Politecnico di Torino

Master of Science in Computer Engineering

Master Degree Thesis

Towards automation of Multi Cluster Network Policies

Supervisors
prof. Riccardo Sisto
prof. Fulvio Valenza
prof. Guido Marchetto
dott. Daniele Bringhenti

Candidate
Giuseppe Sommese

Academic Year 2020-2021

Summary

In recent years, novel cloud technologies have emerged and gained interest in the field of software development. At the same time, the application development methodology has started to change, moving from the monolithic application idea to cloud-native application, composed of micro-services loosely coupled and independent from each other. Developing applications in this way has permitted the various parts to be deployed in different servers and this has contributed to the start of developing new types of software called orchestrators, capable of managing all the resource optimization, deployment, and security parts of a data center or cloud environment. One of the most successful orchestrators at the moment is Kubernetes. Kubernetes allows new applications, developed using micro-services, to be deployed in different parts of data centers and to still be able to communicate with each other. Since its first release, developers have mainly focused on the work of a single Kubernetes instance, called cluster, to try to make everything work within a single cluster. Recently, the attention has shifted to the cooperation of several Kubernetes clusters, which can also belong to different data- centers or companies, with each other to be able to develop parts of an application independently, and then make them communicate to form the entire application. Many projects have also been developed together with Kubernetes that extend it with security networking and monitoring functions. Although Kubernetes doesn't currently provide any features for connecting multiple cluster instances and discovering the services that are within them, the various projects have started to move in this area by proposing different solutions. Even if some parts have been developed using different methodologies, at the moment, the automation part of both the security functions and the creation of connections between clusters and services is missing in these projects. Automating these functions is important to prevent the possibility of a human error in the configuration both of security, which could lead to branches and the connection between clusters and services that could lead to an unwanted interruption of communication with difficult resolution. Furthermore, if it refers to the communication between services of different companies, there is some information in the configurations that could be unknown and that requires cooperation between the parties that could lead to misconfigurations and, therefore, to undesirable effects. The objectives of this work are then the analysis of the current technologies for multi-cluster communications and the development of a Multi Cluster Orchestrator able to configure automatically security and communications in multiple Kubernetes clusters. For the first objective, three software will be analyzed, highlighting their strengths and weaknesses, in particular the lack of automatic configurations features. Based on one of these technologies a new

framework will be proposed, placed at a higher level of Kubernetes, that can automatically configure security policies, connect clusters, and provide service-to-service communication between services in different Kubernetes clusters.

Contents

List of Figures 8

List of Tables 10

Listings 11

1 Introduction 14

1.1 Thesis objective . . . 15

1.2 Thesis description . . . 15

2 Technologies 17

2.1 Kubernetes . . . 17

2.1.1 Cluster . . . 18

2.1.2 Pod . . . 18

2.1.3 Service . . . 20

2.1.4 Multi Cluster . . . 22

2.2 Istio . . . 24

2.2.1 Architecture . . . 24

2.2.2 Multi-cluster . . . 26

2.3 Calico . . . 27

2.3.1 Architecture . . . 27

2.3.2 Multi-cluster . . . 29

2.4 Cilium . . . 29

2.4.1 Architecture . . . 30

2.4.2 Multi-cluster . . . 30

3 Network Isolation in Kubernetes 33
3.1 Kubernetes Network Policy 33
3.2 Network isolation 36
3.2.1 Iptables 39
3.2.2 Ebpf 40
3.3 Istio Network Policy 41
3.4 Calico Network Policy 42
3.5 Cilium Network Policy 43
3.6 Comparison between tecnologies 45

4 Multi Cluster 47
4.1 Introduction to Multi Cluster Orchestrathor 47
4.2 Model 47
4.2.1 Domain 48
4.2.2 Requirements 50
4.2.3 Global Configuration 51
4.2.4 Single Configuration 52

5 Implementation 53
5.1 Entities 54
5.2 Requirements 55
5.2.1 Services 57
5.2.2 Clusters 58
5.2.3 Domains 58
5.3 Policy Refinement 59
5.3.1 Ingress Refinement 62
5.3.2 Egress Refinement 63
5.4 Communication with Kubernetes 65
5.4.1 Kubernetes Java Client 66
5.4.2 Cilium Network Policy API 66
5.5 Service Communication 67
5.5.1 Services in the same Namespace 68
5.5.2 Services in different Namespaces 70

6 Validation 73
6.1 Use Case 1 73
6.2 Use Case 2 79

7 Conclusions 83

Bibliography 86

A Appendix A 89

B Appendix B 96

B.1 Set up 96

B.2 Creating Clusters 96

B.3 Intalling Cilium in the cluster 97

B.4 Installing the cluster-mesh 98

C Appendix C 101

C.1 Use Case 1 101

C.2 Use Case 2 105

C.3 Client 107

List of Figures

2.1 Example of a Kubernetes Cluster 18
2.2 Example of a Pod's structure 19
2.3 Example of a Service 20
2.4 Example of Replication model 23
2.5 Example of Split-by-service model 23
2.6 Architechture of Istio 25
2.7 Architechture of Calico 28
2.8 Architechture of Cilium 31
2.9 Cilium Multi Cluster 32

3.1 Per pod firewall image 38
3.2 Per node firewall image 38

4.1 External Model Of Multi Cluster Orchestrator 48
4.2 Internal Model Of Multi Cluster Orchestrathor 49

5.1 Example of a scenario with two domains and four clusters 60
5.2 Example of configuration with services belonging to the same namespace 69
5.3 Final request path to reach the real service backend from service frontend 69
5.4 Example of configuration with services belonging to different namespaces 71
5.5 Final request path to reach the real service backend from service frontend 72

6.1 Use Case 1 74
6.2 Request failure due to missing service in the cl3 77
6.3 Service2 created in cl3 77
6.4 Service2 External Service created in the default namespace 77

6.5 Namespaces of cl3 after Service2 creation 78
6.6 Successful request to Service2 78
6.7 Services in default namespace of cl3 after policy removal 78
6.8 Services in ns2 namespace of cl3 after policy removal 79
6.9 Namespaces of cl3 after policy removal 79
6.10 Use Case 2 80
6.11 Request from client-service to backend service 81
6.12 Request from client-service in cluster2 to client-service in cluster1 . 81
6.13 request from client-service to backend service 82
6.14 Policies after the backend service removal 82
6.15 Services in backend-ns namespace after service backend removal . . 82

A.1 Resource Graph 90

List of Tables

A.1 RESTful API Design 91

Listings

2.1 Example of a service specification . . . 21
2.2 Example of a Global Service . . . 32
3.1 Example of a Kubernetes Network Policy . . . 34
3.2 Example of a Calico Network Policy . . . 42
3.3 Example of a Cilium Network Policy . . . 43
3.4 Example of a multi-cluster Cilium Network Policy . . . 44
5.1 Example of a requirement . . . 55
5.2 Request to communication with a single service . . . 57
5.3 Request to communication with services of a cluster . . . 58
5.4 Request to communication with services of a domain . . . 59
5.5 Requirement example . . . 61
5.6 Ingress overwiew of Requirement Example 5.5 . . . 62
5.7 Final Ingress refinement Network Policy . . . 63
5.8 Egress overwiew of Requirement Example 5.5 . . . 63
5.9 Final Egress refinement Network Policy . . . 64
5.10 Example of a Context file . . . 65
5.11 YAML file of Service backend . . . 70
5.12 YAML file of Service backend in backend-namespace . . . 70
5.13 External Service . . . 71
6.1 Example of Service with a Deployment . . . 75
6.2 YAML file for Client Deployment . . . 76
6.3 Policy requirement that allows domain1 . . . 77
6.4 Policy requirement that allows service backend . . . 80
6.5 Example of a requirement that allows all services of cluster1 . . . 81
6.6 Example of a requirement that allows service backend . . . 82
B.1 Example of a kind configuration file . . . 97
B.2 Example of a Cilium installation with Helm . . . 97
B.3 Service cilium-etcd-external . . . 98
B.4 Example of a ds.patch file . . . 99
B.5 Commands to aplly the cluster mesh . . . 99
B.6 Example of a Cilium status report . . . 100
C.1 Template of a JSON DomainRequest . . . 102
C.2 Example of DomainRequest . . . 103
C.3 Example of a Policy Requirement request . . . 104
C.4 DomainRequest for use case 2 . . . 105
C.5 Cluster creation request . . . 106
C.6 Service creation request in a single cluster . . . 106
C.7 Service creation request in a domain . . . 107

C.8 postRequest method 107
C.9 Example of creation of a V2OrchestratorNetworkPolicy class from a yaml file 108

Chapter 1

Introduction

In recent years, novel cloud technologies have emerged and gained interest in the field of software development. At the same time, the application development methodology began to change, moving from the monolithic application idea to cloud-native application, composed of micro-services loosely coupled and independent from each other.
Developing applications in this way has permitted the various parts to be deployed in different servers and this has contributed to the start of developing new types of software called orchestrators, capable of managing all the resource optimization, deployment, and security parts of a data center or cloud environment. One of the most successful orchestrators at the moment is Kubernetes.
Kubernetes allows new applications, developed using micro-services, to be deployed in different parts of data-centers and to still be able to communicate with each other.

Since its first release, developers have mainly focused on the work of a single Kubernetes instance, called a cluster, to try to make everything work within a single cluster. The data center is then split into clusters and in each, it's possible to deploy an application working on different servers. Recently, the attention has shifted to the cooperation of several Kubernetes clusters, which can also belong to different data centers or companies, with each other to be able to develop parts of an application independently, and then make them communicate to form the entire application. Many projects have also been developed together with Kubernetes that extend it with security networking and monitoring functions.

Although Kubernetes doesn't currently provide any features for connecting multiple cluster instances and discovering the services that are within them, the various projects have started to move in this area by proposing different solutions.
Even if some parts have been developed using different methodologies, at the moment, the automation part of both the security functions and the creation of connections between clusters and services is missing in these projects. Automating these functions is important to prevent the possibility of a human error in the configuration both of security, which could lead to branches [1], and the connection between clusters and services that could lead to an unwanted interruption of communication with difficult resolution. Furthermore, if it refers to the communication between services of different companies, there is some information in the configuration that could be unknown and that requires cooperation between the parties

that could lead to misconfigurations and, therefore, to undesirable effects.

1.1 Thesis objective

The goals of this thesis, based on the previous introduction, can be divided in:

- Analysis of the current technologies for service-to-service communication, cluster mesh, and security policies
- Implementation of a solution placed at higher level respect clusters, based on an analyzed technology that extends it and provides automatic configurations for service security, discovery, and communications.

The first objective is the analysis of the current state of the art for what concerns the linking between clusters, how services can communicate and can be discovered if they are run in different clusters, and how security it's provided to protect services from undesired traffic. In particular, three projects will be analyzed to compare the different solutions they apply in those three fields. One of these three projects will be the base for the second objective which is to create a Multi Cluster Orchestrator that is placed at a higher level respect the Kubernetes clusters and that uses one of these technologies to automate service security, service-to-service discovery and communication, and creates a cluster mesh when it is needed that connects multiple clusters also if they belong to different companies or data centers.

This Multi Cluster Orchestrator is able, given some information, to carry out a refinement process of high-level requests and automate all configurations by creating links between clusters and services, but also applying security functions in such a way as to have applications that communicate only with those who are enabled to preserve them from undesired traffic.
In particular, in this work, the focus will be mainly on the refinement process of security policies, going to refine a higher-level request, and on the communication between services of different clusters.

1.2 Thesis description

After Chapter 1 briefly introduced the problems to challenge and the goals to achieve, the rest of the thesis is structured as follow:

- **Chapter 2:** describes Kubernetes with its main characteristics, starting from the basic concept of the pod to the advanced concept of multi-cluster, and three software that extends it: Istio, Calico, and Cilium. The focus of this chapter will be on their architecture and how they implement service discovery and Multi Cluster communications, highlighting what are their songstresses and weaknesses.

- **Chapter 3:** describes how the network isolation is provided in Kubernetes, focusing on the high and low-level implementations. First of all, will be introduced the Kubernetes Network Policy, which is the primary tool for securing a Kubernetes cluster. Then will be described how the network of Kubernetes works and what are the technologies for providing isolation and how they are related to network policies. Finally, there will be a description of how the three projects Istio, Calico, and Cilium extend the standard Kubernetes Network Policy and a final consideration about these three projects.

- **Chapter 4:** describes the model of the Multi Cluster Orchestrator, what are the main concepts, and how it is structured at a high level. For each step of the process will be described what the Multi Cluster Orchestrator must do, starting from requirements it needs to work, to the final single configuration for every cluster.

- **Chapter 5:** resumes the previous Chapter and describes how the concepts introduced are implemented and with what implementation's choices, focusing on policy creation, and service-to-service communication implementation and what are the implementation's choices applied.

- **Chapter 6:** provides some use cases to understand how the Multi Cluster Orchestrator works. In particular, there will be two use cases, one to understand how the Multi Cluster Orchestrator operates in a multi-domain environment, and the second describing how the Multi Cluster Orchestrator handles services running on multiple clusters.

- **Chapter 7:**summarizes which goals this thesis work succeeded in reaching, what are the main research directions that could be followed to improve the implementation, and which features could be introduced to enrich the capabilities of the framework.

- **Appendix A:**describes the design of REST-based APIs to interface with the Multi Cluster Orchestrator and the implementation of the corresponding RESTful web service.

- **Appendix B:** provides a guide to install a Cilium cluster mesh between clusters and the installation of Cilium and its components.

- **Appendix C:** describes the steps to do in order to replicate the use cases in Chapter 6. The last part will introduce a client created to simplify the interaction with the Multi Cluster Orchestrator

Chapter 2

Technologies

One of the first chapters of this work is dedicated to the description of technologies representing the actual state of the art for service and multi-cluster communications. Understanding what are the main features and weaknesses will help to introduce the reason that led to thinking of this thesis.
First of all, will be introduced Kubernetes with its characteristics and main features, which represents, at the time of writing this work, one of the most used solutions for management and deployment of containerized applications.
Kubernetes network can be easily extended with some plugins that use a different approach to manage security and network topology.
The second part of the chapter will describe three projects that extend the Kubernetes network for the improvement of security, service, and multi-cluster communication. For all three projects will be given a description of the architecture and how they implement the multi-cluster and service communications, while the next chapter will be dedicated to the description of security policies and a final comparison between the three technologies, highlighting what are their strength and weaknesses.

2.1 Kubernetes

Kubernetes is an open-source platform that helps to automatically manage, scale and deploy containerized applications [2]. It was created by Google and became an open-source software in 2014.
Kubernetes is technically defined as a Cloud Orchestrator. A Cloud Orchestrator is a software that automates tasks computing, networking, and storage infrastructure on behalf of user workloads and is designed also to serve as a platform for building an ecosystem of components and tools to simplify the deployment, scaling, and managing of cloud applications.
Thanks to Kubernetes, all manual tasks involving scaling and deployment of applications are automated and optimized, permitting more efficient management of cloud infrastructure.

The next sections will present the components of Kubernetes starting from the concepts of cluster and pods, which are basic components, and continue with the introduction of services and multi-cluster that are more advanced features.

2.1.1 Cluster

When a Kubernetes instance is created, it's organized in a cluster. A cluster consists of a group of compute nodes, called worker nodes, and at least a management node named the master node.

The master node is responsible for the management of the entire cluster, scheduling the new deployments in the worker nodes, optimizing the resources of the cluster(for example scheduling a new deployment in a worker node that is using fewer resources), and exposing the Kubernetes Application programming interfaces (APIs) for hosts inside and outside the cluster.

Worker nodes, instead, are the execution units of the Kubernetes cluster: they are in charge of exposing compute, networking, and storage resources to applications deployed in the cluster. Each node runs a container run-time (for example Docker), an agent that communicates with the node master via API and additional components for logging, monitoring, service discovery, and other tasks based on plugins.

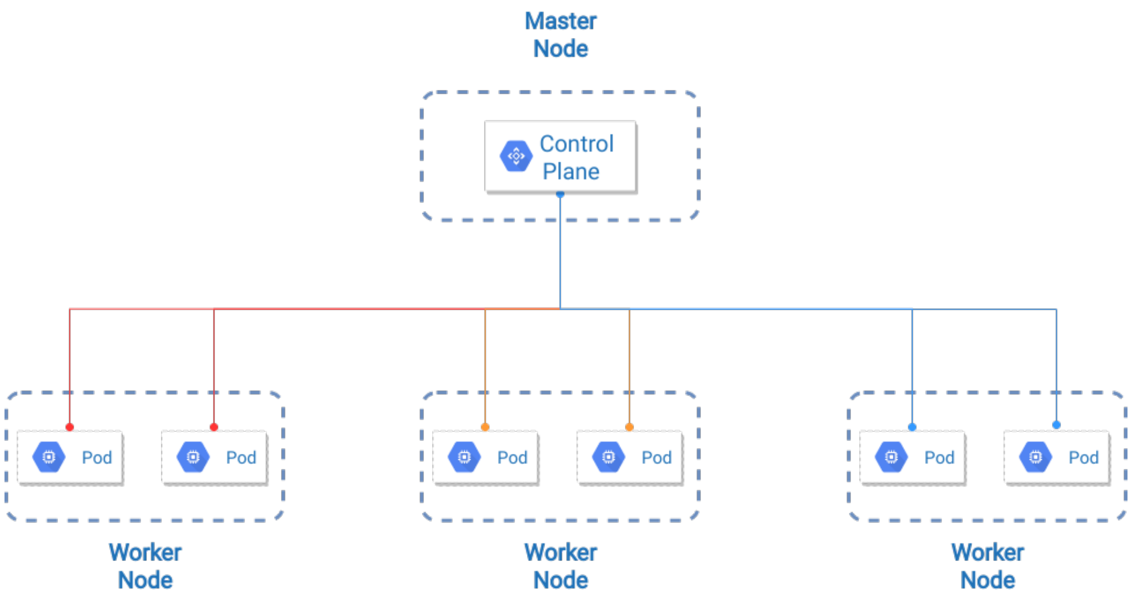


Figure 2.1. Example of a Kubernetes Cluster

2.1.2 Pod

A Pod [3] is the smallest deployable unit of computing that Kubernetes can create and manage.

Creating a pod means defining a REST (Representational state transfer) object using the YAML (Yet Another Markup Language) language that might be composed of one or more containers, sharing storage and network resources, and containing specifications about how containers have to be run.
The containers in a pod run in a shared context and they are always located and

scheduled in the same node. Also if the containers of pod share the same context, which is a set of Linux isolation primitives (like namespaces and cgroups), the whole application, composed by the union and interaction of these containers, may have further sub-isolations applied.

In summary, a Pod in Kubernetes can be used in two ways:

- Run pod with a single container: this is the simplest use case to run a pod, in which the Pod is used as a wrapper around one single container; Rather than managing containers directly, Kubernetes manages Pods.
- Run a pod with multiple containers: an application, that requires more than one container to run, can be deployed in a single pod, making containers strictly coupled and sharing the same resources. In this scenario the pod acts as a wrapper, sharing storage resources and an ephemeral network identity among all containers and acting as a single unit.

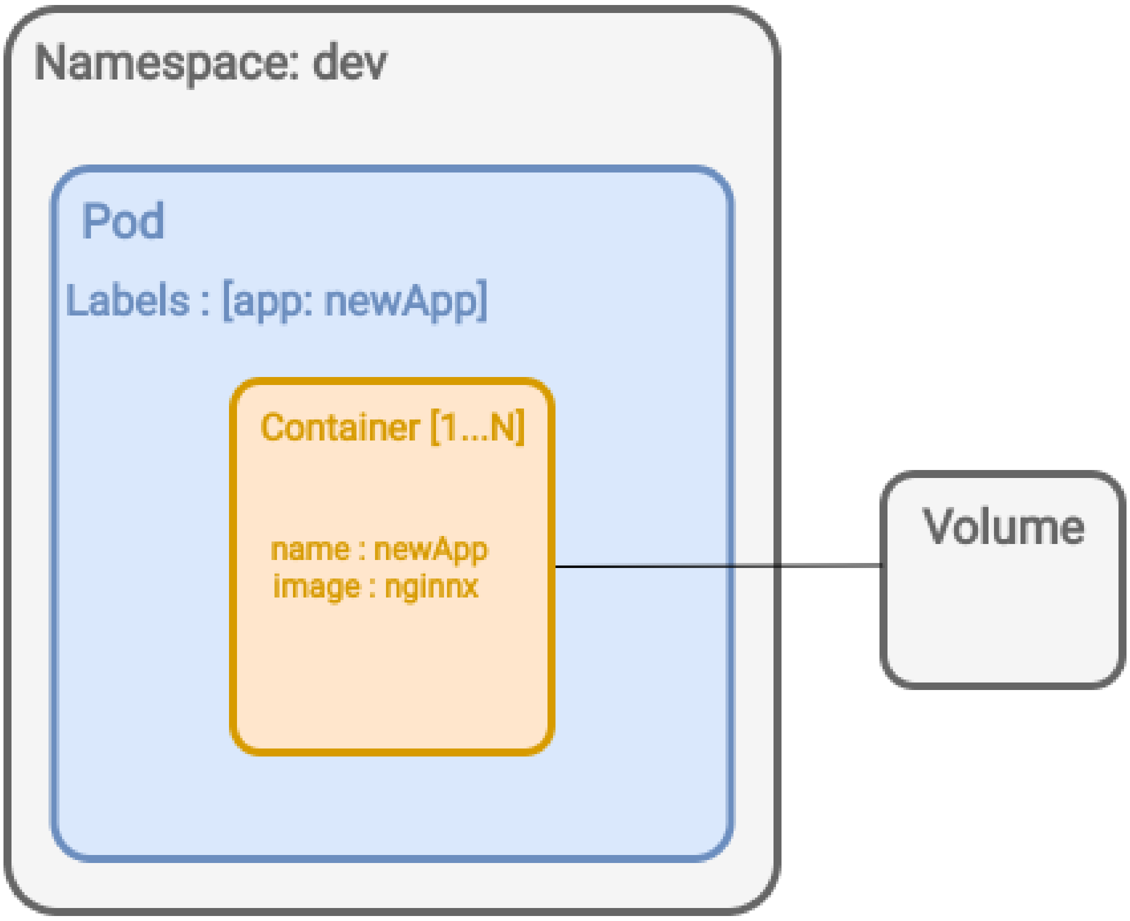


Figure 2.2. Example of a Pod's structure

Each Pod is defined and must run only a single instance of a given application. If it's required to scale an application horizontally (to provide more overall resources by running more instances), it should use multiple Pods, one for each instance.

2.1.3 Service

A service is an abstract way to expose an application running on a set of Pods as a network service [4].
The reason why services exist is that pods cannot be a reliable resource for the service discovery: Kubernetes gives pods unique IP addresses and a single DNS name for a set of pods running the same application, load-balancing across them (to avoid application's modification for the service discovery), but based on the cluster's state and the use of resources, pods are created and destroyed to match the desired state of the cluster. This makes a pod an ephemeral resource that can't be used for network service discovery.
A service is then an abstraction that defines a logical set of pods and a policy by which to access them. The figure 2.3 shows how a request made for a service is transferred to pods associated with it.
Usually, the association between a set of pods and service is done using a selector, which is a key-value pair that helps to identify and group pods. Once pods are linked to a service, they can be reached using always the same IP address, which is the service IP. In this way, a service running into a pod can contact a service in another pod always with the same DNS name and using the IP of the service. Once a request to service IP is done, it will be passed to a proxy(Kube-proxy) that will forward the request to one of the backend pods linked to the service, and it's Kubernetes which dynamically update the list of pods that run the service.

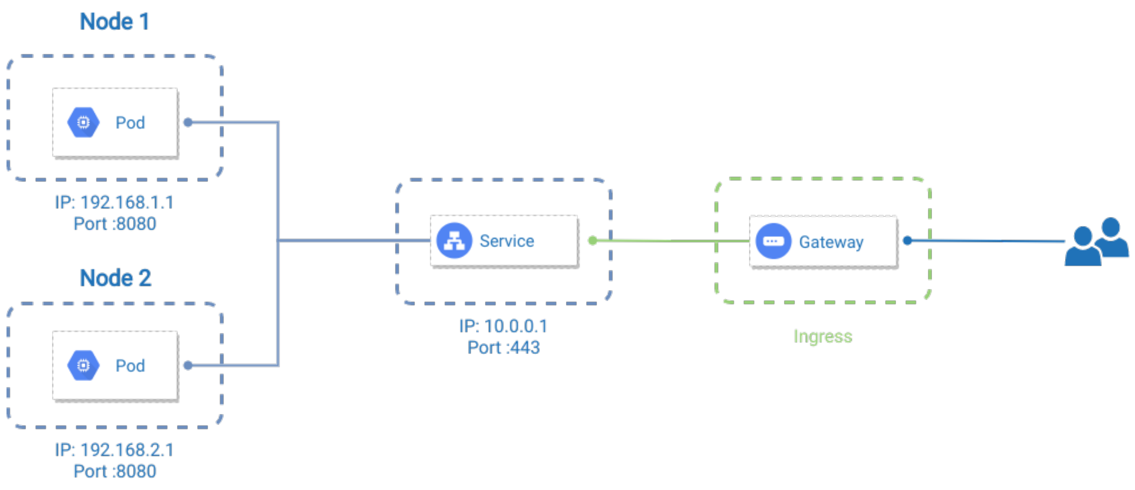


Figure 2.3. Example of a Service

Based on the visibility scope of the service, there are four types of services:

- **Cluster IP services:** This is the default service type used if no type is specified and gives an internal cluster IP to the service. In this way, only entities internal to the cluster can reach the service.
- **Node port:** This is the type used to expose a service outside the cluster. In this type, services are created on each node's IP at a static port and, when accessed, will route on a ClusterIP service (which is automatically created when a Node port service is defined). To contact a service outside the cluster, requests should be done to one of the cluster's nodes at ¡NodeIP¿:¡NodePort¿.
- **Load balancer:** This type of service is created when a cluster is inside a cloud provider that provides a load balancer. This load balancer will route all the requests to an internal service, which can be of CusterIP or Nodeport type (automatically created when Load balancer service is defined).
- **External name:** This type of service is used to refer services outside the cluster. It has a parameter called external name that is used to return a Canonical Name record (CNAME) with its value. With this value, it will be possible to access the actual service.

A Service in Kubernetes is a REST object, similar to a Pod. Like all others REST objects, it can be created doing a POST to the Kubernetes API server to create a new instance. An example of a Service definition is the one in Code Box 2.1:

Listing 2.1. Example of a service specification

```
apiVersion: v1
kind: Service
metadata:
  name: frontend—service
spec:
  selector:
    app: frontend
  ports:
 - protocol: TCP
  port: 80
  targetPort: 8080
```

This specification creates a new Service object named "*frontend-service*", which targets TCP port 8080 on any Pod with the *app=frontend* label. Kubernetes then assigns an IP address to this service, which will be used by service proxies to load balance across pods linked to the service.

When a new pod is created and linked to a service, its IP is added, by the Controller for the Service selector, to another object called Endpoint object(which has the same name of the service, in this case "*frontend-service*"), that is used by proxies to retrieve IP addresses.
Port definitions in pods have names, and it's also possible to reference these names in the *targetPort* attribute of a Service. This works even if there is a mixture of pods in the Service using a single configured name, with the same network protocol

available via different port numbers. This offers a lot of flexibility for deploying and evolving Services. For example, the port numbers that pods expose can be changed, without breaking clients.

The default protocol for Services is TCP and if no type is specified in the *spec* field, *ClusterIp* is used as the default type.
Kubernetes supports multiple port definitions on a Service object if any service needs to expose more than one port(each definition can select also different protocols).

2.1.4 Multi Cluster

Multi cluster is a strategy that permits the deployment of applications across multiple Kubernetes clusters. This could be interesting for new applications based on the cloud, having parts separates in micro-services that interact with each other and that could belong to different companies. Using this strategy can improve:

- **availability:** a service could be available in multiple clusters and if one cluster is not reachable the other can be contacted.
- **isolation:** a particular part of an application or service can run in a single cluster isolated from other parts.
- **scalability:** multiple clusters offers more resources for the application

Also, it could be possible to concentrate part of applications, that needs particular geographic or certification-specific regulations, in one cluster to ensure compliance with different and conflicting regulations. Finally, the safety and speed of software delivery could also be increased, having a separate cluster for the developing team that selectively exposes only services available for testing and release.
Based on how the separation of application is done across clusters, there are two possible models to reach the deployment of an application in a multi-cluster environment:

- Replication for increased availability
- Split-by-service for increased isolation

In the first model presented in the figure 2.4, each cluster runs a full copy of the entire application. With this approach, if there are clusters in different availability zones(i.e clusters running in data centers in Europe and others running in America) the application can scale globally and the traffic can be routed to the nearest cluster respect where the user is located. This model helps also in the fail-over process. For example, if it's coupled with software that checks the health of clusters and can load balance requests across them when a cluster is unreachable, requests are forwarded to any other reachable cluster.

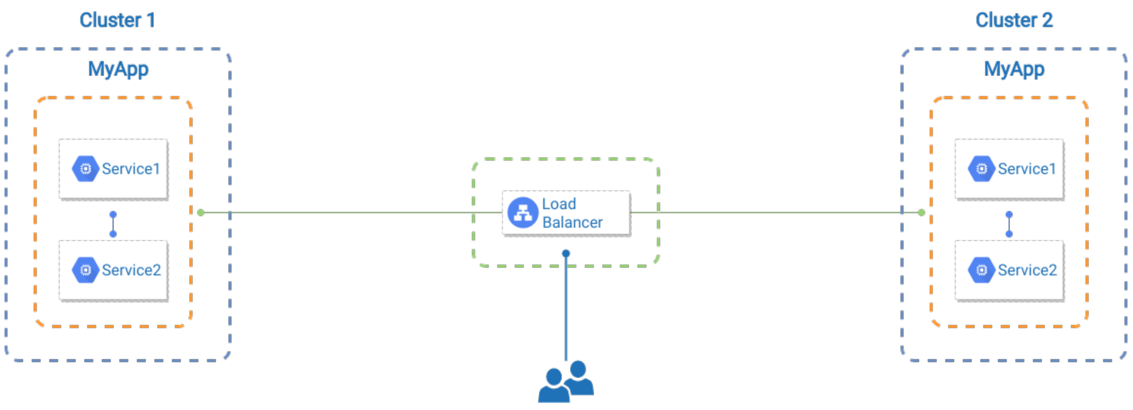


Figure 2.4. Example of Replication model

In the second model instead (figure 2.5), the application is divided across multiple clusters, based on its service composition or different global compliance requirements. This approach is also good if the application is composed of services belonging to different companies: all parts can be isolated from each other, but the complexity is increased(need communication between clusters and more security has to be applied). Other advantages of this model are the independence of parts, speed, and safety during the development and delivery of applications. Different teams can work on their parts in an isolated way, and connect the new releases when all it's finished tested and validated.

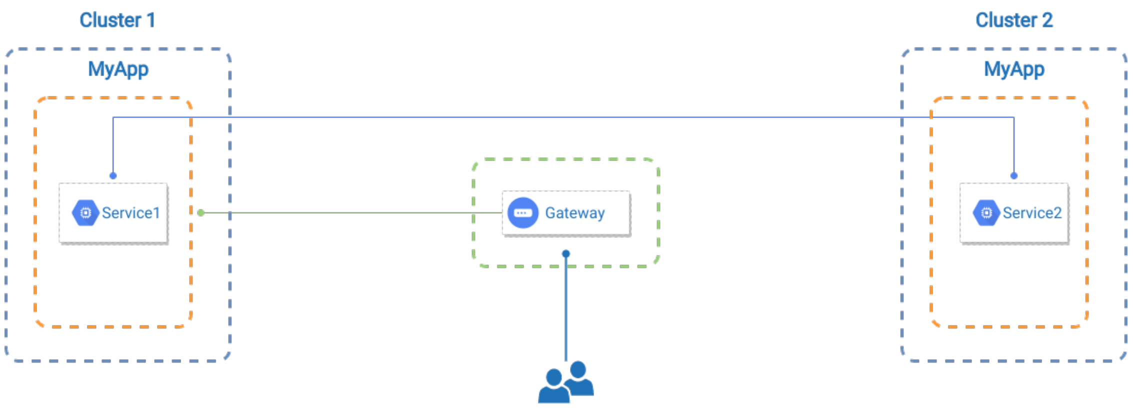


Figure 2.5. Example of Split-by-service model

To realize this goal, it's important that clusters can be able to communicate securely and that is possible to decide the policy to apply to a single service. Three projects can help in this regard and will be arguments for the next sessions. [5]

2.2 Istio

The first project that will be described is a software that helps in the service-to-service communication: Istio.
Istio is an open-source service mesh that layers transparently onto existing distributed applications. [6] As described by Istio's developers

"A service mesh is a dedicated infrastructure layer that can be added to applications. It allows to transparently add capabilities like observability, traffic management, and security, without adding them to the code. The term 'service mesh' describes both the type of software that is used to implement this pattern, and the security or network domain that is created when that software is used."[7]

With the service-mesh provided by Istio, developers can easily move from the development of monolithic applications to cloud-native apps, based on independent services loosely coupled between each other and that communicate, as they can rely on a software that implements security features, load balancing, fault recovery, testing, discovery, metrics, and monitoring for service-to-service communication.

The next sections will present the Istio Architecture and how the multi-cluster and service-to-service communication features are implemented.

2.2.1 Architecture

Istio's Architecture is divided into two main components: data and control plane.
The data plane is the component in charge of the communication between services. When the service mesh is created, the data plane can make decisions based on the type of traffic or who is the sender or receiver. To achieve this the data plane uses a proxy(which is configured by the control plane) and that intercepts all network traffic, allowing a broad set of application-aware features based on configurations.
When a service is started in the cluster or is running in a Virtual machine, Istio deploys an Envoy proxy along with them.
An Envoy proxy is a component instantiated by Istio to interact with data plane traffic. It is deployed as a sidecar to services(i.e. a container coupled to the pod) in the traffic mesh and intercepts all inbound and outbound traffic and provides other features like dynamic service discovery, load balancing, circuit breakers, health checks, and fault injection.
With this sidecar deployment, Istio can enforce policy decisions, traffic management and extract data that can be sent to monitoring systems, providing information about the status of the mesh and its behavior.

The control plane, instead, based on its knowledge of services and the desired configuration given by the user, dynamically programs the Envoy proxy servers, updating them as the rules or the environment changes. Istio's Architecture is summarized in the figure 2.6. Every decision is made by three components: Pilot, Citadel, and Galey:

- **Pilot:** is the component that provides service discovery for the Envoy sidecars, traffic management, deciding the best routes for every service, and resiliency with the application of timeouts or circuit breakers. Pilot can also retrieve Envoy configurations, converting high-level routing rules traffic behaviors, and propagate these configurations to the sidecars at runtime. Finally, it can abstract service discovery mechanisms based on a specific platform and transform them in a standard format consumable by any sidecar conforming to the Envoy API.

- **Citadel:** is the component in charge of the service-to-service and end-user authentication using built-in identity and credential management. Istio uses only application layer security so Citadel can enforce policies based on service identity rather than on layer 3 or layer 4 network identifiers.

- **Galley:** is the component in charge of the Istio's configuration validation, ingestion, processing, and distribution. It is responsible to obtain the user configuration from the underlying platform (e.g. Kubernetes) and propagate this information to all other Istio's components.[8]

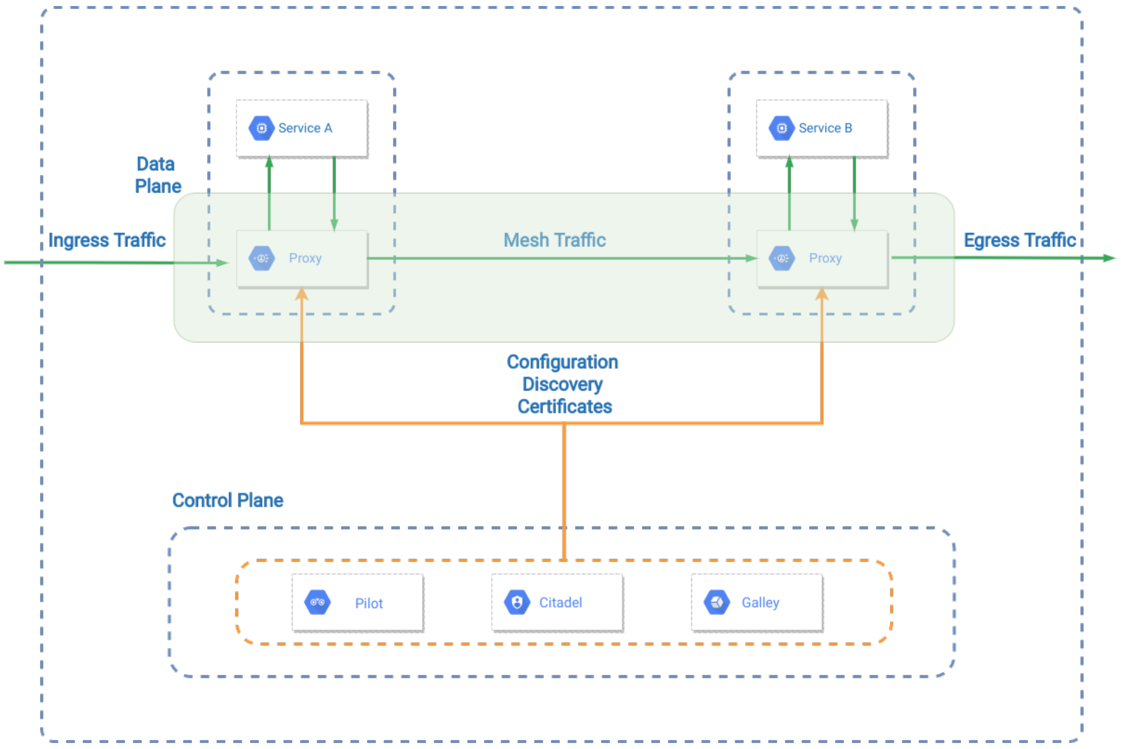


Figure 2.6. Architechture of Istio

2.2.2 Multi-cluster

Istio uses a Multicluster service mesh for communication between services in different clusters. A Multicluster service mesh is a mesh composed of services running in more than one cluster, which can communicate normally as they are deployed in the same cluster. In a Multicluster service mesh, every service with the same name in all clusters connected is considered as the same service, different from a loosely-coupled service mesh where two clusters may have different definitions of the same service (in this case when clusters are integrated and linked together there must be a process that will remove all the ambiguity about services name).

In a Multicluster service mesh then, all services look the same to clients, independently of where the workloads are running, transparently hiding if an application is deployed using services running in multiple clusters or just a single one. To achieve this behavior, services must be coordinated and managed by a single logical control plane, but there could be different physical Istio control planes that in a distributed way will act as a single one.
Based on if the physical control plane is one or distributed among clusters, there are two possible deployments:

- Multiple Istio control planes in which service and routing configurations are replicated among all clusters
- Shared Istio control plane chosen among clusters that can access and configure the services in more than one cluster.

In a multiple control plane topology, in each cluster is installed an Istio control plane (which is the same for all clusters) and a cluster is responsible for the management of its endpoints. To configure the Multicluster service mesh, which is a single logical service mesh, other than a single control plane every single cluster needs also an Istio gateway for traffic forwarding, a common root Certificate Authority (CA) for entrusting security and service entries for service discovery. When applying this deployment, there are no special network requirements (the only one is to have a cloud provider's load balancer implementation needed to load balance requests among clusters), and is the best solution to start when clusters are not connected yet.
With this approach, all shared services need to be replicated in all clusters as all namespaces in which this service's pods run. All clusters are in a shared administrative control for security and policy enforcement which needs also to provide a common root CA configuration in such a way that all traffic over the Istio's gateways is secured.

In the shared control plane deployment, instead, a single Istio control plane is configured which runs in one of the linked clusters. The control plane's Pilot of this cluster is in charge to manage services on the local and remote clusters, configuring all the Envoy proxies coherently.
This approach works also if the underlying network is not the same among clusters: creating Istio's Gateways, which can act as a VPN, all traffic can be forwarded through gateways and act as a single network. If a sidecar sends requests to services

running in its clusters, the normal service IP is used and everything works as the single cluster configuration. Instead, if a request has to be done to a service workload running in a different cluster, the traffic is sent to remote Istio's Gateway which will provide a connection with the service.

The two deployment approaches described can be mixed. If the Multicluster service mesh is large, some clusters could share the control plane while others can have their own. Which approach to use depends on the requirements of applications, on the features and limitations of the underlying cloud deployment platform. [9]

2.3 Calico

Calico [10] is an open-source community project that provides networking for containers and virtual machines.
One of the most important projects of the Calico community is the development of the Kubernetes Container Network Interface (CNI) which provides network and security implementation for pods in a Kubernetes cluster.
the next sections will describe Calico's CNI Architecture, based on a BGP mesh creation, and how Calico provides service-to-service communication and Multicluster connection with its management tool: Calico Enterprise.

2.3.1 Architecture

Calico implements the Kubernetes CNI as a plug-in on the third layer, also known as Layer 3 or the network layer, of the Open System Interconnection (OSI) model. It also provides networking and policy enforcement for containers and pods.
In the first place, Calico creates a flat Layer-3 network, assigning to every pod a fully routable IP address, then it creates a BGP mesh between all cluster nodes and broadcasts container networks routes to only worker nodes. Each worker node has its subnet(assigned by Calico from a larger subnet), which serves connectivity to pod subnets that are hosted on the host, and is configured to act as a Layer 3 gateway for that subnet. The BGP mesh advertises all of the local routes that the worker nodes own to all peers participating in the mesh. It is possible to include BGP peers external to the cluster, but how many BGP advertisements these external peers receive is conditioned by the cluster size. When routing pod traffic, Calico uses the node's local route tables and iptables in such a way that all pod traffic traverses iptables rules before they are routed to their destination. The Architecture of Calico is shown in the figure 2.7 To manage policy and networking Calico uses the following components:

- **Felix:** Agent Daemon running on machine hosts that programs routes and Access Control Lists (ACLs), and anything else required on the host to provide desired connectivity for the endpoints on that host.

- **Bird:** Internet routing daemon running on every node that hosts a Felix agent. This component gets routes from Felix and is in charge of distributing them to BGP peers on the network to guarantee inter-host routing.

- **Confd:** Component that periodically monitors the Calico datastore for changes to BGP configuration. It is also aware of global defaults changing like AS number, logging levels, and IPAM information. When there is an upcoming update, Confd dynamically generates BIRD configuration files based on the modification of the datastore and triggers BIRD to load the new files.

- **Dikastes:** Optional component, runs on a cluster as a sidecar proxy to Istio Envoy and enforces network policy for Istio service mesh using Iptables for levels 3-4 and using Istio policy for higher levels. [11] [12]

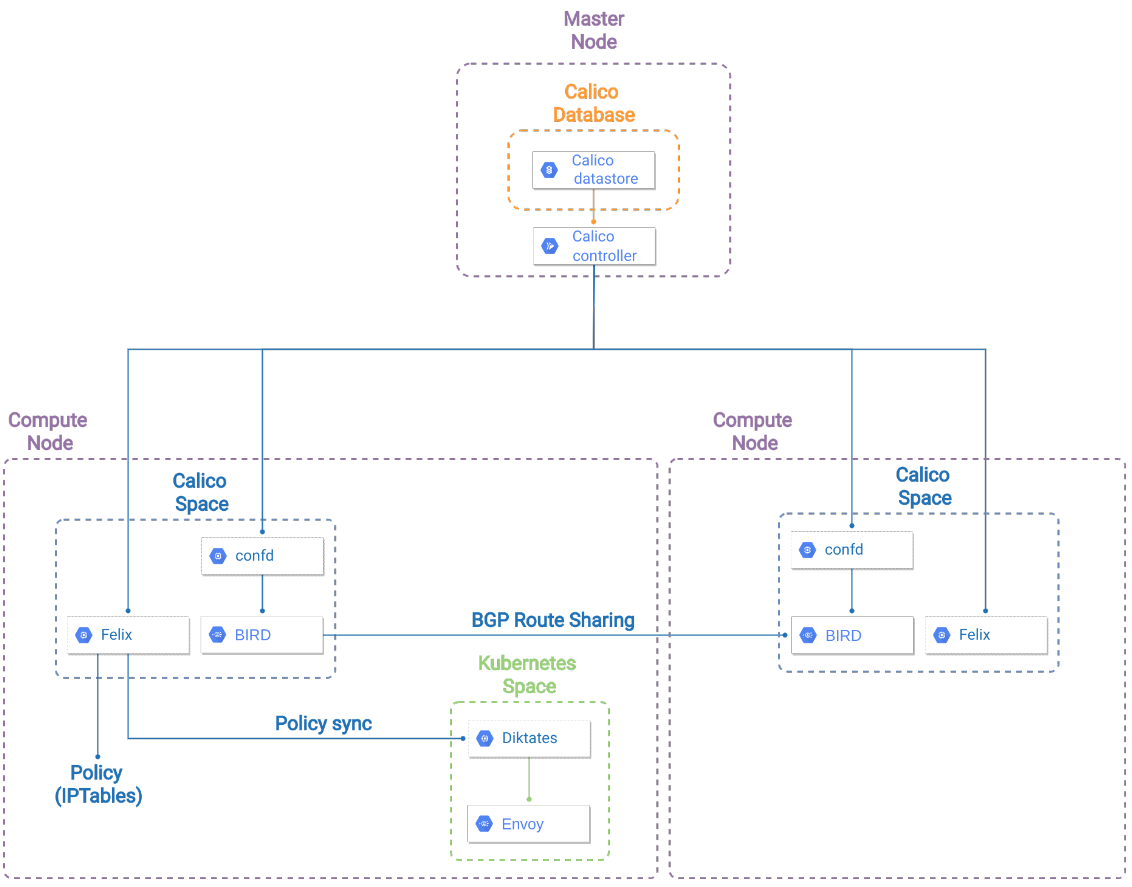


Figure 2.7. Architechture of Calico

2.3.2 Multi-cluster

To achieve multi-cluster and service connection, Calico uses its management tool that is Calico Enterprise.
Calico Enterprise is a solution working on Kubernetes primitives, which are extended with other features to work also on Calico's custom resources. The tool aim is to enable security and observability across multi-cluster, multi-cloud, and hybrid cloud environments, and provide a single point of configuration to ensure consistent application of security controls.
Calico Multi-Cluster Management is structured with a centralized management plane with a single point of control for multi-cluster environments. With this centralized control plane, routine maintenance is simplified and speeded up, in fact, for example, instead of logging into all clusters one at a time to make a policy change, with a single log-in to Calico Enterprise it's possible to apply policy changes consistently across all clusters. Once network security controls are defined, it is also possible to apply them automatically to new clusters as soon as they are added to the management tool.
It is possible to create policies in one cluster that reference pods in another cluster using the federated identity. Federated services provide service discovery of remote pods in another cluster. With these two features, it is possible to create fine-grained security controls between multiple clusters.[13] Using federated tiers and federated policies, security policies can be defined and applied across all clusters, or to a specific group of clusters. If multiple clusters are deployed, federated tiers and policies can extend security controls to each existing and new cluster. This reduces duplication of policies (and maintenance of identical policies per cluster) to simplify the creation and maintenance of security controls. Calico enterprise is not an open-source software and it's possible to use it only by having a subscription to the Calico cloud, for this reason, all details about service discovery and multi-cluster communication are not provided by developers.[14]

2.4 Cilium

Cilium is an open-source software for transparently securing the network connectivity between application services deployed using Linux container management platforms like Docker and Kubernetes. Like Calico, the Cilium project has also developed its own Kubernetes CNI that can be installed easily with cloud tools like Helm. [15]
The main peculiarity of Cilium is that it has focused all the CNI implementation on the new technology of eBPF, from network functions to security policies, making the data plane really fast and able to process a large number of requests and match a big number of policy rules.
In the following sections will be presented the Cilium architecture and how it connects clusters with its Cilium cluster-mesh, providing also the service-mesh communication.

2.4.1 Architecture

When Cilium is deployed in the cluster, some components are instantiated on each node of the cluster. Those components are:

- **Cilium Agent (Daemon):** This component is installed as a daemon in the userspace of all cluster's nodes and it's used to setup networking and security for containers running in the pods (when they are created) and uses plugins to interact with the container runtime and Kubernetes. This component is also in charge to provide an API for configuring network security policies and extracting network visibility data.
- **Cilium CLI Client:** this component is a simple CLI client used for communication with the local Cilium Agent, for configuring network security or visibility policies.
- **Linux Kernel eBPF:** this is not a really Cilium component as it is already integrated as capability of the Linux kernel to accept compiled bytecode that is run in various points of the kernel(called hooks). Cilium uses eBPF to compile programs that are executed by the Linux kernel in the key points of the network stack to have visibility and control over all incoming and outgoing network traffic of all containers.
- **Container Platform Network Plugin:** Each container platform (i.e. Docker, Kubernetes) has its plugin model for how external networking platforms integrate. In the case of Docker, each Linux node runs a process (cilium-docker) that handles each Docker libnetwork call and passes data/requests on to the main Cilium Agent.

In addition to components that run on each node of the cluster, Cilium maintains and manages a key-value store to share data between Cilium Agents running on different nodes. The full architecture of Cilium is summarized in the figure 2.8. [16]

2.4.2 Multi-cluster

Cilium Multi-cluster or ClusterMesh is a mix of Istio's and Calico's multi-cluster because it has the concept of service mesh of Istio but can be extended also to network policies and the federation concept of Calico Enterprise called identity-based here.
ClusterMesh can route Pods traffic across multiple Kubernetes clusters using tunneling or direct-routing and without the use of any gateways or proxies. It is provided also a transparent service discovery using standard Kubernetes services enriched with some annotation (which will be explained later) and the standard Kubernetes DNS (coredns/kube-dns). Finally, all traffic of local nodes and external clusters is encrypted.
To achieve this multi-cluster communication Cilium needs to unify the control plane of all the clusters and to do it needs some requirements :

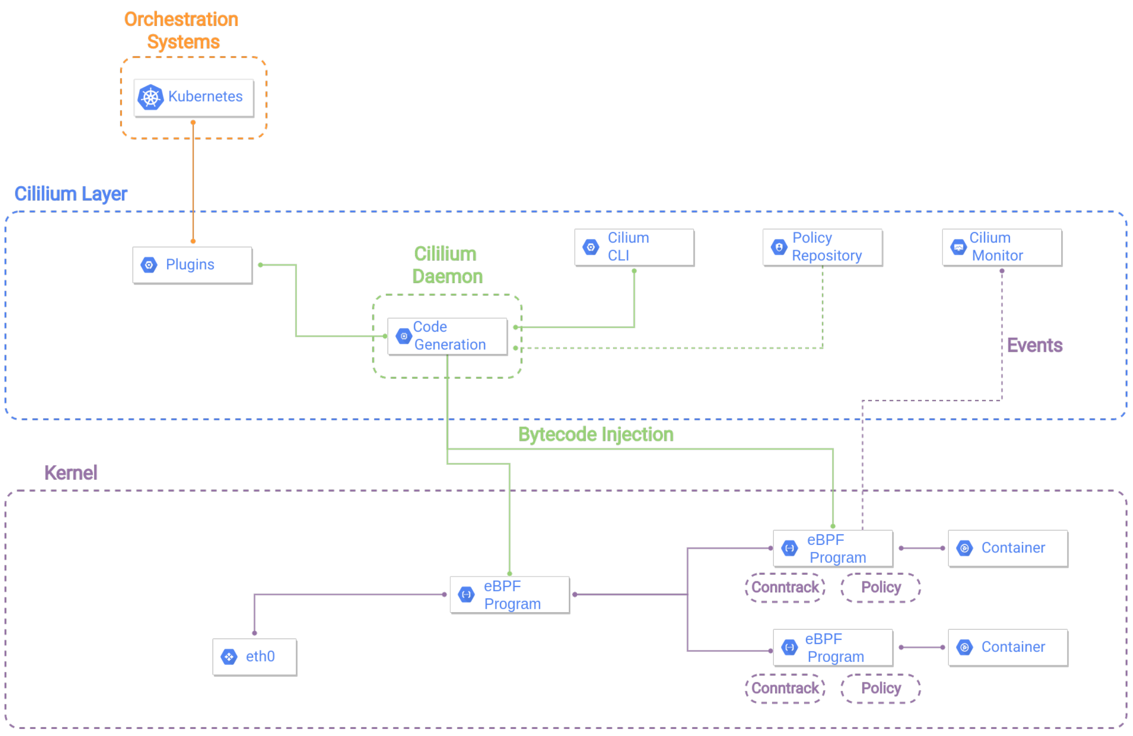


Figure 2.8. Architechture of Cilium

- All Kubernetes worker nodes must be assigned a unique IP address and all worker nodes must have IP connectivity between each other
- All clusters must be assigned unique PodCIDR ranges
- Cilium must be configured to use etcd as the kvstore
- The network between clusters must allow inter-cluster communication. Based if Cilium is configured to work in direct-routing or tunneling mode, the firewalling requirements might change.

The figure 2.9 shows what are the services unified from cilium when creating a ClusterMesh.

To unify the control plane, Cilium uses its key-value store, called cilium-etcd, which is based on the standard Kubernetes etcd. The cilium-etcd is not unique but each Kubernetes cluster maintains its etcd cluster containing the cluster state. Clusters can read the state of other cilium-etcd using their Cilium-agent, to watch for changes and replicate the multi-cluster relevant state into their cluster. Cilium-agents will never connect directly to the store, but each cluster exposes its cilium-etcd via proxies. Accessing the cilium-etcd is always read-only, so the state from multiple clusters is never mixed in etcd itself and ensures also a circumscription of failures: if a cluster has failures this will never propagate in other clusters.
As said before, ClusterMesh uses a similar approach as Istio's mesh for service discovery. In particular, the Cilium service-mesh, transparently to the existing

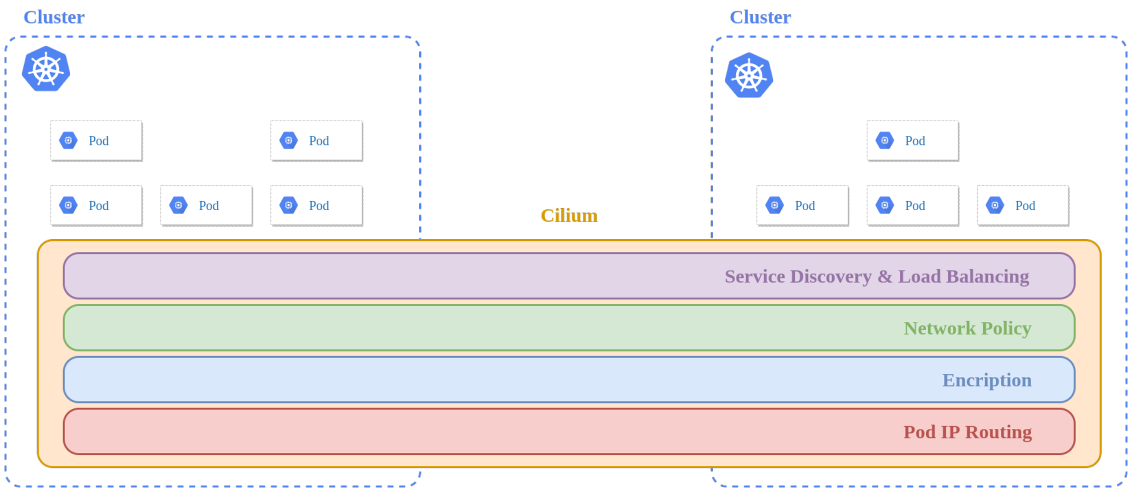


Figure 2.9. Cilium Multi Cluster

Kubernetes deployments, can perform service discovery, using the standard Kubernetes services, adding only a few more information.

Listing 2.2. Example of a Global Service

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  annotations:
    io.cilium/global—service: "true"
spec:
  type: ClusterIP
  ports:
 - port: 80
  selector:
    app: frontend
```

In the Code Box 2.2 there is an example of how service discovery can be performed using the Cilium ClusterMesh: Cilium monitors Kubernetes services and endpoints and watches for services with an annotation *io.cilium/global-service: "true"*. All services which match this requirement, and that have the same name and namespace, are automatically merged along with all their information and can be contacted from all clusters having that service. During this process, Kubernetes and its DNS are not aware of global services in other clusters because each service continues to maintain its ClusterIP. The DNS server will always return a ClusterIP valid only in the local cluster and Cilium will transparently load balance requests to endpoints in all clusters, based on the standard Kubernetes health-checking logic. [17]

Chapter 3

Network Isolation in Kubernetes

In the last Chapter was introduced the Kubernetes Orchestrator with its most relevant features, in particular, it was focused on all those technologies which allow clusters communications and service-to-service communication.
Most of cluster-mesh technologies analyzed focus their operations on creating a logical control plane that is in charge to provide connectivity between pods in all clusters of the mesh. Especially if clusters belong to different companies, some namespaces, pods or services could be wanted to remain private, or would be nice to decide some policies to accept only some requests from specified entities.
Isolation is then an important requirement when connecting clusters, and Kubernetes uses the Network Policy resource to achieve this goal.

Recent works like [18] and [19], have highlighted also the importance of performances when security is applied because the number of connections to control can rise very highly. It's important then know also what technologies are used when security is applied to choose the one that can provide also the best performances.

The following Chapter will then describe at first what is a Network Policy and how its structured, then will be introduced and described the various level of isolation and how it's implemented at a low level using Linux kernel components like iptables or eBPF. In the final part of the chapter there will be a description of how the three projects analyzed in the previous Chapter (Istio, Calico, Cilium) implements Network Policies and a final section that compare the three technologies, based on the information of this and previous Chapter.

3.1 Kubernetes Network Policy

Network policy is the primary tool for securing a Kubernetes network. It allows to easily restrict the network traffic in the cluster so only the traffic that is wanted to flow is allowed.
By default, if no Network policy is applied, pods are non-isolated and they accept traffic from any source. Once a network policy selects a pod or group of pods via a selector, pods become isolated and they will start to reject any connections that are not allowed by any network policy.

Network policy is implemented by the network plugin: a network plugin is a component that provides the implementation of network policy (some plugins translate network policies as rules in Iptables, others as eBPF programs in the Linux kernel) and is responsible for inserting a network interface into the container network namespace of pods that helps in the process of policies rules creation. If a NetworkPolicy resource is created without any network plugin installed that implements it will have no effect as it will not be possible to insert rules in the Linux kernel (via Iptables or eBPF).

If there is more than one network policy that selects a single or multiple pods, their effect will be additive: pods will be restricted to what is allowed for the union of those policies rules, this is valid for the ingress and for the egress specification and the order in which policies are evaluated doesn't affect the finale result.

If two pods are restricted by network policies and want to communicate, it's important that there is a policy allowing the egress traffic to the destination pod (in the source Network policy definition) and a policy allowing the ingress traffic from the source pod(in the destination network policy definition), otherwise, the traffic will be denied.
An example NetworkPolicy might look like this:

Listing 3.1. Example of a Kubernetes Network Policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my—network—policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: backend
policyTypes:
— Ingress
— Egress
ingress:
— from:
 - ipBlock:
      cidr: 172.20.0.0/16
      except:
     - 172.20.1.0/24
 - namespaceSelector:
      matchLabels:
        project: test
 - podSelector:
      matchLabels:
        role: frontend
        namespace: frontend—namespace
ports:
— protocol: TCP
  port: 6379 5480
egress:
— to:
 - ipBlock:
      cidr: 10.0.4.0/24
ports:
— protocol: TCP
  port: 5978 8800
```

As with all other Kubernetes configurations, a NetworkPolicy needs *apiVersion*, *kind*, and *metadata* fields. The field *spec* specifies all the information needed to define a particular network policy in the namespace, in particular the *podSelector* which selects the grouping of pods to which the policy applies. In the example are selected all pods that have the label "*role=backend*" in the *default* namespace.If in the policy it is not specified a selector, it will be applied to all pods belonging to the specified namespace.

Another important field is the *policyTypes*, that is a list which may include either Ingress, Egress, or both specification. The *policyTypes* field indicates if the current policy has to be applied for ingress traffic to the selected pod, egress traffic from selected pods, or both.
For the ingress list each rule specified allows traffic matching both the *from* and *ports* fields. The example policy contains one single rule matching traffic on a single port, and that arrives from one of three sources, specified via *iPBlock*, *namespaceSelector* and *podSelector*.

The egress list instead contains the allowed egress rules. As for the ingress list, each rule allows traffic matching both the *to* and *ports* fields. in the example, the network policy contains only one rule, which allows all traffic on a single port that is intended for any destination in the 10.0.4.0/24 subnet.
Summarizing, the example NetworkPolicy isolates pods with label "*role=backend*" deployed in the "*default*" namespace for both ingress and egress traffic (if they weren't already isolated).
Ingress rules allows connections to all pods in the "*default*" namespace with the label "*role=backend*" on *TCP* port *5480* from:

- any pod with the label "*role=frontend* and that is in the "*frontend-namespace*" namespace
- any pod in a namespace with the label "*project=test*"(namespaces can have labels)
- IP addresses in the ranges 172.20.0.0/16 excluded the subnet 172.20.1.0/24

Egress rules allows connections from any pod in the "*default*" namespace with the label "*role=backend*" to subnet 10.0.4.0/24 on TCP port 8800. There are still some things Network policy can't do and are done using other Kubernetes features like:

- **forcing internal cluster traffic to go through a common gateway:** this might be best served with a service mesh or other proxy
- **anything TLS related:**service mesh or ingress controller for this might be used for this intent
- **targeting of services by name:** it's possible to target pods or namespaces by their labels, which is often a viable workaround ability to log network security events.[20] [21]

3.2 Network isolation

Before introducing the concept of isolation in Kubernetes it's important to understand how the network works. The Kubernetes Network model is defined with the concept of a flat network: a flat network is an abstraction of the underlying physical network in which it's possible to reach the connection of components without the need to map host ports to container ports. With a flat network, every entity deployed in any node of the cluster can be reached through its IP address, no matter how the underlying network is. With this approach, the network design is extremely simplified and allows easy management of the cluster. New workloads can be scheduled, independently and dynamically, anywhere in the cluster with no dependencies on the network design.
To set up the network, Kubernetes has specified a Container Network Interface (CNI) that can be implemented in various ways, which is in charge of set upping network interfaces when entities (pod, nodes, and services) are created for the communication with other entities along with the implementation of security policies(Network Policy).

With the use of this model, pods from different namespaces can be deployed on the same node, and in the first place, they can communicate. Also, when creating a cluster-mesh, the control plane is unified and the final mesh can be seen like one big cluster where every communication is allowed. This rise the problem of isolation and protection of services and pods, especially if the clusters are from different companies (some services or namespace could belong to teams that for example are not interested to communicate with other clusters).

As from the network, Kubernetes introduces a mechanism to abstract network security from topology and leave to CNI the low-level implementation based on different technologies. This abstraction is the Network Policy, which uses label selectors as their primary mechanism for defining which workloads can talk to which other, rather than IP addresses or IP address ranges. Thanks to this level of abstraction, the effort to find an optimal allocation for the firewalls [22] is left to the CNI that uses different approaches based on the technologies that are used.
Even if network policy abstract from the underlying network, it's important to know how those rules are applied because it can affect performance or growth of latency if the number of policies applied increases a lot. there are different levels where a network policy can be applied:

- **Cluster level:** these general policies are defined for every entity on the cluster: applying this policy in the cluster will affect all pods and services in every node. Those policies are applied usually due to specifications valid for every namespace(for example in every namespace some kind of pods have to communicate with another specified type of pod)

- **Node level:** it is possible to give labels to nodes and create Network Policies based on them: every pod deployed in that node will be affected by the policy. Usually, this level is used when a node contains entities that require the same sources for incoming and outgoing traffic.

- **Namespace:** this level will affect all entities belonging to this namespace independently from where they are deployed in the cluster. This level is usually used to allow communications with other namespaces: it's possible to specify then only the namespaces which pods can send or receive requests.

- **Single pod or service:** this is the lowest level and applies to every single entity matching that label and the specified namespace. This is the most commonly used level to specify the allowed communication of a single application.

Applying policies at any level requires to be consequently translated as firewall rules in the cluster's nodes. There are two approaches used based on where the firewalling rules are applied:

- a firewall is placed for every pod in the node and contains only firewalling rules for that pod

- **a** single firewall is placed in the node and contains all rules for all pods deployed in the node.

In the first approach, high-level policies are translated into firewalling rules in the network namespace of the pod. When a pod is created it's assigned to a different network namespace to guarantee (to its containers) isolation from other pods. An example is given in the figure 3.1.

This approach can guarantee a low number of policies applied and it is easy to maintain. When there is an update or an add of a new policy for that pod it's added directly to the network namespace. This guarantee also a fast lookup as it's easy to check if the traffic is allowed to enter or exit from that pod as the only rules present in the firewall are the ones applied to the pod. A drawback for this approach is the redundancy of firewall applied in the node: also if pods run the same instance of application their network namespace is independent, two instances of firewall need to be created, one for each pod.
In the second approach, the firewall is created in the root network namespace of the node and contains all the rules for all the pods present in that node. When pods want to communicate, every traffic that is not for its containers is sent in the root network namespace, in which is present a virtual switch that is connected to

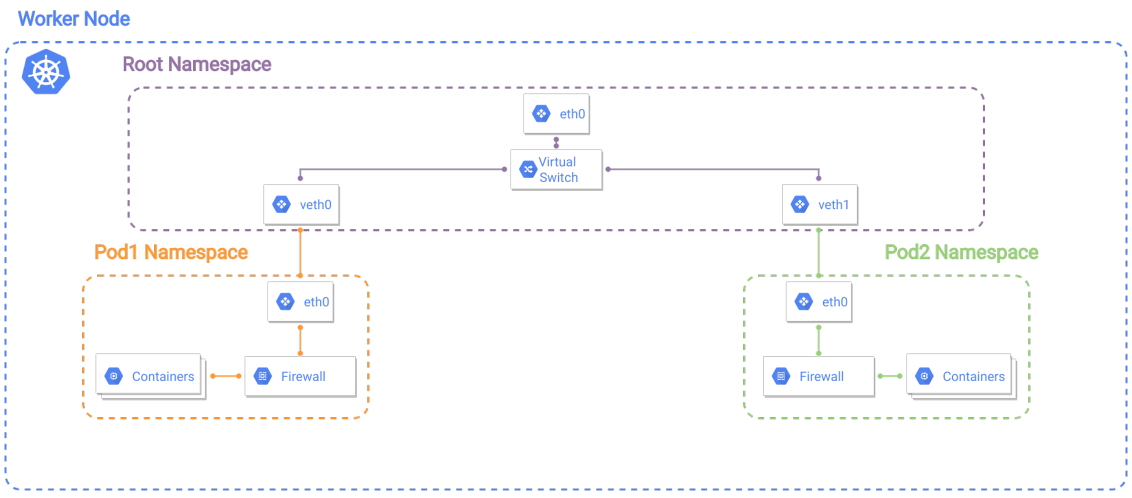


Figure 3.1. Per pod firewall image

all pod's network namespace created in the node and will forward all traffic to the right namespace. Before being forwarded by the virtual switch, as shown in figure 3.2, the traffic is matched with the rules of the firewall.

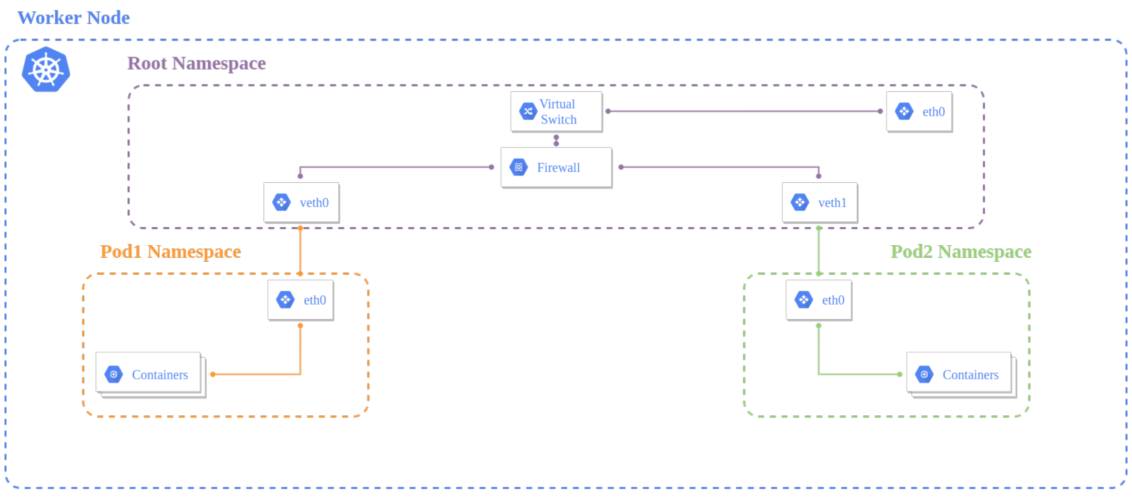


Figure 3.2. Per node firewall image

When a new pod is scheduled for that node there will be a controller in charge of inserting the missing rules for that pod in the firewall. With this approach there is no redundancy because now pods matching the same labels will be matched by a single rule in the firewall. A drawback for this approach is the maintenance of the rules in the firewall because pods can be created and destroyed very easily. If it's not done with a fast technology could introduce latency in the communication or errors in the matching rules(for example using iptables as will be shown in the following section).

Both approaches require the creation of one or more firewalls. The main implementations used to rely on the use of Linux kernel modules:

- iptables that is a user space program to configure a kernel module called Netfilter
- **e**BPF that is a virtual CPU running in the kernel space

In the next two sections, the two technologies will be analyzed and will be evidenced their strength and drawbacks to motive also choices made after in this work.

3.2.1 Iptables

Iptables is the userspace command-line program that allows the network administrator to set up The Linux kernel firewall (from version 2.4.0) by inserting and configuring IP packet filter rules.
Rules are organized in chains and are accessed by a kernel module called Netfilter. Packets entering and leaving the host are captured by the Netfilter module that will match those packets to the lists of iptables rules. Each rule is composed of packet information (source, destination, source port, destination port, and protocol) and the action to do which can be:

- DROP to drop the packet, commonly used in deny firewall type
- ALLOW to allow the packet and go out the system or reach the userspace, commonly used in allow type firewall
- FORWARD to forward the packet to another destination that is not local to the host
- pass the packet to another chain that will match the packet and chose the right action or pass it to another chain of rules

Chains of rules are stored in tables, which might be built-in chains or user-defined chains. By default, iptables has three tables: FILTER, NAT, and MANGLE. For the translation of network policy rules, only the FILTER table is used and will be the only table analyzed in this thesis.
In the FILTER table, there are three important chains where rules are placed once translated from the initial Network Policy:

- **INPUT:** in this chain there are all rules of incoming connections. When a request arrives in the network namespace, is matched with this chain to decide the action to perform with that packet. Rules specified in the Ingress field of Network policies are translated into rules of this chain.
- **OUTPUT:** this is the chain used for outgoing traffic. Before leaving the host, packets are passed to this chain and can be allowed, dropped, or forwarded to another chain. The Egress rules of Network policies are translated into rules of this chain.
- **FORWARD:** this chain is used for forwarding incoming packets that are not destined to the local network. This chain is used for implementing routing behavior on the host and it's not used in the process of Network Policies rules translation.

Using iptables has main advantages. Firstly, they are easy to configure and automate: applying few simple rules in the INPUT and OUTPUT chain provides fast and simple network isolation. Iptables can also be paired with a reactive framework [23] which can react to security threats and configure rules automatically without needing of human interaction.
Secondly, iptables can also be used for blocking or allowing any connection between pods of different namespaces or entire clusters building a full local firewall.
Finally, iptables rules are portable, so the same set of rules can be used to configure different pods or entire Kubernetes namespaces.
However, iptables also has its drawbacks. When the number of rules increases, could be rather complicated to manage or update chains when a pod is created or destroyed with the speed of Kubernetes, especially in the one firewall per node model and also, the network performance could be affected by the high number of rules present in the iptables.
The network isolation solution is implemented by modifying the iptables inside each pod network namespace or the host root namespace.

3.2.2 Ebpf

Extended Barklays Packet Filter(eBPF) is a recent technology introduced in the Linux kernel from version 3.15. It consists of a generic event-based virtual CPU running in the kernel space. With this virtual CPU is possible to write programs that can run sandboxed in the Linux kernel without recompiling it or loading other kernel modules.
Being in the kernel makes execution of these programs really fast because there is no overhead from system calls or context switching from user space to kernel space and vice versa. These programs are executed when a kernel event occurs, for example:

- a Network packet is received from the NIC and has to be processed
- a message at the socket-layer is received, for example, a request made by a program in the userspace

- data written to disk or there is a page fault in memory

For all these cases it's possible to pass the event or a copy to the eBPF program. The main difference is that passing a copy(i.e a copy of network packet) is used for monitoring or counting purposes while if the event could be modified (i.e modify a source destination of a packet) the real event must be passed to the eBPF program. eBPF is interesting especially for the possibility it offers to manage network packets running programs in kernel space that can emulate physical network or security devices. In this way, it's possible to have, for example, a virtual firewall with an optimized algorithm that can avoid the pitfalls of large iptables rulesets that weren't designed for the management of hounded thousands of rule changes in a short time interval.
It is also possible to create a network service function chain because packets can pass from an eBPF program to another and if required, can also transfer packets extremely fast in the user space with a zero-copy operation provided by a shared memory between user and kernel space, but also between eBPF programs.

CNI that decided to implement networking and security policies with eBPF, for example Cilium, can optimize the data path and the process of packets, writing ad hoc programs and making their solution fastest respect the ones that use Kernel modules, such as Netfilter, which were written for general-purpose operations.
In the case of Network Policies, it is possible to write an eBPF program that stores rules like iptables, but using more optimized algorithms for the lookup modification and insertion of firewall rules with high frequency and number.

3.3 Istio Network Policy

Istio's management and implementation of security policies are very different from other solutions in the market as they are application-layer based and, so, policies can be based on a virtual host, URL, or other HTTP headers. To use network policies at Layer 3-4, a network provider should be used, like Calico or Cilium.

The Istio's proxy that implements policies is based on Envoy, which is implemented as a user space daemon in the data plane that interacts with the network layer using standard sockets. This gives it a large amount of flexibility in processing and allows it to be distributed in a container. Network Policies data plane is typically implemented in kernel space using iptables, eBPF filters, or even custom kernel modules. Being in kernel space allows policies to be extremely fast, but not as flexible as the Envoy proxy. On the other side the number of service-to-service communications, based on the number of clusters linked together, can rise very high, so having a lot of rules and a lot of services can dramatically reduce performances.

3.4 Calico Network Policy

Unlike some other network policy implementations, Calico implements the full set of Kubernetes network policy features, but while Kubernetes network policy applies only to pods, Calico network policy provides a way to apply security to multiple types of endpoints like VMs, and host interfaces.[]

In addition to enforcing the Kubernetes network policy, Calico implements its own Network Policy resources that can be namespaced like the standard, but also it's possible to have non-namespaced resources, which provide a way to apply Network policies globally in the cluster. In addition, all namespaced and non-namespaced Network Policies provides other features like the support for policy ordering/priority, deny and log actions in rules, more flexible match criteria for applying policies and in policy rules, including matching on Kubernetes ServiceAccounts, and (if using Istio Envoy) cryptographic identity and layer 5-7 match criteria such as HTTP gRPC URLs and the ability to reference non-Kubernetes workloads in polices, including matching on NetworkSets in policy rules.

To give an example of a Calico Network Policy, the Kubernetes Network Policy in Code Box 3.1 was translated into a Calico Network Policy type in Code Box 3.2.

Listing 3.2. Example of a Calico Network Policy

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
  name: my-network-policy
  namespace: default
spec:
  selector: role == 'backend'
  types:
 - Ingress
 - Egress
  ingress:
 - action: Allow
    protocol:TCP
    source:
      nets: 172.20.0.0/16
        except: 172.20.1.0/24
    source:
      selector: role == 'frontend'
      namespace: frontend-namespace
    source:
      namespaceSelector: project == 'test'
    destination:
      ports:
      -6379
      -5480
  egress:
  - action: Allow
    protocol: TCP
    destination:
      nets: 10.0.4.0/24
      ports:
      —5978
      —6379
```

The resulting policy structure is very similar to the default Kubernetes Network Policy with the addition of a field action, which allows Calico Network Policies to work also in deny mode. The other difference is the compression of fields that can be expressed in one line with the use of operator "==".

Finally, Calico Network Policies are written in a YAML file and then translated as firewall rules in the Linux kernel using iptables. This makes the deployment of network policies easier but not optimized like a deployment using eBPF (see section 3.2.2) because iptables uses standard algorithms to match rules while with eBpf these algorithms can be customized to provide better performances. [24]

3.5 Cilium Network Policy

Like Calico, Cilium provides the full set of standard Network Policy features and enriches it with other ones. One of the features is to add labels to pods based on the cluster, giving the possibility to select only a subset of pods matching a label. Another feature is to add a non-namespaced Network Policy that applies to all nodes in the Cluster.
As done for Calico, in the Code Box 3.3 is present the translation of the standard Network Policy in Code Box 3.1. The structure is very similar to the standard with only changes to the fields name.

Listing 3.3. Example of a Cilium Network Policy

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: my—network—policy
  namespace: default
spec:
  endpointSelector:
    matchLabels:
      role: backend
policyTypes:
— Ingress
— Egress
ingress:
— from:
  - ipBlock:
          cidr: 172.20.0.0/16
        except:
       - 172.20.1.0/24
    - namespaceSelector:
          matchLabels:
            project: test
    - endpointSelector:
          matchLabels:
            role: frontend
            namespace: frontend—namespace
     ports:
    - protocol: TCP
          port: 6379 5480
  egress:
  — to:
    - ipBlock:
          cidr: 10.0.4.0/24
     ports:
    - protocol: TCP
          port: 5978 8800
```

Thanks to ClusterMesh introduced in Chapter 2 it's also possible to apply policies valid only for pods of a certain cluster. This is accomplished by enriching the policies with labels containing the cluster name in the field *endpointSelector* and in the specification of ingress and egress policies(in Cilium Network Policies these fields are *toEndpoints* and *fromEndpoints*).

Listing 3.4. Example of a multi-cluster Cilium Network Policy

```
  apiVersion: "cilium.io/v2"
  kind: CiliumNetworkPolicy
  metadata:
     name: "allow-cross-cluster"
  spec:
     endpointSelector:
        matchLabels:
          name: frontend
          io.cilium.k8s.policy.cluster: cluster2
  egress:
  — toEndpoints:
    - matchLabels:
          role:backend
          io.cilium.k8s.policy.cluster: cluster1
```

The above example policy will allow *frontend* pods deployed in *cluster2* to talk to pods matching the selector *role:backend* in *cluster1*. Pods *frontend* won't be able to talk with other pods backend deployed in the local cluster or other clusters unless additional policies exist that allow the communication.

Cilium uses eBPF for the implementation of its network policies. With the use of Linux eBPF, it's possible to have tables containing security rules based on the identity of an entity (i.e service, pod, container) rather than traditional IP address identification.
Kubernetes networking model is based on assigning an individual IP address for each pod and make them reachable without any network address translation(NAT). This simple architecture has to manage a large number of IPs with the increase of cluster's size or the number of pods, increasing also the number of security rules to add or update based on that IP address. However, network events on the application layer can be filtered by eBPF giving the ability to decouple security from addressing and provide faster and stronger security isolation.
Cilium uses this feature from eBPF to give pods an identity and apply security based on that rather than their IP address. This identity is derived through pods labels and can be shared between pods. When starting the first pod with a label, Cilium assigns an identity to that pod and saves it in its key-value store. The pod is then allowed to start communications with other entities. When another pod with the same label is started, there is first a lookup in the key-value store, which resolves the identity giving to the new pod the same label of the first started. This requires then no other actions to perform on other nodes of the cluster like updating security rules. Rules are the same for every pod with that identity and have to been configured only once, reducing a lot the number of rules applied and consequently speeding up the process of update and lookup.

3.6 Comparison between tecnologies

After a description of the architecture, how the security and isolation are reached, and how service-to-service and cluster-mesh are implemented, in this section there will be a summary of the three technologies evidencing their strengths and weaknesses. Istio is a technology for providing service-to-service communication and a lot of features like monitoring circuit breaking and traffic management. It can also link clusters by sharing the same control plane or let a single control plane configure others(see Chapter 2 section 2.2.2) and provide application-level security. Even if is very advanced in providing communication between services it presents some limitations. The main weakness of this project is the absence of layer 3-4 security policies. Having security at application level is very useful to create fine-granted security policies, which can be based on HTTP requests, but this can lead to network performance and introduce latency if all traffic has to reach the user space, match the rule and return to kernel space. It is possible to pair Istio with the other two technologies, Calico and Cilium, in order to provide also layer 3-4 security, but the two software have to cooperate and presents already their mechanism to provide service-to-service communication.

Calico instead is a complete solution stand-alone, providing multi-cluster connection, service discovery and management, and full implementation of network policies. Thanks to its network layer with BGP it can transport a lot of different information to BGP nodes thanks to the extensiveness of the protocol. For the security instead, it provides also deny rules and the possibility to set priority to namespaced and global network policies allowing complete isolation in case of multi-tenancy in one cluster. In fact giving higher priority to global network policies respect namespaced can deny the overwriting of rules that deny access to other namespaces. One limitation is that it needs a management tool that is not free to connect clusters and reach service-to-service communication. The other problem is the network policy data path implementation with Iptables that can rise a performance problem if there are a lot of policy rules changing in a short time interval.

Finally, Cilium is complete like Calico but presents some advantages. Thanks to eBPF, Cilium can provide a faster data plane and security, using its own optimized programs running in the Linux kernel that can manage the high frequency with which Kubernetes changes the rules to apply to pods. Another advantage is that Cilium provides an open-source cluster-mesh with service-to-service discovery, making it possible to develop other software based on it or to extend it.
The main problem of this solution is that the cluster-mesh and network policy needs to be configured manually for every cluster.

All the projects analyzed present their weaknesses and strengths, but all are missing automated tools which can set security, cluster-mesh, and service-to-service communications automatically. From the time of writing this thesis, all operations described in the last two Chapters have to be done manually, or only a few operations are automated like the policy replication in all clusters provided by Calico Enterprise.

The goal of this thesis then is to propose a Multi Cluster Orchestrator, based on the Cilium project (which is the faster and complete solution for network policies, service-to-service and cluster-mesh communication analyzed in this work), that can automatically configure network policies, the service, and cluster mesh, for every clusters given some high-level inputs.
The next Chapter will present the model of this orchestrator and what are its goals and requirements.

Chapter 4

Multi Cluster

4.1 Introduction to Multi Cluster Orchestrathor

As described in the previous chapters, the lack of technologies that allow automatic configuration of connections and application of security policies between services belonging to clusters of different companies, has given rise to the idea of creating an entity that can manage, in a higher way and globally, from high-level requests, everything needed to put one service in communication with another, starting from the connection between clusters up to the application of security policies which protect the service from all unwanted requests for incoming and outgoing service connections.

The Multi Cluster Orchestrathor it's important because speeds up operations that have been manually done until now, and in meantime prevents human errors that can occur during the configurations. Furthermore, it allows communications to service in clusters where some knowledge is missing(for example the pod selector of the service in an external domain could be unknown to the domain manager, or in which cluster and namespace the service is located)

In this regard, the model of this Orchestrator will be introduced in the following section, specifying its interactions within the network, the entities involved in its operations, and the various steps it will have to perform to pass from high-level requests to a low-level configuration of the various clusters involved.

4.2 Model

The Multi Cluster Orchestrator is a software placed on a higher level of abstraction than a Kubernetes cluster.
Before explaining the model of the Orchestrator it's worth introducing the concept of domain. The domain is a new entity created for the model which groups clusters belonging to the same company to allow the same management towards communications to clusters from other domains. More details will be addressed in subparagraph 4.2.1.

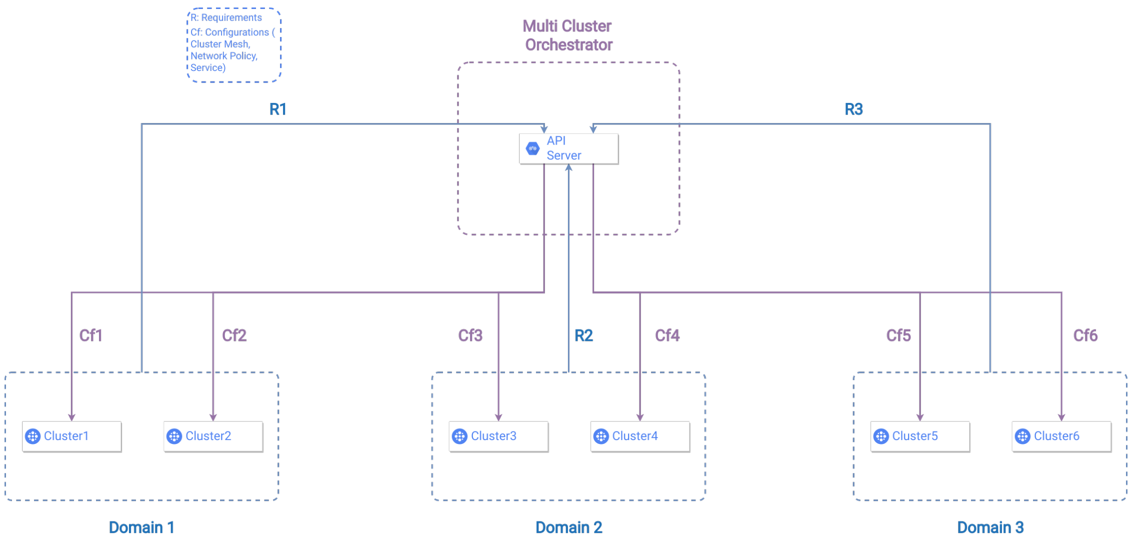


Figure 4.1. External Model Of Multi Cluster Orchestrator

As shown in Figure 4.1, the Multi Cluster Orchestrator receives requests from all domains linked to it, elaborates them, and creates a single configuration for all the clusters of all domains.

Going deeply into the Orchestrator structure, in figure 4.2 it's shown how a request is transformed to a single configuration: first of all, requests are collected and transformed in a global configuration, then the global configuration is refined and a single configuration for a cluster is created and applied.

In the next sections every step will be explained in more detail, focusing on the transformation of requests and information about every phase.

4.2.1 Domain

The concept of domain is introduced in this thesis to allow grouping clusters of a single company. First of all, it's useful to have a definition of what is a domain in this model context and why it is important.

Domain. group of clusters belonging to the same company and under the control of a single manager, called domain manager.

Grouping clusters it's important for the following reasons:

- the management is entrusted to a single person who knows all the details of all the clusters
- it is possible to apply different security policies for infra-domain and inter-domain service communications.

Regarding the first point, the Multi Cluster Orchestrator needs to know some information about clusters and services (that will be fully explained in the subparagraph

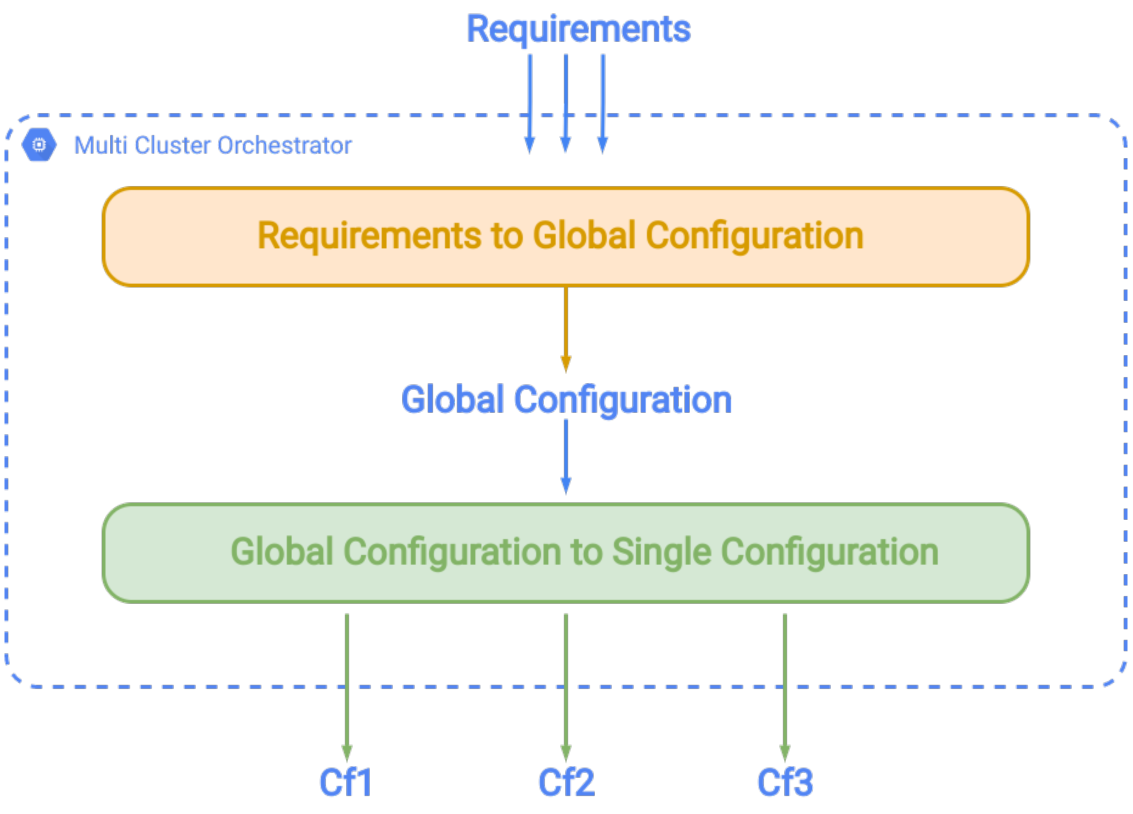

Requirements
Multi Cluster Orchestrator
Requirements to Global Configuration
Global Configuration
Global Configuration to Single Configuration
Cf1
Cf2
Cf3

Figure 4.2. Internal Model Of Multi Cluster Orchestrathor

4.2.2) to reach its goal, and having a domain can simplify the work of the domain manager, because he can give all information about the clusters he manages in one time, reducing the setup time (i.e writing a single requirements file for all cluster of his domain) and the possibility to make human errors writing more requirements files.

Moving to infra-domain communications, defining different security policies other than those applied for external domains can be useful to manage the clusters belonging to different sections of the company(for example the testing department may want to communicate with the cluster of the development department, but not with the human resources section). In this way the Multi Cluster Orchestrator, therefore, offers the possibility of managing clusters belonging to the same domain in a completely automatic and autonomous way concerning the different domains that are connected to the same.

Finally, for inter-domain communications, grouping into domains can be useful to give to the various domains a possibility to communicate with each other without, for example, knowing a service in which specific cluster it is, but leaving to Multi Cluster Orchestrator the task of going to search for the specific service of that domain and going to configure all clusters, so that they can communicate with the service. It is also possible to adjust the application granularity of these security functions by choosing whether to apply them to all clusters of the domain or a single one.

4.2.2 Requirements

The requirements provided by the domain manager to the Multi Cluster Orchestrator are a combination of domain structure and requests of policy application and service communications. In particular, to operate the Multi Cluster Orchestrator requires from domains:

- Knowledge of all clusters that belong to a domain
- Knowledge of all services that a domain wants to expose
- Security policies for the own domain services that can be applied to the incoming and outgoing traffic to and from intra-domain and inter-domain services or particulars IP addresses

The first point of the list is important because the Multi Cluster Orchestrator needs to know all the clusters where it needs to apply the final configuration(more details will be given in the section 4.2.4). The parameters needed for this first requirement are the name of the cluster, which may change internally to the Multi Cluster Orchestrator(see Chapter 5), and the API address, which is where Kubernetes has its REST service API and can be used to create services and apply network policies.

Moving on to the second point, the knowledge of all services that the domain wants to expose and in which cluster they are located is an essential requirement to subsequently allow the other domains to make requests for connections to those specific services. Some pieces of information may be unknown by domain managers of other domains like the namespace where the service belongs, port and protocol used, the cluster where the pods linked to the service are deployed, or the selector that links pods to that particular service. By giving this information to the Multi Cluster Orchestrator, the automation and optimization processes are improved, providing transparently all that is needed to start a secure communication between services.

Finally, the last requirement is a request for security policies that must be applied to services of a domain that will protect from undesired traffic. In this request for each service, the domain manager specifies what are the communications allowed:

- **Services belonging to the same domain:** it's possible to select one or more services that belong to the same domain of the service, but maybe are deployed in a different cluster.
- **Services belonging to a different domain:** it's possible to select services of different domains. In this case the Multi Cluster Orchestrator will add the missing information about that service in a transparent way(see 5 section 5.3.1).

- **External IP addresses not belonging to a domain:** it's possible to specify a range or a single IP address not belonging to any domain registered to the Multi Cluster Orchestrator.

The communications can be allowed for the incoming and outgoing traffic independently. For example a *service1* can start a communication with *service2*, but the *service2* cannot start a communication with the *service1*.
If the policy is applied for outgoing traffic, the Multi Cluster Orchestrator will also create a link between services if they are not able to communicate yet.

4.2.3 Global Configuration

After receiving all requirements from a domain, the Multi Cluster Orchestrator creates or updates, if already exists, a Global Configuration that keeps tracking of the following aspects:

- Which services must be in communication
- Which clusters must be linked together

In the first point, the Multi Cluster Orchestrator maintains a list of all services that communicate together. This is useful because can help in the process of optimization for the mesh of clusters, but also in the optimization process of a single cluster: for example if two services are in the same cluster and needs to talk with a common service outside the cluster, it's possible to apply one single configuration that works for both if some conditions are meet(i.e the two services are in the same namespace). This can be useful when a new communication between two services is started: checking the service linking list it could be possible to skip the operation of creating a path between the two services if it was created before for another one(Communications are connection-oriented, in this case, the two services in the same cluster and namespace started a connection to the same service in another cluster).

Before achieving the goal of communication between services, it is important that clusters in which the two services are deployed can reach each other. For the cluster mesh optimization, the Multi Cluster Orchestrator maintains a list of clusters that are in communication with each other, and they are only the ones that have services talking together. This can give more flexibility to the domain manager, that can choose only some clusters to link other domains with the cluster-mesh and use a different type of technology for the infra-domain cluster communication. Once the cluster-mesh is created, every time a new domain is added or a new link from a service in a cluster not belonging to the cluster-mesh to a service in another domain is created, the mesh is updated by the Multi Cluster Orchestrator which will also derive the new single configuration for the new clusters added.

4.2.4 Single Configuration

Once a Global Configuration is created or updated, the Multi Cluster Orchestrator creates a Single Configuration for every cluster and applies it.
This configuration includes:

- Parameters for the cluster-mesh
- Network Policy for services or pods of the cluster
- New services that refer to services in other clusters to allow service-to-service communication.

For the connection with other clusters, there are some parameters to be set in the cluster and some operations to be done in order to secure connections between all clusters. These parameters and operations are different according to the technology chosen for the implementation. In this thesis, the mesh creation is done using the Cilium cluster-mesh(see Chapter 2) that requires parameters like an id and a cluster name that could be different from the real name given by the domain manager in the requirements, but which will still be transparently mapped by the Multi Cluster Orchestrator. More details will be given in Chapter 5.

For the Network Policy instead, the Multi Cluster Orchestrator will start a process of refinement from the high-level security policy requests, that will create different network policies for all services or pods that have to be protected, allowing communications only to the services specified by the domain manager for both incoming and outgoing traffic.
The Single Configuration will contain, at the end of the refinement process, all the policies for that specific cluster, that will be applied using the Kubernetes Cluster's API.

Finally, to allow services and pods to resolve a service name without the modification of the applications, transparently, in each cluster the Multi Cluster Orchestrator will create new services that will refer to the original service deployed in a different cluster. In this way, applications can resolve the name of the service they want to connect as it belongs to the same namespace in a completely transparent way. The Single Configuration will contain all these services and they will be created using the Kubernetes Cluster's API.
More details about the service creation will be given in the Chapter 5 section 5.5

Chapter 5

Implementation

In this chapter will be described the implementation of security policies and service-to-service communication. The cluster-mesh and Global Configuration creation, described in Chapter 4, will not be included in this thesis but can be implemented as future work extending this one.
From now on there will be the hypothesis that clusters containing services that need to communicate, are in a cluster-mesh created with the Cilium (Chapter 2)and the Global Configuration has been already created, even if for this part a Global Configuration is not needed. However, the process of creation and configuration of the cluster-mesh is described in the Appendix B of this work and can be used to set up the environment where the Multi Cluster Orchestrator can operate properly.

To make it easy to interact with, it was decided to implement the Multi Cluster Orchestrator as a RESTful web service. This can help the domain manager to give all the information needed by the Multi Cluster Orchestrator and the possibility to request security policies and service connections for all the domains or a single cluster. The full APIs provided are described in the Appendix A.

The first part of the chapter will introduce the entities created to represent the model, how they are linked together, and the requirement's format which includes security policies and service connection requests.

The second part will describe the policy refinement process starting from requirements and how they are applied to a single cluster using the Kubernetes Java Client, which has been modified with the addition of methods that interacts with the Cilium APIs.

The last part will describe the service-to-service communication, specifying which operations are executed, what services and namespaces are created when it's needed.

5.1 Entities

To be able to perform all the operations that are required by domain managers, the Multi Cluster Orchestrator maintains some information in a database, saving entities that help in the building of Single Configuration for the clusters.
Entities are linked together with one-to-many and many-to-many associations, in such a way that is possible to find all the necessary information starting from an entity and navigating through others.
The Multi Cluster Orchestrator creates and uses the following entities:

- **Domain:** this is the first entity created when a domain manager registers the domain, he is in charge of, towards the Multi Cluster Orchestarthor.
  It is characterized by a unique name across the Multi Cluster Orchestrator, a one-to-many association with clusters belonging to it(cluster can have only one domain and a domain can have multiple clusters), a one-to-many association with services, which are the ones exposed by the domain manager (service can belong only to one domain and a domain can have multiple services) and a one-to-many association with Network Policies applied in the domain(a policy is applied to clusters belonging in one domain and a domain can have multiple Network Policies)

- **Cluster:** this entity represents a Kubernetes cluster.
  It contains an unique id given by the Multi Cluster Orchestrator, a name which is the real name of the cluster and it is unique across the domain, an unique name given by the Multi Cluster Orchestrator that is the name used to identify the cluster in the cluster-mesh(see Appendix B for more details), the name of the Kubernetes context for the interaction with the Kubernetes API(see section 5.4), a reference to the domain(many-to-one association), a many-to-many association with Network Policies that have been applied to the cluster(Network Policies can be applied in multiple clusters and clusters can have multiple Network Policies), a many-to-many association with services deployed in the cluster(services can be deployed in multiple clusters and clusters can have multiple services) and a one-to-many association with the namespaces created for the service-to-service communication(a namespace belongs only to a cluster while a cluster can have multiple namespaces, see 5.5 for more information about the creation of namespaces)

- **Service:** this entity represents a service exposed by a domain.
  It's characterized by a name that must be unique across all domains, a namespace name that is the namespace where pods linked to the service are deployed, a reference to the domain the service belongs(many-to-one association), the selector used to link the service to pods and that will help in the policy refinement process(see section 5.3), a parameter that identifies the port and protocol of the service, a many-to-many association with clusters which the service is deployed and a many-to-many association with Network Policies that contains the service in the Egress spec field(a service can be referenced by multiple Network Policies and Network Policies can reference multiple services) which helps in the creation of service-to-service communication path(see 5.5 for more details)

- **Policy:** this entity represents a Cilium Network Policy applied to a service or a group of pods with the same selector. The Network Policy can be applied to a single cluster or all clusters in the domain.
  It contains a unique id given by the Multi Cluster Orchestrator, a name that is unique in the domain where the Network Policy is applied, the namespace where the Network Policy is applied, a reference to the domain(many-to-one association), a many-to-many association with the services contained in the egress rules for the configuration of the service-to-service communication path, a many-to-many association with the namespaces (a policy can reference multiple namespaces and a namespace can be referenced by multiple policies) created after the application of this Network Policy(see section 5.5) and a many-to-many association with clusters where the Network Policy is applied

- **Namespace:** this entity represents the namespaces created in a cluster to allow service-to-service communication after the application of an Egress rule to a service in another cluster through Network Policy(see 5.5).
  It's characterized by a unique id given by the Multi Cluster Orchestrator, a name that is unique across the cluster where the namespace is created, a reference to the cluster where the namespace is created(many-to-one association), and a many-to-many association with policies that contains in the egress rule spec a service deployed in a different cluster with that namespace name.

5.2 Requirements

In this section will be presented the format of a request for the application of security policy and service-to-service communication if needed.
The format is very similar to a Network Policy written in YAML with the addition of more semantics.

Code Box 5.1. Example of a requirement

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-cross-cluster"
spec:
# this select all pods in the default namespace with labels
   name=x-wing
  endpointSelector:
    matchLabels:
      name: x-wing
#egress rules: this will allow pods x-wing to connect and
   communicate only with selected services and to external Ip in
   the subnet 108.177.16.0/24
  egress:
  - toEndpoints:
    - matchLabels:
#labels : this will select all services from all clusters in the
   domain1
          service: "*"
          cluster :"*"
          domain: domain1
  - toCIDRSet:
      - cidr: 108.177.16.0/24
#ingress rules: this will allow all connection requests from
   services in cluster1 of domain2 to start a connection with
   pods x-wing
  ingress:
  - frmEndpoints:
    - matchLabels:
#labels : this will select all services from cluster1 in the
   domain2
          service: "*"
          cluster :"cluster1"
          domain: domain2
```

In the Code Box 5.1 there is an example of a requirement and the most important fields are:

- **endpointselector:** in these fields the domain manager specifies what are the pods or nodes that need to be secured. In the example, based if was decided to apply the requirement for a single cluster or all clusters in the domain, it was required to apply a Network Policy and, eventually, the creation of service communication paths for all pods in the default namespace that match the label *"name=x-wing"*. Pods can be linked to a service or can be normal pods running in a node. If the policy and service-to-service communication need to be applied for all namespaces in a cluster it's possible to use *nodeselector* instead of the endpointselector.

- **egress:** in this field will be added all services or IPs that can be reached by pods specified in the endpoint or node selector. In the example pods with label *x-wing* will be able to start a connection with all services in the *domain1* and to all external IP in the *145.0.0.0/32* subnet.

- **ingress:** in this field will be added all services or IPs that can start a connection with pods specified in the endpoint or node selector. In the example pods with label *x-wing* will be able to receive connection from all services in the *cluster1* of *domain1*.

To make it easier for the domain manager to specify ingress and egress rules from and to services outside the domain, without using labels he might not know(i.e. labels of pods linked with a service), some other labels were added. These more abstract labels help to group services also and will be substituted in the process of refinement. The labels added are:

- Service that selects a single service registered to the Multi Cluster Orchestrator with the only specification of the service name
- Cluster that selects all services belonging to the cluster specified
- Domain that selects all services of all clusters in a specified domain

In the next sections will be given some examples for understanding the use of new labels and the combination between them.

5.2.1 Services

In the first example in Code Box 5.2 it's shown how is it possible to specify requirements for a single service, in this case services *rebel-base* and *dns*.
Every service needs to be specified in a different *toEndpoints* field if it's an egress rule (*fromEndpoints* if it is an ingress rule) with its name that is unique across the Multi Cluster Orchestrator.
If the rule is of egress type, like in the example, specifying the service it is equivalent not only to the application of a Network Policy but is also a request of connection with that service. This means that should be possible for the pods with label *x-wing* to request a DNS resolution with the service name(i.e rebel-base), receive an address of a pod serving that service, and be able to establish a connection with it.

Code Box 5.2. Request to communication with a single service

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-rebel-base"
spec:
  endpointSelector:
    matchLabels:
      name: x-wing
    egress:
   - toEndpoints:
     - matchLabels:
          service: "rebel-base"
   - toEndpoints:
     - matchLabels:
          service: "dns"
```

It will be up to the Multi Cluster Orchestrator to do all that is needed transparently if it's not possible to do one of the operations specified before.

5.2.2 Clusters

Selecting single services for requirements can be a long operation to do for domain managers if the specified group of pods has a lot of connections in ingress and egress fields. If the request has to be done like the example in Code Box 5.2, not only requires all time to specify all services but can also introduce errors if some service is missing.
The Multi Cluster Orchestrator can group services based on the cluster where they are deployed allowing to specify them in a single field of requirement.
The second example in Code Box 5.3 shows how is possible to select all services in a cluster. Putting into the service field a value of "*", asks the Multi Cluster Orchestrator to look for a new label which is the cluster label, and to start the refinement for that value, which is *cluster1* in the example. As for a single service, if more clusters needs to be specified, new *toEndpoints* (or *fromEndpoints*) needs to be specified. In addition to cluster labels, to uniquely identify a cluster, also a domain label is required because clusters in different domains can have the same name, so the Multi Cluster Orchestrator needs to know exactly what cluster the requirements refer to without any ambiguity.
Finally, if a service is deployed in two different clusters but only one is selected by the requirement, only pods in that cluster will be allowed for ingress or egress connections.

Code Box 5.3. Request to communication with services of a cluster

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-cluster1"
spec:
  endpointSelector:
    matchLabels:
      name: x-wing
  egress:
  - toEndpoints:
    - matchLabels:
        service: "*"
        cluster: cluster1
        domain: domain1
```

5.2.3 Domains

Another level of aggregation provided by the Multi Cluster Orchestrator is to group all services belonging to a domain.
The example for this kind of aggregation is in Code Box 5.4, in which it's required to establish connections and apply security policies from pods with the label "x-wing" and all services belonging to all clusters in *domain1*. To select this behavior both the service and cluster field needs to have the value of "*". In this case, even if

services are deployed in more than one cluster, all connections and security policies will be applied to all pods running that service.

Code Box 5.4. Request to communication with services of a domain

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-domain1"
spec:
  endpointSelector:
    matchLabels:
      name: x-wing
  egress:
  - toEndpoints:
    - matchLabels:
        service: "*"
        cluster :"*"
        domain: domain1
```

This is the last level of aggregation, which provides a full connection with a domain. Will be now introduced, in the next section, the refinement process of these requirements, which will create Network Policies and connect the services in the request.

5.3 Policy Refinement

- a Network packet is received from the NIC and has to be processed
- a message at the socket-layer is received, for example, a request made by a program in the userspace
- data written to disk or there is a page fault in memory

In this section will be explained the refinement process of the labels introduced in section 5.2 that will create the Network Policy to be applied in the cluster.
Once a requirement is received by the Multi Cluster Orchestrator, a Cilium Network Policy will be created including all services specified with the labels *service*, *cluster* and *domain*. The decision of creating a Cilium Network Policy and not a standard Kubernetes Network Policy is because the first one permits to separate pods linked to a service across clusters with the cluster-mesh name, which simplifies the process of refinement.
Will follow an example to help to understand the refinement process.

In figure 5.1 is presented the following scenario composed by:

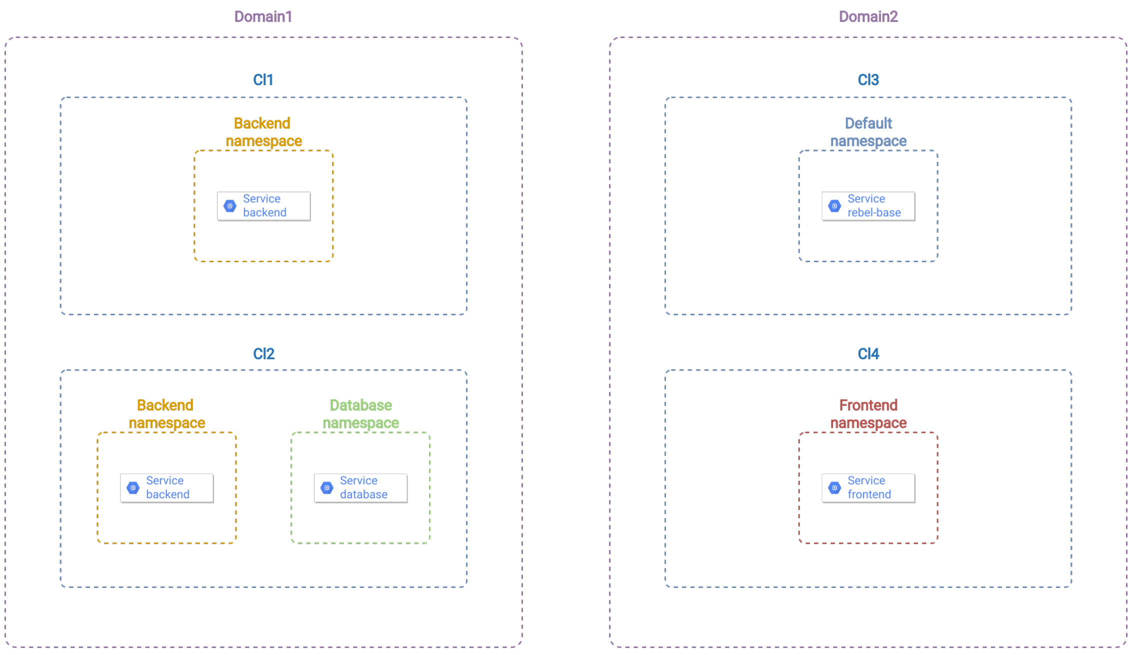


Figure 5.1. Example of a scenario with two domains and four clusters

1. a *domain1* and *domain2* are registered to the Multi Cluster Orchestrator

2. *domani1* is composed by two clusters with name *cl1* and *cl2* while *domain2* is composed by clusters *cl3* and *cl4*

3. in the *domain1* there are two services:

   (a) *backend* that is in the *backend-namespace* and deployed in clusters *cl1* and *cl2*

   (b) *database* that is in *database-namespace* and deployed in *cl2*

4. in the *domain2* there are two services:

   (a) *rebel-base* that is in *default* namespace and deployed in *cl3*

   (b) *frontend* that is in the *frontend-namespace* and deployed in clusters *cl4*

As shown in the Code Box 5.5, the domain manager of *domain2* sends this requirement to the Multi Cluster Orchestrator for the pods linked to the service *frontend* that has to be applied in *cl4*.

Code Box 5.5. Requirement example

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-cluster1"
spec:
  endpointSelector:
    matchLabels:
      app: frontend
  egress:
 - toEndpoints:
   - matchLabels:
        service: "*"
        cluster: "*"
        domain: domain1
  - toCIDRSet:
      - cidr: 108.177.16.0/24
   ingress:
  - fromEndpoints:
    - matchLabels:
         service: "*"
         cluster :"*"
         domain: domain1
   - fromEndpoints:
     - matchLabels:
          service: "*"
          cluster :"cl3"
          domain: domain2
```

With this requirement the domain manager requires:

- **ingress:** the service *frontend* will accept only connection requests from all services of *domain1* (*backend*, *database*) and all services in the *cl3* of *domain2*(*rebel-base*), all other connection requests to the service will not be allowed.
- **egress:** the service frontend is allowed to send connection requests to all services in the *domain1* (*backend*, *database*), and to an external IP subnet (Google IPs).all other connection requests to other services, pods or external IPs will be dropped.

To apply the right Cilium Network Policy needs to substitute the service, cluster and domain labels with the information of all services.
For external IPs, it's not required any modification as it is already in the exact format of a Cilium Network Policy.
The next two sections will describe the information and operations that the Multi Cluster Orchestrator has to do for the ingress and egress requirement request.

5.3.1 Ingress Refinement

In the Code Box 5.6 is reported the section regarding ingress requirements of Example in Code Box 5.5.

Code Box 5.6. Ingress overwiew of Requirement Example 5.5

```
ingress:
— fromEndpoints:
  - matchLabels:
        service: "*"
        cluster :"*"
        domain: domain1
— fromEndpoints:
  - matchLabels:
        service: "*"
        cluster :"cl3"
        domain: domain2
```

For the ingress refinement process, the Multi Cluster Orchestrator will look for every service that has to be inserted in the policy and will add a new *fromEndpoints* field for every new service added.

In particular, the Multi Cluster Orchestrator will add for every service:

- **selector:** this is the selector that links services to pods (at a low level, services are run by pods with the selector specified in the service creation). This parameter is given by the domain manager that registers the service, so could be unknown by another domain manager.
- **namespace:** this parameter is the namespace of pods running the service. In the end, requesting a service means connecting to pods that run the service, so in the policy must be specified the namespace(if no namespace is specified it will be interpreted as the pods to which the Network Policy is applied and the pods in the ingress field are in the same namespace). The label for this parameter has the key "*k8s:io.kubernetes.pod.namespace*"
- **cluster:** this parameter specifies in which cluster are the pods running the service. If this parameter is not specified it's interpreted as the pods runs in the same cluster of the pods specified in the *endpointSelector*(this is not supported in the standard Kubernetes Network Policy and is the reason why the Cilium Network Policy was chosen). The label for this parameter has the key "*io.cilium.k8s.policy.cluster*"

After applying the refinement process the ingress field will look like the Code Box 5.7:

Code Box 5.7. Final Ingress refinement Network Policy

```
ingress:
— fromEndpoints:
  - matchLabels:
        app: backend
        k8s:io.kubernetes.pod.namespace: backend—namespace
        io.cilium.k8s.policy.cluster: cluster1
— fromEndpoints:
  - matchLabels:
        app: backend
        k8s:io.kubernetes.pod.namespace: backend—namespace
        io.cilium.k8s.policy.cluster: cluster2
— fromEndpoints:
  - matchLabels:
        name: database
        k8s:io.kubernetes.pod.namespace: database—namespace
        io.cilium.k8s.policy.cluster: cluster2
— fromEndpoints:
  - matchLabels:
        name: rebel—base
        io.cilium.k8s.policy.cluster: cluster3
```

From Code Box 5.7 it's possible to notice that the names of clusters are different from the ones given by the initial requirement. The reason is that the ones in the final policy are the name given to the cluster when the cluster-mesh was created.

The process of creation of the cluster-mesh is automatic and transparent to the domain managers and requires the uniqueness of cluster names that might not be satisfied with the real cluster names.

In this work it was supposed that the cluster mesh is already created but more information about how to set up a cluster-mesh are given in the Appendix B.

5.3.2 Egress Refinement

As for the ingress, also egress section is reported in Code Box 5.8

Code Box 5.8. Egress overwiew of Requirement Example 5.5

```
egress:
— toEndpoints:
  - matchLabels:
        service: "*"
        cluster: "*"
        domain: domain1
— toCIDRSet:
     - cidr: 108.177.16.0/24
```

The egress refinement is similar to the ingress but it has some operations more to do, because this requirement is not only for the right application of Network

Policy, but is also a requirement to connect the services if it's not possible yet. For this reason, the egress refinement will do, in addition to steps described in section 5.3.1, the following operations:

- apply a policy to communicate with DNS
- save the services contained in the egress field
- connect the service selected by the requirement with all services in the egress field

The first point is important because pods can communicate with everyone until at least one Network Policy is applied. After that, only communications present in Network Policies referring that pods are allowed so if in any isn't specified the communication with the DNS of Kubernetes, pods cannot resolve service names. The Multi Cluster Orchestrator will transparently add this communication path so the DNS resolution requests will always be permitted.

For the second point, the Multi Cluster Orchestrator, during the process of refinement, will save a list of egress services present in the Network Policy in order to permit the service-to-service communication process.

In the last point, based on the list in the previous point, the Multi Cluster Orchestrator will check, for every service in the list, if a connection with the pods selected by the requirement is established and, if it's not, will provide to create a pod-to-service path. These operations will be explained in the section 5.5.
In the Code Box 5.9 is present the final egress policy refinement of the example.

Code Box 5.9. Final Egress refinement Network Policy

```
egress:
— toEndpoints:
  - matchLabels:
        app: backend
        k8s:io.kubernetes.pod.namespace: backend—namespace
        io.cilium.k8s.policy.cluster: cluster1
— toEndpoints:
  - matchLabels:
        app: backend
        k8s:io.kubernetes.pod.namespace: backend—namespace
        io.cilium.k8s.policy.cluster: cluster2
— toEndpoints:
  - matchLabels:
        name: database
        k8s:io.kubernetes.pod.namespace: database—namespace
        io.cilium.k8s.policy.cluster: cluster2
— toEndpoints:
  - matchLabels:
        k8s:io.kubernetes.pod.namespace: kube—system
        k8s—app: kube—dns
```

```
    toPorts:
   - ports:
     - port: '53'
         protocol: UDP

  — toCIDRSet:
     - cidr: 108.177.16.0/24
```

5.4 Communication with Kubernetes

Once the refinement process has ended and all the Cilium Network Policies have been created, the Multi Cluster Orchestrator has to apply this policy in the required clusters.
For this reason, the Multi Cluster Orchestrator contains a client, that is used to connect with a single Kubernetes API in a cluster. This client is written in java and requires as input the cluster context to contact its API. To keep track of the context of a cluster, the Multi Cluster Orchestrator maintains a file, which is shown in the Code Box 5.10:

Code Box 5.10. Example of a Context file

```
apiVersion: v1
clusters:
— cluster:
    certificate—authority—data:
    name: kind—cluster1
— cluster:
    certificate—authority—data:
    server: https://127.0.0.1:39027
    name: kind—cluster2
  contexts:
  - context:
        cluster: kind—cluster1
        user: kind—cluster1
        name: kind—cluster1
  - context:
       cluster: kind—cluster2
       user: kind—cluster2
       name: kind—cluster2
current—context: kind—cluster1
kind: Config
preferences: {}
users:
— name: kind—cluster1
    user:
        client-certificate-data:
        client-key-data:
  - name: kind-cluster2
    user:
      client-certificate-data:
      client-key-data:
```

This file is created putting together all the contexts given by the domain managers and contains, in addition to the context name, also the IP address of the cluster API, the client-certificate-data, and client-certificate-key, which allows to start a TLS session with the Kubernetes server API of clusters. When the Multi Cluster Orchestrator needs to contact a specific server API, it will pass the context name retrieved from the cluster entity and the client will transparently use the information in the file to connect with the right API server.
The next sections will describe more extensively the Kubernetes Java Client and what was added to this to permit the interaction with the Cilium API to create the Cilium Network Policy in the real cluster.

5.4.1 Kubernetes Java Client

The Kubernetes Java Client is a library developed by Kubernetes in order to allow communication with Kubernetes API.
The library contains useful methods for the Multi Cluster Orchestrator like the creation, deletion, modification, and management of a service or a namespace, and also methods to list namespaces and services in a cluster. The library is missing methods for the Network policies and these were added for the scope of this thesis. The next section will describe what methods were added in the library and also the creation of the CiliumNetworkPolicy class which is required for the interaction with Cilium API.

5.4.2 Cilium Network Policy API

When Cilium is installed in a Kubernetes cluster, in addition to installing and start all pods and services to manage the network and network policies, it provides to the Kubernetes API its APIs, which permit the creation of the Cilium Custom Resources Definition (CRDs). One of these resources is the CiliumNetworkPolicy. The Kubernetes Java Client project provides also a generate Java CRD Model tool, which was used to generate the java class of CiliumNetworkPolicy in order to use it as a parameter for the Cilium API.
The Kubernetes Java Client library was then extended with the addition of methods that permits the creation of Network Policies in a synchronous and asynchronous way.

The added methods are:

- **Creation of Cilium Network Policy:** this method requires a new Network Policy given with the CiliumNetworkPolicy class and the namespace where to apply it.
  This method will return an error in case the namespace doesn't exist (with a Not Found Exception) in that cluster or a policy with the same name already exists (Conflict Exception) and the new policy created in the form of CiliumNetworkPolicy class in case of success.

- **Deletion of Cilium Network Policy:** this method deletes a Cilium Network Policy in a namespace. The parameters needed are the policy name and the namespace where is applied. If there is no policy matching in the cluster a Not Found error will be launched, otherwise, it will be returned from the server the policy deleted.

- **Edit of Cilium Network Policy:** this method is used to substitute a policy already created in the cluster, with another one. This method requires the name of the policy to modify, the namespace, and the new policy given with the CiliumNetworkPolicy class. If the policy doesn't exist an error will be raised(Not Found), while in case of success the edited policy will be returned.

- **Read of a Cilium Network Policy:** this method will return the selected policy if it exists. The method requires the name and the namespace where the policy is deployed and will return an error if it not exists(Not Found). Otherwise, the CiliumNetworkPolicy class will be returned.

- **Get Cilium Network Policy List of a namespace:**this method will return the list of network policies in a namespace. The method requires the namespace where the policies are deployed and will return an error if it not exists(Not Found). Otherwise, a list of CiliumNetworkPolicy class will be returned.

5.5 Service Communication

In this section will be described the operations done by the Multi Cluster Orchestrator to connect services.
For clarification will be given first a definition of what is meant for service-to-service communication in this context:

Service-to-service communication. pods running a service A can request a DNS resolution name for service B, receive an IP address of a pod running service B, and send connect requests to it.

This is not trivial in a Kubernetes Cluster because pods running in a namespace that is not the one of a service it wants to connect to, should send a different name to DNS for resolution(see section 5.5.2 for more information) or, if the service is in another cluster, the DNS could not resolve that name because it doesn't know it.
In this regard, the final goal of the Multi Cluster Orchestrator is to set up and create all that is needed for a pod to transparently resolve and connect with a service and act as it is in the same namespace.

First of all, there are two possible scenarios:

1. pods of service A that want to start a connection with pods running the service B are in the same namespace
2. pods of service A that want to start a connection with pods running the service B are in a different namespace

In the next two sections will be presented two examples, one for scenario, and the operations that the Multi Cluster Orchestrator does for the two scenarios will be described.

5.5.1 Services in the same Namespace

The service-to-service communication between services in the same namespace and cluster is trivial: it's always possible to resolve DNS queries and connect to pods in the same cluster's namespace, the only requirement is that pods running the service are included in the egress Network Policy rules.
If the services are in two different clusters but share the same namespace name, it is required to import the service to connect in the cluster's namespace and link to the service in the other cluster using the annotation introduced in Chapter 2 section 2.4.2.
Will be given an example to list and describe what are the operations done by the Multi Cluster Orchestrator in this case.
For the example will be considered the following scenario summarized also in figure 5.2:

1. a service *frontend* is deployed in the *default* namespace of *cluster1* in the *domain1*
2. a service *backend* is deployed in the *default* namespace of *cluster2* in *domain2* and is the one in Code Box 5.11
3. the domain manager of *domain1* applies a policy like the one in Code Box 5.2, specifying *backend* as the only service in the egress rules.
4. there is no service with name *backend* in the cluster1.
5. the two clusters are already in a cluster-mesh

After the policy refinement process described in 5.3, the Multi Cluster Orchestrator will check if there is already a service named *backend* and if it doesn't find it, will collect all the information about the service in its database (selector and port introduced section 5.1) and will create the service in the *cluster1*.
It is important that both the service named *backed* in *cluster1* and *cluster2* have the annotation *"io.cilium/global-service: 'true'"*, because with this parameter Cilium will group the two services and act like it is one service working on two clusters(Chapter 2 section 2.4.2).

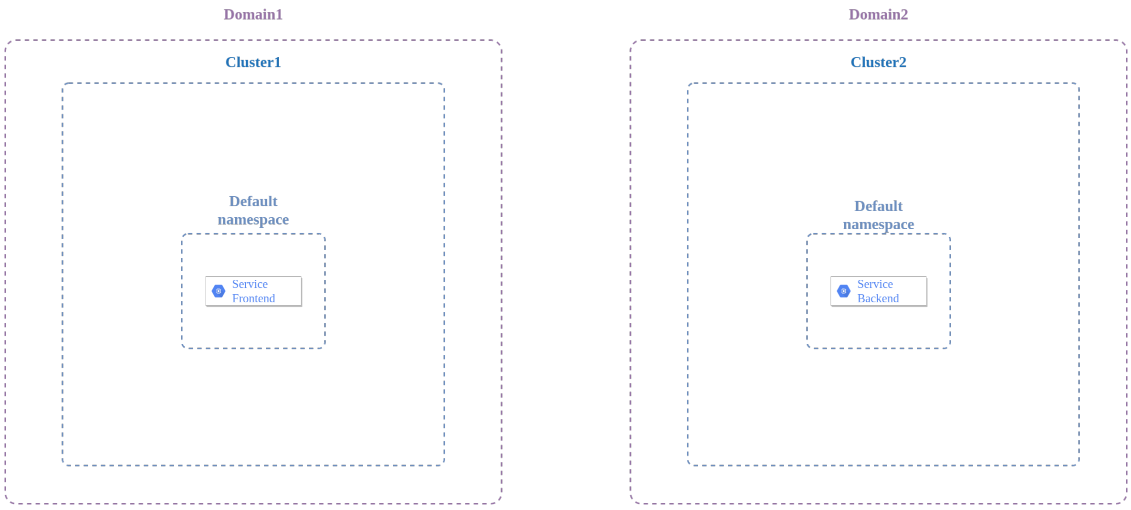


Figure 5.2. Example of configuration with services belonging to the same namespace

Doing in this way pods of service *frontend* now can resolve the *backend* service name and because in *cluster1* there are no pods matching the labels of service *backend*, after the DNS resolution of service *backend*, Cilium will resolve only IPs of pods in *cluster2*. The final status is shown in the figure 5.3.

The Multi Cluster Orchestrator will also maintain a link between the policy created and the service *backend* created in *cluster1*. In case the policy will be deleted, the Multi Cluster Orchestrator will also delete the service *backend* in the *cluster1*, if there are no other policies referencing the service *backend* in *cluster2*.

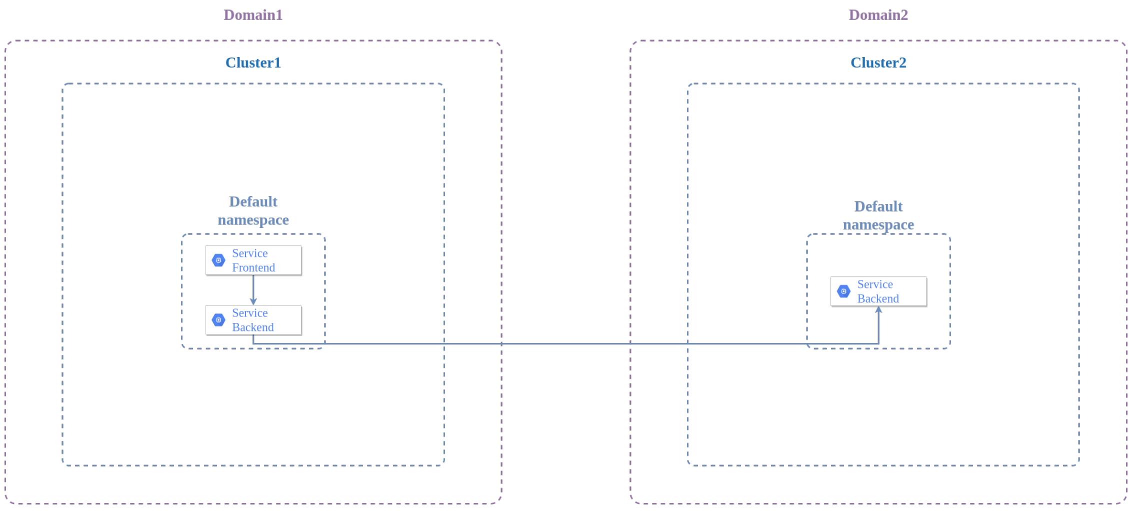


Figure 5.3. Final request path to reach the real service backend from service frontend

Code Box 5.11. YAML file of Service backend

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
  annotations:
    io.cilium/global—service: "true"
spec:
  type: ClusterIP
  ports:
 - port: 80
  selector:
    name: backend
```

5.5.2 Services in different Namespaces

Will be now presented an example where the pods of requirement's service are in a different namespace. The example in section 5.5.1 will be slightly modified, in particular:

1. the pods running service *frontend* are now deployed in the namespace "*frontend-namespace*"
2. the service *backend* is now the one in Code Box 5.12 with pods running in the same namespace(*backend-namespace*)

The example is summarized in figure 5.4.

Code Box 5.12. YAML file of Service backend in backend-namespace

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
  namespace: backend—namespace
  annotations:
    io.cilium/global—service: "true"
spec:
  type: ClusterIP
  ports:
 - port: 80
  selector:
    name: backend
```

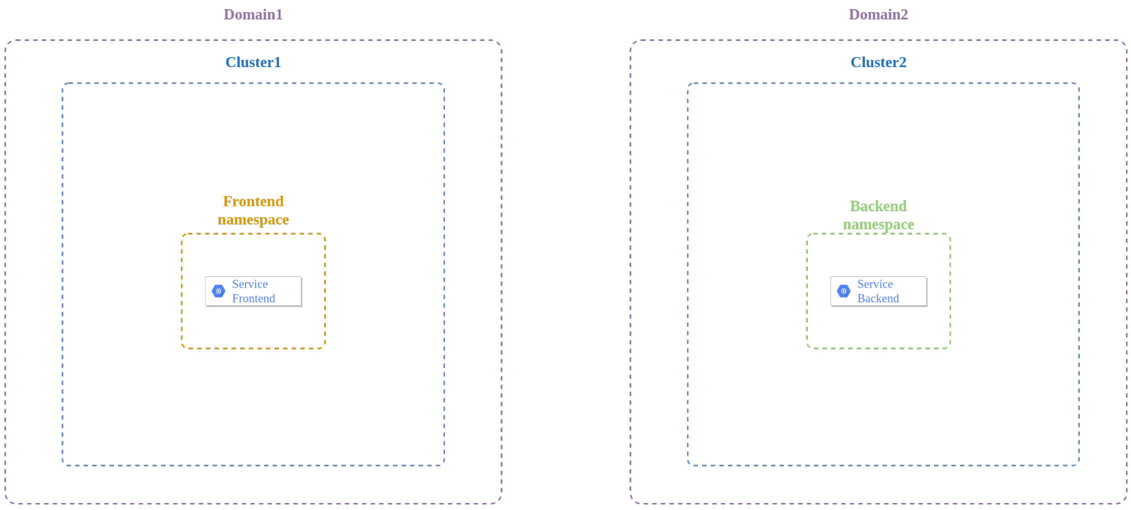


Figure 5.4. Example of configuration with services belonging to different namespaces

In this case, if all operations are done like in section 5.5.1, pods of *frontend* service still can't resolve the DNS name of the *backend* service because they are in different namespaces.
Kubernetes allows to resolve DNS service name in different namespace changing the request to "*service-name.service-namespace.svc.cluster.local*", but this means that applications need to be modified in order to request the right resolution name in the right namespace.
The solution adopted in the Multi Cluster Orchestrethor is to create a different service in the *frontend-namespace*, that reference the *backend* service in the *backend-namespace*.
In the Code Box 5.13 there is an example of the service deployed in *frontend-namespace* and it's a service of type external name. In an external name type service, the field *externalName* has the same structure as the DNS resolution requests for services in different namespaces. In this way, applications don't need to be modified as when they try to resolve the service *backend*, thanks to the external service the name will be substituted with the right one, and pods can receive the address of pods running the *backend* service.

Code Box 5.13. External Service

```
kind: Service
apiVersion: v1
metadata:
  name: backend
  namespace: frontend—namespace
  annotations:
    io.cilium/global—service: "true"
spec:
  type: ExternalName
```

```
    externalName: backend.backend−namespace.svc.cluster.local
    ports:
   - port: 80
```

In the end for the service-to-service communication between two services in different clusters and namespaces the Multi Cluster Orchestrator will do the following operations:

- in the cluster of the service selected by the requirement, create a namespace and a mirror service with the same name of the service to connect and the annotation *"io.cilium/global-service: 'true'"* if they don't exist yet
- create a service of type External name in the service namespace selected by the requirement adding as *externalName* parameter *"service-to-connect-name.service-to-connect-namespace.svc.cluster.local"*

The final path and the final configuration after the operations of the Multi Cluster Orchestrator are shown in the figure 5.5.

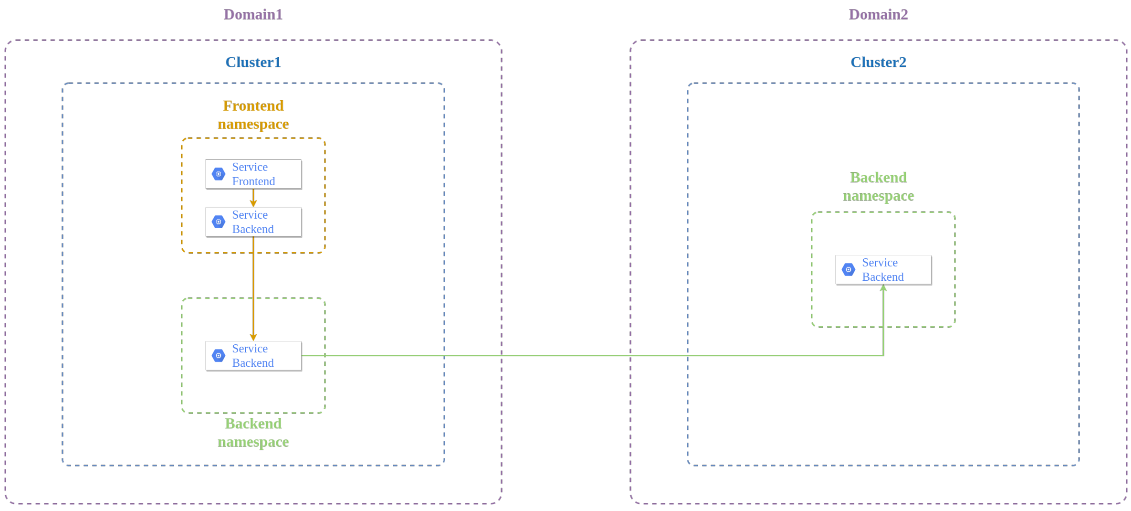


Figure 5.5. Final request path to reach the real service backend from service frontend

Chapter 6

Validation

This chapter will present some use cases to help understanding how the Multi Cluster Orchestrator works. In particular, two use cases will be presented, one that will expose an example of how the Multi Cluster Orchestrator operates in a multi-domain environment, what operations it performs and what are the final configurations for every cluster. In the second example, instead, a use case will be presented in a single domain environment to analyze the possible operations, in particular the interaction with services present on different clusters such as the second model presented in the Chapter 2 section 2.1.4.
In order to make concepts simple and help to understand every operation done by the Multi Cluster Orchestrator, in this Chapter the commands to perform for reaching the final state of the examples are not included. As mentioned in Chapter 5, in the examples, there will be the hypothesis that clusters are already linked with a Cilium ClusterMesh. Full information about how to install Cilium and set up the cluster mesh is presented in the Appendix B.
If the examples want to be replicated, in the Appendix C are present all commands and requests to do to the Multi Cluster Orchestrator to replicate the examples of this Chapter along with the data that is needed for the requests.

6.1 Use Case 1

The first use case that will be presented is an example including a multi-domain configuration in which two companies have some services that need to communicate with each other and want to do it automatically and securely so as not to have configuration problems that can lead to a disservice or a data breach. The following scenario is the one presented in the figure 6.1:

1. a *domain1* and *domain2* are registered to the Multi Cluster Orchestrator.
2. *domani1* is composed by two clusters with name *cl1* and *cl2* while *domain2* is composed by clusters *cl3* and *cl4*
3. in the *domain1* there are two services:
   (a) *service1* that is in the *ns1* namespace and deployed in cluster *cl1*

   (b) *service2* that is in *ns2* namespace and deployed in cluster *cl2*

4. in the *domain2* there are two services:

   (a) *service3* that is in *ns1* namespace and deployed in cluster *cl3*

   (b) *service4* that is in the *ns2* namespace and deployed in cluster *cl4*

All services used in this example are like the one in Code Box 6.1, in which it is included, other than the service specification, also a deployment that creates two pods running the service, and a ConfigMap, which contains a message that specifies where the service is running and it's returned when a request is made to the service.

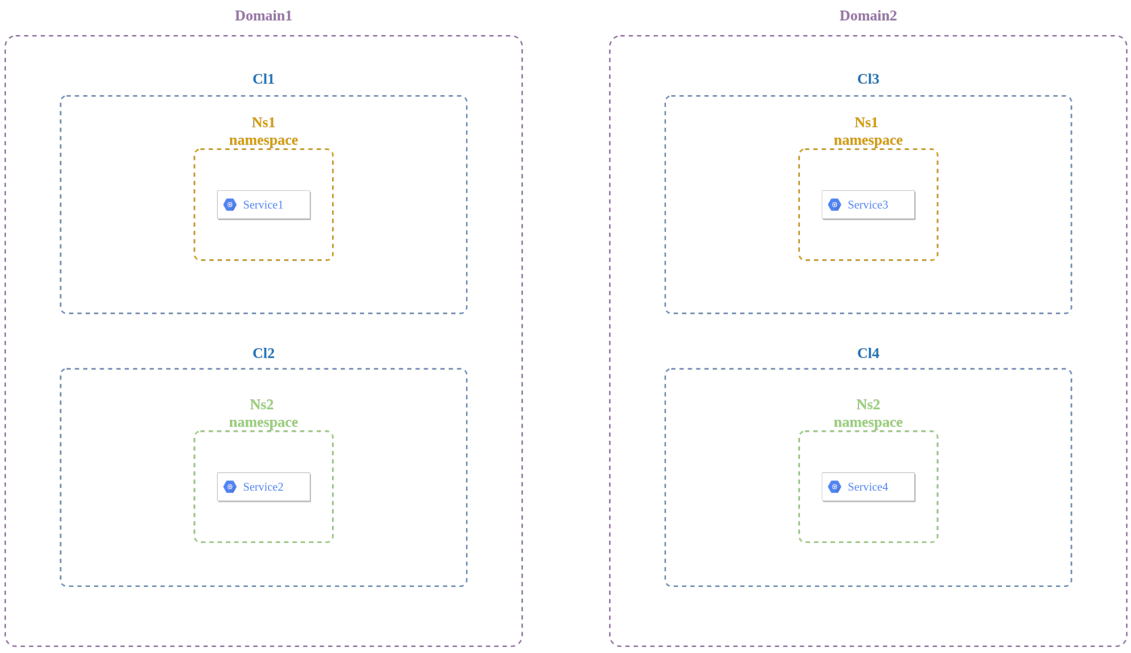


Figure 6.1. Use Case 1

Code Box 6.1. Example of Service with a Deployment

```
---
  apiVersion: v1
  kind: Service
  metadata:
    name: service
    namespace: service—namespace
    annotations:
      io.cilium/global—service: "true"
  spec:
    type: ClusterIP
    ports:
   - port: 80
    selector:
      name: service
---
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: service
  spec:
    selector:
      matchLabels:
        name: service
    replicas: 2
    template:
      metadata:
        labels:
          name: service
      spec:
        containers:
       - name: rebel—base
          image: docker.io/nginx:1.15.8
          volumeMounts:
           - name: html
            mountPath: /usr/share/nginx/html/
          livenessProbe:
            httpGet:
              path: /
              port: 80
            periodSeconds: 1
          readinessProbe:
            httpGet:
              path: /
              port: 80
        volumes:
            - name: html
            configMap:
              name: service—response
              items:
                - key: message
                  path: index.html
---
  apiVersion: v1
  kind: ConfigMap
  metadata:
    name: service—response
  data:
    message: "{\"Domain\":␣\"1\",␣\"Cluster\":␣\"Cluster-2\"}\n"
```

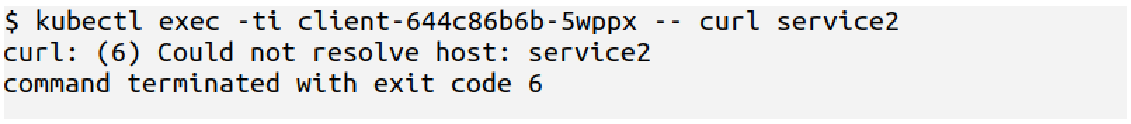
To retrieve a message from services will be used a client specified in Code Box 6.2 that creates a Deployment with two pods that are able to request HTML pages using the curl command.

Code Box 6.2. YAML file for Client Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client
spec:
  selector:
    matchLabels:
      name: client
  replicas: 2
  template:
    metadata:
      labels:
        name: client
    spec:
      containers:
     - name: x—wing—container
        image: docker.io/cilium/json—mock:1.2
        livenessProbe:
          exec:
            command:
           - curl
           - —sS
           - —o
           - /dev/null
           - localhost
         readinessProbe:
           exec:
             command:
```

```
            - curl
            - —sS
            - —o
            - /dev/null
            - localhost
```

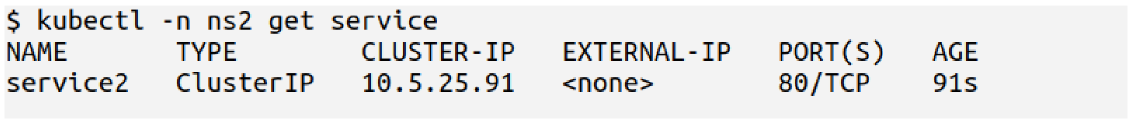
First of all the *client* deployment is applied in the cluster *cl3* in the *default* namespace and in fig 6.2 it is shown how the client is not able to find and contact the *service2* of the *domain1*.



```
$ kubectl exec -ti client-644c86b6b-5wppx -- curl service2
curl: (6) Could not resolve host: service2
command terminated with exit code 6
```

Figure 6.2. Request failure due to missing service in the cl3

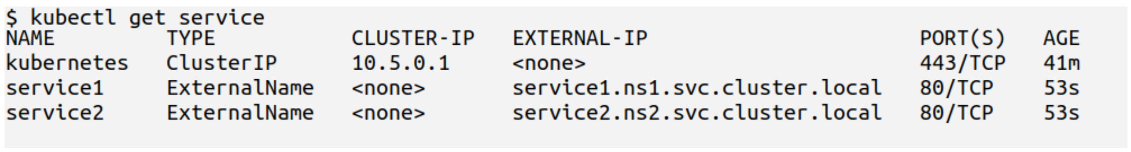
Applying the policy requirement in Code Box 6.3, the Multi Cluster Orchestrator will create the services and namespaces needed to start the communication with the *service1* and *service2* in the cluster *cl3* as shown in figures 6.3, 6.4, 6.5 that show the creation of *service2* in the namespace *ns2*, witch is created as well, and in default namespace.



```
$ kubectl -n ns2 get service
NAME        TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
service2    ClusterIP   10.5.25.91    <none>        80/TCP    91s
```

Figure 6.3. Service2 created in cl3

Along with that also a CiliumNetworkPolicy has been created with the name of "*allow-domain1*" and contains the labels of pods running the *service1* and *service2* in order to allow only this two traffics for the client's pods.



```
$ kubectl get service
NAME         TYPE           CLUSTER-IP   EXTERNAL-IP                      PORT(S)   AGE
kubernetes   ClusterIP      10.5.0.1     <none>                           443/TCP   41m
service1     ExternalName   <none>       service1.ns1.svc.cluster.local   80/TCP    53s
service2     ExternalName   <none>       service2.ns2.svc.cluster.local   80/TCP    53s
```

Figure 6.4. Service2 External Service created in the default namespace

Code Box 6.3. Policy requirement that allows domain1

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
```

```
metadata:
  name: "allow-domain1"
  spec:
    endpointSelector:
      matchLabels:
        name: client
    egress:
    - toEndpoints:
      - matchLabels:
          service: "*"
          cluster: "*"
          domain: "domain1"
```

```
$ kubectl get namespace
NAME                 STATUS   AGE
default              Active   42m
kube-node-lease      Active   42m
kube-public          Active   42m
kube-system          Active   42m
local-path-storage   Active   42m
ns1                  Active   28m
ns2                  Active   103s
```

Figure 6.5. Namespaces of cl3 after Service2 creation

If the client now requests the *service2* it will be able to contact it and receive a response as shown in the figure 6.6

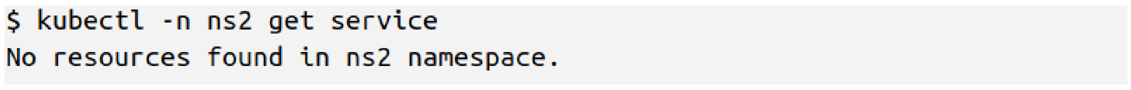
```
$ kubectl exec -ti client-644c86b6b-5wppx -- curl service2
{"Domain": "1", "Cluster": "Cluster-2"}
```

Figure 6.6. Successful request to Service2

If the policy is deleted, also the services and namespaces created will be deleted, because there are no other policies applied in the clusters that need that service and namespace. In the figures 6.7, 6.8, and 6.9, it's shown how now the services and namespaces have been deleted.

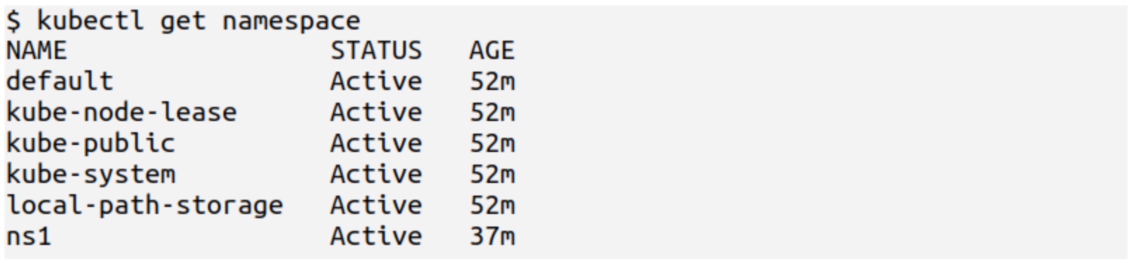
```
$ kubectl get service
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.5.0.1     <none>        443/TCP   51m
```

Figure 6.7. Services in default namespace of cl3 after policy removal



```
$ kubectl -n ns2 get service
No resources found in ns2 namespace.
```

Figure 6.8. Services in ns2 namespace of cl3 after policy removal



```
$ kubectl get namespace
NAME                 STATUS   AGE
default              Active   52m
kube-node-lease      Active   52m
kube-public          Active   52m
kube-system          Active   52m
local-path-storage   Active   52m
ns1                  Active   37m
```

Figure 6.9. Namespaces of cl3 after policy removal

6.2 Use Case 2

In this use case, the focus will be on services running in multiple clusters of the same domain and how the Multi Cluster Orchestrator handles the removal of policies or services. for this example, summarized in the figure 6.10, the main characteristics are:

1. a *domain1* is registered to the Multi Cluster Orchestrator
2. *domani1* is composed by two clusters with name *cluster1* and *cluster2*
3. in the *domain1* there are three services:
   (a) *backend* that is in the *backend-ns* namespace and deployed in clusters *cluster1* and *cluster2*
   (b) *database* that is in *database-ns* and deployed in *cluster1*
   (c) *client-service* that is in *default* and deployed in *cluster1* and *cluster2*

The services are deployed as the ones presented in the Code Box 6.1 of use case 1 and the client presented in Code Box 6.2 is deployed in both clusters in the *default* namespace and both are linked to the client service.

The Multi Cluster Orchestrator handles services running in multiple clusters in two different ways based on the requirement request. The first is shown in the Code Box 6.4 and is applied in the *cluster1* to the client's pods. In this case, selecting only the name of the service for the egress rules, the communication is allowed for both pods of *cluster1* and *cluster2*. Even if in the ingress rule is specified that the only allowed traffic is the one of *cluster1*, as shown in figure 6.11, the clients

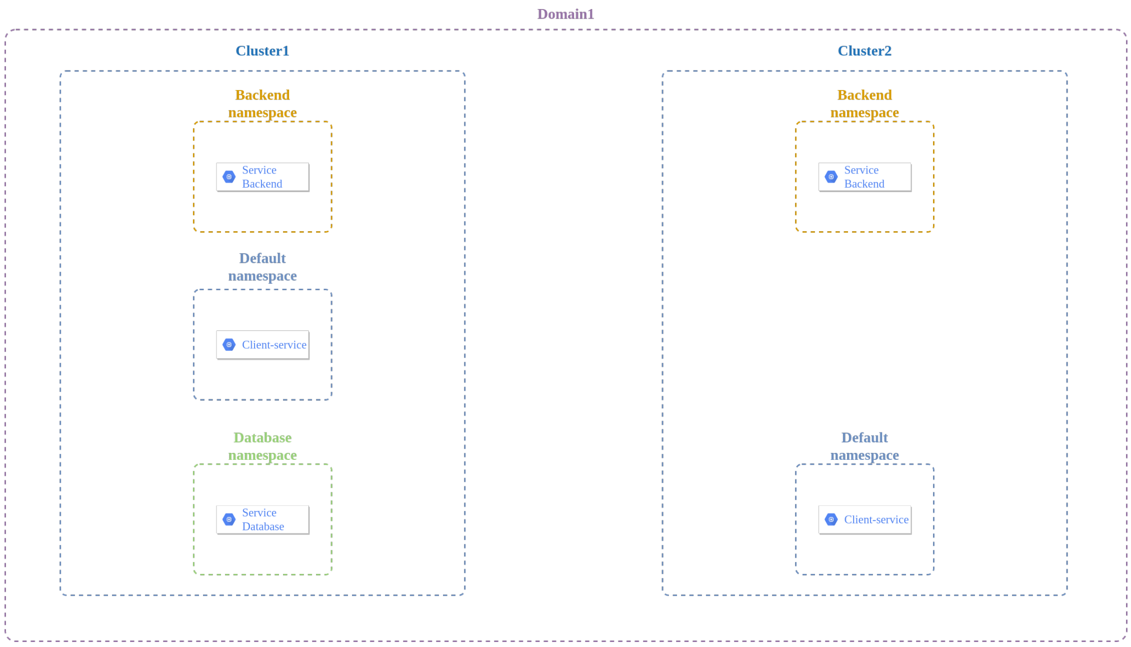


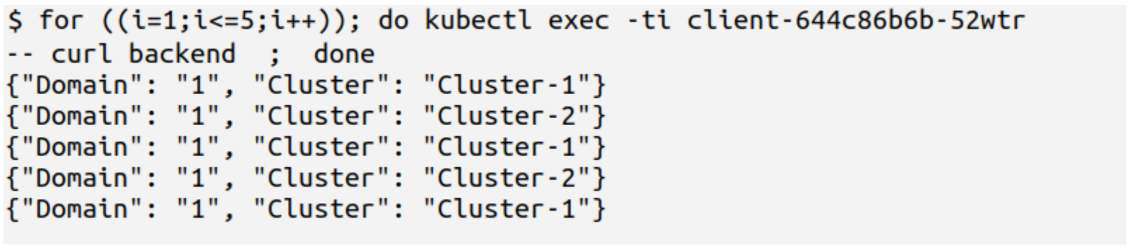
Figure 6.10. Use Case 2

receive a response for both pods of *cluster1* and *cluster2*: this happens because the traffic will be denied when the service *backend* tries to start a connection with the client-service while if the traffic is allowed in egress is guaranteed to receive the answer because *client-service* started the connection. This can be tested, as shown in fig 6.12, trying to request from *client-service* pods of *cluster2* to client-service pods of *cluster1*. In this case, the connection is refused because the ingress rule was only specified for the *cluster1*.

Code Box 6.4. Policy requirement that allows service backend

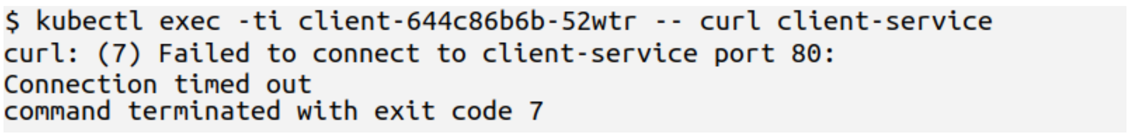
```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-domain1"
spec:
  endpointSelector:
    matchLabels:
      name: client
  egress:
  - toEndpoints:
    - matchLabels:
        service: "backend"
   ingress:
   - fromEndpoints:
     - matchLabels:
       service: "*"
       cluster: "cluster1"
```

```
          domain: "domain1"
```



```
$ for ((i=1;i<=5;i++)); do kubectl exec -ti client-644c86b6b-52wtr
-- curl backend  ;  done
{"Domain": "1", "Cluster": "Cluster-1"}
{"Domain": "1", "Cluster": "Cluster-2"}
{"Domain": "1", "Cluster": "Cluster-1"}
{"Domain": "1", "Cluster": "Cluster-2"}
{"Domain": "1", "Cluster": "Cluster-1"}
```

Figure 6.11. Request from client-service to backend service



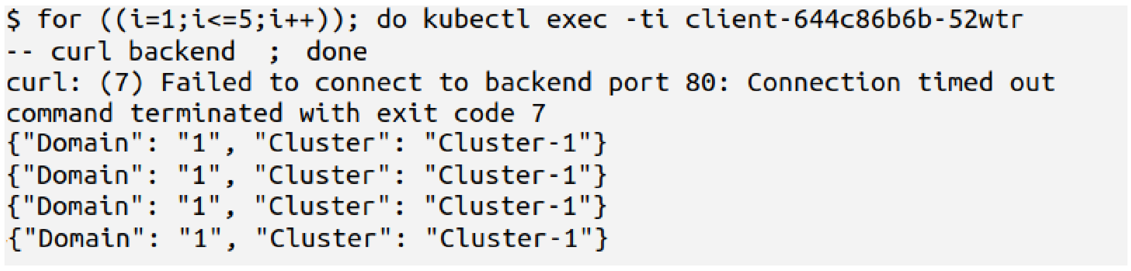
```
$ kubectl exec -ti client-644c86b6b-52wtr -- curl client-service
curl: (7) Failed to connect to client-service port 80:
Connection timed out
command terminated with exit code 7
```

Figure 6.12. Request from client-service in cluster2 to client-service in cluster1

In the second case in Code Box 6.5, the requirement in the egress rules specifies only services in *cluster1* so only pods of *cluster1* running the service *backend* are allowed. The figure 6.13 shows how *client-service* pods are able only to contact pods of *backend* service of *cluster1*, while the other requests are blocked.

Code Box 6.5. Example of a requirement that allows all services of cluster1

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-cluster1"
spec:
  endpointSelector:
    matchLabels:
      name: client—service
  egress:
 - toEndpoints:
   - matchLabels:
        service: "*"
        cluster: "cluster1"
        domain: "domain1"
```



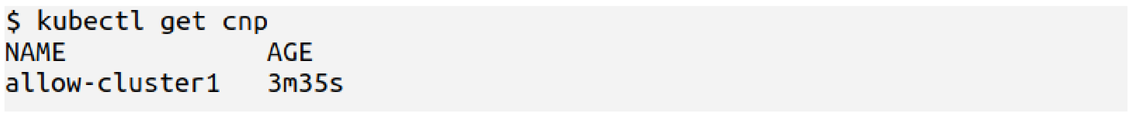
```
$ for ((i=1;i<=5;i++)); do kubectl exec -ti client-644c86b6b-52wtr
-- curl backend  ;  done
curl: (7) Failed to connect to backend port 80: Connection timed out
command terminated with exit code 7
{"Domain": "1", "Cluster": "Cluster-1"}
{"Domain": "1", "Cluster": "Cluster-1"}
{"Domain": "1", "Cluster": "Cluster-1"}
{"Domain": "1", "Cluster": "Cluster-1"}
```

Figure 6.13. request from client-service to backend service

Will now be added a new policy containing only the service backend. the policy is summarized in Code Box 6.6. If now the service *backend* is removed, the Multi Cluster Orchestrator will look for all the applied policies containing that service, and if it's the only one, will remove also the policy. In this case, as shown in the figures 6.14 and 6.15, the last policy is removed because the service *backend* was the only one available for that policy.

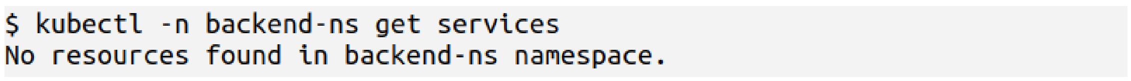
Code Box 6.6. Example of a requirement that allows service backend

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-service-backend"
spec:
  endpointSelector:
    matchLabels:
      name: client—service
egress:
— toEndpoints:
  - matchLabels:
      service: "backend"
```



```
$ kubectl get cnp
NAME             AGE
allow-cluster1   3m35s
```

Figure 6.14. Policies after the backend service removal



```
$ kubectl -n backend-ns get services
No resources found in backend-ns namespace.
```

Figure 6.15. Services in backend-ns namespace after service backend removal

Chapter 7

Conclusions

During the work of this thesis, a lot of points were analyzed to reach the goals, which were the analysis of the current technologies and the configuration's automation of clusters mesh, service-to-service communications between services from different Kubernetes clusters, and security policies that allow traffic protection from other entities that are not allowed to communicate with a particular service or group of pods.

First of all, technologies representing the state of the art for the Multi Cluster and service-to-service communications, and for the network isolation were analyzed to understand how they work and what are their strengths and weaknesses. Based on the analysis results, it emerged that all technologies have no tools for the automatic configuration of cluster mesh, service discovery, and application of policies. Cilium was chosen as a base to start with a more abstracted orchestrator, which can be placed at a higher level from clusters to configure them based on users' requirements.

To resolve all problems of these technologies, it was decided to create a Multi Cluster Orchestrator that was able to provide automation for the configurations to be made, and also to fill the lack of information on clusters of different companies, if necessary. A high-level model was proposed which, based on high-level requirements given by domain managers, would create a global configuration of the connections between clusters and services which would then be used to derive the individual cluster configurations.

The next step was to pass to the implementation phase which concerned in particular the initial development of the REST server and the implementation of the modules for the application of security policies and communication between services. It was described at first the choice made to implement the requirements format request, how the subsequent refinement took place and how the Multi Cluster Orchestrator communicates with the individual clusters through the Cilium and Kubernetes APIs in order to apply Network Policies and create global services for the service-to-service communication.

Finally, two use cases were provided to fully understand how we can interact with the Multi cluster Orchestrator and what operations it does initially. One use case was about the interaction of the Multi Cluster Orchestrator with multiple domains, while the second use case was focused on the operations done with services running on multiple clusters of the same domain.

This work laid the foundation for the development of a new framework that could help in the process of integration of multiple clusters of different companies, simplifying and automate all the tasks required to perform to securely make services communicate with each other.

The development of this thesis focused mainly on the first creation of a model that could work to achieve the proposed objectives and to start the implementation of the most important parts which are the application of network policies in the clusters and the creation of paths and namespaces to automate the communication between services.

One future work could concern the development of the module for the creation of the cluster-mesh between clusters, which can optimally understand which clusters to introduce into the mesh and automate all the required operations with also the creation of the global configuration that keeps track of the clusters and related services to speed up the creation of communication services. The development can include also a verification process of reachability requirements for clusters and services as described in [25] and [26].
This is a very important part that could provide great help to anyone who needs to link clusters together in an automated way. The tasks to be performed manually concern both the addressing spaces of the individual clusters and the assignment of unique names in order to allow the mesh cluster to understand where to forward the traffic. These operations, if done manually and on a large scale, can lead to configuration errors that can cause unwanted events such as the application of policies in a different cluster or the sending of traffic to an unwanted cluster. In addition, some information that is now required in this version of the Multi Cluster Orchestrator such as the cluster's name attributed by Cilium, or the context of a cluster, would no longer be requested from the domain manager but would be automatically deduced during the creation of the mesh cluster.

Another development point could concern the creation of an interface that helps the domain manager to set the requests in the most intuitive way possible so that he can easily decide which services can communicate with external services and the application of policies.
Currently, to create domains, policies, and services, it is necessary to provide the server with objects that can be more or less complicated. For example, the requirement of the policy and communication with services has a very long and complicated structure like that of the Cilium Network Policy. Going to create an interface that visually allows to express the concepts at the highest level and which are then translated into objects to be supplied to the Multi Cluster Orchestrator, would reduce and simplify the work for the domain manager who will no longer have to worry about having to create these objects so complex, but perhaps simpler objects with less information that will then be elaborated and transformed in the final request. Having an interface can permit also a more refinement process as described in [27] and [28] for network functions, which allow to having a high-level language that can be then translated into formal verification models of different verification tools to detect misconfigurations in the requirements.

Another future work could be the development of a CNI to make the Multi Cluster Orchestrator independent from Cilium and that can provide more efficient management of the service discovery or that provides for the insertion of the domain identity.
As for service discovery, currently with the solution adopted it is not possible to interact with the DNS to make it understand that some clusters running the service, for example, are not enabled by network policies, so the resolution is also made towards those clusters and is subsequently blocked from policy controller. Going to modify or create a new CNI that foresees to make the DNS aware that particular addresses are blocked by the policies, it would make possible a more efficient communication between the services without having to make requests to addresses that will then be blocked by the control of the network policy. As for the introduction of the domain entity, it would make it easier to apply global policies for the domain without the need to derive different policies for each cluster and apply them individually, also reducing the configuration time as already it occurs for the application of policies on services running on different clusters, which is done at the kernel level using hashmaps.

Bibliography

[1] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. 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.

[2] "What is kubernetes?" https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, accessed: 2021-08-16.

[3] "Pods—kubernetes," https://kubernetes.io/docs/concepts/workloads/pods/, accessed: 2021-08-16.

[4] "Services—kubernetes," https://kubernetes.io/docs/concepts/services-networking/service/, accessed: 2021-08-17.

[5] "Understanding multi-cluster kubernetes," https://www.getambassador.io/learn/multi-cluster-kubernetes/, accessed: 2021-08-17.

[6] "What is istio?" https://istio.io/latest/about/service-mesh/, accessed: 2021-08-17.

[7] "What is a service-mesh?" https://istio.io/latest/about/service-mesh/, accessed: 2021-08-17.

[8] "Istio/architecture," https://istio.io/latest/docs/ops/deployment/architecture/, accessed: 2021-08-17.

[9] "Istio/multicluster deployments," https://istio.io/v1.2/docs/concepts/multicluster-deployments/, accessed: 2021-08-18.

[10] "Project calico," https://www.tigera.io/project-calico/, accessed: 2021-08-18.

[11] "Calico/ component architecture," https://docs.projectcalico.org/reference/architecture/overview, accessed: 2021-08-18.

[12] "Calico," https://www.ibm.com/docs/en/cloud-private/3.2.0?topic=ins-calico, accessed: 2021-08-18.

[13] "Calico-enterprise/federation," https://docs.projectcalico.org/security/calico-enterprise/federation, accessed: 2021-08-20.

[14] "Calico-enterprise an overview," https://www.tigera.io/blog/calico-enterprise-an-overview/, accessed: 2021-08-20.

[15] "Cilium/introduction to cilium and hubble," https://docs.cilium.io/en/v1.10/intro/, accessed: 2021-08-18.

[16] "Cilium/architecture guide," https://docs.cilium.io/en/k8s-doc/architecture/, accessed: 2021-08-18.

[17] "Cilium/deep dive into cilium cluster-mesh," https://cilium.io/blog/2019/03/12/clustermesh, accessed: 2021-08-20.

[18] D. Bringhenti, G. Marchetto, R. Sisto, and F. 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.

[19] I. Pedone, A. Lioy, and F. Valenza, "Towards an efficient management and orchestration framework for virtual network security functions," *Security and Communication Networks*, vol. 2019, p. 2425983, Nov 2019. [Online]. Available: https://doi.org/10.1155/2019/2425983

[20] "Network policies—kubernetes," https://kubernetes.io/docs/concepts/services-networking/network-policies/, accessed: 2021-08-17.

[21] "About network policy," https://docs.projectcalico.org/about/about-network-policy, accessed: 2021-08-17.

[22] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated optimal firewall orchestration and configuration in virtualized networks," in *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1109/NOMS47738.2020.9110402

[23] ——, "Introducing programmability and automation in the synthesis of virtual firewall rules," in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, 2020, pp. 473–478.

[24] "Calico/about network policy," https://docs.projectcalico.org/about/about-network-policy, accessed: 2021-08-18.

[25] D. Bringhenti, G. Marchetto, R. Sisto, S. Spinoso, F. Valenza, and J. Yusupov, "Improving the formal verification of reachability policies in virtualized networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 713–728, 2021.

[26] F. Valenza, S. Spinoso, and R. Sisto, "Formally specifying and checking policies and anomalies in service function chaining," *Journal of Network and Computer Applications*, vol. 146, pp. 102–419, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S108480451930253X

[27] G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "A framework for verification-oriented user-friendly network function modeling," *IEEE Access*, vol. 7, pp. 99 349–99 359, 2019.

[28] C. Basile, F. Valenza, A. Lioy, D. R. Lopez, and A. P. Perales, "Adding support for automatic enforcement of security policies in nfv networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 2, pp. 707–720, 2019.

[29] "Cilium—setting up cluster mesh," https://docs.cilium.io/en/v1.9/gettingstarted/clustermesh/, accessed: 2021-08-22.

[30] "Github/cilium/clustermesh-tools," https://github.com/cilium/clustermesh-tools.git, accessed: 2021-08-22.

Appendices

Appendix A

Rest API

This Appendix is dedicated to the Multi Cluster Orchestrator's API server and will describe all resources and paths to make requests.
In the figure A.1 its shown what is the resource graph, while in the tables are described all the path with methods that is possible to request.

Some resources for POST requests in paths are composed by different classes, in particular:

- **DomainRequest:** this is a resource required in the POST method for the path *API/domains* to create a new domain. It is possible to specify only the domain to create but also the clusters and servicese belonging to this domain. For this reason the resource is composed of:
  - a Domain class that specifies the name of the domain
  - a list of Clusters class that specifies what are the clusters belonging to the domain
  - a list of Services class that specifies what are the services belonging to the domain
  - a Map with key the service name and value a cluster to link services with clusters

  if the Service list is specified, also the Map must exists or there will be no service creation.
- **ServiceRequest:** this is a resource required in the POST method for the path *API/domains/services/create* to create a new service in the domain. To create the service it must be specified also what are the clusters running that service. For this reason the resource is composed of:
  - a Service class that specifies the service's attribute
  - a list of String that specifies what are the clusters name that run the service.

Figure A.1. Resource Graph

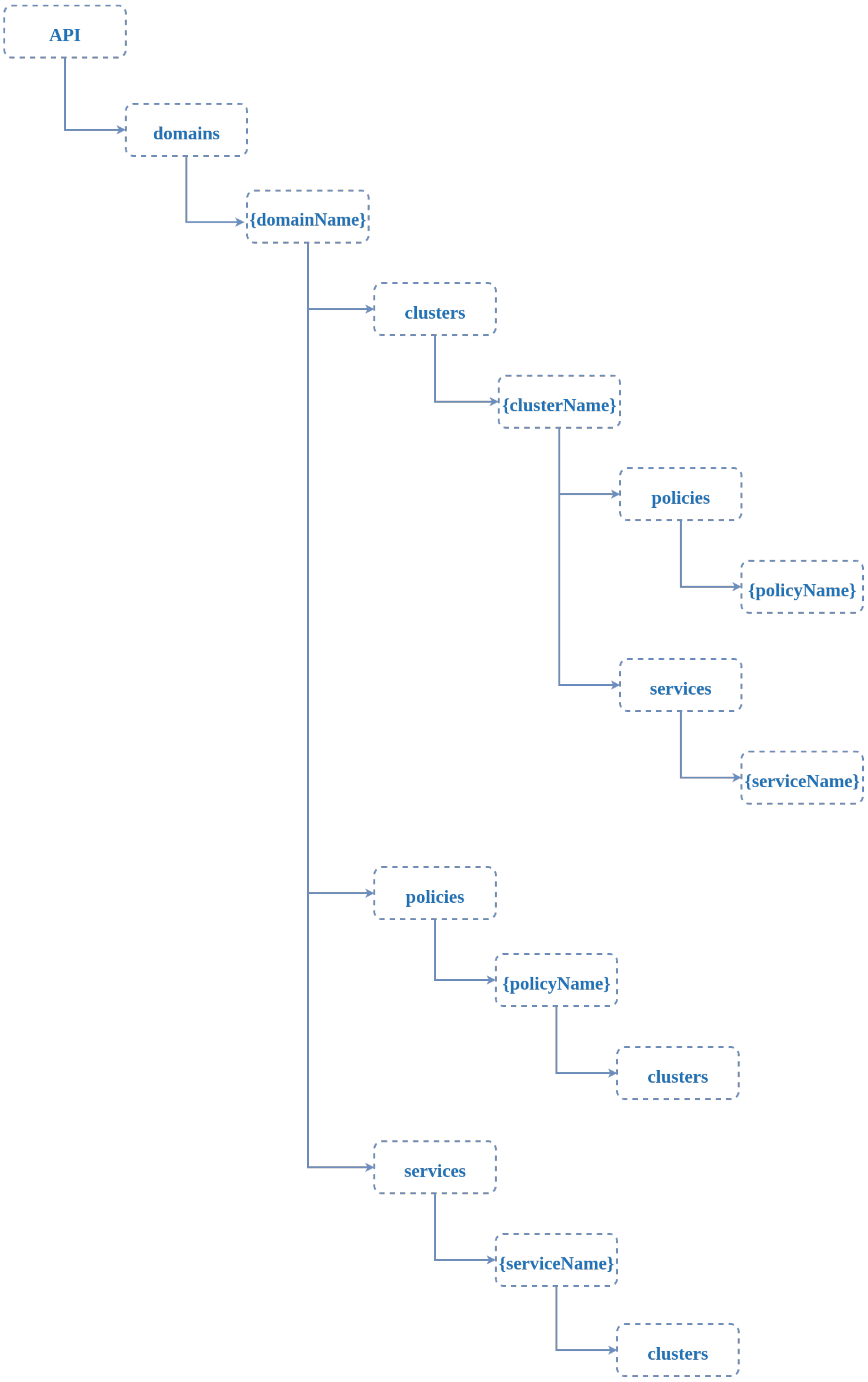


Table A.1: RESTful API Design

Resource	*Verb*	*Request Body*	*Status*	*Response Body*	*Description*
API/domains	GET		200 OK 404 Not Found	Domains	Retrieve all domains in the database
API/domains	POST	DomainRequest	201 Created 409 Conflict 400 Bad Request	Domain	Create a new Domain.
API/domains/ {domainName}	GET		200 OK 404 Not Found	Domain	Retrieve a domain by its name
API/domains/ {domainName}/remove	DELETE		200 OK 404 Not Found	boolean	Delete the selected domain.
API/domains/ {domainName}/clusters	GET		200 Ok 404 Not Found	Clusters	Retrieve all clusters in the specified domain.
API/domains/ {domainName}/clusters /remove	DELETE		200 OK 404 Not Found	boolean	Remove a cluster from the selected domain.
API/domains/ {domainName}/clusters /create	POST	Cluster	201 Created 400 Bad Request 409 Conflict 404 Not Found	Cluster	Create a new cluster in the domain.

Resource	*Verb*	*Request Body*	*Status*	*Response Body*	*Description*
API/domains/ {domainName}/clusters /{clusterName}	GET		200 OK 404 Not Found	Cluster	Retrieve a cluster by its name and domain's name.
API/domains/ {domainName}/clusters /{clusterName}/policies	GET		200 Ok 404 Not Found	Policies	Retrieve all policies applied in the selected cluster in the selected domain.
API/domains/ {domainName}/clusters /{clusterName}/policies /create	POST	Policy	201 Created 400 Bad Request 409 Conflict 404 Not Found	Policy	Create a new Policy in the selected cluster of the selected domain.
API/domains/ {domainName}/clusters /{clusterName}/policies /{policyName}	GET		200 Ok 404 Not Found	Policy	Retrieve a Policy in the selected cluster of the selected domain.
API/domains/ {domainName}/clusters /{clusterName}/policies /{policyName}/remove	DELETE		200 OK 404 Not Found 400 Bad Request	boolean	Delete a Policy in the selected cluster of the selected domain.

Resource	*Verb*	*Request Body*	*Status*	*Response Body*	*Description*
API/domains/{domainName}/clusters/{clusterName}/services	GET		200 Ok 404 Not Found	Services	Retrieve all services in a cluster of selected domain.
API/domains/{domainName}/clusters/{clusterName}/services/create	POST	Service	201 Created 400 Bad Request 409 Conflict 404 Not Found	Service	Create a new service in the selected cluster of selected domain.
API/domains/{domainName}/clusters/{clusterName}/services/{serviceName}	GET		200 Ok 404 Not Found	Service	Retrieve a service in the selected cluster of selected domain.
API/domains/{domainName}/clusters/{clusterName}/services/{serviceName}/remove	DELETE		200 OK 404 Not Found 400 Bad Request	boolean	Delete a service in the selected cluster of selected domain.

Resource	*Verb*	*Request Body*	*Status*	*Response Body*	*Description*
API/domains/ {domainName}/services	GET		200 Ok 404 Not Found	Services	Retrieve all services in the specified domain.
API/domains/ {domainName}/services /create	POST	ServiceRequest	201 Created 400 Bad Request 409 Conflict 404 Not Found	Service	Create a service in the selected clusters of specified domain.
API/domains/ {domainName}/services /{serviceName}	GET		200 Ok 404 Not Found	Service	Retrireve a service in the specified domain.
API/domains/ {domainName}/services /{serviceName}/remove	DELETE		200 OK 404 Not Found 400 Bad Request	boolean	Delete a service in the specified domain.
API/domains/ {domainName}/services /{serviceName}/clusters	GET		200 Ok 404 Not Found	Clusters	Retrieve a list of clusters in the specified domain that run the service.

Resource	*Verb*	*Request Body*	*Status*	*Response Body*	*Description*
API/domains/ {domainName}/policies	GET		200 Ok 404 Not Found	Policies	Retrieve all policies in the specified domain.
API/domains/ {domainName}/policies /create	POST	Policy	201 Created 400 Bad Request 409 Conflict 404 Not Found	Policy	Create a policy in the selected clusters of specified domain.
API/domains/ {domainName}/policies /{policyName}	GET		200 Ok 404 Not Found	Policy	Retrireve a policy in the specified domain.
API/domains/ {domainName}/policies /{policyName}/remove	DELETE		200 OK 404 Not Found 400 Bad Request	boolean	Delete a policy in the specified domain.
API/domains/ {domainName}/policies /{policyName}/clusters	GET		200 Ok 404 Not Found	Clusters	Retrieve a list of clusters in the specified domain that have the policy applied.

Appendix B

Cluster Mesh

B.1 Set up

The first step is to set up the environment where the Multi Cluster Orchestrator can operate. A multicluster sandbox can be created using kind, a tool that can generate Kubernetes clusters using docker containers. On top of that after installing the desired number of clusters, a cluster-mesh will be added between all clusters using the documentation provided by Cilium [29].

B.2 Creating Clusters

As mention before, kind can be used to create all clusters. Kind requires some dependencies to work in particular:

- a stable version of docker
- kubectl version $\geq$ v1.14.0
- helm $\geq$ v3.0.3

After the installation of all dependencies, it is possible to download kind and install it. To work kind requires a YAML configuration file for every cluster that will be created. In the YAML file, it's possible to set the number of control nodes for every cluster, worker node, pod, and service subnet. It is important to set different pod and service subnets for every cluster because the cluster-mesh requires different address spaces for all the clusters in the mesh. A configuration YAML file for kind might looks like this:

Code Box B.1. Example of a kind configuration file

```
kind: Cluster
apiVersion: kind.x—k8s.io/v1alpha4
nodes:
— role: control—plane
— role: worker
networking:
disableDefaultCNI: true
podSubnet: "10.0.0.0/16"
serviceSubnet: "10.1.0.0/16"
```

In this example a cluster with two nodes will be created, one control node, and one worker node and the pods' addresses will be in the 10.0.0.0/16 subnet, while the services addresses will be in the 10.1.0.0/16 subnet.

To create the cluster, the following command must be launched:

$sudokindcreatecluster - -name = cluster1–config = kind - cluster1.yaml$

where the *name* parameter is the name of the cluster and the *config* parameter is the configuration YAML file.

B.3 Intalling Cilium in the cluster

After the creation of the cluster, the next step is to install Cilium as CNI. For this step the tool used is Helm. Helm is a tool that helps in the installation of Kubernetes applications because it has its repository where all commons Kubernetes applications are located and can be pulled and installed. First of all its needed to pull the Cilium repository with the following command:

$helmrepoaddciliumhttps : //helm.cilium.io$

After the setup of the repository, the cilium image must be pre-loaded in kind with these commands:

$dockerpullcilium/cilium : v1.9.9; kindloaddocker - imagecilium/cilium : v1.9.9$

To install Cilium in the cluster1 for example, it's possible to launch this command:

Code Box B.2. Example of a Cilium installation with Helm

```
helm install cilium cilium/cilium ——version 1.9.5 \
——namespace kube—system \
——set nodeinit.enabled=true \
——set kubeProxyReplacement=partial \
——set hostServices.enabled=false \
——set externalIPs.enabled=true \
——set nodePort.enabled=true \
--set hostPort.enabled=true \
--set etcd.enabled=true \
--set etcd.managed=true \
--set identityAllocationMode=kvstore \
--set cluster.name=cluster1 \
--set cluster.id=1 \
--set ipam.operator.clusterPoolIPv4PodCIDR=10.0.0.0/16 \
--set ipam.operator.clusterPoolIPv4MaskSize=24
```

In this command it is possible to choose the version of Cilium desired, and in witch namespace the Cilium pods will be deployed. It's important to set the parameters *etcd.enabled* and *etcd.managed* to true because it's required for the correct set up of the cluster-mesh. The *cluster.name* and the *cluster.id* parameters must be unique for every cluster and the *pam.operator. clusterPoolIPv4PodCIDR* parameter must be the same as *podSubnet* value in the kind configuration file, while the *pam.operator.clusterPoolIPv4MaskSize* defines the range of addresses for every node.

B.4 Installing the cluster-mesh

To set the cluster-mesh there are other dependencies required to be satisfied in order to work properly. In the first place, every cluster needs to expose its etcd in read-only mode, in order to be reachable by all other clusters in the mesh. For this reason in the namespace where cilium was deployed it's mandatory to have the following service:

Code Box B.3. Service cilium-etcd-external

```
apiVersion: v1
kind: Service
metadata:
  name: cilium-etcd-external
  namespace: kube-system
spec:
  type: NodePort
  ports:
 - port: 2379
  selector:
    app: etcd
    etcd cluster: cilium-etcd
    io.cilium/app: etcd-operator
```

This service must be applied to all clusters that have to join the cluster-mesh. For the next steps the git repository of Cilium [30] will be used: the repository contains all tools required to extract the TLS key of every cluster and to generate the YAML files that will setup the cluster-mesh. For every cluster is required to extract the TLS key and root CA authority of local etcd using the script:

$./extract - etcd - secrets.sh$ (Note: these scripts extract only the secret of the cluster in the current context. To move thought contexts it's possible to use kubectl for example $kubectlconfiguse - contextkind - cluster2$)

After all secrets in all clusters are being extracted, a single Kubernetes secret from all the keys and certificates extracted must be created. This secret contains the etcd configuration that specifies the service IP or hostname of the etcd and includes the keys and certificates to access it. To create this secret the following script can be used:

./generate-secret-yaml.sh ¿ clustermesh.yaml

The next step is to create the patch to be applied to all Cilium DemonSet of all clusters:

./generate-name-mapping.sh ¿ ds.patch

This script will generate a patch that might look like this:

Code Box B.4. Example of a ds.patch file

```
spec:
  template:
    spec:
      hostAliases:
     - ip: "10.0.0.18"
        hostnames:
       - cluster1.mesh.cilium.io
     - ip: "10.0.2.19"
        hostnames:
       - cluster2.mesh.cilium.io
```

The next step is to apply the patch in all clusters, apply the Kubernetes secrets, and restarting the cilium agents using the commands:

Code Box B.5. Commands to aplly the cluster mesh

```
kubectl —n kube—system patch ds cilium —p "$(cat␣ds.patch)"

kubectl —n kube—system apply —f clustermesh.yaml

kubectl —n kube—system delete pod —l k8s—app=cilium

kubectl —n kube—system delete pod —l
    name=cilium—operator
```

After the cilium agents restart it's possible to check the status of the cluster-mesh by searching the Cilium-agent (for example cilium-asrtvg) and launching the following command:

$kubectl - nkube - systemexec - ticilium - asrtvg - -ciliumstatus$

the output should be similar to the one in the Code Box and should be checked for every cluster in the cluster-mesh. If one cluster doesn't find other clusters or clusters keep being unreachable it's wort to restart again the Cilium-agent end Cilium-operator in that cluster using the last two commands in Code Box.

Code Box B.6. Example of a Cilium status report

```
KVStore: Ok etcd: 1/1 connected, lease-ID=6b797bc07c20fb19, lock lease-ID=6b797bc07c20fb21, has-quorum=true: https://cilium-etcd-client.kube-system.svc:2379 - 3.3.12 (Leader)
Kubernetes: Ok 1.20 (v1.20.2) [linux/amd64]
Kubernetes APIs: ["cilium/v2::CiliumClusterwideNetworkPolicy", "cilium/v2::CiliumNetworkPolicy", "core/v1::Namespace", "core/v1::Node", "core/v1::Pods", "core/v1::Service", "discovery/v1beta1::EndpointSlice", "networking.k8s.io/v1::NetworkPolicy"]
KubeProxyReplacement: Partial [eth0 (Direct Routing)]
Cilium: Ok OK
NodeMonitor: Listening for events on 8CPUs with 64x4096 of shared memory
Cilium health daemon: Ok
IPAM: IPv4: 9/255 allocated from 10.0.0.0/24,
ClusterMesh: 3/3 clusters ready, 0global-services
BandwidthManager: Disabled
Host Routing: Legacy
Masquerading: IPTables
Controller Status: 75/75 healthy
Proxy Status: OK, ip 10.0.0.13, 0redirects active on ports 10000-20000
Hubble: Ok Current/Max Flows: 4096/4096 (100.00%), Flows/s: 141.92 Metrics: Disabled
Cluster health: 4/4 reachable (2021-09-07T13:45:19Z)
```

Appendix C

Use Case replication

This appendix will contain all commands and data to replicate the use cases presented in the Chapter 6. Before reading this Appendix is worth reading the Appendix B that describes how to install and set up a Cilium ClusterMesh, which is used as a prerequisite for the operations described here.

Although these use cases can be replicated manually following the next sections, it was also implemented a client that can communicate with the REST APIs of the Multi Cluster Orchestrator and that contains already all operations to replicate these use cases. The last section of this Appendix will be dedicated to this client explaining what methods it contains and commands to launch to replicate the two use cases.

C.1 Use Case 1

For the first use case presented in Chapter 6 it's needed first to install the services presented in the Code Box of the Chapter 6 using the following command in any cluster that needs a service:

kubectl apply -f servicefile.yaml

servicefile.yaml is the YAML file containing the service definition and the deployment. If the namespace where the pods' services need to be started does not exists, it is possible to create it using the command:

kubectl create namespace namespaceName

Once installed all services and the client, the next step is to register the two domains to the Multi Cluster Orchestrator. It is possible to register the domain only with the name and add services and clusters after using other APIs paths or use the path "*/API/domains/create*" using the JSON object described in Code Box C.1. In the Code Box C.2 there is the example for the domain1.

Code Box C.1. Template of a JSON DomainRequest

```

{
"domain":{
        "name":"",
        "links":[]
 },
"clusters":[
        {
                "domain":"",
                "name":"",
                "cilium_cluster":"",
                "context":"",
                "links":[]
        },
        ],
"services":[
        {
                "name":"service3",
                "domain":"",
                "namespace":"",
                "selector":"",
                "port":"",
                "links":[]
        },
        ],
"servicesMap":{
        "serviceName":[""],
        }

}
```

Once the domain are registered to the Multi Cluster Orchestrator, to apply policies and start the service discovery it's possible to make request to apply to all domains making requests to the API path "*/API/domains/{domainName}/policies/create*" or "*/API/domains/{domainName}/clusters/{clusterName}/policies/create*" giving as parameter the policy specified in the Code Box of Chapter 6.

Code Box C.2. Example of DomainRequest

```

{
"domain":{
        "name":"domain2",
        "links":[]
},
"clusters":[
        {
                "domain":null,
                "name":"cluster1",
                "cilium_cluster":cluster3,
                "context":"kind-cluster3",
                "links":[]
        },
        {
                "domain":null,
                "name":"cluster2",
                "cilium_cluster":cluster4,
                "context":"kind-cluster4",
                "links":[]
        }
],
"services":[
        {
                "name":"service3",
                "domain":null,
                "namespace":"ns1",
                "selector":"name:service3",
                "port":"TCP:80",
                "links":[]
        },
        {
                "name":"service4",
                "domain":null,
                "namespace":"ns2",
                "selector":"name:service4",
                "port":"TCP:80",
                "links":[]
        }
],
"servicesMap":{
        "service4":["cluster2"],
        "service3":["cluster1"]
}

}
```

In the Code Box C.3 is present the JSON object for the request to apply the policy 6.3 in Chapter 6.

Code Box C.3. Example of a Policy Requirement request

```

{
"apiVersion":"cilium.io/v2",
"kind":"CiliumNetworkPolicy",
        "metadata":{
        "annotations":null,
        "clusterName":null,
        "creationTimestamp":null,"
        deletionGracePeriodSeconds":null,
        "deletionTimestamp":null,
        "finalizers":null,
        "generateName":null,
        "generation":null,
        "labels":null,
        "managedFields":null,
        "name":"allow-domain1",
        "namespace":null,
        "ownerReferences":null,
        "resourceVersion":null,
        "selfLink":null,
        "uid":null
},
"spec":{
        "description":null,
        "egress":[
                {
                        "toCIDR":null,
                        "toCIDRSet":null,
                        "toEndpoints":[
                                {
                                  "matchExpressions":null,
                              "matchLabels":
                                     {
                                             "service":"*",
                                             "cluster":"*",
                                             "domain":"domain1"
                                         }
                                }
                        ],
                        "toEntities":null,
                        "toFQDNs":null,
                        "toGroups":null,
                        "toPorts":null,
                        "toRequires":null,
                        "toServices":null
                                        }
        ],
        "egressDeny":null,
        "endpointSelector":{
                "matchExpressions":null,
                "matchLabels":{
                        "name":"client"
                }
        },
        "ingress":null,
        "ingressDeny":null,
        "labels":null,
        "nodeSelector":null
},
"specs":null,
"status":null
}
```

It is also possible to retrieve the policy class using the YAML file with the client create for the communication with the APIs of Multi Cluster Orchestrator (more information will be given in the section C.3).

Finally to delete the policy to check that also namespace and services created are deleted by the Multi Cluster Orchestrator, a DELETE request to ”*/API/domains/domain2/policies/allow-domain1/remove*” can be done.

C.2 Use Case 2

In the second use case, the initial setup is similar to the first use case but this time will be shown how to register a domain with only the name and add clusters and services after the creation of the domain.

First, in the Code Box C.4 it’s reported the JSON object for the registration of the *domain1* and one cluster making a request to the Multi Cluster Orchestrator at the ”*/API/domains/create*” path, services, and the other cluster will be added after using other paths.

Code Box C.4. DomainRequest for use case 2

```

{
"domain":{
        "name":"domain1",
        "links":[]
},
"clusters":[
        {
                "domain":null,
                "name":"cluster1",
                "cilium_cluster":cluster1,
                "context":"kind-cluster1",
                "links":[]
        }
],
"services":[]
"servicesMap":[]

}
```

After registering the domain, using the JSON object in the Code Box C.5 and making a request to "*/API/domains/domain1/clusters/create*" it is possible to create another cluster and add it to the domain.

Code Box C.5. Cluster creation request

```

{
        "domain":null,
        "name":"cluster2",
        "cilium_cluster":"cluster2",
        "context":"kind-cluster2",
        "links":[]
}
```

To add a service instead, the path to use is "*/API/domains/domain1/clusters/cluster2/services/create*" and the service's JSON object it's shown in the Code Box C.6.

Code Box C.6. Service creation request in a single cluster

```

{
        "name":"database",
        "domain":null,
        "namespace":"database-namespace",
        "selector":"name:database",
        "port":"TCP:80",
        "links":[]
}
```

To add services in both clusters instead it is also possible to make a request to "*/API/domains/domain1/services/create*". In the Code Box C.7 there is an example of a JSON object for the creation of service backend on both *cluster1* and *cluster2*.

Code Box C.7. Service creation request in a domain

```
{
"service":{
       "name":"backend",
       "domain":null,
       "namespace":"backend-namespace",
       "selector":"name:backend",
       "port":"TCP:80",
       "links":[]
},
"clusters":["cluster1","cluster2"]}
```

To delete the service *backend* as done in the use case 2 example, it is possible doing a DELETE request to the path "*/API/domains/domain1/services/backend /remove*", this will delete the service in both clusters and will also delete the policy.

C.3 Client

In this section will be described a client created to interact with the Multi Cluster Orchestrator API. In the main of the class there are all the operations to replicate the two use cases presented in the Chapter 6.

In the Client project is present an ApiClient Class that contains generic methods which can be used for all type of requests. An example is in the POST method in Code Box C.8.

Code Box C.8. postRequest method

```

 public <T> T postRequest(String path,Object
     request,Class<T> aClass){
WebClient.UriSpec<WebClient.RequestBodySpec> uriSpec =
    this.client.post();
WebClient.RequestBodySpec bodySpec = uriSpec.uri(path);
WebClient.RequestHeadersSpec<?> headersSpec =
    bodySpec.body(Mono.just(request), request.getClass());
Mono<T> res = headersSpec
               .header("Content-Type", "application/json")
               .accept(MediaType.APPLICATION_JSON,
                   MediaType.APPLICATION_XML)
               .acceptCharset(StandardCharsets.UTF_8)
               .ifNoneMatch("*")
               .ifModifiedSince(ZonedDateTime.now())
               .retrieve()
```

```
                .bodyToMono(aClass);
return res.block();
}
```

This method requires the path where the POST request has to be performed, an Object Class which is the request, and a Class¡T¿ object that correspond to the response object class from the server.
Another class called MultiClusterOrchestratorAPI was created and uses the generic methods of ApiClient to perform all the REST requests present in the Server. This class contains methods like createDomain, createPolicyinCluster etc. which can be easily extended as soon as new server's REST API are created.

Finally, to help domain managers to write requirements with the simple YAML language and do not use a Class full of fields like the V2OrchestratorNetworkPolicy or the V2CiliumNetworkPolicy classes, another class was created called yaml class which translates a YAML file in the corresponding Java Class.
An example is given in the Code Box C.9, which translates a YAML file in the V2OrchestratorNetworkPolicy class.

Code Box C.9. Example of creation of a V2OrchestratorNetworkPolicy class from a yaml file

```

 yaml.addModelMap("MultiCLusterOrchestrator.io", "v2",
     "OrchestratorNetworkPolicy",
     "orchestratornetworkpolicies",
     V2OrchestratorNetworkPolicy.class);

File file = new File("/path/to/the/YAML file/YAMLfile.yaml");
V2OrchestratorNetworkPolicy yamlNp
    =(V2OrchestratorNetworkPolicy) yaml.load(file);
```