur, and the SeC service will shut down, there is the possibility that some events could be lost after being read from the topic: by this way, at the restart, the events won't be used to align the Verefoo Graph; this is a big issue because the Kubernetes cluster and the Verefoo graph could not be aligned anymore coherently
- To solve the previous issue, a persistent queue will be initialized in order to locally save all the events that weren't sent correctly to the Verefoo Graph

At the beginning, the service must check if some events are inside the Persistent Queue and, eventually, process them: these messages represent all the events that weren't processed correctly before the application's termination. After processing all the events inside the local queue, the SeC will continue, as usual, by consuming all the events directly from the Kafka Topic.

Internal Persistence - BigQueue project

As was already mentioned, in order to avoid some problems and errors with the overall mechanism, each controller will instantiate locally a persistent queue.

We choose the BigQueue project [21] because of its lightweight and performances.

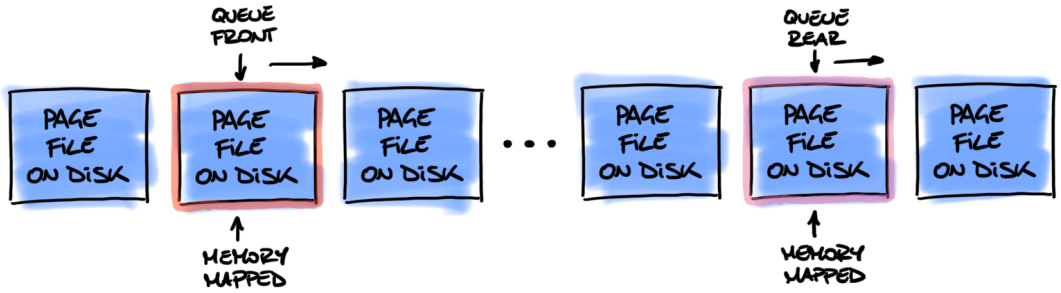

Figure 3.8: BigQueue library: Memory management

The project is characterized by the following features:

- is **Fast**: both the enqueue and dequeue operations are close to O(1) memory access
- is **Big**: the size of the queue is strictly dependent on the available space on the disk
- is **Persistent**: all data are saved on the physical disk. No data loss will occur if some crash will happen
- is **Reliable**: the operating system will proceed by saving the data, even if the process will shut down unexpectedly
- is **Realtime**: new messages will be available to all the consumers immediately
- is **Memory-efficient**: the queue will work with the paging and swapping memory mechanism
- is **Thread-safe**: it is designed to work with several consumers and producers
- is **Simple and Light-weight**

Here in the following, some of the BigQueue performance highlights:

- In a **concurrent scenario** (producing then consuming): the average throughput is 166M bytes/second.
- In **sequential scenario** (producing then consuming): the average throughput is 333M bytes/second.

By supposing an average message size of 1KB, the big queue can potentially produce and then consume 166.000 messages per second. The final throughput is limited by the disk IO bandwidth.

The main reason for its adoption is because of its blocking queue features:

- By **popping** from the queue, the service will be stopped if the queue is empty
- By **pushing** on the queue, the service will be stopped if the queue is full

We can easily understand that, in a concurrent scenario, the above features are pretty useful to implement a proper multi-thread solution.

3.3 Architecture limits

Before continuing with our discussion, as it was already anticipated, we have to underline that the previous solution represents the very beginning of the thesis work: for this reason some constraints, or details, weren't considered; some of them will be treated later on through the thesis work. Some of these constraints, once considered, will change also the overall application's architecture.

3.3.1 Node's Neighbours

Inside the Kubernetes cluster, there is the possibility to limit the network traffic between the pods, thanks to the Network Policies. This aspect, during the beginning of the thesis, wasn't considered: to start with a simple scenario we considered that all the pods inside the cluster (and all the nodes inside the Verefoo graph) were able to communicate to all the other services, without constraints (like in a complete graph).

In summary, when a pod is created inside the Kubernetes cluster it will be added, in the Verefoo Graph, as it is already connected to all the other nodes.

Remember that, in this first scenario, we were only interested to communicate the Kubernetes topology changes to the Verefoo graph, without considering the network aspect (that was considered later on).

3.3.2 Kubernetes's Containers

Each Kubernetes Pod could contain different Containers.

In this first scenario, we had considered only one container per pod, representing the running business logic inside the pod itself.

3.3.3 Managed Events

The Fabric8io Watcher can listen to the following event types [22]:

- **ADDED**
- **MODIFIED**
- **DELETED**
- **ERROR**

For the above architecture, were only managed the **ADDED** and **DELETED** event types.

The **MODIFIED** event could have been useful if it would let us understand if some Pod's parameters have changed in runtime, such as the Ip address and the Port. But, in general, you should not rely on the pod itself as Pods are ephemeral and replaceable.

What does it mean? An user gave the following response for a question opened on StackOverflow [23]:

"As the Pod is the smallest deployable object in Kubernetes and is easily replaceable, we should avoid direct interaction with a pod itself".

In fact, this is the reason why the interaction with the Pod should be demanded to a kubernetes Service, which select Pods based on their labels not on their IP so, the IP and Port related to a Pod, aren't fundamental for the cluster's knowledge.

Anyway, the Pod IP and port won't change as long as the pod is running, but there are no promises that your pod will stay running and there is no generic way in Kubernetes to get long-term static IP on a pod so, when it will restart, there is no guarantee that the pod will maintain the same IP address of the previous run (this isn't a problem because, after the pod's restart, we'll receive an ADDED event message with the new POD IP and PORT and we'll be informed of the new pod configuration).

As a final aspect, the Fabric8io library is limited because there is only the possibility to watch IP changes on endpoints and not pods itself.

In summary for our purpose, the Pod ip and port changes aren't useful for two main reasons:

- The service (which represents the endpoint of the cluster) interacts with the pods based on their labels and not on their IP, so it doesn't matter if the IP changes because the service should be able to interact with the correct POD, even if its IP will change
- the POD ip and port won't change while the pod is running

and so the ADDED and DELETED events contain enough information to reach the purpose of this work, and to communicate the topology changes inside the Kubernetes cluster.

3.3.4 Pod Events

For this thesis work, were only managed the events generated by the Kubernetes pods.

The Fabric8io project let the user to listen to many different kind of events: for example, the ones generated by the services.

As we know from the Kubernetes API, services can be exposed in one of the three forms [24]: internal, external and load balanced.

- **Internal**: There are some services that don't need an interaction with the outside (for example cache endpoints or databases). These kind of services won't be exposed to the outside because they will be consumed only from the inside of the cluster. These services, once started, will obtain an IP address that's visible only from the inside. Kubernetes let the developer to obscure the service by exposing an endpoint that is still available only inside the cluster. This component will usually be used to hide the pods inside the cluster from the public.
- **External**: Services that need an interaction from the outside (for example some web services) can be exposed through this component. By this way, the service can be reached through the endpoint defined inside the component itself.
- **Load Balanced**: There are some scenarios in which cloud providers offers a load balancer. This Kubernetes service can be wired to a third-party load balancer to balance and distribute the incoming traffic.

As we can understand from the previous points, the service component is totally independent from the pod one. For the previous reason, the service events won't be treated because they won't increase our knowledge about the pod's status.

In summary, only the pod events were the one listened by the Agent Master to communicate the topology changes happened inside the Kubernetes cluster.

3.3.5 Incomplete Flow

Considering the above architecture, the overall mechanism isn't complete: at this point, there isn't a way to deploy the Verefoo networking rules on the Kubernetes Cluster.

We can align the Verefoo Graph thanks to the Kubernetes events, but the other way around wasn't possible with this architecture: this feature will be covered and developed later on, as it is showed in the next chapters sections.

Chapter 4

Network changes: reflect Verefoo changes inside Kubernetes

This chapter represents the thesis research core: from the previous chapter, we were able to reflect only the Kubernetes changes inside Verefoo. Now we must take a step forward to understand how to reflect also the Network rules produced by Verefoo into Kubernetes and close the loop. In order to do that, inside this chapter, were analyzed many different technologies to better understand if it was possible, through them, to reflect the **Neighbour** and **Forward** rules, produced by Verefoo, inside Kubernetes. Because of the magnitude of the material contained in this chapter, we decided to describe here only the technologies that were analyzed to reflect the network rules inside the Kubernetes cluster and then, in the two next chapters, understand if it is possible, or not, to extend the application, developed until now, to exploit these technologies (with the aim of closing the loop and reflecting the network rules in the cluster).

What do we mean for Network changes?

Verefoo, by producing the whole graph, will list the neighbours for each node: by this way the output won't be a complete graph (in which each node is able to talk to all the others) but each node has the duty to talk only to its neighbourhood. So, we need a mechanism to force some kind of network rules on Kubernetes, in order to reflect these constraints also inside the cluster. The collection of these network rules is called **Neighbour Mechanism**.

On the other side, Verefoo is also able to produce some rules that must be

followed by the packets to reach the destination: it is a sort of path, between the source and the destination of the packet, inside which there is a sequence of nodes through which the traffic must be processed. So, the Kubernetes cluster must be extended with some mechanism that can forward the packets through a list of pods, in order to emulate a chain of services that must be invoked to process, sequentially, the traffic itself. The collection of these network rules is called **Forwarding Mechanism**.

To summarize:

- this chapter will face the problem from a theoretical point of view: illustrate all the technologies that were analyzed to reflect the Neighbour and Forwarding mechanism inside Verefoo, and if it is possible or not to reflect these network mechanisms inside the cluster
- the next two chapters (**Inter-Cluster use case** and **Intra-Cluster use case**) will focus on some practical aspects: after having understood which technologies we can use (and which rules can be reflected on Kubernetes), we have to know in which environment we are able to work with (like the inter-cluster or intra-cluster scenarios) and how the application's architecture, developed until now, must be extended apply the network rules on the cluster through these technologies

4.1 Neighbour Mechanism

After the deployment of all the nodes (from Verefoo into Kubernetes) we need a mechanism inside the cluster to manage the traffic between the deployed pods. This information will be obtained by the Verefoo graph that will produce the network rules needed by the graph.

This mechanism will deny and allow some traffic routes, inside the cluster, in order to make visible to each pod a list of different pods: by default, Kubernetes will let each pod communicate with each other; this mechanism will create a totally connected graph and also some network-security problems: if someone finally goes inside a pod, will have the possibility to send traffic to each pod inside the cluster. For this reason, we want to limit the accepted incoming traffic on each pod, in order to respect the neighbours rules generated by the Verefoo tool and increase the overall cluster security.

First of all, we have to understand how we can programmatically set the pod's routing rules inside the cluster: what technologies can be adopted? And how do

they work?

Inside this chapter, we analysed two totally different technologies: CNI and Service Mesh. We'll go into the detail of these two technologies to understand how they will be applied into our cluster and what changes they will perform inside it. As we'll see in the following chapters, these two technologies will impact the pod traffic routes with two totally different constructs and behaviours so, the purpose of this section is compare these technologies to better understand their differences.

In the beginning, we'll go into the details of the CNI, and then we'll focus on the Service Mesh technology.

4.1.1 CNI

It is a group of API introduced to provide and enforce the network connectivity between the cluster's pods. Except some particular cases, only one CNI can be enforced inside the cluster: once the CNI will be applied inside the cluster, the network configuration will be established for all the elements that compose the cluster. There is a group of CNI APIs grouped in a file called CNI Plugin that are fundamental for the correct implementation of the CNI itself.

CNI providers [25] usually implements their fabric products choosing between two alternatives:

- encapsulated network model: like a Virtual Extensible Lan
- unencapsulated network model: like Border Gateway Protocol

For this thesis work we'll make an high level overview of the unencapsulated model because is the one followed by the Calico CNI technology, the one tested and studied in this paper, as shown in the following chapters.

Unencapsulated Network

The Unencapsulated Network model [25] is built over a Level-3 Network protocol skeleton to manage the traffic between the cluster's containers. The model won't isolate the network at Level-2: this will make the Kubernetes workers to manage any kind of routing distribution that is needed. Finally, this model will use the BGP network protocol to distribute the routing informations between the Kubernetes components.

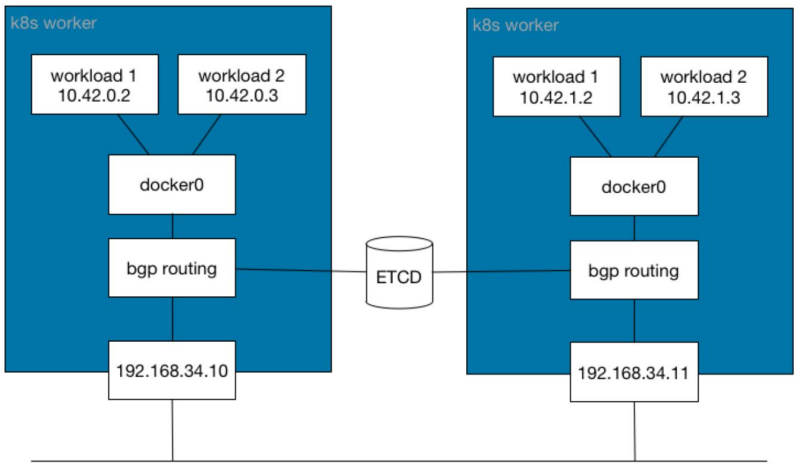

Figure 4.1: Neighbour Mechanism: Unencapsulated Network schema

To summarize, this model will generate a sort of "distributed" router between all the Kubernetes components: this router will provide informations regarding the routing path (in order to reach/filter all the internal pods).

When we need more performances (less sensibility to network latency or network changes phases) or when we prefer a routed L3 network model, this model will be the best choice. Some of the network providers that follows this model are, for example, Calico [26] and Romana [27] CNI.

Calico CNI

Figure 4.2: Neighbour Mechanism: Calico CNI Logo

Calico is one of the CNI technologies [28] which aim is that of enabling complex networking inside the clusters spread all over the cloud. Calico, as mentioned, is built by following the unencapsulated IP network policy engine. If necessary, Calico provides an IP-in-IP encapsulation modality.

This technology will offer some rules and policies to enforce network constraints (such as isolation), allowing the developer to secure the cluster from unexpected

traffic flows. As we'll see later on in this paper, the CNIs will offer some components to enforce both egress and ingress rules.

As mentioned inside the paper titled "Integrating VNF Service Chains in Kubernetes Cluster" inside the chapter called "Analysis of different CNIs" [29], when performances are needed inside our cluster, Calico is the best choice. This aspect is obtained thanks to the fact that Calico implements a proprietary network policy solution, that substitutes the Kubernetes native one.

To provide network connectivity to pods within the same node, Calico CNI installs three fundamental elements in each node of the cluster [30]:

- **Calico node**: it is the agent that will be instantiated inside the node. It will contains and manage some processes like Bird and Felix.
- **Felix**: it will manage programs routes and ACLs, and everything required on each host to provide connectivity to all the other hosts. It will run as an agent daemon (BGP daemon process that will exchange route informations).
- **Bird**: it will get the routes from the Felix daemon and will distribute them to all the other BGP peers inside the cluster.

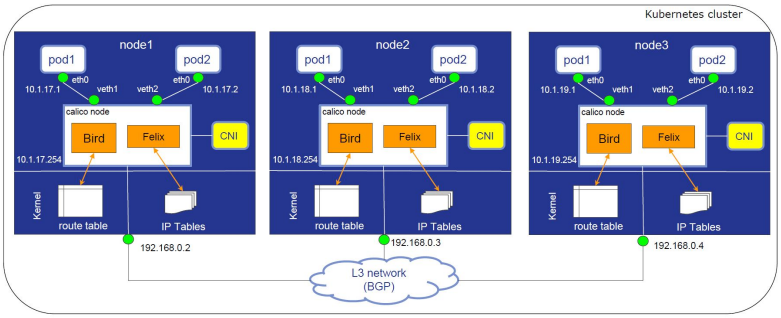

Figure 4.3: Neighbour Mechanism: Calico CNI

To sum up, the Bird component, by exchanging messages with the other Bird components in the other nodes, it will know exactly where the traffic must be routed and sent (also outside the node itself). Thanks to the CNI apis and expressivity, it is possible to create and configure rules describing how the pods should communicate and send/accept/deny incoming/outgoing traffic.

For a better understanding, Calico (and all the other CNIs) is a simple technology that let the cluster to apply some filtering rules inside the cluster: by enabling

the CNI policy and specifying the technology (for example, Calico), we are enabling the cluster to accept all the future network rules.

The question could be the following one: How Calico or better, the CNI engine, can help us creating the pod's neighbour rules, produced by Verefoo, inside the Kubernetes cluster?

Kubernetes and Calico provide network policy APIs to help you secure and constraint the workloads with the Network Policy construct. So, in order to answer to the previous question, we have to enter into the detail of the Network Policy object.

Network Policy

The Network Policy is an API [31] natively offered by the Kubernetes group: after enabling the CNI policy, which is just a technology, we'll be able to apply the network policies inside the cluster, otherwise they will not be considered. Please note that, after applying the CNI inside our cluster, we can use both the Network Policies provided by the Kubernetes and the Calico group.

In the past, the networking and the security was provided by designing a physical topology through the installation and use of network devices (like routers, firewall, switches, etc.). Historically the physical network topology defined the security constraints and boundaries (for example, which traffic is allowed or not) of the overall network. With the coming of the virtualization, the physical devices and network constructs were "virtualized" inside the cloud: by "reflecting" the techniques used in the past, and using the new virtualized paradigm, the user is able to provide the network security. In any case, by adding a new application, or network security logic, inside the environment often required additional changes and updates to the overall design.

In contrast to these approaches, the Kubernetes model used for networking will define a sort of flat architecture: every pod inside the cluster can communicate with all the others by using a specific pod IP address. This mechanism will simplify drastically the network design phase because there is no dependency on the network design itself. In summary, this approach let:

- new components to be added with no efforts and
- new workloads to be scheduled in a dynamic way

In Kubernetes, instead of defining network topology boundaries to enforce some network security constraints, we'll use the NetworkPolicy component to define the

network rules inside the cluster: this component is totally abstracted from the network topology of the cluster itself, in fact it will use selectors, like labels, to recognize and define which pod can communicate to which one (instead of using IP addresses or ranges).

In the end, inside Kuberentes, we'll largely use the NetworkPolicy resource to describe the network rules, that must be followed, inside the cluster.

The main features of this component are:

- The policies will use the namespace scope (inside each scope the user can create resources like pods). By using the namespace, the policy can be applied to a group of different resources.
- The network policies will be applied to the pods by using some specific labels. The label will be chosen during the resource creation (for example, it could be defined inside the YAML file).
- These rules will specify which traffic is allowed or not (ingress and egress). It is possible to use namespaces, labels or CIDRs to express a specific endpoint.
- The Network Policies can specify both the port ranges and the protocols that must be respected.

Kubernetes, by itself, won't enforce any network policy inside the cluster (the plugin itself must be installed separately from the k8s initial setup). The enforcement of the network policies is delegated to a specific Network Plugin (like the Calico CNI) so, in order to apply the NetworkPolicy component, we need to install a specific CNI inside the cluster. All the CNIs must implement the APIs or elements by following the Kuberentes guidelines (in order to be compatible with the Kubernetes APIs and operations) and most of them implement only a portion of the Kubernetes APIs (not all of them).

Of course, each CNIs can implement their own resources and APIs: for example Calico, in addition to the Kubernetes Network Policy, supports its own resources like the GlobalNetworkPolicy which provide additional behaviours and features to the one carried by the native Network Policy. These features include support for:

- policies priority and ordering mechanism
- logs specification inside the rules
- there are more flexible matching mechanism for applying policies (for example the Kubernetes ServiceAccounts can be matched) and (by using the Istio

service mesh with Calico) there is the possibility to define layer 5-7 matching criteria such as gRPC or HTTP URLs.

The main difference between the Kuberentes and the Calico network policy is that the Kubernetes one can be only applied to pods while the Calico Network Policy can be applied to different types of endpoints (VM, Pods, Host interfaces etc.).

Network Policy Schema

As described in the Calico web site [32]: *"A network policy resource represents an ordered set of rules which are applied to a collection of endpoints that match a label selector".*

As already mentioned, the Kubernetes network policy is a resource that can be exploited to enforce network isolation or rules in the case of a "soft" multi-tenancy environment. We cannot satisfy the "hard" multi-tenant environments because we don't have the guarantee that nobody can modify, delete or add new network rules: for this reason, the Calico CNI, introduces a new resource called Global Network Policy (GNP) which remedies to these problems because the tenants can't edit or delete these objects because can be managed and modified only by the administrators that created them. By this way, we avoid any kind of malicious or accidental modification action performed over the cluster networking. Inside this work of thesis, the GNP won't be used because, for a first approach, we didn't need to work with an hard multi-tenant system.

In general, the Network Policy is a name-space dependent resource: can only be applied to the endpoint resources inside the name-space specified in the configuration of the Network Policy itself (two pods will share the same name-space if the name-space value is the same on both the resources).

Here there is an example of a Network Policy object taken from the Kubernetes official website [33]:

```
---
apiVersion : networking . k8s . io /v1
kind : NetworkPolicy
metadata :
    name : Matthew
    namespace : Alice
spec :
```

```
    podSelector : {}
    Ingress :
        - to:
            - namespaceSelector :
                matchLabels :
                    name : Alice
            - namespaceSelector :
                matchLabels :
                    name : Bob
    policyTypes :
    - Ingress
---
```

```
podSelector: a field that will tell the cluster on which pod the rule
    must be applied (if no pods will be specified, like in the previous
    example, the rule will be applied to all the pods inside the
    namespace Alice)
policyTypes: define the network policy rule (Ingress or Egress)
```

Before continuing with the discussion of these studies, we made an high level overview of other two CNIs like Flannel and Canal to make some comparison with Calico and understand why we finally have choosen Calico as CNI.

Flannel CNI overview

Flannel is one of the most popular CNI plugin available on the market. It also represents one of the most advanced and mature examples of CNI products [34] for the container orchestration and management. It is specialized on allowing a better networking between containers and hosts.

If we compare Flannel to other solutions, it is relatively easier to install and configure. It is deployed and distributed as a single binary file called flanneld and can be installed (like other CNI solutions) by default inside the Kubernetes cluster. The Kubernetes etcd can be used by Flannel to store some information: this avoid Flannel to provision a dedicated data store to manage this data.

Flannel works over a layer IPv4 network: this network is created by spanning across every node inside the cluster. Inside this network, will be provided a sub-net to each node to allocate their IP addresses internally. Once the pods will be allocated inside the cluster, the Docker bridge interface will allocate a new address

for each container: pods inside the same host will communicate through the Docker bridge; pods inside different hosts will encapsulate the traffic inside UDP packets (by using the flanneld) and send them to destination.

Flannel has different types of backends that can be used for both encapsulation and routing. If we compare VXLAN to the other options, it is recommended as default approach because it will offers both good performance and it will need less manual intervention.

To summarize, Flannel will offer a simpler networking model and, if we only need a basic approach and technology, it is suitable in the most of the available environments. In general, if the user started using these technologies, to enforce networking inside the cluster, Flannel will be the best choice but, when the user will need something more it is better to move to other solutions like Calico, because they will offer a complete API.

Canal CNI overview

The Canal project, initially, aimed to integrate the network layer provided by the Flannel CNI with the network policies defined in the Calico CNI [34]. So Canal, even if it isn't a full combination of Flannel and Calico, tried to combine some of the features provided by these two CNIs. The Canal's benefits are also at the intersection of this combination:

- The Networking layer is as simple as the one provided by the Flannel CNI (without the need of addition manual configuration)
- The Networking policies are layered on top of the Calico's base network (with all the advantages and power of the Calico API)

Canal, as the Flannel CNI, is a good way to start experimenting and gaining experience with the cluster networking policies, before obtaining enough expertise in order to focus on a more complex CNI technology.

To summarize, Canal is a good compromise if the user like the usability and simplicity of the Flannel CNI and the expressivity and capabilities taken by the Calico CNI.

CNI - Final Considerations

To summarize, the Calico CNI [34] should be taken in consideration when:

- the user is searching for performances and flexible features

- the user wants to apply all of its requirements on the environment it is working with.
- the user wants to fully control the networking (instead of configuring it only once and forgetting)
- the user wants to open (also later on) a commercial support for the product
- the cluster should be integrated with service mesh like Istio (this is the most relevant thing that was considered before chosing the Calico CNI)
- there is the need of powerful rules that describe how the pods inside the cluster should accept or deny some traffic (this aspect is fundamental to improve the overall network security)

Because of its network-controlling capabilities and expressiveness, Calico represents the best choice for our thesis work.

4.1.2 Service Mesh

The Service Mesh [35] is an infrastructure layer designed to stand over the cluster architecture and to guarantee high network performances and communication between the application services inside the cluster itself. According to the CEO of Buoyant William Morgan [36]:

"A service mesh is a dedicated infrastructure layer for making service-to-service communication safe, fast, and reliable. If you're building a cloud native application, you need a service mesh!".

Typically, a service mesh will be composed by the control and data plane: it will provide some programming interfaces to interact with its infrastructure and enforce some particular constraints or behaviour. The service mesh was also designed to ensure that the communication between the containers and the application services will be secure and fast, by providing services and capabilities like:

- service discovery
- encryption
- load balancing and circuit breaker pattern
- traceability
- authentication

- authorization

Usually, the service mesh is implemented by adopting and introducing a new component called **sidecar**. The sidecar is a proxy instance that will be instantiated for each service and stays up and running. Its aim is to manage anything that can be "abstracted" from the service's business logic:

- enforce security mechanisms
- monitoring
- handle inter-services communication

The sidecar's usage is one of the most important features carried by this infrastructure: both designers and developers, during the development, don't have to manage inter-services communication rather than monitor the system. By this way, the business logic won't be affected by these "horizontal" aspects.

One of the most known service mesh architecture is Istio [37], and it was designed to support, natively, the Kubernetes orchestator framework.

Here in the following, some of the functionalities and components provided by the Istio service mesh:

- **Sidecar proxy**: usually it is a single container that will run in parallel with the pod that contains the business logic. Its aim is to proxy or route the container's traffic that runs with it. The sidecar will communicate to all the other proxies in order to exchange some routing information (all the sidecars will be managed by the cluster orchestrator). The most of the service mesh technologies, like Istio, will use an implementation of the sidecar proxy to manage and intercept all the incoming and outgoing traffic to the Pod.
- **Service discovery**: the service discovery is the mechanism needed by all the instances that want to interact with a specific service. During the discovery, the service performs a DNS lookup (the container orchestrator will keep a list of containers that can receive and manage the traffic) to find an available and healthy service's instance that can manage the traffic.
- **Load balancing**: the load balancing is already performed by Kubernetes. The main difference with the mechanism provided by the service mesh is that the native one works over the transport level (layer 4) and the load balancing provided by the mesh will work over the application level (layer 7). By working on a higher layer, the service mesh can implement more sophisticated and powerful traffic algorithms to manage the traffic in a richer way: for example, there is the possibility to orchestrate canary deployments via API.

- **Encryption**: it represents one of the greatest advantages took by the service mesh. By performing the encryption outside of the pod's business logic, the developer doesn't have to manage it inside the code: this aspect also improve the performances by prioritizing and reusing existing connections instead of creating new ones. One of the most used encryption mechanism is the mutual TLS that is largely supported by service mesh like Istio.
- **Authentication and authorization**: the mesh can authenticate and authorize all the requests that are made both from outside or within the cluster: only the validated requests will be accepted or sent.
- **Circuit breaker pattern**: it is a mechanism to support the debug and recovery from errors. With this pattern we can isolate unhealthy instances and, eventually, bring them back once recovered from error.

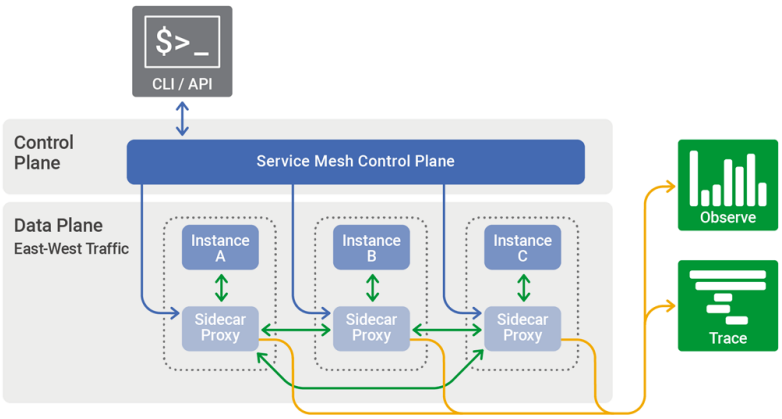

Figure 4.4: Neighbour Mechanism: Service Mesh generic topology

Typically, a service mesh is split into two main planes:

- **Data Plane**: the portion of the service mesh that manages the traffic between the container/pod instances inside the cluster. It is represented by the proxies inside the cluster. Usually, the service mesh will instantiate a proxy per service so that all its network traffic will be intercepted and managed by the proxy itself: by this way the pod/service/resource doesn't have to manage the traffic because the proxy will know exactly where to properly route all the packets. Of course, by using this approach, the proxies will need a mechanism (or better, an external component) to configure properly their behaviour: for example, each proxy will need to know exactly the authorizations/authentication constraints, the endpoints (the services to which route the traffic) and the policies that

they must follow to forward the traffic in a correct way. Accordingly to these aspects, the data plane will influence the overall traffic inside the cluster. The proxy component is the perfect place into which collect data and information: which service talks to whom, the overall time of the requests, if some packets goes wrong or if there is some problems in the communication. All these information will be collected inside some specific metrics and then will be used by some tools to visualize them.

- **Control Plane**: it will generate and deploy the configuration that will control and coordinate the data plane. Usually it is connected to a GUI, a command line interface or an API to orchestrate the applications. The control plane is the interface that will manage the cluster network traffic and will provide the configuration for the correct traffic routing. To summarize all the traffic performed inside the cluster will be controlled and configured by the control plane which must provide the correct policies and configurations to the proxies.

But before continuing we have to understand, when do someone really need a service mesh [38]?

First of all, one of the greatest advantage of deploying a service mesh inside Kubernetes (for example, by choosing Istio) is that the application won't need to implement and apply features like security, observability, traffic split, reliability mechanism etc. inside the application itself: this is great for the developing teams because the developer will only focus on its responsibilities without writing features that don't pertain to the application's business logic.

In any case, the application won't be aware of the service mesh: the application logic won't be affected from a "code" point of view, even if the service mesh will apply some new horizontal features.

Service Mesh open source projects

As Sachin Manpathak wrote in the article titled *"Kubernetes Service Mesh: A Comparison of Istio, Linkerd, and Consul"* [39]:

"There are three leading contenders in the Kubernetes ecosystem for Service Mesh" and continued "Each solution has its own benefits and downfalls, but using any of them will put your DevOps teams in a better position to thrive as they develop and maintain more and more microservices".

- **Consul**: Consul [40] is part of the HashiCorp's suite of products [41]. Initially it started as a system to manage resources running on Nomad [42] and later

added the support to manage other platforms like Kubernetes. It will work by installing an agent DaemonSet that will communicate to the Envoy sidecar proxy to handle the routing mechanism and forward the traffic.

- **Linkerd**: Linkerd [43] is a service mesh technology included in the Cloud Native Foundation consortium [44]. It is probably the second most used service mesh on Kubernetes. After some refactoring performed later on, the architecture reflect some of the Istio's architectural choices. If we compare this service mesh to all the others, it results to be easier to use because of its low complexity architecture.
- **Istio**: Istio [37] is a solution natively compatible with Kubernetes. It represents the service mesh technology of choice of many companies like IBM, Microsoft or Google. The Istio architecture, as we will see later on in this paper, is composed by the data plane and the control plane. The Data plane is composed by all the sidecar proxies that will cache information and metrics that will be used by some tools to visualize them. The control plane is composed by some Istio pods that will run inside the Kubernetes cluster.

Istio service mesh: architecture

Inside this chapter we will focus better in detail on the Istio architecture: other technologies won't be covered in detail because, for this thesis work, we only used the Istio service mesh.

As mentioned, service mesh is a pattern or a paradigm and Istio is one of its implementations. Inside the Istio architecture the proxies will be implemented by exploiting the Envoy project: it is a independent open source project that Istio, as well as many other service mesh implementations, uses.

The control plane component is managed the Istiod element, which manages and injects the envoy proxies in each of the microservices pods.

In the earlier versions of Istio (until v1.5) istio control plane was a bundle of multiple components (citadel, mixer, galley and many others components) so we had multiple pods when we deployed Istio in the cluster. However in version 1.5 all these separate components where combined into the Istiod component to make easier, for the operators, to configure and operate Istio.

Istio architecture is comprised of:

- **Control Plane**: the control plane is composed by the Istiod component that will manage the data plane. It will control, manages and configures all

the proxies to route the traffic. Additionally, the control plane configures the mechanisms to enforce traffic policies and collect metrics for telemetry. The pilot is a component that interprets and converts the high level routing rules, configured to control the traffic behaviour, into some Envoy specific configurations; finally it will propagate these rules to the running sidecars inside the cluster. The Citadel component also provides a strong end user and service to service authentication with credential management so that the user doesn't have to configure and manage it manually. The Galley component will receive the user configurations from the Kubernetes cluster. The configuration can be obtained in the form of a YAML file.

- **Data Plane**: is composed by all the Envoy proxies inside the cluster. These proxies will control all the network traffic between the micro-services.

As mentioned, Istio will use the envoy proxy with a sidecar pattern and what happens is that all the requests, that comes to the container, will be catch by the proxy: this component is fundamental because it knows what traffic should flow through, what is the data that needs to be generated out of that traffic; it is also involved in the dynamic service discovery or load balancing mechanisms and helps the developer to do and establish TLS secure communication (by this way the application doesn't need to do it from the inside).

The proxy is deployed as a sidecar container inside the Pod configuration: remember that a Pod is a collection of containers so, one container will represents the main application (the pure business logic) and the sidecar would be another docker container which runs the Envoy proxy to observe traffic and applies policies to them.

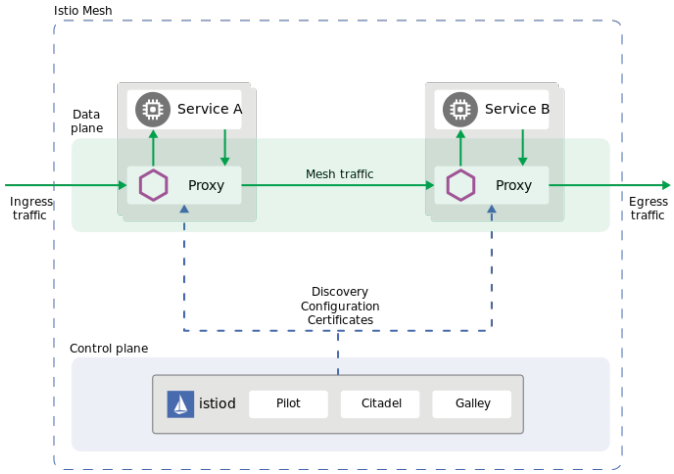

Figure 4.5: Neighbour Mechanism: Istio Architecture

To configure the service mesh, the user simply needs to define and deploy some customized Kubernetes objects: by applying the YAML files through the Kubernetes API, we are able to interact with the Istio control plane, in the same way as we interact with the Kubernetes cluster. The Istio resource objects will provide many different functionalities that will be covered later on in this paper.

Istio service mesh: traffic management

Kubenetes traffic management is mainly infrastructure based: it gives you the control over how traffic reaches all the pods. Rather than specifying which pods should the traffic reach to, Istio does it differently: Istio lets the user define configuration rules that allow final control over APIs and traffic between the services. As we can expect, it simplifies a lot the configuration of service level properties like circuit breakers, timeout or retries.

One of the most important component is the Pilot. It is essentially the core component of the traffic management: it manages and configures all the Envoy proxies and converts high level routing rules into envoy specific configurations. This behaviour happens in the backend, so the user doesn't have to do anything by himself: the user simply has to provide APIs to interact with the control plane.

By looking at the basic traffic configuration model offered by the Istio tool, we can use many different APIs like:

- **Virtual Service**: essentially, it is a set of rules that control how a request for a service is routed within the mesh.
- **Destination Rule**: it talks about how, after a virtual service routing has been done, a specified set of policies are applied.
- **Service Entry**: any request that needs to be sent outside the mesh we control it via this component.
- **Gateway**: this resource allows the user to set up both HTTP or TCP load balaning mechanisms, allowing the user to enable traffic from outside the cluster to reach an internal running application.

Istio Main functionalities

The core functionalities of Istio are divided into three main categories: security, traffic management and observability (metric collection). The following table was taken from the paper work done by Ronja Mara Josh titled "Managing Microservices

with a Service Mesh" and it will summarize the Istio's main features, considering the Istio's website, resources and articles [45] [46] [47].

Category	Overall Feature	Specific Functionalities
Traffic Management	Traffic routing (virtual services routing rules)	Header & path-based traffic splits (URI prefix...)
		Percentage-based traffic splits: canary rollout
		Fault injection
		Timeouts
		Retries
		Mirroring Traffic
	Traffic Routing (destination rules)	Load Balancing
		Circuit breaking: outlier detection or connection pool
	Gateways	Ingress traffic control
		Egress traffic control
Security	Authentication Policies	Peer authentication: Mutual Transport Layer Security (mTLS)
		Request authentication: JSON Web Token (JWT) validation
	Authorization Policies	Auth Rules: allow/deny sources if a specific condition will match
Observability	Metrics and Logs	External tool: Prometheus collects telemetry data. Latency, traffic ...
	Distributed Tracing	External tools: e.g. Zipkin, Jaeger, LightStep
	Dashboards	Visualising the mesh with external tools (e.g. Kiali, Grafana ...)

Figure 4.6: Neighbour Mechanism: Istio main functionalities

Authorization Policies

The Authorization Policy [48] is a special resource, available from the Istio API, that enables user access control on the traffic inside the mesh.

The Authorization policy let the user to define both deny and allow policies on the traffic. By this way there is the possibility do define which traffic can be admitted or not. If, for some reasons, both allow and deny policies are defined for a specific workload in the same time, there are some "priority" order rules that are followed to evaluate the workload access:

- If a deny policy will match the request, the request will be denied.
- If no allow policy was defined for the workload, the request will be allowed.
- If an allow policy will match the request, the request will be allowed.
- If none of the previous cases has happened, the request will be denied.

The Authorization Policy also supports some particular mechanism that not affect directly the workload mechanism: for example, it allows an audit mechanism to print and save some logs. By this way, we can trace all the operations and all the traffic that we receive from the inside/outside and which traffic will give error

or not. In order to properly audit a request, this one should be marked internally to understand if there is an audit policy that will match with the request. To support and fulfill the audit mechanism, we need to configure and install a specific plugin: if there is no plugin, the request can't be audited. Actually the only plugin, supported by the Istio service mesh, is the Stackdriver plugin [49].

The following Authorization Policy example configures the action to **ALLOW**, to accept some specific traffic. By default, the action will be interpreted as an **ALLOW** action but it is useful to be explicit when we define these rules, like it is done in the next example.

The following authorization policy allows the requests from:

- service account called "cluster.local/ns/default/sa/sleep"
- namespace called "test"

and the workload will be allowed if it is:

- a "GET" method on the path with the prefix "/info"
- a "POST" method on the path "/data"

In the following example it is also specified that the request will be accepted if it has a valid JWT token issued by "https://accounts.google.com".

All the other requests, that won't match with the previous rules, will be denied by the authorization policy itself.

```
---
apiVersion: security . istio . io /v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/sleep"]
    - source:
```

```
        namespaces: ["test"]
    to:
    - operation:
        methods: ["GET"]
        paths: ["/info*"]
    - operation:
        methods: ["POST"]
        paths: ["/data"]
    when:
    - key: request.auth.claims[iss]
      values: ["https://accounts.google.com"]
---
```

The following Authorization Policy is another example that sets the action to **DENY** to block some specific traffic. It denies all the requests from the namespace called "dev" to the **POST** requests in the namespace called "foo".

```
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  action: DENY
  rules:
  - from:
    - source:
        namespaces: ["dev"]
    to:
    - operation:
        methods: ["POST"]
---
```

The Authorization Policy scope is determined by the **metadata/namespace** and the **selector** property fields: the scope, or target, is fundamental to understand on which resources we must apply the policies defined inside.

- **metadata/namespace**: it will tell on which namespace the policy will be applied. By using the root namespace, we can apply the policy to all the namespaces inside the cluster.

- **selector**: it is optional and can be used to restrict the target on which to apply the policy.

In the following example, the authorization policy will be applied to the workloads that contain the label “app: httpbin” in "namespace: bar".

```
---
apiVersion: security . istio . io /v1beta1
kind: AuthorizationPolicy
metadata:
  name: policy
  namespace: bar
spec:
  selector:
    matchLabels:
      app: httpbin
---
```

The source property field specifies the source of a request or better, from which entity we expect the traffic. All the fields inside the source property will be considered like in an **AND** clause: it will represent the group of sources by which we expect to receive some requests.

In the following authorization policy, the source field matches if:

- the principal will be “admin” or “dev”
- the namespace is “prod” or “test”
- the ip is not “1.2.3.4”

```
---
apiVersion: security . istio . io /v1beta1
kind: AuthorizationPolicy
metadata:
  name: policy
  namespace: bar
spec:
  selector:
    matchLabels:
```

```
      app: httpbin
principals:  ["admin", "dev"]
namespaces: ["prod", "test"]
not_ipblocks: ["1.2.3.4"]
---
```

As we can see, by using the Authorization Policy component, we will be able to deny and allow the pod's incoming traffic. This component will be later tested and used to understand if we can implement the Verefoo Neighbour rules inside the cluster.

4.1.3 Differences between Istio Access control and Calico Network management

Until now, we realised how these technologies are built (from an architectural point of view), how these technologies interact with Kubernetes (by using some specific API components) and what components they offer to "potentially" reflect the neighbour rules inside the Verefoo Graph (the CNI NetworkPolicy and the Istio AuthorizationPolicy).

The **Authorization Policy** is a powerful api that can be used to enforce access control and restrict/allow traffic inside the cluster workload. As Marton Sereg wrote in an article titled *"Introduction to Istio access control"* [50]:

"It basically answers the question: who can access what, under which specific conditions?"

Also the **NetworkPolicy** is an api that can be configured to restrict or allow the traffic between the cluster's pods. In the Kubernetes documentation, we can find the following definition:

"A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints."

In the article mentioned before, there was covered an interesting comparison between the **AuthorizationPolicy** and the **NetworkPolicy** [33] resources: they are similar under some aspects because they try to solve the access problem. In any case, there isn't an official answer on which API is better for a certain use case: they aren't the same, and are good and preferable on different environments. For example, the Istio authorization policy will work over the application level (because the service mesh or better, the Envoy proxy, works over the layer 7) while

the Network Policy will work, directly, over the network and transport levels.

Working over the application level (L7) has its own advantages: at that level, the Envoy sidecar is able to understand and interpret many different protocols (most of the time HTTP or HTTPS). By this way, the proxy is able to apply a rich set of attributes to the policy decisions like decisions based on the URI or the HTTP headers etc.

The network policy, on the other side, cannot perform these operations because the network/transport layer is unaware of these concepts. But operating on a lower level has its own advantages too: it is an universal approach because all the network applications will use the IP protocol. This means that the user/developer can apply policies that will be enforced at the kernel level. This mechanism improves drastically the performances but of course isn't as powerful and flexible as the Envoy proxy mechanism.

One other important difference that we must mention is that the NetworkPolicy will work with a white-list model: only communications that are explicitly allowed will be accepted, all the others will be denied by the policy itself. The policies won't conflict but simply follow an additive and order sensitive approach. The AuthorizationPolicy supports the allow/deny mechanism: of course this approach will be more complicated but, on the other side, gives lot of flexibility to the configuration phase.

Finally what the user should choose it depends on the specific use case it has to deal with:

- if the user needs a finer grained authorization mechanism, Istio would be a better choice
- if the user simply need a mechanism to choose which resource should communicate to which one, the Network Policy are good enough to cover this behaviour.
- there is also a third option: the user can define also an hybrid configuration, in fact Istio can work perfectly with a CNI plugin like Calico.

To summarize, the Istio service mesh can enforce and apply an access control inside the cluster by using the Authorization Policy resource: this control is enforced at the application layer by exploiting the Envoy proxy functionalities. On the other side, the NetworkPolicy should be used with simpler scenarios in which the user will simply need a mechanism to allow/deny traffic between resources like pods.

4.1.4 Use-cases and Tests

Before proceeding with the modification of the application's architecture, we need to understand which technology better reflects the neighbour rules produced by Verefoo inside the cluster and if there are some limitations relative to these technologies: purpose of this chapter is to illustrate two different use-cases in which we applied, separately, the Calico and the Istio technologies inside the Kubernetes cluster. At the end of the chapter, a final comparison will be performed, between these two technologies, to better understand the best technology to be adopted to reflect the Neighbour Mechanism.

Istio Use Case

First of all, to proceed we need to deploy some pods into our cluster: for these use-cases we deployed 5 different pods simulating a "production" environment (db-pod, web-pod and an api-pod etc.). After the deployment, by getting the list of the running pods, the situation could be like the following one:

kubectl get pods

NAME	READY	STATUS	RESTARTS	AGE
playlists-api-684fb4c5d7-8p6gb	1/1	Running	0	47h
playlists-db-7d68bbf5c4-pz22s	1/1	Running	0	47h
videos-api-ccc8f5b46-nd8gj	1/1	Running	0	47h
videos-db-85ccd8bc-g5fwp	1/1	Running	0	47h
videos-web-cdbc466f4-s8vtf	1/1	Running	0	47h

After the cluster setup, it was installed Istio into the cluster itself through the following command:

istioctl install –set profile=default

After the Istio installation, we were ready to inject the sidecar container inside our pods. To resume the overall mechanism, Istio will inject, inside each pod, a **sidecar-container** in order to collect stats and enable some authorization policy for the incoming traffic.

By enabling the auto-inject behaviour with the following command:

kubectl label namespace/default istio-injection=enabled

a specific sidecar was deployed inside each pod; in fact, after getting the pod list, we can notice that the number of running (READY) containers, per each pod, is two (one representing the previous business logic and the other one representing the new proxy-sidecar).

NAME	READY	STATUS	RESTARTS	AGE
playlists-api-684fb4c5d7-tjsqg	2/2	Running	0	98s
playlists-db-7d68bbf5c4-b45ql	2/2	Running	0	98s
videos-api-ccc8f5b46-d7n96	2/2	Running	0	98s
videos-db-85ccd8bc-pjlpm	2/2	Running	0	98s
videos-web-cdbc466f4-5m8hm	2/2	Running	0	98s

Before performing some routing tests, we installed also a graphic tool called **Kiali** in order to represent graphically all the traffic cached from the sidecars.

After executing the following command:

kubectl apply -f /tmp/istio-1.6.12/samples/addons/kiali.yaml

we were able to run the Kiali dashboards by instantiating a local server with the following command

kubectl -n istio-system port-forward svc/kiali 20001

After looking, from the main-page, at the Kiali dashboards we have seen the following traffic topology:

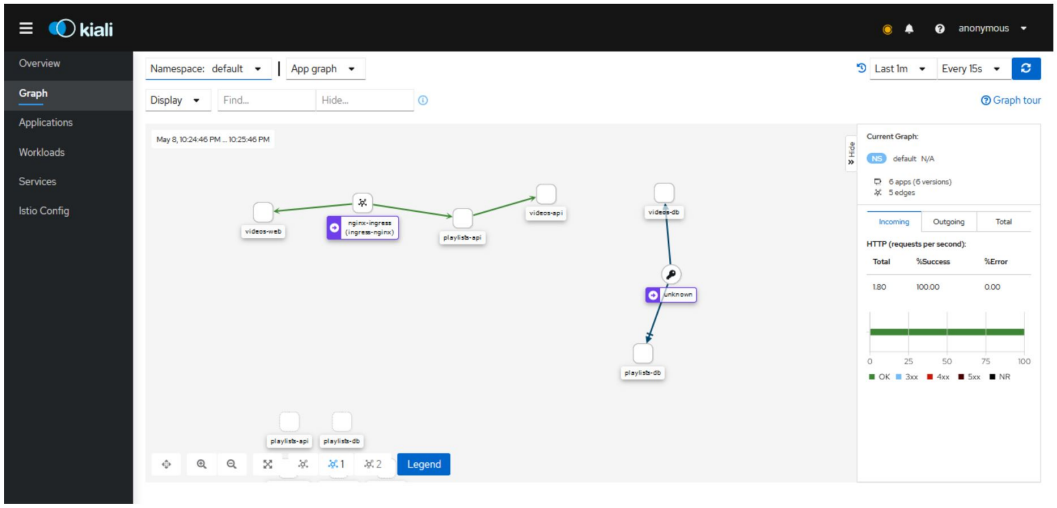

Figure 4.7: Neighbour Mechanism - Istio scenario: Cluster Traffic Topology with Kiali

As we can see from the previous image, the nginx-ingress (an Ingress component) communicate with the videos-web pod and also with the playlists-api pod (which redirect the request to the videos-api pod).

By inspecting some more details for the pod playlists-api, we can get

- **Overview details**: it will display information regarding the properties of the pod and some health statistics
- **Inbound metrics**: these dashboards will collect some metrics regarding the incoming traffic
- **Outbound metrics**: these dashboards will collect some metrics regarding the outgoing traffic (as we can see, the overall outbound playlist-api pod's traffic is directed to the videos-api pod)

Figure 4.8: Neighbour Mechanism - Istio scenario: Outbound Traffic Metrics with Kiali

As we can understand, if we don't apply any constraints or restrictions, each pod is able to talk to all the others. But, how can we reflect the Verefoo neighbours rules, inside our cluster, by using Istio? In order to restrict some incoming traffic, we used the **Authorization Policy** resource.

First of all, we needed to declare a **ServiceAccount** for each pod, the reason is simple: the authorization Policy construct is able to recognize the traffic related to a specific K8S account; as we already know, the AuthorizationPolicy is an Istio component that was built to authorize or restrict the traffic for a specific

entity/account/organization. In fact, the authorization policy will simply work on top of ISO/OSI level 7 (if we compare it to the Calico 3-4 working levels, Istio will work over an higher layer).

For example, we tried to deny the traffic from the playlists-api to the videos-api. We started deploying the Service Accounts for both the pods:

```
# VIDEOS-API Service Account

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: videos-api
  namespace: default
---

# PLAYLISTS-API Service Account

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: playlists -api
  namespace: default
---
```

With the previous service accounts, we were able to label the pod's traffic. These accounts were used to define the following AuthorizationPolicy rule:

```
---
apiVersion: security . istio . io /v1beta1
kind: AuthorizationPolicy
metadata:
  name: videosapiauthpolicy
  namespace: default
spec:
  selector:
```

```
    matchLabels:
      app: videos –api
  action: DENY
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/playlists-api"]
---
```

By applying the previous resource with the following command:

Kubectl apply –f authPolicy.yaml

we saw immediately some results.

By looking at the previous Kiali dashboards, we noticed that the traffic between the videos-api and the playlists-api was not coloured anymore with green: this means that the playlists-api wasn't able to talk with the videos-api anymore (on the right side we can see that the percentage of error http-response rapidly increased).

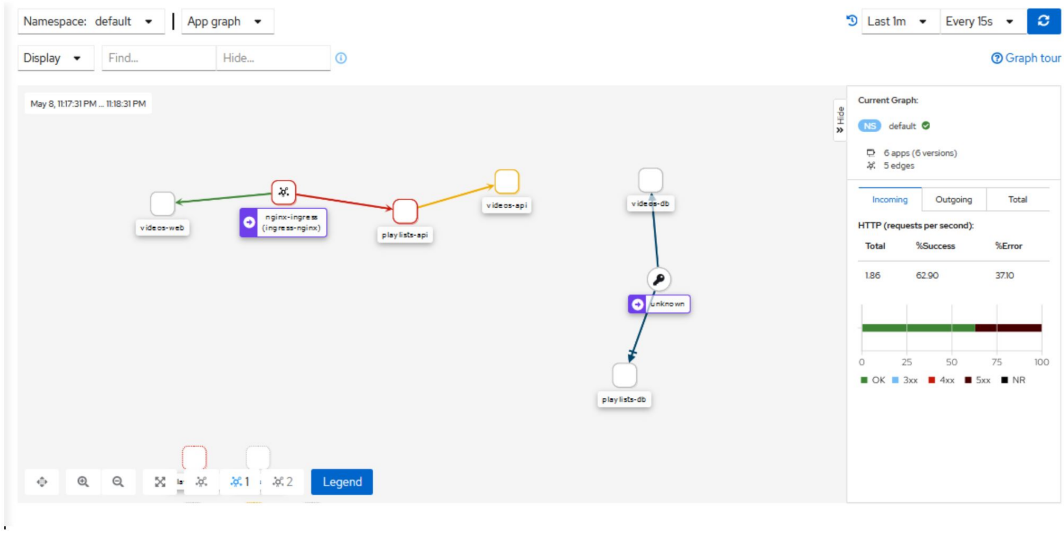

Figure 4.9: Neighbour Mechanism - Istio scenario: Cluster Traffic Topology with network constraints

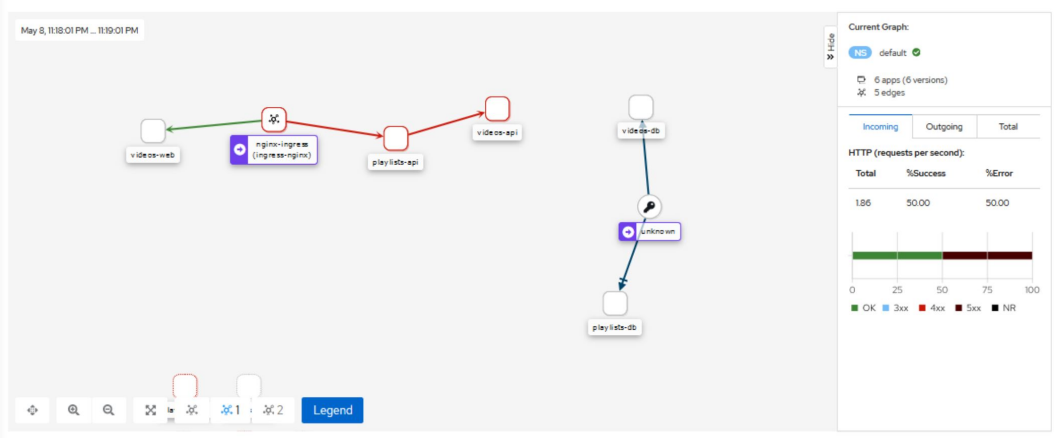

Figure 4.10: Neighbour Mechanism - Istio scenario: Cluster Traffic Topology with network constraints

By looking also at the overview section, we can see that all the HTTP traffic responded with error.

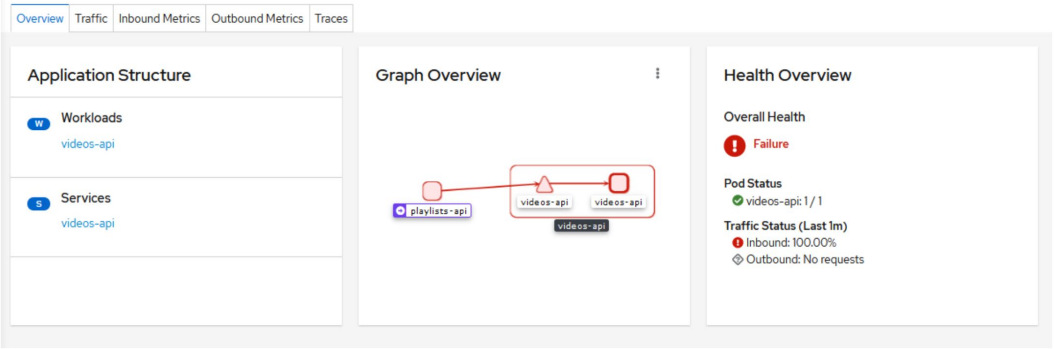

Figure 4.11: Neighbour Mechanism - Istio scenario: Network Traffic Overview with network constraints

To summarize, as we can see from the previous tests, thanks to the AuthorizationPolicy and the ServiceAccount, we are able to restrict the incoming traffic for each pod. By using this resource we are able, potentially, to enforce some traffic-restrictions between the pods: this mechanism can be used to reflect the neighbour mechanism from Verefoo, by telling to each pods the list of accounts (the neighbour list) to which they can communicate or not.

On the other side, we have to consider that this mechanism carries some limitations:

- It needs to define a ServiceAccount for each pod
- It is necessary to maintain a unique service account per each pod: if for some reasons, two (or more) pods share the same ServiceAccount, the overall filter-mechanism could be compromised
- With the AuthorizationPolicy, we are able to define only incoming rules: the outgoing rules cannot be covered by this component
- This mechanism works on top of ISO/OSI level 7
- with the Istio AuthorizationPolicies the developer can specify which user can access under specific condition and there isn't the possibility to specify directly which pod can communicate to the others

All these limitations are strictly bounded to the nature of Istio itself: as we already treated, the service mesh, natively, works on top of level 7, and so the user can only define high-level rules. For example, with the AuthorizationPolicy, there is the possibility to filter the GET/POST http requests that will match a specific URL prefix-path.

For the purposes of this thesis (we simply need to define lower-level rules, to deny/allow IP traffic on some specific ports) and for the reasons shown below, we have searched for different technologies and approaches. This is the reason why we compared and tested this solution with the Calico CNI technology.

Calico Use Case

For this use-case we installed the Calico CNI inside our cluster and maintained the previous pod configuration.

First of all, we launched a Kubernetes cluster, defining the Calico CNI plugin we wanted to use. After having verified the Calico installation in the cluster, we were able to see something similar to:

NAME	READY	STATUS	RESTARTS	AGE
calico-node-85s5b	1/1	Running	0	3m31s

it means that the Calico node agent was running, correctly, inside our cluster.

By checking the actually running pods inside the cluster, we saw the following results:

NAME	READY	STATUS	IP
playlists-api-684fb4c5d7-wrdms	1/1	Running	10.244.120.69
playlists-db-7d68bbf5c4-5gtll	1/1	Running	10.244.120.70
videos-api-ccc8f5b46-v724r	1/1	Running	10.244.120.73
videos-db-85ccd8bc-gfwg8	1/1	Running	10.244.120.72
videos-web-cdbc466f4-brsvk	1/1	Running	10.244.120.71

After the Calico installation, the pods are still able to communicate and send traffic to each other, without restrictions. As we can see from the following screenshot, by sending some traffic, from the pod videos-web into the pod playlists-api, it will be accepted and received by the destination.

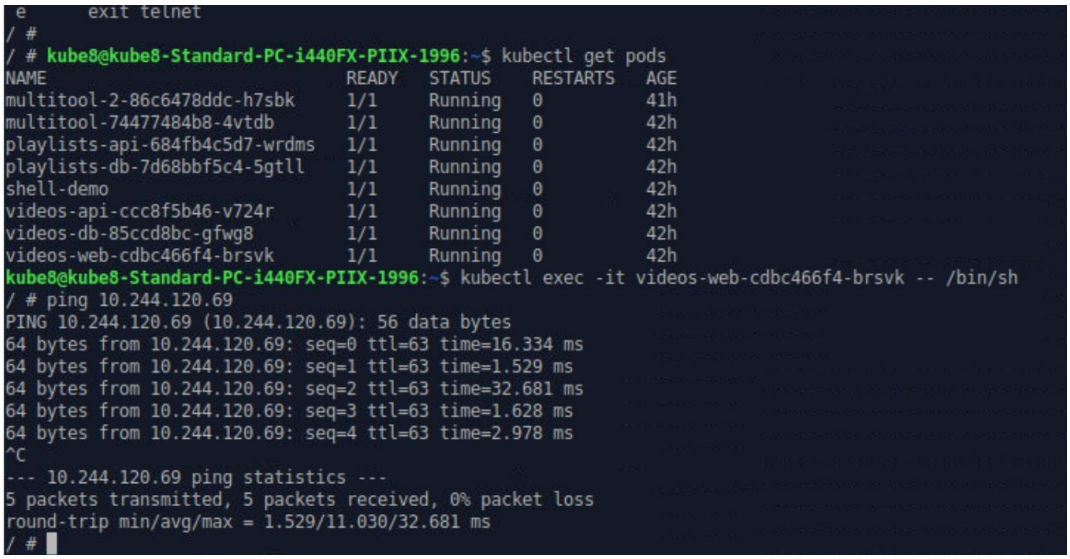

Figure 4.12: Neighbour Mechanism - Calico scenario: successful ping test

In order to limit the incoming traffic of the playlists-api pod, it was applied the following NetworkPolicy: the purpose of this constraint is to limit all the incoming traffic, except the one from the pod playlists-db, to the pod playlists-api.

```
---
apiVersion: networking . k8s . io /v1
kind: NetworkPolicy
metadata:
  name: default -deny-ingress
spec:
  podSelector:
```

```
    matchLabels:
      app: playlists   -api
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: playlists   -db
---
```

For a better understanding, the previous NetworkPolicy is composed of an INGRESS rule to ALLOW only the incoming traffic from the playlists-db pod. Once the previous policy was applied with success, after trying to ping the playlists-api from the videos-web pod the request will be avoided, as we can see from the following screenshot:

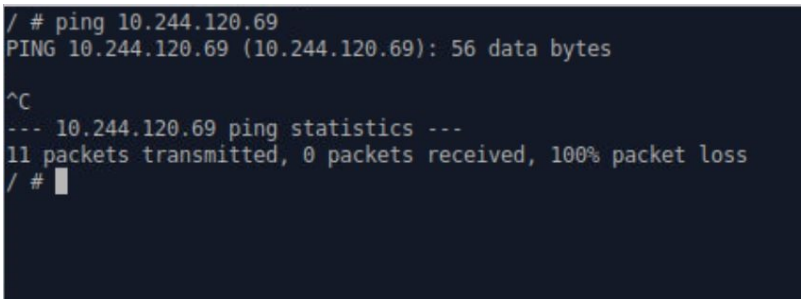

Figure 4.13: Neighbour Mechanism - Calico scenario: denied ping test

On the other side, after pinging the playlists-api from the playlists-db pod, the request was accepted with no problem, as it is showed in the following screenshot:

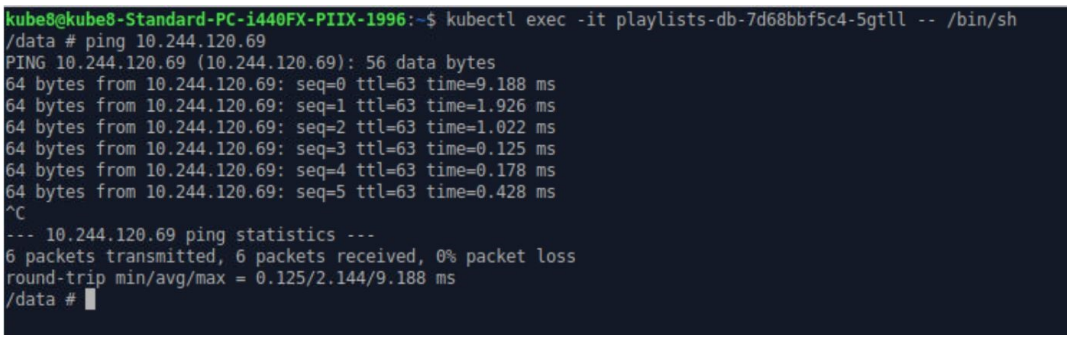

Figure 4.14: Neighbour Mechanism - Calico scenario: denied ping test with the Network Policy constraint

Differently from Istio, with Calico we are able to define also the outgoing firewall

rules (not only the incoming ones). In order to demonstrate the potentiality of this technology, we have defined also a NetworkPolicy in which it was defined an **EGRESS** rule for a pod. For our last test with this use-case, we used the following NetworkPolicy:

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
spec:
  podSelector:
    matchLabels:
      app: playlists-api
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: playlists-db

  policyTypes:
  - Egress
---
```

With the previous NetworkPolicy, from the playlists-api pod we were able to send traffic to the playlists-db pod only. After going inside the playlists-api pod, we tried to ping the videos-web pod and then the playlists-db pod: by pinging the pod videos-web from the playlists-api pod, all the requests were denied but, all the traffic to the pod playlists-db was allowed instead.

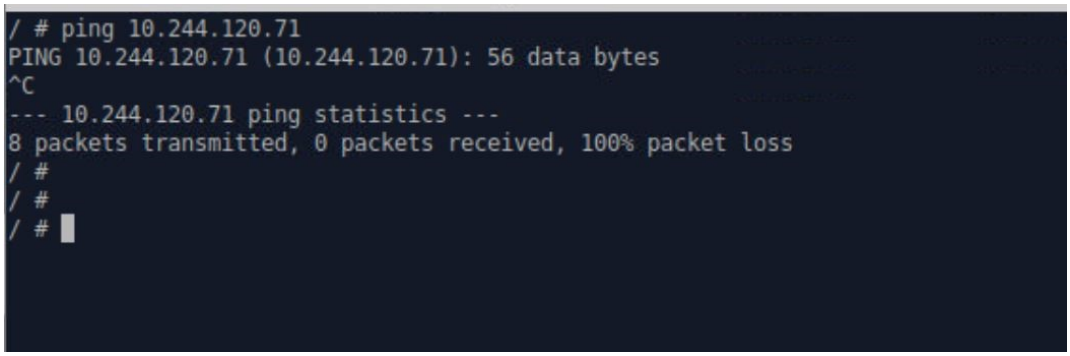

Figure 4.15: Neighbour Mechanism - Calico scenario: denied egress traffic

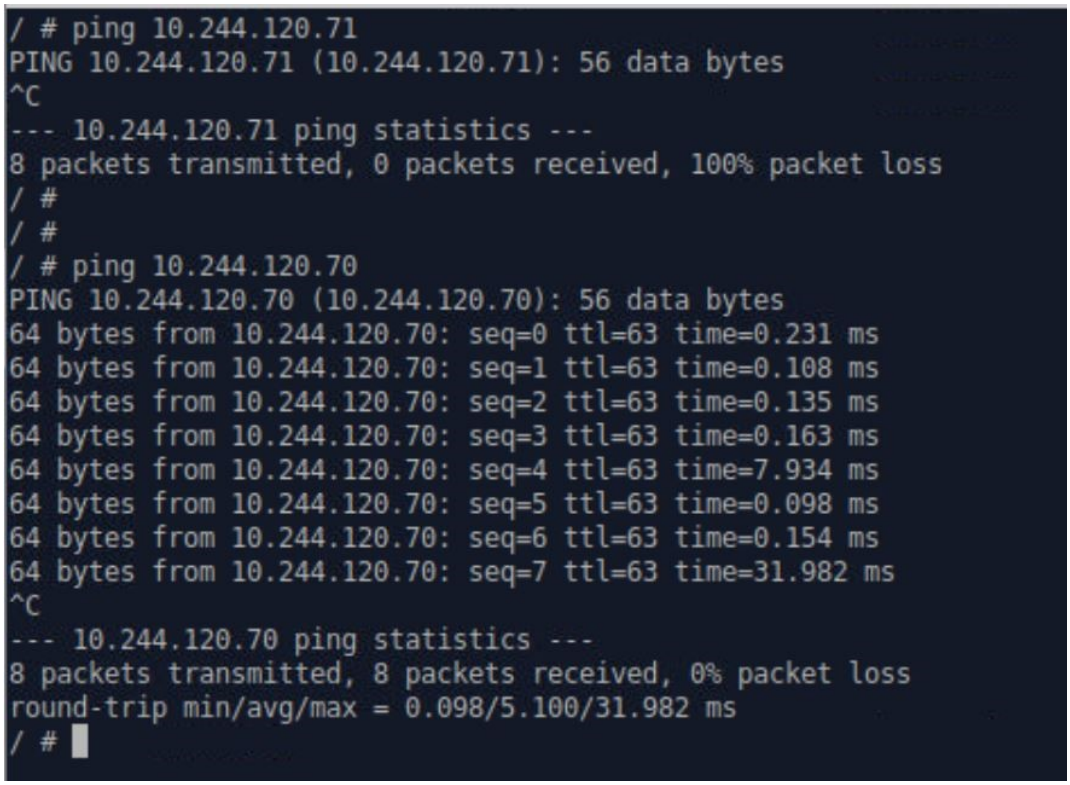

Figure 4.16: Neighbour Mechanism - Calico scenario: allowed egress traffic

To summarize, the Calico solution is characterized by the following aspects:

- the developer can specify which pod can communicate to each other
- the NetworkPolicies will filter the L3/L4 traffic (it works on both the transport and network layer)

- the developer can specify both ingress and egress traffic rules
- the definition of the traffic rules is simple and require less K8S components definition

Final considerations

As we have seen in the first use case, Istio offers a lot of functionalities and flexibility but, if it must deal with the network traffic, it is only able to work with the L7 traffic and define the Incoming Rules inside each pod.

On the other side, Calico offers less functionalities than Istio, but we are able to fill the previous gap thanks to the outgoing NetworkPolicies.

In summary, from the point of view of the Traffic Management, with Calico we are able to cover all the traffic-routing needs, by working on the same level (level 3-4 of the ISO-OSI protocol) used by Verefoo to produce the neighbour and firewall rules.

For the reasons showed previously, the Calico technology is more suitable for the thesis' purpose but, if we need, maybe in future, to work in a more complex scenario, we should switch to the Istio service mesh.

4.2 Forwarding Mechanism

Another important Verefoo's feature, that we had to deal with, is the forwarding mechanism: Verefoo is able to define a path composed by a chain of different services (nodes) that the traffic must follow to be processed correctly.

By default, Kubernetes isn't able to reflect this mechanism: internally, it is able to define only point-to-point constraints. For this reason, were explored three different solutions that will work over the kuberentes cluster to extend its functionalities and support the forwarding rules outputted by Verefoo.

To come up with a solution, we analyzed three different technologies:

- Service Mesh
- Serverless functions
- Serverless workflows

In the following section, we will discuss better in details only the serverless technologies, because the service mesh one was already treated in this paper.

It is important to underline the fact that, none of the three solutions analyzed were sufficient to reflect the Verefoo forwarding rules because, as we will discuss later in more detail, all the solutions will add too much complexity and will simply work over the application layer (one of the major limits that we had to face once working over the kubernetes cluster). To sum up, we didn't found a mechanism to reflect these rules on the network or transport layer, which are the most suitable working layers to reflect the forwarding rules.

4.2.1 Verefoo Forwarding Concept

The Verefoo tool, as we already know, is able to retrieve in output all the node's neighbours: the neighborhood represents all the nodes that a specific end-host is able to communicate with.

This isn't the only feature that the tool has to offer to us: it's also possible to define a specific forwarding path that can be used to **redirect** the messages to a specific node that the source can't contact directly through the topology schema (its neighbours list doesn't contain that node).

In the example shown below we can better understand what does it means.

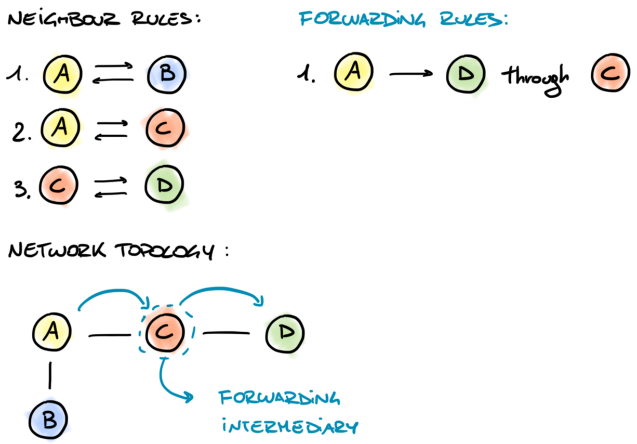

Figure 4.17: Forwarding Mechanism: Verefoo Forwarding Rule example

First of all, Verefoo will retrieve to us the **neighbour rules**, represented by the three **black** rules in the previous image. These rules are the ones that the graph

must consider in order to allow point-to-point communication inside the graph itself (the neighbourhood).

The previous chapter focused the attention on the black rules and on the technologies that can constraint the Kubernetes pods to only communicate with their neighbours.

The **forwarding rule**, represented by the **blue** rule in the previous image, is the one that let the node A to communicate, indirectly through the node C, to the node D: by this way, the node A and D doesn't have to be near to each other, but can communicate through an intermediary node (the node C).

The following sections will focus on the "blue rule" and on the technologies that can reflect this mechanism over the Kubernetes cluster.

On a first instance, we will explore all the technologies that were analyzed for our studies and then we will discuss all the designed architectures that were proposed to let these technologies to work with the Kubernetes cluster.

As we anticipated, the main purpose of this chapter is to explain the studied technologies and the solutions that were proposed to extend the cluster functionalities: there won't be illustrated any use cases, as performed in the previous chapter, because the solutions weren't enough flexible and suitable to support the Verefoo forwarding mechanism.

4.2.2 Serverless functions

As Adam Bavosa wrote in the article titled *"What is a Serverless Function"* [51], the principal aim of the serverless technology is to provide a platform-agnostic experience to all the developers, a way to write and run code independently on the language or platform. By this way the developer will be able to simply write the business logic, without having to focus on a specific language in particular.

Typically these functions are hosted and managed by some particular infrastructure that won't belong to a specific language or framework. These functions will be run and invoked remotely (through internet) and hosted by cloud companies. The company's team has to guarantee some aspects:

- these functions were completed with no delay time or computation problems
- there must be some redundant instances (replica set) to guarantee a fault tolerant approach: if some instance will shut down, there must be the possibility

to compute the request/operation on a different replica

- the infrastructure must scale all the incoming requests that will come from all around the cloud

Because of the fact that the serverless functions will run a specific business logic, these functions can be considered, in a certain way, micro-services. As we know, in the last year there was a growing interest, in the developing communities, in moving from the classical monolithic approach into the multiple/distributed micro-services one: the monolithic application is difficult to maintain and to extend with new code (even the single bug fixes could be hard to be done).

Of course, monoliths applications have their own advantages because in some scenarios it could be enough to have all the logic into one point but, with the grown of the application/project, in the future some task could become cumbersome.

As we know, an architecture composed of micro-services will improve the maintainability of the code itself and increase the productivity of the software developing team: the team could be split into small group of people working on different tasks that can be performed in parallel and completely autonomously.

Two of the greatest advantages of adopting the serverless functions is that the code deployment will be scaled: the developer has to simply write the logic of the functions, while the infrastructure has to call the function guaranteeing some services (the cloud host will provide these services as products, Functions as a Service).

Functions as a Service platform (FaaS platform) has its own unique scenarios and will offer different solutions and functions.

AWS Lambda

AWS Lambda is the product offered and available on the Amazon Web Services platform. It will support some programming languages like:

- Java
- Go
- PowerShell
- Node.js
- Python

- Ruby

One of the most usefull scenario in which to use the Lambda Function is where there is the need of computing the requests on-demand: for example, file conversion or file processing.

Azure Functions

Azure Functions is the product proposed by the Microsoft Azure environment. The supported languages are the following ones:

- JavaScript
- Java
- Python
- TypeScript

Using these lambda functions is ideal in environments or scenarios in which there is the need of a single-purpose API: for example, build up an HTTP endpoint to enable a secure communication.

Google Cloud Functions

Google Cloud Functions is the product proposed by the Google Cloud Platform environment. The supported languages are the following ones:

- Go
- JavaScript
- Python

One of the greatest advantage of using these functions is that they can interact, potentially, with all the other Google cloud services and applications. For this reason, this product is ideal when we have to process data (for example, retrieving data from videos or images).

Kubeless

Kubeless [52] is a project designed to work over the Kubernetes infrastructure: the user will have to simply deploy small functions inside Kubernetes without having to manage manually the infrastructure and deployment on the cluster. By being designed over the Kubernetes cluster, it will take all the advantages of the Kubernetes primitives. In a certain way, Kubeless will offer the functionalities provided by the AWS, Google or Azure infrastructures inside the Kubernetes environment.

Kubeless includes some of the following features:

- Support for a large group of programming languages: Java, Node.js, PHP, Python etc.
- It is CLI compliant with the AWS Lambda CLI offered by Amazon
- It will support the event triggering mechanism by using the Kafka message systems or the HTTP protocol
- It can include the Prometheus monitoring systems for the functions calls or the functions execution latency

Kubeless uses a Custom Resource Definition (in compliance to the standard followed by Kubernetes [53]) to be able to translate the functions into custom Kubernetes resources (as we will see later on, this operation is performed automatically by the Kubeless controller). Kubeless then runs an internal controller that watches and monitors these resources. The controller dynamically translate and inject the function codes into resources to make them available through Kafka (publish-subscribe mechanism) or through the HTTP protocol.

To summarize, the Kubeless project will bring both functions and events into the Kubernetes cluster.

As we know, the function is an independent unit of work (pure business logic), and it is similar to a micro-service. It's represented by the user's code that is most often written to compute a single job, for example:

- Interacting with the database
- Process a file or extract some information from it
- Perform a repetetive task (scheduled task)

The event, on the other side, is the trigger that will execute a specific Kubeless function. The event and the trigger will be managed entirely by the Kubeless infrastructure.

Here there are some kubeless event examples:

- An HTTP endpoint requested by some entities in the cloud (REST API)
- A Kafka message published over the queue
- A timer (this functionalities must be added in the newer version of Kubeless) to perform some scheduled task

As we can understand, this project was born to help the developer to easily deploy the logic into the cluster without considering how to inject the logic inside the pod (the developer, as was already mentioned in this paper, will simply focus on the pure business logic).

Kubeless: Installation and Functions deployment

First of all, the developer has to create a kubeless namespace where he will deploy the controller

```
kubectl create ns kubeless
```

Then, he must install a kubeless stable version with the **kubectl create** command

```
kubectl create -f kubeless-v1.0.0-alpha.8.yaml
```

After a while, the user can see that some pods are running in the "kubeless" namespace. The controller, on the other side, will watch for the function objects to be created.

Entering the following command, the kubeless controller has to be shown on the screen:

```
kubectl get pods -n kubeless
```

As soon as the controller starts, the developer can deploy a function.

To deploy a function it can be used the kubeless CLI. The developer need to specify a runtime which specifies which language his function is in. Then, he need to specify the file that contains his function, how the function will get triggered (here we use an HTTP trigger) and finally he can specify the function name as a handler.

```
kubeless function deploy toy –runtime python2.7
–handler toy.handler
–from-file toy.py
```

Then he could list the function with the kubeless CLI:

```
kubeless function ls
```

Behind the curtains, Kubeless automatically create a Kubernetes deployment and service. The developer can check that a Pod containing his function is running:

```
kubectl get pods
```

At this point we have two alternatives:

- send an HTTP request through the kubeless CLI
- expose a proxy, and then send an HTTP request (with curl for example)

To summarize, the developer can call the function using the kubeless CLI convenience wrapper:

```
kubeless function call toy –data '"hello":"world"'
```

Or he can run a proxy

```
kubectl proxy –port 8080
```

and then invoking the function directly via curl:

```
curl –data '"hello":"world"'
localhost:8080/api/v1/namespaces/default/services/toy:8080/proxy/
–header "Content-Type:application/json"
```

As we can see from the following image, the overall mechanism is quite simple:

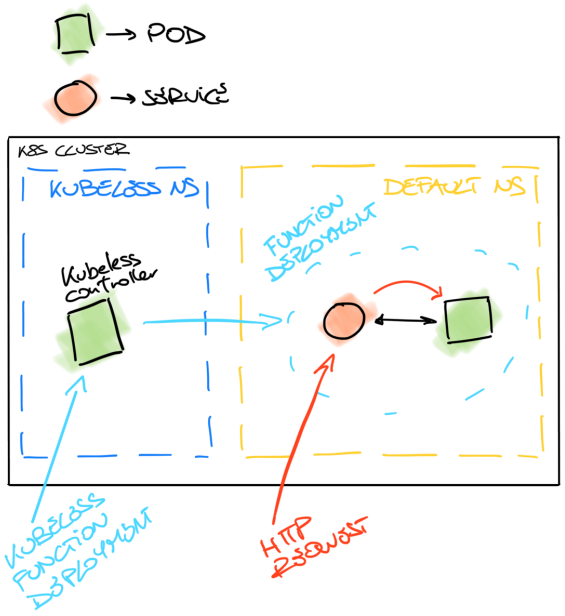

Figure 4.18: Kubeless Deployment and Request mechanism

Once we had installed Kubeless into our cluster, a custom controller called **function-controller** will be instantiated to watch continuously the changes to the resources and respond to them. By default the controller will be instantiated inside the "kubeless" namespace. It will listen to the creation event moved by the user when he will deploy some functions into the cluster. Each time the user will add a new function, the controller will create a deployment for the function itself and exposes it with a clusterIP service. The service and the deployment will be managed by the user thanks to the function.spec field. To summarize, the controller will be a sort of translator: it will take the kubeless function configuration passed by the user and translate it into Kubernetes components.

During the creation (or the deployment phase) of a function, the user can specify the way which the external world can communicate with the function: by specifying the trigger type, for example, we can communicate through Kafka or the HTTP protocol. In the following examples we will define the HTTP trigger; of course, we have to ensure that the deployment will be reachable from the outside. As mentioned, the user has to define only the business logic inside the function file: the controller will move everything inside the function into the Kubernetes deployment, in order to be executed each time the pod will be instantiated.

4.2.3 Serverless Workflow

The Serverless Workflow is a mechanism and a cloud service that is used to coordinate and manage multiple tasks in parallel or sequentially. It is usually committed to simplify some tasks (or simply the coordination of them) such as: error/state management, task coordination etc. Typically, is a mechanism used to execute CI/CD pipelines in production environments.

One of the strongest features offered by these tools is that, sometimes graphically, the user can arrange tasks in parallel, branch, sequence or with some if-then-else logic: by this way the user can define, exactly, what must be done in order to complete the overall task (the user has to simply focus on the logic and the tool will cares about the resource management and the trigger-phase). The workflow tools, usually, will offer some services to guarantee some specific behaviours like the error handling, retry logic, etc. to offer a smooth completion mechanism.

Another great advantage of using these technologies is that the user, by simply focusing on the code, doesn't have to care about some "horizontal/transversal" logic (like the error handling) and by this way the overall amount of necessary code will be reduced drastically: sometimes these tools offer also a GUI to let the user define the pipeline-logic (means as a sequence of tasks) without writing this mechanism by hand. This helps also by reducing the amount of duplicated code.

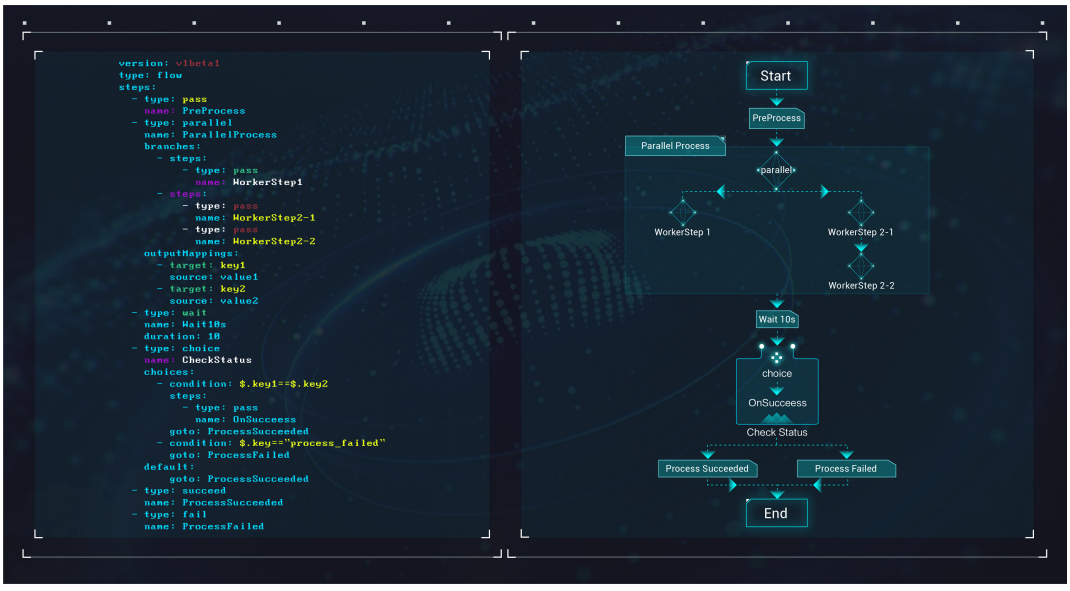

Figure 4.19: Forwarding Mechanism: Serverless Workflow Logic

The most important features introduced by the serverless workflow can be

resumed in the following list:

- **Service Orchestration Capabilities**: the Workflow tool, according to the process definition (the steps that must be executed), will execute the pipeline's steps in a precise order.
- **Operation and Maintenance**: the Serverless Workflow tools provides some features to free the user from maintaining the infrastructure by hand (for example, provide security or fault-tolerant mechanisms).
- **Process State Management**: the tool will provide a mechanism to manage the state (execution step's tracking, input and output between the steps, etc.). The user doesn't need to manage the state by itself or build complex State Machines.
- **Visual Monitoring**: the tools provide some visual interface (GUI) to define the workflows and view their execution state.
- **Long Running Process**: the tool will monitor and track the process to ensure its completion for a long time (some tasks may need to be executed for days, others for hours etc.).
- **The Distributed Component**: these tools will be a sort of coordinator between the applications: the applications will be written in many different languages and run in different network places.

Argo Project

The Argo Project [54] is a group and collection of different projects, natively compatible with the Kubernetes infrastructure, proposed for the continuous delivery and integration (CI/CD).

Inside the Argo suite, we have the following tools:

- Argo Workflows engine
- Argo CD continuous deployment
- Argo Events dependency manager

The user will be able to use these tools through both the Argo CLI and the Kubernetes kubectl.

Argo Workflow

Before entering in the detail of this product, we have to understand why it is necessary to think about a task orchestrator. As Coussement Bruno wrote in the article titled *"What to consider before choosing Argo?"* [55]:

"Imagine you are tasked with four items: cleaning data, training a model, evaluating a model, using the model to make inferences on unseen data. In the beginning, this happens ad-hoc, because you're the only one in the organization performing them".

One of the main problems that these tools aim to solve it's a practical one: the task management. Initially, with the starting of a project, the team is a small-sized one and the management of the tasks could be enough simple. But with the:

- growing of the developing team
- by repeating some tasks during the software life cycle
- by increasing the number of products or teams that will be involved in the project

the management of these tasks could become cumbersome and hard to deal with. This is the typical scenario in which the task orchestrator is necessary. When it comes into play, the user can leverage on it to model a graph of tasks (DAG) by defining the node (representing the tasks that must be done) and the edge/arrow (the task's execution dependency). The user will define the graph and the orchestrator will monitor and trigger the flow.

Argo Workflows like other similar products, is an open source project and will offer a task orchestrator with the advantage of being compatible with Kubernetes natively: this means that the engine will orchestrate the tasks inside the Kubernetes cluster, with no intervention from the user.

The Argo workflow will offer the following features:

- the user will define the flow in which each step will be a Kubernetes running container.
- it will use the DAG graphs to model the sequence of tasks (the final flow).
- it can run intensive tasks with no delays, inside Kubernete.
- it can run CI/CD pipelines without using complex development products.

The Workflow tool will perform two important tasks/functions:

- definition of the workflow that must be executed.
- workflow's state saving.

Because of these responsibilities [56], a Workflow must be considered a "live" object. It is not only a static definition: every time the workflow will be executed, it becomes a running instance inside the Kubernetes cluster.

```
---
apiVersion: argoproj . io /v1alpha1
kind: Workflow
metadata:
  generateName: hello-world-    # Name of this Workflow
spec:
  entrypoint: whalesay          # Will run "whalesay" step first
  templates:
    - name: whalesay
      container:
        image: docker/whalesay
        command: [cowsay]
        args: ["hello world"]
---
```

As we can see from the previous example, the workflow that must be executed will be defined through a CRD resource called "Workflow": the flow that must be triggered by the orchestrator is defined inside the Workflow.spec field. The main structure of a workflow.spec will be a group of entry-points or templates.

To summarize, the Argo Workflow will be a DAG composed by the tasks that must be executed in a proper order: by adopting this approach the user doesn't have to define also the logic to orchestrate the tasks, but simply the business logic.

Argo Events

Even if it isn't strictly related to the serverless workflow concepts, it is another product offered by the Argo suite that was studied and used inside this paper, for this reason is necessary to give at least an high level overview of this project. Argo Events is an event-drive framework to automate the triggering of many different tasks types: Webhook, Serverless Workloads, K8S resoruces, Argo workflows, messaging queues, etc.

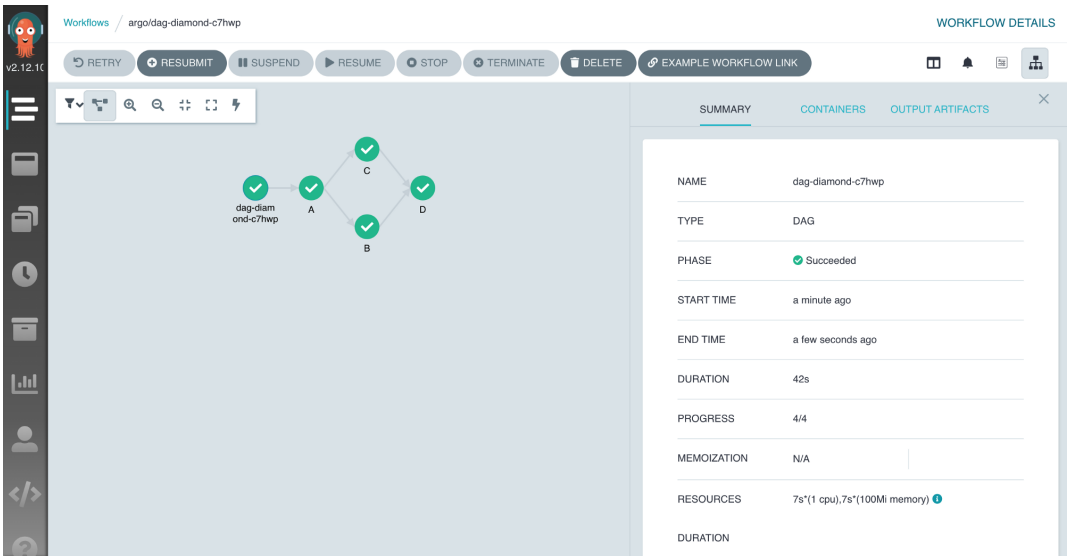

Figure 4.20: Forwarding Mechanism: Argo Workflow example

To summarize, the Argo Events main features can be listed in the following ones:

- supports many different triggering type events
- it gave the possibility to customize the workflow logic (automation) with if-then-else logics
- compatible with the Kubernetes infrastructure
- compatible with the Argo Workflow product

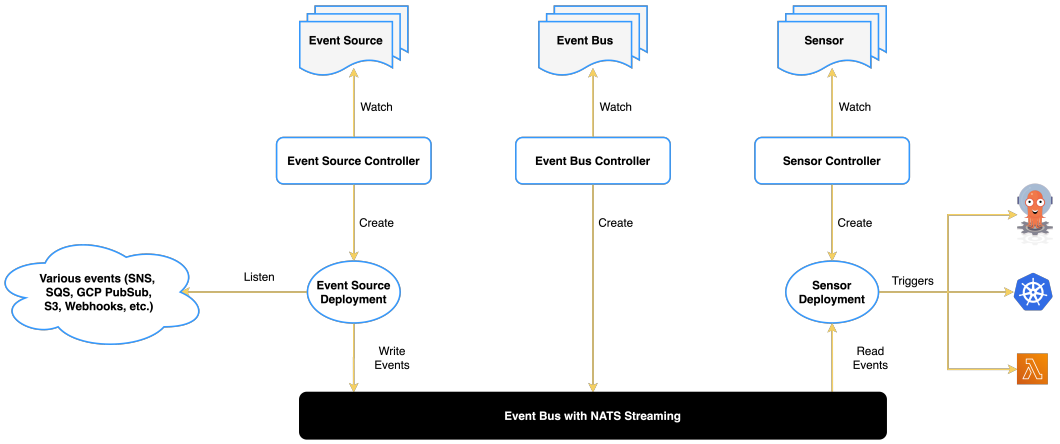

Figure 4.21: Forwarding Mechanism: Argo Events architecture

The Argo events product is characterized by many different components:

- **Event Source**: it defines the event source type that will generate the event (Webhooks, PubSub etc.). Once the event is received, this component will dispatch it inside the cluster thanks to the EventBus component
- **Sensor**: it will define a group of event dependencies (that act as inputs) and triggers (that act as outputs). It will receive the events from the event bus and manage them to finally execute the trigger. The event that the Sensor is waiting for is called "Dependency".
- **Eventbus**: it is a sort of event-broker. It is the transport layer of the Argo-Events product it is responsible of the connection between the EventSource (the event publisher) and the Sensor (the event subscriber)
- **Trigger**: it represents the resource or the workload that must be executed and triggered by the Sensor.

To summarize, the Argo events is the proposed tool to manage the event generation and management: it will be used, as we will see in the following sections, to trigger an Argo Workflow and perform some tasks.

4.2.4 Architecture Design

In this chapter will be treated all the architectures that were thought and designed in order to reflect the Verefoo forwarding mechanism inside the Kubernetes cluster. There will be proposed three different designs that will use the technologies that were previously introduced. For each design we will focus on:

- its features
- the complexity added in the cluster
- its weakness

Istio Design

The first architecture that was analyzed to reflect the forwarding mechanism was the one which exploits the functionalities provided by the Istio Service Mesh. As we have already introduced, the service mesh will add a Software layer over the cluster that will work only with the application level (L7).

One of the first limits that we had to face with this technology was the **application level constraint**: Istio isn't capable to understand the lower level of the ISO/OSI protocol, for this reason we are able to work only over the level 7.

The design of this architecture was strongly influenced by the previous constraint and, for this reason, all the forwarding logic was built inside the application level: as we can understand, this is the reason why the Istio architecture can't be flexible enough to be adopted for the forwarding mechanism (we aren't able, for example, to inspect information or fields relative to the layer 3/4).

In the following schema, we can see in more detail all the components that must be added in the cluster to reflect the forwarding mechanism: the majority of these components come from the Istio API.

This design is characterized by a strong architecture-constraint: we need to **define a different namespace per each pod** (the component which contains our business logic). This limit comes from the Sidecar component: as we will see later on, if we want to force the Sidecar to send the traffic to a specific destination,

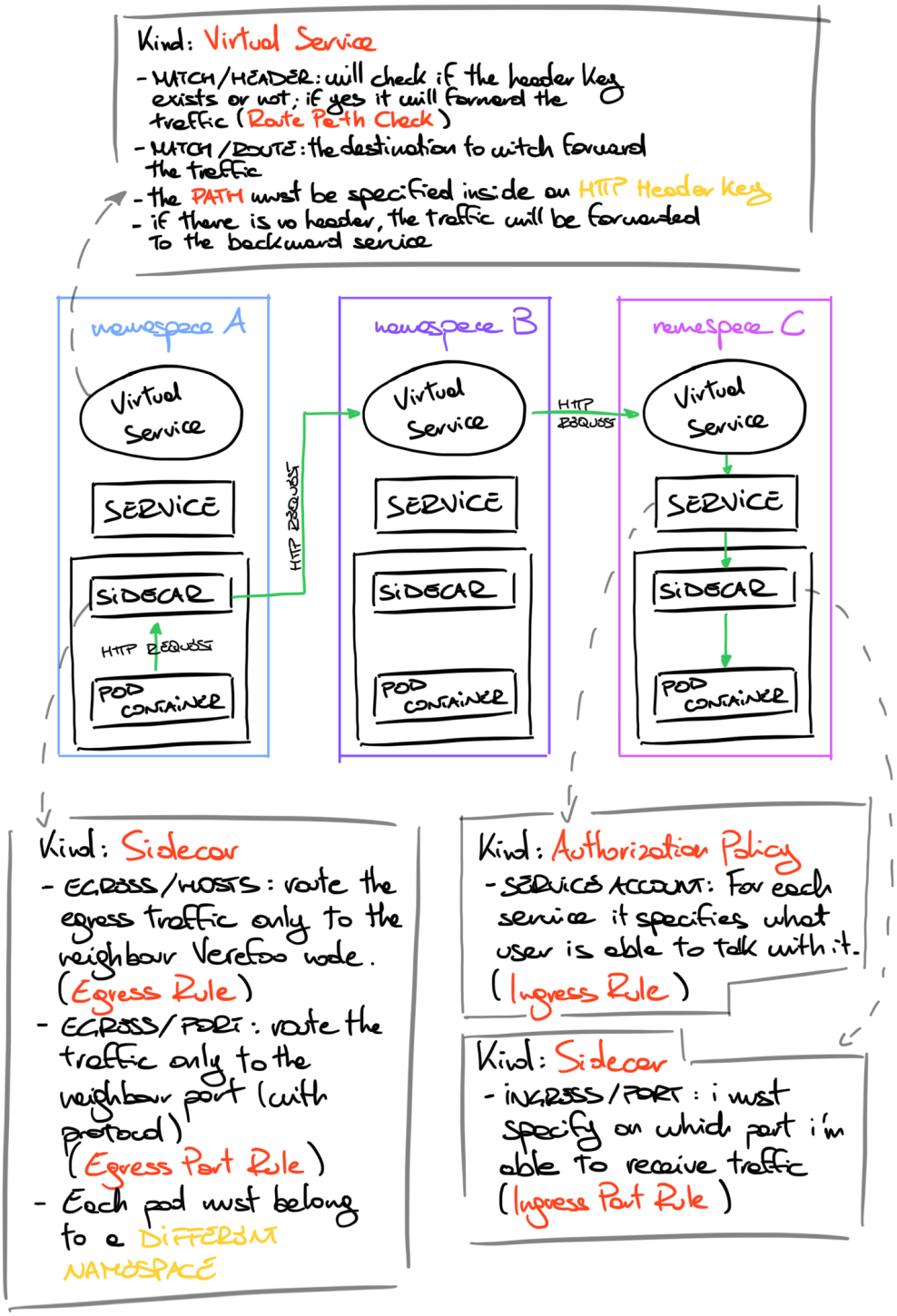

Figure 4.22: Forwarding Mechanism: design schema with the Istio Service Mesh

we can specify only a **destination namespace** and if we want to specify as destination only one pod we need to have only one pod per namespace.

As we already know, after having installed Istio inside our cluster, inside each Pod we will have a **sidecar** container instance: the Sidecar describes the configuration of the proxy container that manages all the incoming and outgoing traffic to the instance it is belonged to. By default, Istio will instantiate all sidecars in the cluster with the configuration required to communicate with all the others resources in the cluster, and in order to accept also the traffic on all the available ports on the associated container. The Sidecar configuration will provide to the user some features to change the default configuration and customize the set of protocols or ports that the proxy will accept (in ingress and egress) to the proxy. It is also possible to reduce the group of services that the sidecar can reach when it has to forward the egress traffic coming from the container it is belonged to.

By default, the Istio controller will instantiate an initial **Sidecar** inside each pod but it is possible to change its configuration. In the following example, we modified the Sidecar configuration available in the prod-us1 namespace with some custom settings that overwrites the default container ones, and will configure all the sidecars in the declared namespace (default in the example) to allow the outgoing traffic to the services in the istio-system, prod-apis and the prod-us1 namespaces.

```
---
apiVersion: networking . istio  . io /v1alpha3
kind: Sidecar
metadata:
  name: default
  namespace: prod-us1
spec:
  egress:
  - hosts:
    - "prod-us1/*"
    - "prod-apis/*"
    - "istio-system/*"
---
```

The sidecar resource APIs will let the programmer to customize two main features:

- **Egress hosts**: by customizing this property we enable the relative Sidecar to communicate with the services/resources linked to that namespace (we are

constraining the sets of services/endpoints with which we can communicate with): this is also a limitation because we are only able to allow the traffic to a specific namespace but, if we maintain one namespace per pod, we can specify an Egress Rule for a specific pod. This aspect will be treated later on in this paper. As we can see, the namespace constraint, already treated, comes from this limitation.

- **Egress port**: we can also customize not only the allowed destination ports, but also the destination protocol through which we can communicate with.

In the design schema presented in this chapter, we have a forwarding rule that redirect the traffic from the Pod A to the Pod C through the Pod B. In order to reflect this forwarding rule we can modify, for example, the Sidecar component of the Pod A:

- in the **Egress/hosts property** we have to specify the namespace B (pod B) as the only admitted destination
- in the **Egress/port property** we have to specify, eventually, some destination ports

By this way, all the HTTP traffic sent from the Pod container will be catch from the Sidecar and, eventually, redirected to the destination (if the Pod will try to communicate with a destination that wasn't specified inside the Sidecar, it will be refused by teh Sidecar itself).

After having configured properly the Sidecar component, for each pod we have to deploy another component called **Virtual Service**: it defines a set of routing rules that must be respected when a host is addressed. Each one of the routing rules will define some matching criteria for a specific protocol's traffic. If the traffic match that specific rule, then it will be sent to the desired destination service defined in the configuration.

The following configuration, will route all the HTTP traffic to the reviews service with the label "version: v1". Finally, with this configuration, when the sidecar will receive the HTTP requests with the path starting with /consumercatalog/ or /wpcatalog/, it will rewrite them with the new /newcatalog path and send them to the pods with label "version: v2".

---

apiVersion: networking . istio . io /v1alpha3

```
kind: VirtualService
metadata:
  name: reviews-route
spec:
  hosts:
  - reviews.prod.svc.cluster.local
  http:
  - name: "reviews-v2-routes"
    match:
    - uri:
        prefix: "/wpcatalog"
    - uri:
        prefix: "/consumercatalog"
    rewrite:
      uri: "/newcatalog"
    route:
    - destination:
        host: reviews.prod.svc.cluster.local
        subset: v2
  - name: "reviews-v1-route"
    route:
    - destination:
        host: reviews.prod.svc.cluster.local
        subset: v1
---
```

The Virtual service component will work "in front of" the service component. All the incoming traffic will be catch by the Virtual Service itself: by this way, it can inspect and forward all the traffic that receives. This component will represent the **forwarding core** of this architecture: it will be responsible of the forwarding mechanism inside the cluster.

The Virtual service component can inspect the HTTP request fields, such as the URI or the headers: by inspecting these two fields the Virtual Service understands if the traffic must be forwarded to another Virtual Service (another pod) or backward to the Service it is belonged to.

In the design schema showed in this chapter, for example, the pod-A's Sidecar will send some HTTP traffics to the pod-B's namespace and then the Virtual Service (the one inside the "purple" area) will understand if it has to forward the traffic to the pod-C (to its Virtual Service) or to the pod-B.

Once we have to create the VirtualService configuration, we have to consider the following properties/aspects:

- **Match Header**: it will check if the header key exists or not; inside this architecture the Header will contain the information regarding a specific path. Here there is one of the strongest constraints of this architecture: **the path** (and so, the information to understand which forwarding rule it must be followed) will be a custom label contained inside the HTTP header of the request. For this reason, we have to add an header (representing the "coloured" path) to an HTTP request in order to "force" a forwarding mechanism. If no path/header is found, the traffic will be forwarded to the backward service (it will be considered as traffic directed to the pod it belongs to).
- **Match Route**: the destination to witch forward the traffic. It can be enriched with some interesting behaviours like the URI-rewriting feature: you can rewrite the header/URI of the request to eventually match the path expected by the destination. All the traffic that match one of the routes, defined in this property, will be forwarded to the route-destination, defined in the configuration file.

Another component that we have to configure is the **AuthorizationPolicy**: we have already introduced it in the previous chapters. As we know, the Istio Authorization Policy will perform access control on the incoming/outgoing traffic in the mesh. This resource supports allow/deny or custom actions for the access control.

For example, with the following authorization policy configuration we'll deny all the requests make to the workloads in the namespace foo.

```
---
apiVersion: security . istio . io /v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-all
  namespace: foo
spec:
---
```

The following authorization policy allows all requests to workloads in namespace foo.

```
---
apiVersion: security . istio . io /v1beta1
kind: AuthorizationPolicy
metadata:
 name: allow-all
 namespace: foo
spec:
 rules:
 - {}
---
```

Inside the proposed architecture, the Authorization Policy resource is needed to control if the traffic can be accepted or not: for each service it specifies what user is able to talk with. So, to correctly use the authorization policy, we will need to associate a different user for each pod/service (remember that the authorization policy can refers only to a specific **Service Account**). By this way we can control which traffic is admitted and so we are able to force some ingress rules/checks.

Finally we have to specify, for each Sidecar, also some Ingress Rule to specify on which port the sidecar is able to receive some traffic.

To summarize, the design that was introduced, in this chapter, is characterized by the following aspects/constraints:

- we need to specify a different namespace for each pod.
- the path information is contained inside the HTTP Header. It will be a custom HTTP header, interpreted as the colour of the path to which the packet belongs to.
- we need to specify an user per pod (Service Account). It will be needed by the respective Authorization Policy.
- the forwarding mechanism can be only performed over the application level.
- good modularity: the forwarding logic and the business logic are totally separated. By this way we are not constrained to build the forwarding logic inside the application business logic. As we will see better in detail later on, with the next two designs, that will be analyzed, we weren't able to guarantee this aspect: one of the strongest weakness of the next designs is that the business and the forwarding logic will be put in the same place and must be built, by hand, by the developer.

- confusing forwarding management: the forwarding logic is spread over many different components (Virtual Service, Sidecar and Authorization Policy).

Considering the whole forwarding logic management and the network working level this architecture is not suitable for the purposes of this thesis: because of the fact that the forwarding logic is spread over many different components and we can't perform the packet's forward below the application level, this architecture cannot reflect, in a proper way, the forwarding mechanism produced by Verefoo. For the reasons showed before, we moved the scope of these researches to other technologies and mechanisms, like the serverless ones.

Argo Design

As we introduced yet, the Argo project is an Open source Kubernetes native project that includes tools to manage workflows, events, CI and CD.

In the architecture design that will be showed later on, were mainly used two tools: the Argo Events and the Argo Workflow.

When we talk about the Argo events we mean: **event bus** (the one that manages all the events, a bridge between sources and triggers), **event sources** (the source of events that publish some messages, it is a sort of event-publisher) and **triggers** (the entity that trigger certain actions depending on the events created through the event sources; it is a sort of event-consumer).

The event source can be of many different types: for example, we can create a web hook as the source through which we can generate events, like in the following configuration file:

```
---
apiVersion: argoproj .o/v1alpha1
kind: EventSource
metadata:
 name: webhook
spec:
    service:
        ports:
            - port: 12000
              targetPort: 12000
    webhook:
        devops-toolkit:
```

```
      port: "12000"
      endpoint: /devops-toolkit
      method: POST
---
```

Here we have an **EventSource** called webhook: we are going to expose an entry point in our cluster through which we can send **HTTP requests** that will become the source of our events. Of course, we need a service in our cluster that will expose a port (the service will be automatically created by the Argo framework). The event-type is the webhook one (there can be others): inside we have to specify the port through which that webhook should be exposed, the entry-point (in this case is called "/devops-toolkit") and finally the communication method which is set to POST.

After including this component in our cluster, we will be able to send HTTP requests to that entry-point because the event-source "webhook" will be created with both a service and a pod. The Event Source will listen to some events but we need also a mechanism to do something when those events will be received. For this reason, we need to deploy a **Sensor**: this component will be linked to a specific EventSource and each time someone will send a web hook event this sensor will be activated to make some operations.

To summarize, the purpose of the Sensor is to connect a trigger with an event source, as shown in the next example:

```
---
apiVersion: argoproj .io /v1alpha1
kind: Sensor
metadata:
  name: webhook
spec:
  template:
    serviceAccountName: operate-workflow-sa
  dependencies:
    - name: test-dep
      eventSourceName: webhook
      eventName: example
  triggers:
    - template:
        name: webhook-workflow-trigger
```

```
        k8s:
          group: argoproj . io
          version: v1alpha1
          resource: workflows
          operation: create
          source:
            resource:
              apiVersion: argoproj . io /v1alpha1
              kind: Workflow
              metadata:
                generateName: webhook-
              spec:
                entrypoint: whalesay
                arguments:
                  parameters:
                  - name: message
                    # the value will get overridden by event
    payload from test-dep
                    value: hello  world
                templates:
                - name: whalesay
                  inputs:
                    parameters:
                    - name: message
                  container:
                    image: docker/whalesay: latest
                    command: [cowsay]
                    args: ["{{inputs.parameters.message}}"]
          parameters:
            - src:
                dependencyName: test-dep
              dest: spec . arguments . parameters .0. value
---
```

- The **dependencies property** defines the EventSource to which the Sensor belongs to.

- The **trigger property** defines the operation to be performed once the event will be received.

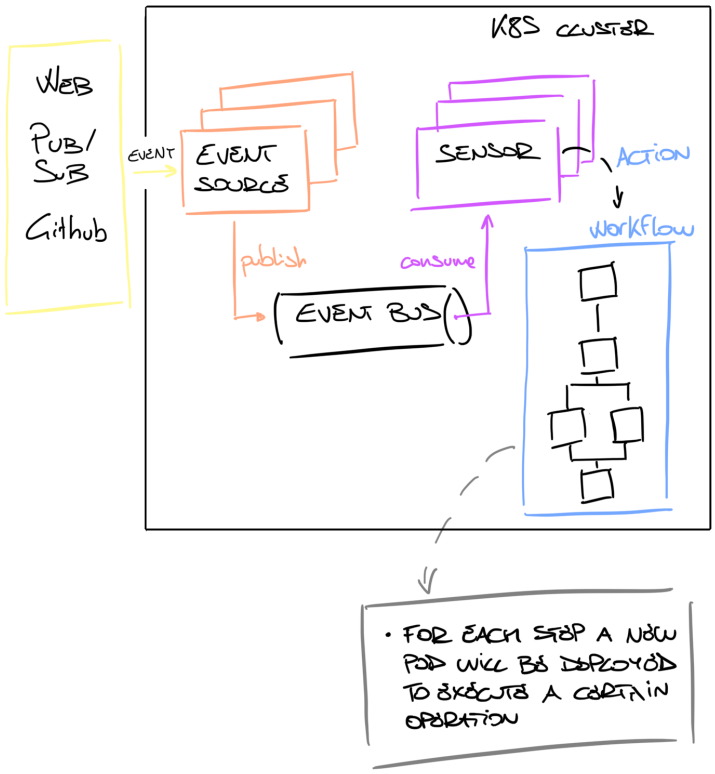

Figure 4.23: Forwarding Mechanism: Architecture Design Schema with Argo

Now we have to understand, how will this architecture reflect the forwarding mechanism produced by Verefoo?

As we can see from the previous image, the core component of this architecture is represented by the **Argo Workflow** component: as mentioned before, the workflow component represents a pipeline of tasks that can be run in parallel, sequentially or with some custom if-then-else logic. Inside this architecture, the workflow will execute all the controls that must be performed on the received packets: as we'll better understand later, this solution is similar to the one proposed with the Kubeless architecture because the forwarding logic will be written inside the Workflow business logic, and so inside the code that the user has to develop and deploy to the cluster.

To sum up, inside the workflow business logic, we'll have to write the mechanism to understand how to forward and redirect the traffic once the packets will be received.

The **event** will be an HTTP one and represents the request performed by some pods inside the cluster: as we can understand, once the pod will perform an HTTP request, the request will be catch by an **Argo Sensor** which will finally trigger an **Argo Workflow** that must perform some controls on packet itself.

We can say that the architecture will perform, sequentially, the following steps:

- The sensor will receive an HTTP event
- The sensor will trigger a specific action: in this case we'll be an Argo Workflow
- The workflow will contain all the controls that must be performed to process the received packet and, eventually, forward it to a different destination.

Inside the workflow configuration, there can be specified 6 different types of templates: these templates define the work that must be done. One of the templates, that could be used, is the **script-template**: it is the same as a container resource in which the user can directly define a script in-place (a piece of code) to be executed by invoking it. The script will be saved into a file and executed on demand.

Here in the following, there is an example of a script:

```
---
  - name: gen-random-int
    script:
      image: python:alpine3 .6
      command: [python]
      source: |
        import random
        i = random.randint(1, 100)
        print(i)
---
```

Inside the architecture, the script represents one of the steps that will compose the workflow itself and, inside it, we can specify the operations that must be performed after the event will be received: to summarize, we receive a certain event, we perform some computations on it and then we will forward it to the next steps. So, the source-script section will be used to write the forwarding logic and process the events once they will be received.

As mentioned, thanks to the Argo Workflow resource, we can specify (also graphically) the steps necessary to process the incoming events but, as already

mentioned, this solution wasn't flexible enough to reflect the forwarding mechanism produced by the Verefoo tool. There are two main problems with this solution:

- The main purpose of the forwarding mechanism is to process the requests step by step following a sort of pipeline (by processing and forwarding the traffic to the next step), like we were doing through the Argo Workflow but, on the other side, the forwarding mechanism need to be performed by looking at the network and transport layers fields: with the Argo workflow, we were simply processing and passing the events as input to the next steps without the possibility of looking inside them. Like in the architecture proposed with Istio, we are obliged to work over the application layer by inspecting the object used by the Argo framework.
- Another big problem to deal with is that, to execute the workflow, Argo creates and instantiate a pod per step: this aspect perfectly match with the CI/CD scenarios in which this framework is proposed to work with. In a CI-CD environment, we don't need real-time performances because these tasks will be carried out periodically, each weeks or months: inside these scenarios we don't have to care about duration or performances because these operations will be performed rarely or scheduled. The scenario proposed in this paper is very different from the previous one: we have to consider the network communication between the various pods so, we need a performing and real time mechanism to deal, potentially, with the enormous amount of requests that the cluster has to manage. If we have to wait the pod creation each time we send a request the overall mechanism will become unmanageable and long-time consuming.

For the reasons showed before, we searched and studied another serverless framework to reflect the forwarding logic inside the Kubernetes cluster: Kubeless.

Kubeless Design

The following architecture is the last design that was proposed to overcome the limits introduced by the two previous architectures. This design will use the Kubeless framework: as we introduced in the previous chapters, Kubeless is a Kubernetes-native serverless framework that lets the user deploy small bits of code (functions) without having to worry about the underlying infrastructure.

Kubeless will be deployed over the Kubernetes cluster and will listen to the serverless functions that will be written by the programmer.

Before continuing, we have to understand how these functions can be used to reflect the forwarding mechanism. In the same way done in the previous architecture,

the logic to forward or process the incoming traffic will be put inside the function written by the user. The function will receive some parameters and one of them is the event/message that was sent. As we can easily understand, like before, the forwarding and the business logic will be in the same place.

By inspecting the event object, we'll be able to perform some controls and make decisions on how to forward it.

Here we have the example of a serverless function, written in Java, that can be deployed inside the cluster.

```
package io.kubeless;

import io.kubeless.Event;
import io.kubeless.Context;

import com.google.gson.Gson;
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.annotations.Expose;
import com.google.gson.annotations.SerializedName;

public class Hello {
    public String sayHello(io.kubeless.Event event, io.kubeless.Context
    context) {

      try {

          Gson gson = new Gson();

          // get input element
          String requestData = event.Data;
          JsonObject jRequest = gson.fromJson(requestData,
    JsonObject.class);

          JsonElement jMember = jRequest.get("source");

          if(jMember != null) {
             String member = jMember.toString();
             return member;
          }
```

```
            return "Source field wasn't found!";

        } catch (Exception e) {
            return e.getMessage();
        }
    }
}
```

In the previous example, the function will receive an event (here through the HTTP protocol), compute it and return a response.

Inside the function's logic, we can put some forwarding-mechanism decisions: for example, by inspecting the event fields, we could get some information regarding the "colored path" and forward the traffic to the proper destination function (remember that, for each function, we will have a pod and a service). Of course, this aspect must be customized and thought by the developer who choose the way through which perform the forwarding itself.

As we can see from the following example, inside the function we can mix the forwarding logic with the business one: the forwarding mechanism (like in the Istio architecture design) is performed over the application level. We can't inspect the lower packet fields (regarding the network or the transport layer) but simply the fields contained in the event that was received by the function, which is the object supported by the framework itself.

```
package io.kubeless;

import io.kubeless.Event;
import io.kubeless.Context;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

import com.google.gson.Gson;
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.annotations.Expose;
import com.google.gson.annotations.SerializedName;

public class Hello {
```

```
public String sayHello(io.kubeless.Event event, io.kubeless.Context
context) {

  try {

     Gson gson = new Gson();

     // get input element
     String requestData = event.Data;
     JsonObject jRequest = gson.fromJson(requestData,
JsonObject.class);

     JsonElement jMember = jRequest.get("destination");
     String dst;

     if(jMember != null) {
        dst = jMember.getAsString();
     } else {
        return "destination field wasn't found";
     }

     if(dst.equals("pod-b")){

        URL obj;
        obj = new URL("http://pod-b:8080");
        HttpURLConnection con = (HttpURLConnection)
obj.openConnection();
        con.setRequestMethod("GET");
        int responseCode = con.getResponseCode();
        System.out.println("GET Response Code :: " + responseCode);
        if (responseCode == HttpURLConnection.HTTP_OK) { // success
           BufferedReader in = new BufferedReader(new
InputStreamReader(
                 con.getInputStream()));
           String inputLine;
           StringBuffer response = new StringBuffer();

           while ((inputLine = in.readLine()) != null) {
              response.append(inputLine);
           }
           in.close();

           // print result
           return "Received response from B: " +
response.toString() + " with status-code: " + responseCode;
```

```
            } else {
               return "GET request didn't work; status-code received
    from B: " + responseCode;
            }

         } else if (dst.equals("pod-d")) {

            URL obj;
            obj = new URL("http://pod-c:8080");
            HttpURLConnection con = (HttpURLConnection)
    obj.openConnection();
            con.setRequestMethod("GET");
            int responseCode = con.getResponseCode();
            System.out.println("GET Response Code :: " + responseCode);
            if (responseCode == HttpURLConnection.HTTP_OK) { // success
               BufferedReader in = new BufferedReader(new
    InputStreamReader(
                     con.getInputStream()));
               String inputLine;
               StringBuffer response = new StringBuffer();

               while ((inputLine = in.readLine()) != null) {
                  response.append(inputLine);
               }
               in.close();

               // print result
               return "Received response from C: " +
    response.toString() + " with status-code: " + responseCode;
            } else {
               return "GET request didn't work; status-code received
    from C: " + responseCode;
            }

         } else {

            return "destination: " + dst + " is not supported";

         }
      } catch (Exception e) {
         return e.getMessage();
      }
    }
}
```

The previous example is the serverless function developed for a pod called "Pod-A". To better understand its structure, we need to inspect the cluster's topology that was choosen for this example. Here we have the topology schema that was considered:

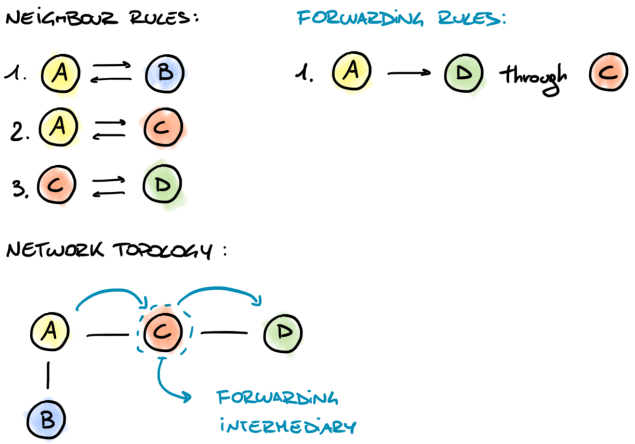

Figure 4.24: Forwarding Mechanism: Kubeless design architecture topology example

Inside the Pod-A function, after inspecting the "destination" field contained in the received event, the computation will pass through an if-case logic:

- if the destination is the **Pod-B**: the request will be redirected to the service that expose the DNS name "pod-b". We can suppose that the pod-B belongs to the pod-A's neighborhood and for this reason the request will be forwarded directly to the Pod B.
- if the destination is the **Pod-D**: the request will be forwarded to the Pod-C. The Pod-A can't communicate directly to the Pod-D but has to communicate through the Pod-C: The Pod-D doesn't belong to the Pod-A neighbours. This represents an example of how we can enforce the forwarding rules inside the serverless functions.

We can easily deploy all the needed functions inside the cluster and, eventually, enrich them with some forwarding logic. Otherwise the simplicity and flexibility of this technology we have to deal with some already known problems:

- The business logic is mixed with the forwarding logic: the mechanism to forward the traffic is performed and wrote inside the serverless function. With large projects or logic, it could become difficult to maintain such a mechanism.

- The forwarding logic must be written by the programmer, with the risk of wrong configurations.

- There is no way to inspect the request's lower payloads, like the network and transport payload. For this reasons, we are obliged to work over the application layer.

- The mechanism proposed doesn't follow a hundred percent compatible forwarding mechanism: we aren't redirecting any traffic, but simply check the destination, prepare a new object, and send a new object to the next destination.

For the problems showed before, also this design can't reflect the forwarding mechanism inside the kubernetes cluster.

4.2.5 Final considerations

As shown in this chapter, we followed and studied many different technologies with the aim of reflecing the forwarding mechanism inside the Kubernetes cluster.

All the three designs that were analyzed, share a common problem: we can simply work over the application level. We aren't able to inspect any lower details regarding the requests received: for these reasons the forwarding mechanism can't be reflected (in a way suitable for the Verefoo tool) inside the Kubernetes cluster.

From the previous chapter, we know that we are able to reflect the neighbour rules (point-to-point communication) inside the Kuberentes cluster by using the Calico Network Policies. Because of the impossibility of finding a proper way to forward the traffic inside the cluster, we can only reflect the Neighbour Mechanism inside Kubernetes.

Now we have to understand how to extend the application developed until now to support the Neighbour Mechanism: in which scenarios we can work and what changes we must perform over the application's architecture.

As we know, the Verefoo tool can retrieve a list of nodes with some point-to-point communication rules (neighbourhood). If we consider the Verefoo graph as a collection of clusters, not pods anymore, the neighbour rules could be interpreted as the network rules between the Kubernetes clusters, to allow or not the communication between them.

The goal for the next chapter is to extend the application, developed until now, to work inside an **inter-cluster scenario**: the application has to deal with a chain of clusters (not only one cluster) and has to reflect the neighbour rules, produced by Verefoo, between them (not between the pods).

Chapter 5

Neighbour Mechanism: Inter-Cluster use case

As we have understood from the previous chapters, we have found a strong limitation: inside the kubernetes cluster, we have no way to reflect the Verefoo forwarding rules, by inspecting the traffic on the lower ISO/OSI levels (network and transport). For the reasons shown before, we are able to only reflect the **Neighbour Mechanism**, provided by Verefoo, inside the K8S cluster.

Until now, we have only worked inside the cluster (we had a collection of pods which had to talk to each other) and, for this reason, interpreted the neighbour (and the forwarding) mechanism as a way to enforce some network rules/constraints between the internal pods. We can call it **Intra-Cluster** scenario (inside the cluster).

To outline the scope for the next steps, and before starting with the code development to extend the application functionalities, we had to better understand the application working environment: in which scenarios do we have to work with? Are we obliged to work only inside a specific cluster, as it was done until now, or can we work also with a **chain of clusters**?

Due to the questions raised above, purpose of this chapter is to extend the application, developed until now, to work with a chain of cluster and use the Network Policy resources as a way to deny/allow the network communication between a collection of K8S clusters.

Because of this decision, we had to reinterpret some main concepts followed during the work done until now, in particular:

- The Verefoo graph will be a collection of clusters, not nodes
- The Verefoo neighbour rules will be some point-to-point communication constraints between the clusters: they will tell which cluster can communicate to another one
- The Verefoo forwarding rules won't be considered anymore
- We can't consider only a Kubernetes cluster but a collection of them

We just want to anticipate that, due to the technological limitations due to Kubernetes, the inter-cluster scenario is indeed feasible due to the mechanism, used by Kuberentes itself, of networking within the cluster. For these reasons, in the next chapter we decided to go back to the intra-cluster scenario, with which we opened the studies for this thesis.

5.1 Verefoo: what's changed

The Verefoo concepts were the first aspects that we had to reinterpret. To better understand what's changed, we can compare the previous graph topology-architecture with the newer one.

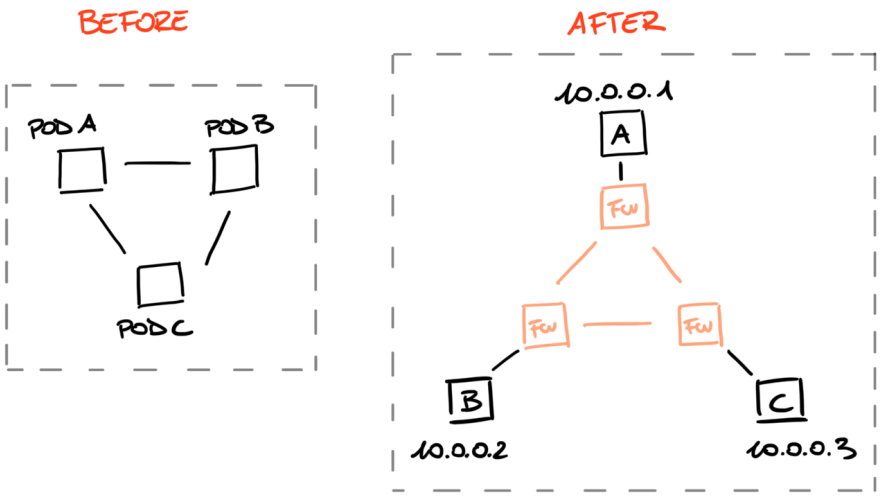

Figure 5.1: Inter-Cluster scenario: Verefoo topology changes

In the **older version**, the Verefoo graph is simply considered as a collection of nodes: each node is able to communicate to all the others (or to a small subset) through the neighbour rules (the mechanism to tell which node can communicate to another one). The graph will be composed by simple nodes that will get the name from the pod deployed in the Kubernetes cluster, to which the graph is belonged to.

In the **newer version**, first of all, we had to change the node interpretation that we followed until now: as mentioned, the node (the black one) will represent the cluster, not the pod; for this reason, the name of the Verefoo node will be the public IP address of the machine in which the Kuberentes cluster it is actually up and running. To summarize we'll interpret the graph as collection of kubernetes clusters, not pods. In the newer version, we also introduced another component, the Firewall. Because of the changes and the choices that we made inside Kubernetes (we'll be largely covered later on), we must instantiate a firewall per each node (cluster). The node will be connected only to its firewall and the firewall nodes will be connected all together (through the neighbour rules).

From a topology point of view this architecture reflects what we have inside each cluster: as we'll discuss later on, inside each cluster we define some Network Policies to constraint the networking (the traffic that is allowed or not). It is like having a large firewall that is spread all over the clusters: for this reason we decided to "reflect" this aspect also inside the Verefoo graph by instantiating multiple firewalls (one per each node/cluster).

Inside each firewall we'll define the networking rules we have to reflect inside the Kubernetes cluster. Here, in the following, we have an example of a Firewall configuration supported by Verefoo:

```
<?xml version="1.0" encoding="utf-8"?>
<node functional_type="FIREWALL" name="30.0.0.1">
    <neighbour name="10.0.0.1"/>
    <neighbour name="10.0.0.2"/>
    <configuration description="A simple description"
   name="configurationName">
        <firewall defaultAction="DENY">
            <elements>
                <action>ALLOW</action>
                <source>10.0.0.1</source>
                <destination>10.0.0.2</destination>
                <protocol>ANY</protocol>
```

```
                <src_port>*</src_port>
                <dst_port>*</dst_port>
            </elements>
        </firewall>
    </configuration>
</node>
```

The networking rules will be defined inside the configuration firewall attribute. Inside this property we can define a list of elements, each of them will define a 5-tuple rule to specify which traffic is allowed or not. In the previous example, we are allowing all the traffic that comes from the IP 10.0.0.1 and it is directed to the 10.0.0.2 address.

In the firewall configuration we can also specify the default action that must be followed (ALLOW or DENY): here we have set DENY as default action.

As we can understand, inside the firewall configuration we'll put all the rules that must be reflected by the Kubernetes NetworkPolicy: each time the Verefoo FW configuration will change, we'll reflect the new rules on the respective cluster (identified by the IP of the node this firewall will belong to).

5.2 Kubernetes: what's changed

As already mentioned, now we must consider a group of clusters spread all over the cloud. Each cluster will be represented by the public IP address of the machine in which it is running: because of this, each pod inside the cluster will share the same public IP address (the one of the machine in which the cluster is running).

By sharing the same IP, inside Verefoo we don't have any way to distinguish a pod from another one: Verefoo doesn't have any kind of mechanisms (or more specific XML tags) that can be used, inside the graph configuration, to define a finer topology (for example, a subset of nodes that belong to an upper-grade node). Using the private IP addresses, assigned to each pods, wouldn't be a good idea because, as we know, many pods can use the same IP, even if inside different clusters.

To summarize, the problem is that we can't identify all the pods inside the cluster. For this reason, we decided to consider only one pod per cluster. We changed also the Pod's event interpretation: these events will be used as a way to communicate to Verefoo if the cluster is running or not. For example, once we delete a pod (and no pod is actually running inside the cluster) we communicate a "disconnection" event to Verefoo and it will delete the cluster node from its graph.

Another important aspect that we have to deal with is linked to the NetworkPolicy resource: as we discussed yet, this component will be used to reflect the Firewall 5-tuple rule. Here in the following, we have a NetworkPolicy example:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

As we can see, both in the ingress and the egress properties, we can specify the IP address with which we can communicate or not. For example, as we'll do later on, inside the **ipBlock** property we can put the IP address of the cluster with which we can communicate (both in ingress or egress). So, by using the ipBlock property, we are able to reflect the 5-tuple rules produced by Verefoo, and communicate to the cluster through the application.

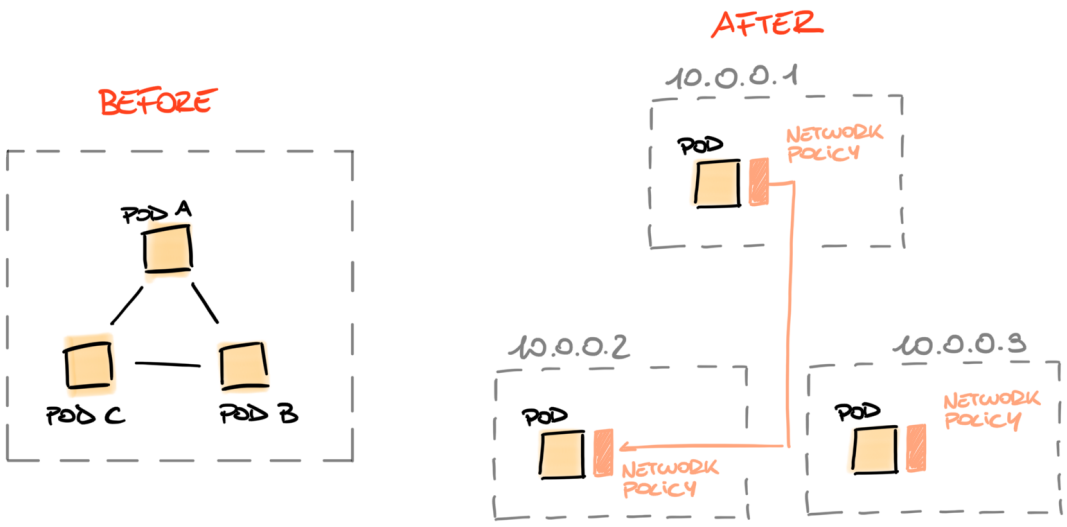

Figure 5.2: Inter-Cluster scenario: Kubernetes topology changes

In order to understand how the Network Policy was used for this purpose, we can inspect the previous schema: inside each cluster, there will be a pod and a network policy resource that is used define the network rules to communicate with the other clusters (by default, each cluster will be isolated from the outside).

For example, in the previous example, the cluster with the IP 10.0.0.1 can send traffic to the cluster with 10.0.0.2 as network address, but not the vice-versa, and the cluster 10.0.0.3 cannot send or receive traffic from the cloud (it will be isolated).

5.3 Architecture: what's changed

Because of the changes performed on Kubernetes and Verefoo, we had to change something also inside the architecture developed until now.

As we can see by the following schema, new components were introduced as the **Dep Master** and **Slave** (the components responsible for the network policies deployment) and some of the older ones changed their disposition (the Agent Master and Slave switched their positions).

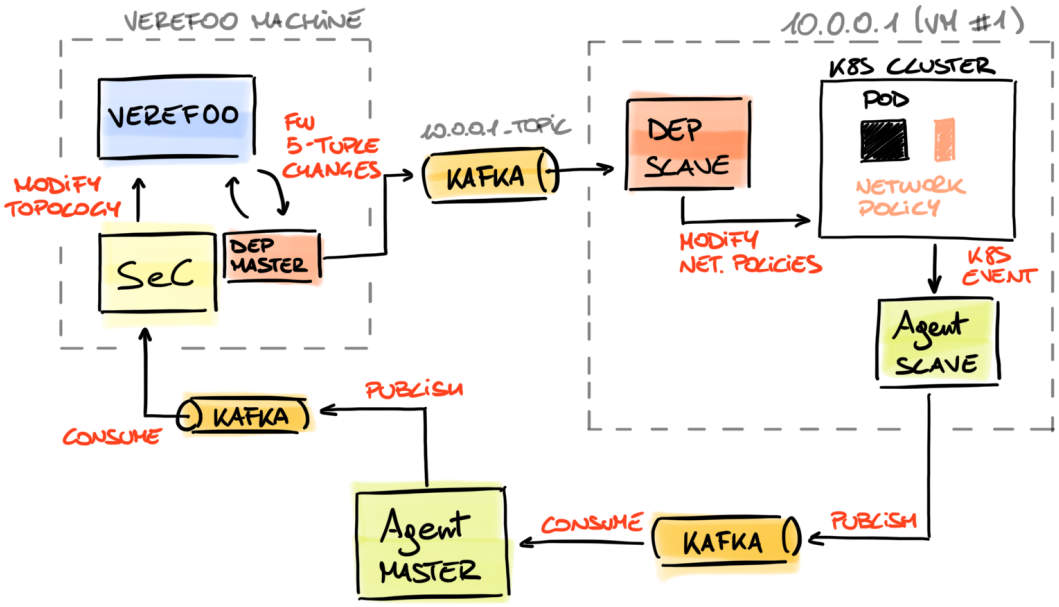

Figure 5.3: Inter-Cluster scenario: Application architecture changes

The main difference from the previous architecture is represented by the new Deployment (Dep) components:

- Inside the Verefoo machine, the Security Controller will manage a secondary thread called **DepMaster** which is responsible to listen to the Firewall changes inside Verefoo: each time a firewall will change its configuration (a new 5-tuple rule is added/removed or modified) it will publish, on a specific **Kafka topic**, the new event (the name of the topic will be the public IP of the machine that must change its network configuration inside the Kubernetes cluster). To summarize, the DepMaster micro-service will control all the DepSlaves components inside each Virtual Machine.
- We added a new Kafka Topic per each Kubernetes cluster: as already mentioned, it will get the IP Address of the machine in which the Kubernetes cluster is running.
- Inside each hostname machine (the bare-metal or virtual machine in which a Kubernetes cluster is up and running), we introduced the **DepSlave** component: it will listen and consume the messages from the Kafka Topic to which it is linked to and will deploy the new rules inside the local Kubernetes instance. To summarize, it will translate the Kafka Event into a specific NetworkPolicy configuration that will be deployed into the cluster.

All the new application components represent the logic proposed to enforce the Verefoo network changes inside each Kubernetes cluster.

Another important change to mention is the position-switch between the Agent Master and Slave: now, potentially, we have to deal with many different Kubernetes clusters so the AgentSlave will represent the component instance that we have to duplicate, inside each machine, and which must communicate with the AgentMaster.

5.4 Ingress traffic problem

Once we started studying the overall solution we encountered some problems linked to the kubernetes ingress networking policy.

First of all we needed to expose the pod outside of the cluster in order to be reachable from the external environment: in order to do that we could exploit the **Service API** component offered by Kubernetes (LoadBalancer or NodePort components).

The Service component was putted in front of the pod and intercepted all the incoming traffic to forward it to the backward pod. As we already know, the service will expose an external IP, the same IP that will be used by some external clients to reach the pod itself. Here we encountered the first problem with this solution: as mentioned in the blog titled "A Deep Dive into Kubernetes External Traffic Policies", written by Andrew Sy Kim [**deepdivekubtrafficpolicies**], the kube-proxy component will **Source-NAT** the client source IP with the cluster's node IP. By this way, the proxy will redirect all the traffic to the pod with the IP address of the local node, not the one used by the client: this will led to a big problem, especially if we are using in our solution the NetworkPolicy. By exploiting the CIDR ip ranges offered by the NetworkPolicy component, we need to specify the IP address of the incoming traffic we can accept/deny. Fortunately, by setting the property called **service.spec.externalTrafficPolicy** to "Local" we are able to avoid the client' source IP to be NAT-ed with the node's IP. By this way we can specify the client source IP directly inside the NetworkPolicy.

Another problem that, unfortunately, wasn't solved is related to the IP that we had to consider for the overall architecture: the only way to communicate with a pod is by exposing it through a service and using the public IP exposed by the service itself. The main problem is that, to send the traffic we must specify the service's public IP and to receive the traffic we must use the pod's IP address.

From an algorithmic point of view (the business logic that must be developed inside the Dep Master and Slave components), there could be the possibility to associate two different IPs to a specific pod: one IP must be used to enforce the Egress traffic (service's IP) and the other one that must be used to enforce the Ingress traffic (pod's IP). The main problem is related to the mechanism used by Verefoo to manage the rules inside the firewall node: two different IPs will be interpreted as two different machines running in the cloud. By this way, the overall mechanism could be compromised.

To summarize, we strictly need to refer to each pod only by using a single IP address. As we will understand later on, this aspect is still open and wasn't solved for the thesis' purpose.

5.5 Egress traffic problem

Another big problem to deal with was the Kubernetes egress networking mechanism.

The traffic exiting from a certain pod won't be intercepted, as in the ingress direction, by the Kubernetes service: if a pod will send some traffic outside the cluster, all the packets will be intercepted by the kubernetes node interface and will be source NAT-ed with the source ip of the node in which the pod is running. For example, the Calico CNI will offer an instrument to disable or enable the SNAT (Source NAT) for specific destination machines [57]: in order to do that, we can used the **IPPool API** offered by the Calico library.

```
apiVersion: projectcalico .org/v3
kind: IPPool
metadata:
  name: default-ipv4-ippool
spec:
  cidr: 192.168.0.0/16
  natOutgoing: true
```

In the previous yaml configuration example, we have created an egress ip pool for which the NAT was enabled: all the traffic that will be sent to the ip range 192.168.0.0/16 will be source natted and the source ip will be substituted with the external node's ip address. In order to prevent the NAT to change the source ip address we can exploit the same component.

For example, in the following example, we are disabling the NAT mechanism from the pods to 10.0.0.0/8 IPs.

```
apiVersion: projectcalico .org/v3
kind: IPPool
metadata:
  name: no-nat-10.0.0.0-8
spec:
  cidr: 10.0.0.0/8
  disabled: true
```

Of course, we have to ensure that the network between the 10.0.0.0/8 adresses and the cluster is able to route the pod IPs.

Even if we were able to solve this aspect through the Calico API components, the problem opened in the previous section is still open: because of the mechanism used by the Kubernetes Networking, we need to use two different IPs (one referred to the Pod and the other one to the Service). We can't manage all the networking mechanism with only one IP address per pod.

Because of this limitation, we abandoned the idea of enforcing the network communications between different clusters to focus on another scenario: the initial **Intra-Cluster** networking model.

Chapter 6

Neighbour Mechanism: Intra-Cluster use case

Because of the network model adopted by Kubernetes, as shown in the previous chapter, we weren't able to work in a multi-cluster scenario: we encountered some limitations on the appliance of the network rules between a collection of clusters.

For the reasons showed before, we come back to the scenario with which we started this paper. For Intra-Cluster scenario we mean to work only inside a specific cluster and to enforce the network rules between the pods that compose the cluster itself. As already mentioned, this is the simplest scenario that we could think about: only one cluster, composed by many different pods which communication is established and allowed through the Network Policies resources.

All the concepts that were changed during the previous chapter (to match an Inter-cluster scenario) were refreshed to the version used at the beginning:

- The Verefoo graph will be a collection of nodes, not clusters
- The Verefoo neighbour rules will tell which node can communicate or not to another one
- We will consider only one cluster
- The Kubernetes cluster will be composed by a collection of many different pods.

As we can understand, now we don't carry anymore on the networking between different clusters, but simply on the internal cluster networking: we will work only inside the **private** cluster network domain.

6.1 Verefoo: what's changed

As it was already anticipated, each Verefoo node will represent a specific Kubernetes pod, up and running. From an architectural point of view, is the same topology that was considered in the beginning of this paper.

The major changes will involve the way the firewall will be allocated inside the Verefoo graph. For this reason, we studied and proposed two ways of allocating the firewalls inside the graph:

- **Star-centered firewall**: we allocate only one firewall in the center of the graph and each pod will be connected to the firewall itself. The firewall will contain all the rules referred to all the nodes that compose the graph. The Network policies, that we must allocate per each pod, will be extracted from the firewall configuration rules. Inside this configuration, the Verefoo firewall will correspond to many different NetworkPolicies resources inside the Kubernetes cluster.

- **Distributed firewall**: we allocate one firewall per each node (like it was performed in the inter-cluster communication model). By this way the firewall will contain only the rules that refer to the pod with which it is belonged to. Each Kubernetes NetworkPolicy will correspond, exactly, to a Verefoo firewall.

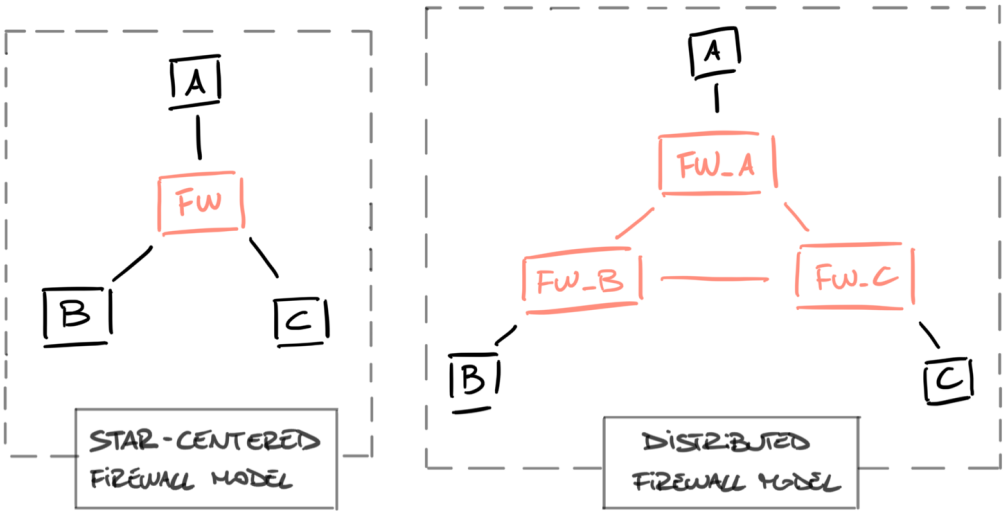

Figure 6.1: Intra-Cluster scenario: Verefoo Firewall models proposed

6.2 Kubernetes: what's changed

Now we have to consider many different pods inside the cluster. Because of this aspect, inside the cluster, we have also many different Network Policies (one per pod). One of the biggest difference from the previous networking model is that, inside the NetworkPolicy, we won't use anymore the IP range with the CIDRs: to deny/allow the traffic between the pods, we can use directly some specific labels to recognize the pods that were instantiated with them. Remember that now we will work only inside the cluster, for this reason we are able to recognize the pod by using the Kubernetes APIs.

For example, in the following NetworkPolicy configuration file, we can see that through the **podSelector** label we can specify, for both the ingress and egress directions, with which pod we can communicate.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - podSelector:
        matchLabels:
          role: frontend
```

```
ports:
- protocol: TCP
  port: 5978
```

With the same mechanism, through the podSelector label, we can specify also to which pod we can associate the NetworkPolicy.

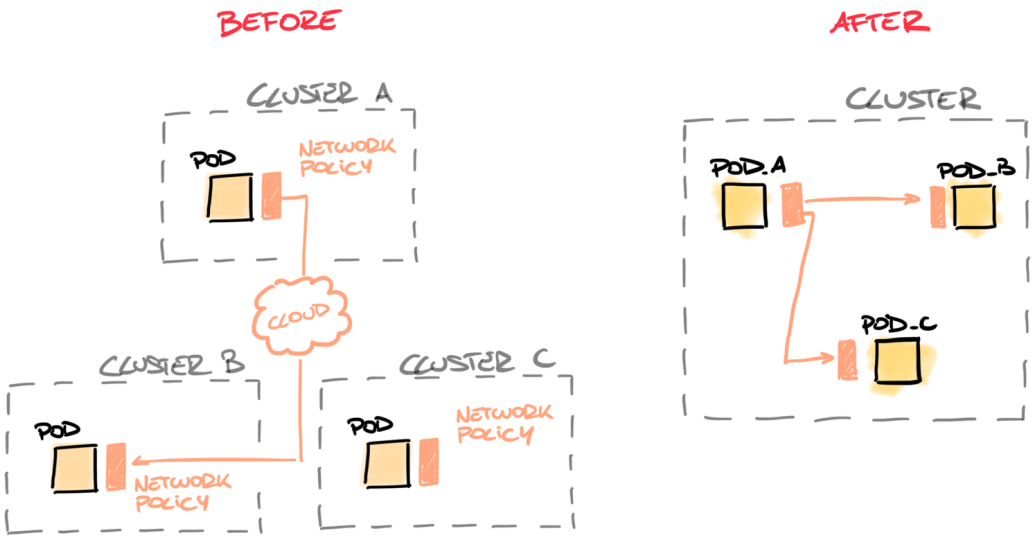

Figure 6.2: Kubernetes Cluster changes

From an architectural point of view, inside Kubernetes small changes took place: the only difference between the previous networking model is that now inside the cluster we have many different pods and network policies resources.

6.3 Architecture: what's changed

As we can see from the following schema, the overall component's architecture is similar to the previous design.

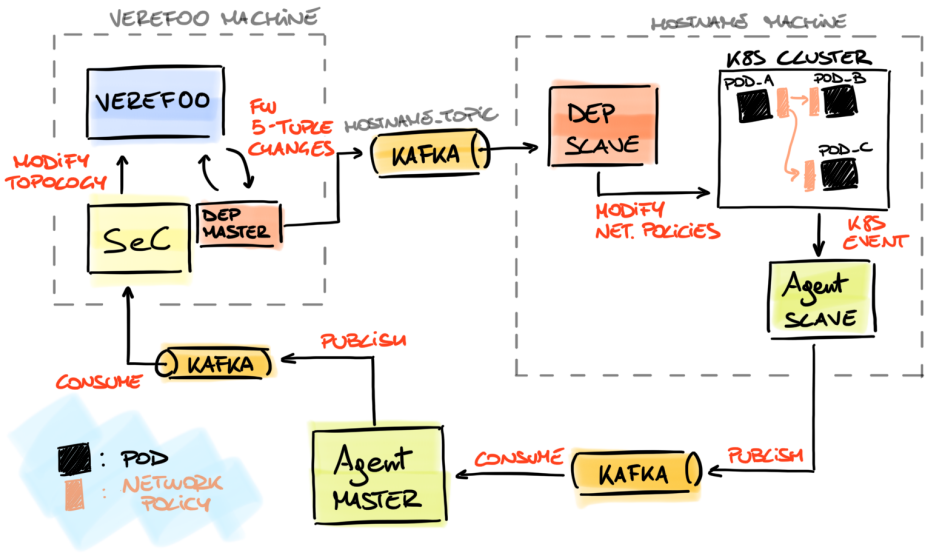

Figure 6.3: Intra-Cluster scenario: Architecture Design Schema

On the Kubernetes-side, we have got two main components:

- **Agent Slave**: it is the micro-service that will listen to the topology changes from the Kubernetes cluster (adding/removing pods) and communicate them to the Security Controller.
- **Dep Slave**: it is the micro-service that will listen to the firewall rule changes from the Verefoo tool. As we will see later on, its behaviour will be the same in both the **Star-Centered** and the **Distributed** firewall models proposed.

By looking at the Verefoo-side, as previously, we have got two main components:

- **Agent Master**: it is the micro-service that will listen to the Kubernetes changes communicated through the Kafka topic.
- **Dep Master**: it is the micro-service that will listen to the Verefoo's firewalls changes and will communicate them to the Dep Slave through Kafka.

6.4 Demo

Inside the following chapter we'll perform some kind of tests regarding:

- the deployment of the micro-services inside the local machine
- the event collection and communication through Kafka

- the Kubernetes topology changes (add or remove some pods)
- the communication of the cluster changes through Kafka
- the Verefoo firewall configuration changes
- the communication of the firewall changes through Kafka

6.4.1 Kubernetes configuration

Before starting with the demo we have to setup correctly the Kubernetes environment: for the purpose of this chapter we have allocated initially three different pods that ran a specific service on the port 80. To better understand what we mean, here in the following we can inspect the Pod yaml file that we used for the demo.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-a
  labels:
    app: pod-a
spec:
  containers:
    - name: pod-a
      image: praqma/network-multitool
      ports:
      - containerPort: 80
```

As was already mentioned, the Pod will run some business logic on the port 80: this port will be used to test the correctness of the Network Policies deployed in the cluster and the reachability of the port, coherently to the Network Policy.

After having deployed three instances of the same pod, what we got from the kubectl command was the following output:

NAME	READY	STATUS	RESTARTS	AGE
pod-a	1/1	Running	0	38s
pod-b	1/1	Running	0	26s
pod-c	1/1	Running	0	14s

Because of the fact that initially there aren't any kind of Network Policy to restrict the traffic, each pod will be able to communicate to all the others: if we get the pod IPs, we can try to ping or send some traffic inside the cluster and we will see that the traffic will be allowed.

By querying the detailed list of pods we obtained the following result:

NAME	READY	STATUS	RESTARTS	AGE	IP
pod-a	1/1	Running	0	13m	10.244.120.90
pod-b	1/1	Running	0	13m	10.244.120.91
pod-c	1/1	Running	0	13m	10.244.120.92

After that, we moved inside the pod-a, and we tried to ping all the pods. All the requests weren't reject, as shown in the next screenshot.

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 16:DF:C6:FD:72:C1
          inet addr:10.244.120.90  Bcast:0.0.0.0  Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1440  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1221 (1.1 KiB)  TX bytes:915 (915.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # ping 10.244.120.91
PING 10.244.120.91 (10.244.120.91) 56(84) bytes of data.
64 bytes from 10.244.120.91: icmp_seq=1 ttl=63 time=0.037 ms
64 bytes from 10.244.120.91: icmp_seq=2 ttl=63 time=0.056 ms
64 bytes from 10.244.120.91: icmp_seq=3 ttl=63 time=0.044 ms
64 bytes from 10.244.120.91: icmp_seq=4 ttl=63 time=0.043 ms
64 bytes from 10.244.120.91: icmp_seq=5 ttl=63 time=0.062 ms
^C
--- 10.244.120.91 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4123ms
rtt min/avg/max/mdev = 0.037/0.048/0.062/0.009 ms
/ # telnet 10.244.120.91 80
Connected to 10.244.120.91
```

Figure 6.4: Intra-Cluster scenario: reachability tests with no restrictions

From the ifconfig command we can understand that we were inside the pod-a (the ip-network interface eth0 was listening on the address 10.244.120.90, the same obtained from the kubectl command) and after pinging and telnetting the address 10.244.120.91 (pod-b) we reached the destination with no problems.

Starting from the previous cluster configuration, we ran both the Agent Slave and the Dep Slave services: as it was already anticipated, their behaviour won't change by changing the firewall structure inside the Verefoo graph. After having started the Agent Slave, the first operation that it performed was the pod isolation: initially each pod mustn't be able to communicate to all the others.

By looking to the Network Policies that were allocated, after the Agent Slave start, we saw something similar to the following configuration yaml file:

```
apiVersion: v1
items:
— apiVersion: networking . k8s . io /v1
  kind: NetworkPolicy
  metadata:
    creationTimestamp: "2021-09-06T10:25:24Z"
    generation: 1
    managedFields:
    - apiVersion: networking . k8s . io /v1
      fieldsType: FieldsV1
      fieldsV1:
        f: spec:
          f: egress:   {}
          f: ingress:   {}
          f: podSelector:
            f: matchLabels:
              .: {}
              f: app: {}
          f: policyTypes:   {}
      manager: okhttp
      operation: Update
      time: "2021-09-06T10:25:24Z"
    name: pod—a
    namespace: default
    resourceVersion: "102720"
    selfLink:
    /apis /networking . k8s . io /v1/namespaces/default  /networkpolicies   /pod—a
    uid: b67f9fc3  —c4e5—455e—8476—8cd9f586e98c
  spec:
    egress:
    - to:
```

```
      - podSelector:
          matchLabels:
            app: pod-a
    ingress:
    - from:
      - podSelector:
          matchLabels:
            app: pod-a
    podSelector:
      matchLabels:
        app: pod-a
    policyTypes:
    - Ingress
    - Egress
- apiVersion: networking.k8s.io/v1
  kind: NetworkPolicy
  metadata:
    creationTimestamp: "2021-09-06T10:25:24Z"
    generation: 1
    managedFields:
    - apiVersion: networking.k8s.io/v1
      fieldsType: FieldsV1
      fieldsV1:
        f:spec:
          f:egress: {}
          f:ingress: {}
          f:podSelector:
            f:matchLabels:
              .: {}
              f:app: {}
          f:policyTypes: {}
      manager: okhttp
      operation: Update
      time: "2021-09-06T10:25:24Z"
    name: pod-b
    namespace: default
    resourceVersion: "102721"
    selfLink:
    /apis/networking.k8s.io/v1/namespaces/default/networkpolicies/pod-b
    uid: 35663282-6938-430e-b75d-2e5e4e6f2354
  spec:
```

```
    egress:
    - to:
      - podSelector:
          matchLabels:
            app: pod-b
    ingress:
    - from:
      - podSelector:
          matchLabels:
            app: pod-b
    podSelector:
      matchLabels:
        app: pod-b
    policyTypes:
    - Ingress
    - Egress
- apiVersion: networking.k8s.io/v1
  kind: NetworkPolicy
  metadata:
    creationTimestamp: "2021-09-06T10:25:24Z"
    generation: 1
    managedFields:
    - apiVersion: networking.k8s.io/v1
      fieldsType: FieldsV1
      fieldsV1:
        f:spec:
          f:egress: {}
          f:ingress: {}
          f:podSelector:
            f:matchLabels:
              .: {}
              f:app: {}
          f:policyTypes: {}
      manager: okhttp
      operation: Update
      time: "2021-09-06T10:25:24Z"
    name: pod-c
    namespace: default
    resourceVersion: "102719"
    selfLink:
    /apis/networking.k8s.io/v1/namespaces/default/networkpolicies/pod-c
```

```
    uid: e9181a24-b1b5-4a6c-86f4-c9f78fc99425
  spec:
    egress:
    - to:
      - podSelector:
          matchLabels:
            app: pod-c
    ingress:
    - from:
      - podSelector:
          matchLabels:
            app: pod-c
    podSelector:
      matchLabels:
        app: pod-c
    policyTypes:
    - Ingress
    - Egress
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

By focusing on the pod-a's Network Policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
    name: pod-a
    namespace: default
spec:
    egress:
    - to:
        - podSelector:
            matchLabels:
                app: pod-a
    ingress:
    - from:
      - podSelector:
            matchLabels:
```

```
            app: pod-a
  podSelector:
    matchLabels:
      app: pod-a
  policyTypes:
    - Ingress
    - Egress
```

we can see that the pod-a was able to accept only the traffic from himself. All the other traffic (outgoing or incoming) would have been denied by the policy himself. By trying to reach the other pods from the pod-a, as expected, all the traffic was denied:

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 16:DF:C6:FD:72:C1
          inet addr:10.244.120.90  Bcast:0.0.0.0  Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1440  Metric:1
          RX packets:23 errors:0 dropped:0 overruns:0 frame:0
          TX packets:23 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2043 (1.9 KiB)  TX bytes:1737 (1.6 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # ping 10.244.120.91
PING 10.244.120.91 (10.244.120.91) 56(84) bytes of data.
^C
--- 10.244.120.91 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7165ms

/ # telnet 10.244.120.91 80
telnet: can't connect to remote host (10.244.120.91): Operation timed out
```

Figure 6.5: Intra-Cluster scenario: reachability tests with network constraints

As we can see from the previous screen, the ping requests were denied and the telnet connection refused, without completing the communication.

Now we have to understand how the Agent Slave will communicate the Kubernetes pod changes to the Agent Master. First of all, the **Agent Slave** will read from a local configuration file, that will contain some information like the address of the Kafka Broker and the Kafka topic on which it must publish the Kubernetes events.

```
{
    "kafkaConsumerGroupId": "DemoConsumer",
    "kafkaProducerGroupId": "DemoProducer",
    "kafkaTopic": "masterTopic",
    "kafkaServerUrl": "localhost",
    "kafkaServerPort": "9092"
}
```

For example, from the previous json configuration file, the Agent Slave will understand that it must write all the events into the topic called **masterTopic**.

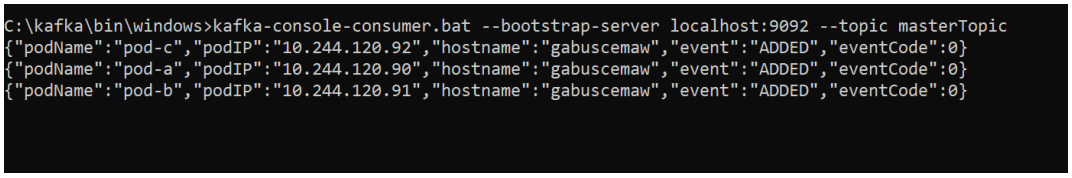

Figure 6.6: Intra-Cluster scenario: Kafka events published by the Agent Slave

On the other side, the **Dep Slave**, will subscribe to the Kafka topic named with the hostname of the local machine (gabuscemaw in this demo). Inside this topic it will receive all the events published by the **Dep Master** when some firewall changes took place. We will see later on how this mechanism works.

6.4.2 Verefoo configuration

Before continuing with the demo, we should setup properly the environment around the Verefoo tool. We should instantiate both the **Agent Master** and the **Security Controller** micro-services.

By requesting the graph's configuration, through the following endpoint offered by the Verefoo API

$$http://localhost:8085/verefoo/adp/graphs/graph_id$$

when the Security Controller will start for the first time, we can see that the graph will be allocated with no nodes inside it.

6.4.3 Star-centered firewall

As we already know, with this configuration, the Security Controller will allocate initially an empty firewall that will be enriched with the network rules that must

be reflected in the Kubernetes Cluster.

When the Security Controller will start for the first time, it will allocate an initially empty firewall, like the one that we can see from the following configuration.

```
{
    "node": [
        {
            "neighbour": [],
            "configuration": null,
            "id": 317,
            "name": "10.0.0.1",
            "functionalType": "FIREWALL"
        }
    ],
    "id": 215,
    "serviceGraph": false
}
```

After we received the first events through Kafka, both the graph configuration and the firewall neighbourhood will change. In the following example, the Security Controller has received the events coming from the three pods that were instantiated inside the Kubernetes cluster for the demo.

```
{
    "node": [
        {
            "neighbour": [
                {
                    "id": 199,
                    "name": "10.0.0.1"
                }
            ],
            "configuration": null,
            "id": 323,
            "name": "10.244.120.90",
            "functionalType": "ENDHOST"
        },
        {
```

```
            "neighbour": [
                {
                    "id": 234,
                    "name": "10.244.120.91"
                },
                {
                    "id": 248,
                    "name": "10.244.120.90"
                },
                {
                    "id": 228,
                    "name": "10.244.120.92"
                }
            ],
            "configuration": null,
            "id": 317,
            "name": "10.0.0.1",
            "functionalType": "FIREWALL"
        },
        {
            "neighbour": [
                {
                    "id": 293,
                    "name": "10.0.0.1"
                }
            ],
            "configuration": null,
            "id": 291,
            "name": "10.244.120.91",
            "functionalType": "ENDHOST"
        },
        {
            "neighbour": [
                {
                    "id": 308,
                    "name": "10.0.0.1"
                }
            ],
            "configuration": null,
            "id": 272,
            "name": "10.244.120.92",
```

```
            "functionalType": "ENDHOST"
        }
    ],
    "id": 215,
    "serviceGraph": false
}
```

As we can notice from the previous graph's configuration, all the nodes were connected only to the initial firewall. The firewall was put in the middle of the topology: in fact its neighbourhood was composed by the collection of all the nodes that were created.

Then, we tried to update the firewall configuration with the following configuration

```
{
   "neighbour":[
      {
         "id":234,
         "name":"10.244.120.91"
      },
      {
         "id":248,
         "name":"10.244.120.90"
      },
      {
         "id":228,
         "name":"10.244.120.92"
      }
   ],
   "configuration":{
      "firewall":{
         "elements":[
            {
               "action":"ALLOW",
               "source":"10.244.120.90",
               "destination":"10.244.120.91",
               "protocol":"TCP",
               "srcPort":"80",
               "dstPort":"80",
```

```
                "directional":true
            }
        ],
        "defaultAction":"DENY"
    }
    "id":0,
    "name":"string",
    "description":"string"
  },
  "id":317,
  "name":"10.0.0.1",
  "functionalType":"FIREWALL"
}
```

to allow the TCP communication from the IP 10.244.120.90 (pod-a) to the destination IP 10.244.120.91 (pod-b) on the port 80. After the firewall update, on Kubernetes we saw that two different Network Policies were updated.

On one side, the pod-a NetworkPolicy was updated with the following configuration:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: pod-a
  namespace: default
spec:
  egress:
  - ports:
    - port: 80
      protocol: TCP
    to:
    - podSelector:
        matchLabels:
          app: pod-b
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: pod-a
```

```
  podSelector:
    matchLabels:
      app: pod-a
  policyTypes:
  - Ingress
  - Egress
```

As we can see, on Kubernetes the NetworkPolicy was updated to enable, in the egress direction, the traffic directed to the pod-b.

On the other side, if we look to the pod-b NetworkPolicy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: pod-b
  namespace: default
spec:
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: pod-b
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: pod-a
    ports:
    - port: 80
      protocol: TCP
  podSelector:
    matchLabels:
      app: pod-b
  policyTypes:
  - Ingress
  - Egress
```

we can see that, in the ingress direction, the traffic sent by the pod-a was accepted. Also the telnet from the pod-a to the pod-b was allowed.

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 16:DF:C6:FD:72:C1
          inet addr:10.244.120.90  Bcast:0.0.0.0  Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1440  Metric:1
          RX packets:27 errors:0 dropped:0 overruns:0 frame:0
          TX packets:42 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2267 (2.2 KiB)  TX bytes:3247 (3.1 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # telnet 10.244.120.91 80
Connected to 10.244.120.91
```

Figure 6.7: Intra-Cluster scenario - Star-centered firewall: telnet connection after NetworkPolicy update

Then, we tried to delete the pod-a inside the Kubernetes cluster: because of the Kubernetes topology change, the firewall and the graph configuration in Verefoo were updated accordingly. Each time the cluster topology change, the network configuration inside the Verefoo graph will be updated coherently.

```
PS C:\Users\gabuscema> kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
pod-a   1/1     Running   0          4h56m
pod-b   1/1     Running   0          4h56m
pod-c   1/1     Running   0          4h56m
PS C:\Users\gabuscema> kubectl delete pod pod-a
pod "pod-a" deleted
PS C:\Users\gabuscema> kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE     IP              NODE       NOMINATED NODE   READINESS GATES
pod-b   1/1     Running   0          4h56m   10.244.120.91   minikube   <none>           <none>
pod-c   1/1     Running   0          4h56m   10.244.120.92   minikube   <none>           <none>
PS C:\Users\gabuscema>
```

Figure 6.8: Intra-Cluster scenario - Star-centered firewall: deletion of pod-a from the cluster

Then, the Verefoo graph was updated to the following configuration

```
1  {
2   "node":[
3      {
```

```
        "neighbour":[
           {
              "id":293,
              "name":"10.0.0.1"
           }
        ],
        "configuration":null,
        "id":291,
        "name":"10.244.120.91",
        "functionalType":"ENDHOST"
     },
     {
        "neighbour":[
           {
              "id":228,
              "name":"10.244.120.92"
           },
           {
              "id":234,
              "name":"10.244.120.91"
           }
        ],
        "configuration":{
           "firewall":{
              "elements":[
              ],
              "defaultAction":"DENY"
           },
           "id":351,
           "name":"string",
           "description":"string"
        },
        "id":317,
        "name":"10.0.0.1",
        "functionalType":"FIREWALL"
     },
     {
        "neighbour":[
           {
              "id":308,
              "name":"10.0.0.1"
```

```
            }
        ],
        "configuration":null,
        "id":272,
        "name":"10.244.120.92",
        "functionalType":"ENDHOST"
    }
  ],
  "id":215,
  "serviceGraph":false
}
```

As we can see from the previous configuration json, both the firewall's rules and the firewall's neighbours were updated correctly (the pod-a was removed from both the neighbourhood and the firewall rules).

On the AgentSlave-side, the NetworkPolicies were updated coherently (the pod-a doesn't exists anymore so it was removed from all the pod's network rules): in fact, if we inspect the pod-b NetworkPolicy, it came back to the initial version (pod-b fully isolated).

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: pod-b
  namespace: default
spec:
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: pod-b
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: pod-b
  podSelector:
    matchLabels:
      app: pod-b
```

```
  policyTypes:
  - Ingress
  - Egress
```

To summarize, after the deletion of a pod:

- the graph node list will be updated by removing the old node (equivalent to the deleted pod)
- the firewall configuration will be updated by removing the pod from the neighbourhood and from the networking rules
- the NetworkPolicies will remove the rules that refers to the deleted pod

6.4.4 Distributed firewall

In order to extend the flexibility of the Security Controller, another mechanism was introduced to deploy the firewalls inside the Verefoo graph with another configuration: with this modality, the firewall will be split in many nodes, as many as the number of the nodes that were deployed inside the Kuberentes cluster. By this way, per each pod we will instantiate a specific firewall on Verefoo.

The overall mechanism is similar to the previous one, with the only exception that the management of the firewalls (especially after the pod deletion) is a little bit different. When the Security Controller will be launched for the first time, it will allocate an empty graph (now, the firewalls will be instantiated only after the pod's events reception):

```
1  {
2    "node": [],
3    "id": 416,
4    "serviceGraph": false
5  }
```

To better understand how it behaves, we duplicated the previous scenario by deploying three different pods on Kubernetes:

```
PS C:\Users\gabuscema\Desktop> kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP             NODE       NOMINATED NODE   READINESS GATES
pod-a   1/1     Running   0          27s   10.244.120.80  minikube   <none>           <none>
pod-b   1/1     Running   0          18s   10.244.120.81  minikube   <none>           <none>
pod-c   1/1     Running   0          8s    10.244.120.82  minikube   <none>           <none>
PS C:\Users\gabuscema\Desktop>
```

Figure 6.9: Intra-Cluster scenario - Distributed Firewall: Kuberentes deployed pods

After we had instantiated the Agent Slave, the Security Controller received the first events through Kafka and both the graph configuration and the firewall neighbourhood will change. As we can see from the following configuration, the Security controller created as many firewall as the pods that were allocated inside the cluster:

```
{
        "node": [
          {
            "neighbour": [
              {
                "id": 220,
                "name": "10.0.0.1"
              },
              {
                "id": 538,
                "name": "10.244.120.81"
              },
              {
                "id": 307,
                "name": "10.0.0.2"
              }
            ],
            "configuration": {
              "firewall": {
                "elements": [],
                "defaultAction": "DENY"
              },
            "id": 359,
            "name": "10.0.0.0",
            "functionalType": "FIREWALL"
          },
```

```
        {
          "neighbour": [
            {
              "id": 222,
              "name": "10.0.0.0"
            },
            {
              "id": 218,
              "name": "10.244.120.80"
            },
            {
              "id": 286,
              "name": "10.0.0.2"
            }
          ],
          "configuration": {
            "firewall": {
              "elements": [],
              "defaultAction": "DENY"
            },
          "id": 443,
          "name": "10.0.0.1",
          "functionalType": "FIREWALL"
        },
        {
          "neighbour": [
            {
              "id": 393,
              "name": "10.0.0.0"
            },
            {
              "id": 289,
              "name": "10.244.120.82"
            },
            {
              "id": 558,
              "name": "10.0.0.1"
            }
          ],
          "configuration": {
            "firewall": {
```

```
                "elements": [],
                "defaultAction": "DENY"
              },
            "id": 456,
            "name": "10.0.0.2",
            "functionalType": "FIREWALL"
          },
          {
            "neighbour": [
              {
                "id": 424,
                "name": "10.0.0.1"
              }
            ],
            "configuration": null,
            "id": 559,
            "name": "10.244.120.80",
            "functionalType": "ENDHOST"
          },
          {
            "neighbour": [
              {
                "id": 306,
                "name": "10.0.0.0"
              }
            ],
            "configuration": null,
            "id": 537,
            "name": "10.244.120.81",
            "functionalType": "ENDHOST"
          },
          {
            "neighbour": [
              {
                "id": 223,
                "name": "10.0.0.2"
              }
            ],
            "configuration": null,
            "id": 524,
            "name": "10.244.120.82",
```

```
            "functionalType": "ENDHOST"
          }
        ],
        "id": 416,
        "serviceGraph": false
      }
```

All the nodes are connected to the firewall that was created in front of them, and all the firewalls are connected together.

Of course, all the pods are totally isolated from the external environment: if we try to connect from the pod-b to the pod-c, the communication will be denied, as we can see from the following screen:

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr DA:38:2F:D9:11:BE
          inet addr:10.244.120.81  Bcast:0.0.0.0  Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1440  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # ping 10.244.120.82
PING 10.244.120.82 (10.244.120.82) 56(84) bytes of data.
^C
--- 10.244.120.82 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5155ms

/ # telnet 10.244.120.82 80
telnet: can't connect to remote host (10.244.120.82): Operation timed out
/ #
```

Figure 6.10: Intra-Cluster scenario - Distributed Firewall: denied reachability tests

Then we tried to update the pod-b's firewall configuration (firewall with address 10.0.0.0) with the following one:

```
{
  "neighbour": [
```

```
      {
        "id": 220,
        "name": "10.0.0.1"
      },
      {
        "id": 538,
        "name": "10.244.120.81"
      },
      {
        "id": 307,
        "name": "10.0.0.2"
      }
    ],
    "configuration": {
      "firewall": {
        "elements": [
            {
          "action": "ALLOW",
          "source": "10.244.120.81",
          "destination": "10.244.120.82",
          "protocol": "TCP",
          "srcPort": "80",
          "dstPort": "80",
          "directional": true
         }
       ],
        "defaultAction": "DENY"
      },
    "id": 359,
    "name": "10.0.0.0",
    "functionalType": "FIREWALL"
  }
```


to allow the TCP communication from the IP 10.244.120.81 (pod-b) to the destination IP 10.244.120.82 (pod-c). After the update, both the pod-b and pod-c NetworkPolicies changed to the following configuration:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```

```
  name: pod-b
  namespace: default
spec:
  egress:
  - ports:
    - port: 80
      protocol: TCP
    to:
    - podSelector:
        matchLabels:
          app: pod-c
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: pod-b
  podSelector:
    matchLabels:
      app: pod-b
  policyTypes:
  - Ingress
  - Egress


apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: pod-c
  namespace: default
spec:
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: pod-c
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: pod-b
    ports:
```

```
    - port: 80
      protocol: TCP
  podSelector:
    matchLabels:
      app: pod-c
  policyTypes:
  - Ingress
  - Egress
```

As we can see, the network policies changed in order to enable the traffic directed from the pod-b to the pod-c. By trying to connect from the pod-b to the pod-c, the communication was allowed.

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr DA:38:2F:D9:11:BE
          inet addr:10.244.120.81  Bcast:0.0.0.0  Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1440  Metric:1
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:42 (42.0 B)  TX bytes:1000 (1000.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # telnet 10.244.120.82 80
Connected to 10.244.120.82
```

Figure 6.11: Intra-Cluster scenario - Distributed Firewall: reachability tests

Finally, after the pod-c deletion from the cluster, we saw that some changes take place on Verefoo:

```
PS C:\Users\gabuscema\Desktop> kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP             NODE       NOMINATED NODE   READINESS GATES
pod-a   1/1     Running   0          31m   10.244.120.80  minikube   <none>           <none>
pod-b   1/1     Running   0          31m   10.244.120.81  minikube   <none>           <none>
pod-c   1/1     Running   0          30m   10.244.120.82  minikube   <none>           <none>
PS C:\Users\gabuscema\Desktop> kubectl delete pod pod-c --force
warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "pod-c" force deleted
PS C:\Users\gabuscema\Desktop> kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP             NODE       NOMINATED NODE   READINESS GATES
pod-a   1/1     Running   0          31m   10.244.120.80  minikube   <none>           <none>
pod-b   1/1     Running   0          31m   10.244.120.81  minikube   <none>           <none>
PS C:\Users\gabuscema\Desktop>
```

Figure 6.12: Intra-Cluster scenario - Distributed Firewall: pod deletion tests

To sum up, after the pod deletion, the following changes will took place:

- **graph configuration changes**: both the node and its firewall will be deleted from the graph
- **firewall rules changes**: all the firewalls that contains some rules that refers to the deleted pod, will be cleared


```
{
"node": [
  {
    "neighbour": [
      {
        "id": 434,
        "name": "10.0.0.0"
      },
      {
        "id": 412,
        "name": "10.244.120.80"
      }
    ],
    "configuration": {
      "firewall": {
        "elements": [],
        "defaultAction": "DENY"
      },
    "id": 443,
    "name": "10.0.0.1",
    "functionalType": "FIREWALL"
  },
  {
    "neighbour": [
      {
        "id": 429,
        "name": "10.0.0.1"
      },
      {
        "id": 417,
        "name": "10.244.120.81"
      }
    ],
    "configuration": {
```

```
            "firewall": {
              "elements": [],
              "defaultAction": "DENY"
            }
          },
          "id": 359,
          "name": "10.0.0.0",
          "functionalType": "FIREWALL"
        },
        {
          "neighbour": [
            {
              "id": 424,
              "name": "10.0.0.1"
            }
          ],
          "configuration": null,
          "id": 559,
          "name": "10.244.120.80",
          "functionalType": "ENDHOST"
        },
        {
          "neighbour": [
            {
              "id": 306,
              "name": "10.0.0.0"
            }
          ],
          "configuration": null,
          "id": 537,
          "name": "10.244.120.81",
          "functionalType": "ENDHOST"
        }
      ],
      "id": 416,
      "serviceGraph": false
    }
```


On the AgentSlave-side, the NetworkPolcies were updated as a consequence of the previous changes: the pod-c doesn't exist anymore, so it was removed from all the pod's network rules (the pod-b will be isolated again from the external

environment).

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: pod-b
  namespace: default
spec:
  egress:
  - ports:
    - port: 80
      protocol: TCP
    to:
    - podSelector:
        matchLabels:
          app: pod-b
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: pod-b
  podSelector:
    matchLabels:
      app: pod-b
  policyTypes:
  - Ingress
  - Egress
```

Chapter 7

Conclusions and Future Works

The aim of this thesis work was to understand the containerized scenarios in which Verefoo could be reflected.

In order to do that, we grouped some of the Verefoo functionalities and tried to apply them on Kubernetes.

Some of the proposed scenarios/functionalities cannot be reflected properly because of two main reasons:

- **Kubernetes application working level**: as mentioned many times during this paper, both the Kubernetes cluster and the tools built over the K8S API (like the Istio service mesh) will work over the ISO/OSI layer 7. This was the reason why the Verefoo forwarding mechanism couldn't be applied properly inside Kuberentes. Because of this aspect, the forwarding mechanism could only be applied over the application level by building some custom logic, that doesn't exploit the lower ISO/OSI layers: it would be interesting to inspect the cluster traffic's lower level fields (L3 or L4) in order to build a proper forwarding mechanism. The NetworkPolicy construct is already able to inspect these layers, but it doesn't offer any proxy-forwarding networking mechanisms.
- **Kubernetes network's architecture**: as described in the chapter titled **Inter-Cluster mechanism**, also the communication between many different clusters could be problematic. Kubernetes is able to expose a specific pod, on a certain address, and let it communicate with another endpoint (which could be another pod or an endhost). The aspect that wasn't covered is linked to the network rules management: because of the network-interface system hold

by Kubernetes, we weren't able to control the network communication rules between many different clusters.

By considering simple scenarios, the integration was done with no big effort (like the work done for the neighbour mechanism): by working with an **intra-cluster environment** (one cluster composed by many different pods) and simple point-to-point communications, we were able to reflect the Verefoo configuration over the Kubernetes cluster with no problems.

On one side, if we complicate the overall scenario, by enriching the functionalities that we must deploy inside the cluster, the integration could be hard or time-consuming: of course, this paper is not meant to be exhaustive, and maybe there could be some specific technology that can (at least partially) cover all the aspects that weren't closed in this paper.

On the other side, if we work in simpler scenarios by using some specific libraries, with no need of implementing by hand the software interface, to communicate with Kubernetes through its APIs (like it was done with the NetworkPolicy API), we can integrate easily the Verefoo and the Kubernetes functionalities together.

Bibliography

[1] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. «Automated optimal firewall orchestration and configuration in virtualized networks». In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 2020, pp. 1–7. DOI: 10.1109/NOMS47738.2020.9110402 (cit. on p. 2).

[2] Mohamed Ahmed. «Docker Containers Changed How We Deploy Software». In: (July 2019). URL: https://www.magalix.com/blog/kubernetes-101-concepts-and-why-it-matters (cit. on p. 3).

[3] netgroup-polito. *verefoo*. https://github.com/netgroup-polito/verefoo. 2021 (cit. on p. 5).

[4] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. «Towards a fully automated and optimized network security functions orchestration». In: *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. 2019, pp. 1–7. DOI: 10.1109/CCCS.2019.8888130 (cit. on p. 6).

[5] Kevin Casey. «How to explain Kubernetes in plain English». In: (Sept. 2020). URL: https://enterprisersproject.com/article/2017/10/how-explain-kubernetes-plain-english (cit. on p. 7).

[6] Jahongir Rahmonov. «Introduction to Kubernetes». In: (Feb. 2018). URL: https://rahmonov.me/posts/introduction-to-kubernetes/ (cit. on p. 7).

[7] Matthew O'Riordan. «Everything You Need To Know About Publish/Subscribe». In: (). URL: https://ably.com/topic/pub-sub (cit. on p. 11).

[8] «How does Kafka work?» In: (). URL: https://kafka.apache.org/intro (cit. on p. 13).

[9] Bo Gowan. «What is NFV?» In: (May 2016). URL: https://business.comcast.com/community/browse-all/details/what-is-nfv (cit. on p. 15).

[10] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, and Fulvio Valenza. «A novel approach for security function graph configuration and deployment». In: *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. 2021, pp. 457–463. DOI: 10.1109/NetSoft51509.2021.9492654 (cit. on p. 16).

[11] Cataldo Basile, Fulvio Valenza, Antonio Lioy, Diego R. Lopez, and Antonio Pastor Perales. «Adding Support for Automatic Enforcement of Security Policies in NFV Networks». In: *IEEE/ACM Transactions on Networking* 27.2 (2019), pp. 707–720. DOI: 10.1109/TNET.2019.2895278 (cit. on p. 16).

[12] Antonio Lioy Ignazio Pedone and Fulvio Valenza. «Towards an Efficient Management and Orchestration Framework for Virtual Network Security Functions». In: (Nov. 2019). URL: https://doi.org/10.1155/2019/2425983 (cit. on p. 16).

[13] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Serena Spinoso, Fulvio Valenza, and Jalolliddin Yusupov. «Improving the Formal Verification of Reachability Policies in Virtualized Networks». In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 713–728. DOI: 10.1109/TNSM.2020.3045781 (cit. on p. 16).

[14] «RedHat: What is NFV?» In: (Aug. 2019). URL: https://www.redhat.com/en/topics/virtualization/what-is-nfv (cit. on p. 16).

[15] Rohan Kumar. «Difference between Fabric8io and Official kubernetes java libraries». In: (July 2020). URL: https://itnext.io/difference-between-fabric8-and-official-kubernetes-java-client-3e0a994fd4af (cit. on pp. 21, 22).

[16] Peter Wilcsinszky Mate Ory. «Understanding Kubernetes cluster events». In: (Jan. 2020). URL: https://banzaicloud.com/blog/k8s-cluster-logging/ (cit. on p. 26).

[17] kubernetes. *kubernetes events*. https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/events/event.go. 2021 (cit. on p. 26).

[18] Lovisa Johansson. «Apache Kafka for beginners - What is Apache Kafka?» In: (Mar. 2019). URL: https://www.cloudkarafka.com/blog/part1-kafka-for-beginners-what-is-apache-kafka.html (cit. on pp. 27–29).

[19] Pei-Ming Wu. «Preventing Systemic Failure: Backpressure—What It Is and How It Works». In: (Mar. 2019). URL: https://glasnostic.com/blog/preventing-systemic-failure-backpressure (cit. on p. 31).

[20] «Kafka Java Client». In: (). URL: https://docs.confluent.io/clients-kafka-java/current/overview.html (cit. on p. 31).

[21] bulldog2011. *bigqueue*. https://github.com/bulldog2011/bigqueue. 2016 (cit. on p. 36).

[22] Keilan Jackson. «Types of Kubernetes events». In: (Feb. 2020). URL: https://www.bluematador.com/blog/kubernetes-events-explained (cit. on p. 39).

[23] *When will the kubernetes POD IP change?* https://stackoverflow.com/questions/52362514/when-will-the-kubernetes-pod-ip-change. 2018 (cit. on p. 39).

[24] Janakiram MSV. «The Kubernetes Way: Pods and Services». In: (May 2016). URL: https://thenewstack.io/kubernetes-way-part-one/ (cit. on p. 40).

[25] «What is CNI?» In: (). URL: https://rancher.com/docs/rancher/v2.x/en/faq/networking/cni-providers/ (cit. on p. 44).

[26] *What is Calico?* https://docs.projectcalico.org/about/about-calico. 2021 (cit. on p. 45).

[27] *Just the Basics*. https://romana.io/how/romana_basics/. 2021 (cit. on p. 45).

[28] Madhura Maskasky. «Kubernetes Networking: Achieving High Performance with Calico». In: (2018). URL: https://platform9.com/resource/kubernetes-networking-achieving-high-performance-with-calico/ (cit. on p. 45).

[29] Raffaele Giuseppe Trani. «Integrating VNF Service Chains in Kubernetes Clusters». In: 2020, pp. 18–19. URL: http://webthesis.biblio.polito.it/id/eprint/14516 (cit. on p. 46).

[30] *Calico components*. URL: https://docs.projectcalico.org/reference/architecture/overview (cit. on p. 46).

[31] *About Network Policy*. URL: https://docs.projectcalico.org/about/about-network-policy (cit. on p. 47).

[32] *Network Policy*. URL: https://docs.projectcalico.org/reference/resources/networkpolicy (cit. on p. 49).

[33] *Network Policies*. URL: https://kubernetes.io/docs/concepts/services-networking/network-policies/ (cit. on pp. 49, 63).

[34] Justin Ellingwood. «Comparing Kubernetes CNI Providers: Flannel, Calico, Canal, and Weave». In: (Mar. 2019). URL: https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/ (cit. on pp. 50, 51).

[35] Owen Garrett Floyd Smith of NGINX. «What Is a Service Mesh?» In: (Apr. 2018). URL: https://www.nginx.com/blog/what-is-a-service-mesh/ (cit. on p. 52).

[36] «Service mesh: A critical component of the cloud native stack». In: (Apr. 2017). URL: https://www.cncf.io/blog/2017/04/26/service-mesh-critical-component-cloud-native-stack/ (cit. on p. 52).

[37] *The Istio service mesh*. URL: https://istio.io/ (cit. on pp. 53, 56).

[38] William Morgan. «What's a service mesh? And why do I need one?» In: (). URL: https://buoyant.io/what-is-a-service-mesh (cit. on p. 55).

[39] Sachin Manpathak. «Kubernetes Service Mesh: A Comparison of Istio, Linkerd, and Consul». In: (Oct. 2019). URL: https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/ (cit. on p. 55).

[40] *Consul Architecture*. URL: https://www.consul.io/docs/architecture (cit. on p. 55).

[41] *The HashiCorp Stack*. URL: https://www.hashicorp.com/ (cit. on p. 55).

[42] *Nomad Project*. URL: https://www.nomadproject.io/ (cit. on p. 55).

[43] *Linkerd Architecture*. URL: https://linkerd.io/2.10/reference/architecture/ (cit. on p. 56).

[44] *Cloud Native Computing Foundation*. URL: https://www.cncf.io/projects/ (cit. on p. 56).

[45] *Istio Traffic Management*. URL: https://istio.io/latest/docs/concepts/traffic-management/ (cit. on p. 59).

[46] *Istio Security*. URL: https://istio.io/latest/docs/concepts/security/ (cit. on p. 59).

[47] *Istio Observability*. URL: https://istio.io/latest/docs/concepts/observability/ (cit. on p. 59).

[48] *Istio Authorization Policy*. URL: https://istio.io/latest/docs/reference/config/security/authorization-policy/ (cit. on p. 59).

[49] *Istio Stackdriver Config*. URL: https://istio.io/latest/docs/reference/config/proxy_extensions/stackdriver/ (cit. on p. 60).

[50] Marton Sereg. «Introduction to Istio access control». In: (July 2020). URL: https://www.ciscotechblog.com/blog/istio-authorization-policies/ (cit. on p. 63).

[51] Adam Bavosa. «What is a Serverless Function?» In: (May 2019). URL: https://www.pubnub.com/blog/what-is-a-serverless-function/ (cit. on p. 78).

[52] *Kubeless*. URL: https://kubeless.io/ (cit. on p. 81).

[53] *Extend the Kubernetes API with CustomResourceDefinitions*. URL: https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/ (cit. on p. 81).

[54] argoproj. *argoproj*. https://github.com/argoproj/argoproj. 2021 (cit. on p. 86).

[55] Coussement Bruno. «What to consider before choosing Argo Workflow?» In: (Mar. 2021). URL: https://medium.com/datamindedbe/what-to-consider-before-choosing-argo-workflow-54f6067307a8 (cit. on p. 87).

[56] *Argo Workflow Core Concepts*. URL: https://argoproj.github.io/argo-workflows/workflow-concepts/ (cit. on p. 88).

[57] *Configure outgoing NAT*. URL: https://docs.projectcalico.org/networking/workloads-outside-cluster#create-an-ip-pool-with-nat-outgoing-enabled (cit. on p. 119).

Ringraziamenti

Ci tenevo a mantere questa sezione in italiano per poter ringraziare chi mi ha accompagnato e sostenuto lungo questo mio percorso, chi ha riso e pianto con me.

Ringrazio mio padre, mia madre e mia sorella. Nonostante le distanze e le difficoltà, il loro supporto è stato fondamentale per poter raggiungere questo traguardo.

Ringrazio i miei parenti, amici, colleghi e compagni di squadra. Ho imparato durante questo mio percorso che la felicità è tale solo se è condivisa e in questi anni ho avuto la fortuna di condividere con voi dei momenti indelebili che porterò per sempre con me. Vi ringrazio di cuore.

Un saluto speciale infine va a mia nonna. Non so se esiste un dio ma so che lei mi ha sempre guardato da lassù.

Gaetano