POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Enabling Service Mesh in a Multi-Cloud Environment

Supervisor	Candidate
Prof. Fulvio RISSO	Giandonato FARINA

A.Y. 2020-2021

Summary

In the last two decades the cloud has gained a lot of importance, indeed the current trend is to engineer the new web applications to be cloud native, thus to be split up in loosely-coupled micro-services, each one containerized and deployed as a part of a bigger application. The use of containers allows to cut oneself off the hosting physical hardware and operating system, letting to focus on the main purposes of a web application: to be widespread and high-available. The cloud allows to achieve this goal, by gathering the infrastructure control under the cloud provider tenants and implementing the IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) paradigms: the computational, networking and storage resources are provided on demand to the cloud provider's customers as if they were services. A technology that broke through the cloud market is Kubernetes, a project kicked off by Google in 2014 that allows to automate deployment, scaling, and management of containerized applications. Beside the cloud, in recent years the edge computing has gained a lot of importance: it is a distributed computing paradigm that brings the computational and storage resources close to the final user, in order to improve the QoS standards in terms of latency and bandwidth.

This thesis is involved in Liqo project which has the goal of creating a federation of Kubernetes clusters that cooperate at the edge of the network: many different tenants are connected together to cooperate in creating a federation of clusters with computational, storage and networking resources shared between them. In this scenario every tenant can make its own resource cluster available to the federation by sharing or leasing them out in a federated environment.

In this context would be useful to enable service mesh, in order to obtain its features like observability, security, better load balancing and so on. However, there are several issues in using service mesh on Liqo architecture. This work analyzes issues of the main service mesh available and proposes a prototype solution.

Acknowledgements

Un ringraziamento speciale va alla mia famiglia,
in particolare ai miei genitori che,
durante questo percorso, mi hanno sempre sostenuto
e aiutato a superare le difficoltà.

Table of Contents

List of Figures VIII

Acronyms XI

1 Introduction 1
- 1.1 Why Service Mesh on Liqo . . . 1
- 1.2 Goal of the thesis . . . 2

2 Kubernetes 4
- 2.1 Kubernetes: a bit of history . . . 4
- 2.2 Applications deployment evolution . . . 5
- 2.3 Container orchestrators . . . 6
- 2.4 Kubernetes architecture . . . 7
  - 2.4.1 Control plane components . . . 8
  - 2.4.2 Node components . . . 10
- 2.5 Kubernetes objects . . . 11
  - 2.5.1 Label & Selector . . . 12
  - 2.5.2 Namespace . . . 12
  - 2.5.3 Pod . . . 12
  - 2.5.4 ReplicaSet . . . 13
  - 2.5.5 Deployment . . . 13
  - 2.5.6 Service . . . 14
- 2.6 RBAC . . . 15
  - 2.6.1 ServiceAccount . . . 16
  - 2.6.2 Role and ClusterRole . . . 16
  - 2.6.3 RoleBinding and ClusterRoleBinding . . . 17
- 2.7 Virtual-Kubelet . . . 17
- 2.8 Kubebuilder . . . 18

3 Liqo 20
3.1 Liqo Idea . . . 20
3.2 Concepts . . . 21
3.2.1 Discovery . . . 21
3.2.2 Peering . . . 22
3.2.3 Networking . . . 23
3.2.4 Offloading . . . 25
3.3 Liqoctl . . . 26

4 Service Mesh 28
4.1 Issues of micro-services approach . . . 28
4.2 Features . . . 28
4.3 How it works . . . 29
4.4 Linkerd . . . 30
4.4.1 A bit of history . . . 30
4.4.2 Architecture . . . 31
4.4.3 Extensions . . . 34
4.5 Istio . . . 36
4.5.1 Concepts . . . 37
4.5.2 Architecture . . . 39
4.5.3 Multi-cluster . . . 41

5 Service Mesh on Liqo architecture: state of art 43
5.1 Linkerd Analysis . . . 43
5.1.1 Namespace Reflection . . . 44
5.1.2 mTLS Certificates . . . 44
5.1.3 Service Profiles . . . 45
5.1.4 Endpoints . . . 46
5.2 Istio Analysis . . . 49
5.2.1 Namespace Reflection . . . 49
5.2.2 Downward API . . . 50
5.2.3 Authentication . . . 51
5.2.4 Endpoints . . . 53

6 Integrating Linkerd and Liqo 54
6.1 Service Profiles . . . 54
6.2 Endpoints . . . 57
6.3 mTLS Certificate . . . 65
6.4 Namespace Reflection . . . 66

7 Evaluation 67
7.1 Performance analysis . . . 67

8 Conclusion and future work 73
8.1 Future works 73

Bibliography 75

List of Figures

2.1 Evolution in applications deployment. 5
2.2 Container orchestrators use [9]. 7
2.3 Kubernetes architecture 8
2.4 Kubernetes master and worker nodes [1]. 11
2.5 Kubernetes pods [1] 13
2.6 Kubernetes Services [1] 15
2.7 Virtual-Kubelet concept [2] 18

3.1 No Change in Kubernetes API 21
3.2 Discovery 22
3.3 Peering 23
3.4 Network Architecture 24
3.5 Pod offloaded on the foreign cluster 26

4.1 Service Mesh Architecture 30
4.2 Linkerd Timeline [15] 30
4.3 Linkerd Architecture [16] 32
4.4 Linkerd Dashboard [20] 35
4.5 Linkerd Multicluster Overview [21] 35
4.6 Linkerd Buoyant Dashboard [20] 36
4.7 Istio Security Architecture [25] 38
4.8 Istio Architecture [28] 40

5.1 Service Profile Example 46
5.2 Offloaded proxy tries to contact offloaded endpoint 47
5.3 Offloaded proxy tries to contact home endpoint 48
5.4 Endpoint resolution failure in a three cluster scenario 48

6.1 Solution to Service Profiles issue 55
6.2 Solution to Endpoints issue 57

7.1 Testing environment 68

7.2 P95 Latency in first testing case 69
7.3 Error Rate in first testing case 69
7.4 P95 Latency in second testing case 70
7.5 Error Rate in second testing case 70
7.6 P95 Latency in third testing case 71
7.7 Error Rate in third testing case 72

Acronyms

DNS
Domain Name System

mDNS
Multicast DNS

HTTP
HyperText Transfer Protocol

gRPC
gRPC Remote Procedure Calls

API
Application Programming Interface

TLS
Transport Layer Security

TCP
Transmission Control Protocol

IP
Internet Protocol

NAT
Network Address Translation

LAN
Local Area Network

WAN

Wide Area Network

K8s

Kubernetes

CRD

Custom Resource Definition

CR

Custom Resource

Chapter 1

Introduction

It the last decades the ICT world has seen an incredible innovation with the introduction of virtualization first, then with containerization and finally with orchestration. In this last field, one of the main actors is Kubernetes, an open-source system for managing containerized applications in a clustered environment. The spread of Kubernetes is rapidly increasing, even in small and medium companies that need to execute their jobs in smaller clusters that may not have enough resources to deal with peaks of traffic. In this context is born the Liqo project which enables the creation of a multi-cluster environment with liquid resource sharing between different clusters.

Recently, a new technology that is being developed is the Service Mesh. It provides features like observability, reliability, security and even a better Load Balancing than Kubernetes ones, which considers also new communication technologies for distributed applications, as gRPC. Solutions like Istio and Linkerd, the main open-source service mesh on the market, provide these features.

The idea behind this thesis is to combine the flexibility of Liqo multi-cloud environment with the features of the service mesh. This will enable the creation of mesh that can expand dynamically on other clusters, without forcing any installation other than Liqo. Service mesh developers create some solutions to extend their products on multiple clusters, even though they are less flexible than Liqo and require more complex setups.

1.1 Why Service Mesh on Liqo

Liqo suffers an issue with micro-services application which uses gRPC protocol for communications due to a limitation of the Kubernetes default load balancer. In fact, it does not provide load balancing of gRPC traffic and this is problematic when application scales up: the original replica receives all the traffic directed to

the service while other replicas have no load. This brings the first replica to failure and all the traffic is redirected to the new replica, which in the same way will fail due to overhead and so on. The application became unusable because of high latency and error rate.

The solution to this issue is `Service Mesh`. However, Liqo does not support them and so it is necessary design a solution and implement it. Moreover, enabling service mesh on Liqo environment will bring not only gRPC load balancing, but all the advantages of the service mesh, like observability, security and reliability.

Since solutions for multi-cluster service mesh already exist, the question is why not use them directly instead of implement a new one based on Liqo? The answer is simple: Liqo offers a simplicity of usage and, mostly, a dynamism in peering with other clusters and in offloading pods.

In fact, using solutions by service mesh providers implies users have to deploy micro-services on clusters separately and then connect the clusters. This can be a limitation when managing the application (e.g. move a service from a cluster to another one, scale up/down services, etc). In addition, these solutions may have some requirements, for example create a trust between clusters, share API server credentials manually and so on.

Liqo automatize all these processes, using a simple CLI to install it and peer clusters and manages pod offloading, very useful in case of loss of connectivity, where pods are rescheduled on local nodes, or in autoscaling, where pods may be scheduled on remote clusters.

1.2 Goal of the thesis

The goal of the thesis is to develop a prototype of service mesh which works in the Liqo multi-cluster environment. Obviously, the starting point for this fork is represented by the existing service mesh, in particular Linkerd and Istio which are the main-open source solutions on the market.

To achieve our result, we need first to understand why these service mesh do not work in Liqo environment and find some solutions to overcame issues found. In fact, the first part of this work is dedicated to an analysis of these problems and limitations they introduce.

Next we choose one technology between the analyzed as basis of the prototype and we design some modifications, both on it and on Liqo, necessary to make service mesh working on our multi-cluster environment.

After implementing them, we evaluate the result by comparing it with existing solutions, like multi-cluster proposed by service mesh developer, and the current version of Liqo, so that we can confirm the improvements on performance.

We will analyze all this work, starting from the technologies involved, as follows:

- **Chapter 2** presents Kubernetes, its architecture and concepts.
- **Chapter 3** presents Liqo, its architecture and concepts.
- **Chapter 4** describes what are Service Mesh and presents Linkerd and Istio.
- **Chapter 5** presents the analysis of issues and limitations of Linkerd and Istio on Liqo environment.
- **Chapter 6** describes how limitations are overcame in a prototype of Linkerd service mesh which works on Liqo.
- **Chapter 7** analyses performance of the prototype and compares it to other solutions.
- **Chapter 8** summarizes results achieved and describes possible future works.

Chapter 2

Kubernetes

In this chapter we present Kubernetes, the technology behind for all the work exposed in this thesis. In particular, we analyze its architecture, history and evolution through time. Since Kubernetes (often shortened as K8s) is a huge framework, a deep examination of it would require much more time and discussion and so here we limit to a description of its main concepts and components. To know more about this technology we recommend to consult the official documentation [1].

Moreover, in this chapter we introduce other technologies and tools involved in this project, in particular **Virtual-Kubelet** [2], which provides the possibility to create virtual nodes with a specific behaviour, and **Kubebuilder** [3], a tool that allow us to build custom resources.

2.1 Kubernetes: a bit of history

Around 2004, Google created the **Borg** [4] system, a small project with less than 5 people initially working on it. The project was developed as a collaboration with a new version of Google's search engine. Borg was a large-scale internal cluster management system, which "ran hundreds of thousands of jobs, from many thousands of different applications, across many clusters, each with up to tens of thousands of machines" [4].

In 2013 Google announced **Omega** [5], a flexible and scalable scheduler for large compute clusters. Omega provided a "parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, in order to achieve both implementation extensibility and performance scalability".

In the middle of 2014, Google presented **Kubernetes** as on open-source version of Borg. Kubernetes was created by Joe Beda, Brendan Burns, and Craig McLuckie, and other engineers at Google. Its development and design were heavily

influenced by Borg and many of its initial contributors previously used to work on it. The original Borg project was written in C++, whereas the language chosen for Kubernetes was **Go**, developed by Google itself.

In 2015 Kubernetes v1.0 was released. Along with the release, Google set up a partnership with the Linux Foundation to form the **Cloud Native Computing Foundation** (CNCF) [6]. Since then, Kubernetes has significantly grown, achieving the CNCF graduated status and being adopted by nearly every big company. Nowadays it has become the de-facto standard for container orchestration [7, 8].

2.2 Applications deployment evolution

Kubernetes is a portable, extensible, open-source platform for running and coordinating containerized applications across a cluster of machines. It is designed to completely manage the life cycle of applications and services using methods that provide consistency, scalability, and high availability.

What does "containerized applications" means? As illustrated in figure 2.1, in the last decade the deployment of applications has seen significant changes.

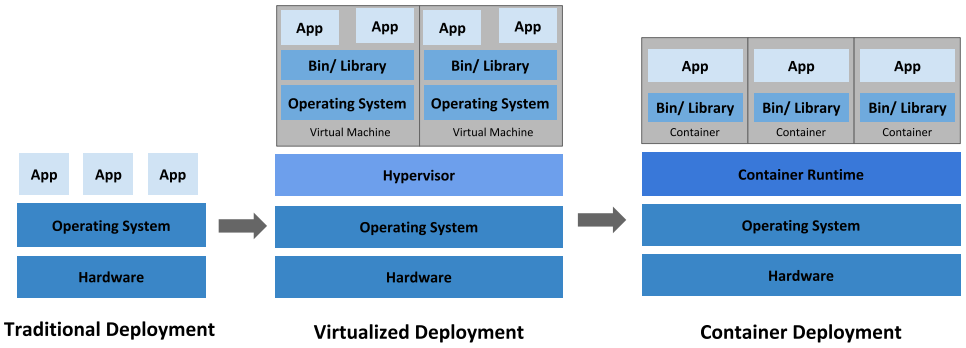

Figure 2.1: Evolution in applications deployment.

Traditionally, organizations used to run their applications on physical servers. One of the problems of this approach was that resource boundaries between applications could not be applied in a physical server, leading to resource allocation issues. For example, if multiple applications run on a physical server, one of them could take up most of the resources, and as a result, the other applications would starve. A possibility to solve this problem would be to run each application on a different physical server, but clearly it is not feasible: the solution could not scale, would lead to resources under-utilization and would be very expensive for organizations to maintain many physical servers.

The first real solution has been **virtualization**. Virtualization allows to run

multiple Virtual Machines on a single physical server. It grants isolation of the applications between VMs providing a high level of security, as the information of one application cannot be freely accessed by another application. Virtualization enables better utilization of resources in a physical server, improves scalability, because an application can be added or updated very easily, reduces hardware costs, and much more. With virtualization it is possible to group together a set of physical resources and expose it as a cluster of disposable virtual machines. Isolation certainly brings many advantages, but it requires a quite 'heavy' overhead: each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

A second solution which has been proposed recently is **containerization**. Containers are similar to VMs, but they share the operating system with the host machine, relaxing isolation properties. Therefore, containers are considered a lightweight form of virtualization. Similarly to a VM, a container has its own filesystem, CPU, memory, process space etc. One of the key features of containers is that they are portable: as they are decoupled from the underlying infrastructure, they are totally portable across clouds and OS distributions. This property is particularly relevant nowadays with cloud computing: a container can be easily moved across different machines. Moreover, being "lightweight", containers are much faster than virtual machines: they can be booted, started, run and stopped with little effort and in a short time.

2.3 Container orchestrators

When hundreds or thousands of containers are created, the need of a way to manage them becomes essential; container orchestrators serve this purpose. A container orchestrator is a system designed to easily manage complex containerization deployments across multiple machines from one central location. As depicted in figure 2.2, Kubernetes is by far the most used container orchestrator. We provide a description of such system in the following.

Kubernetes provides many services, including:

- **Service discovery and load balancing** A container can be exposed using the DNS name or using its own IP address. If traffic to a container is high, a load balancer able to distribute the network traffic is provided.
- **Storage orchestration** A storage system can be automatically mounted, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** The desired state for the deployed containers can be described, and the actual state can be changed to the desired state at a controlled rate. For example, it is possible to automate the

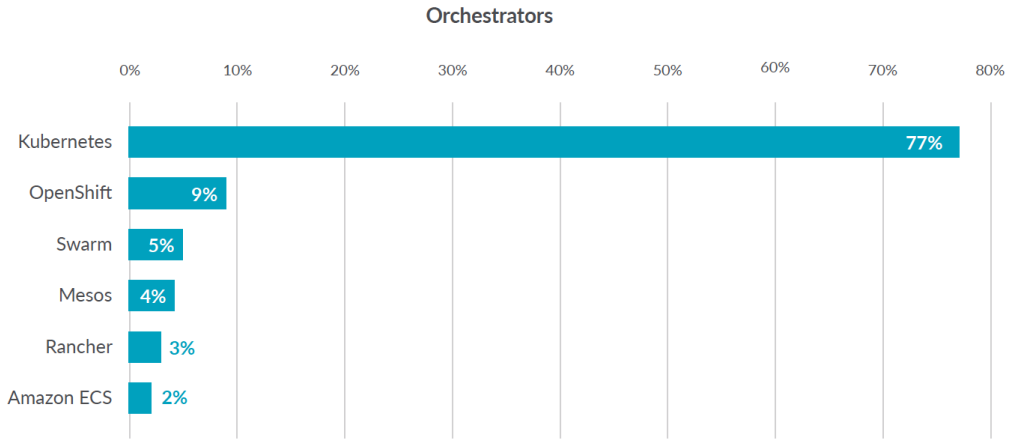

Figure 2.2: Container orchestrators use [9].

creation of new containers of a deployment, remove existing containers and adopt all their resources to the new container.

- **Automatic bin packing** Kubernetes is provided with a cluster of nodes that can be used to run containerized tasks. It is possible to set how much CPU and memory (RAM) each container needs, and automatically the containers are sized to fit in the nodes to make the best use of the resources.
- **Secret and configuration management** It is possible to store and manage sensitive information in Kubernetes, such as passwords, OAuth tokens, and SSH keys. It is possible to deploy and update secrets and application configuration without rebuilding the container images, and without exposing secrets in the stack configuration.

2.4 Kubernetes architecture

When Kubernetes is deployed, a cluster is created. A Kubernetes cluster consists of a set of machines, called **nodes**, that run containerized applications. At least one of the nodes hosts the control plane and is called **master**. Its role is to manage the cluster and expose an interface to the user. The **worker** node(s) host the **pods** that are the components of the application. The master manages the worker nodes and the pods in the cluster. In production environments, the control plane usually runs across multiple machines and a cluster runs on multiple nodes, providing fault-tolerance and high availability.

Figure 2.3 shows the diagram of a Kubernetes cluster with all the components linked together.

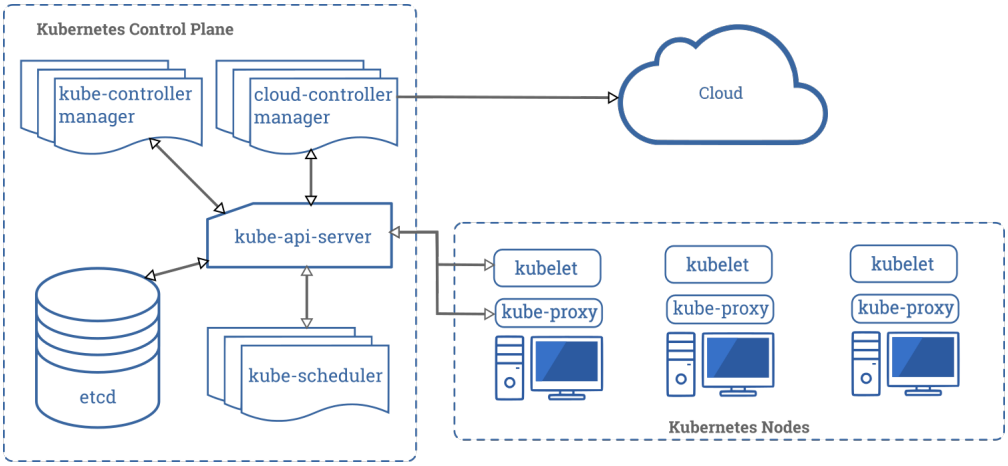

Figure 2.3: Kubernetes architecture

2.4.1 Control plane components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod). Although they can be run on any machine in the cluster, for simplicity, they are typically executed all together on the same machine, which does not run user containers.

API server

The API server is the component of the Kubernetes control plane that exposes the Kubernetes REST API, and constitites the front end for the Kubernetes control plane. Its function is to intercept REST request, validate and process them. The main implementation of a Kubernetes API server is `kube-apiserver`. It is designed to scale horizontally, which means it scales by deploying more instances. Moreover, it can be easily redounded to run several instances of it and balance traffic among them.

etcd

`etcd` is a distributed, consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. It is based on the Raft consensus algorithm [10], which allows different machines to work as a coherent group and survive to the breakdown of one of its members. `etcd` can be stacked in the master node or external, installed on dedicated host. Only the API server can communicate with it.

Scheduler

The scheduler is the control plane component responsible of assigning the pods to the nodes. The one provided by Kubernetes is called `kube-scheduler`, but it can be customized by adding new schedulers and indicating in the pods to use them. `kube-scheduler` watches for newly created pods not assigned to a node yet, and selects one for them to run on. To make its decisions, it considers singular and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

kube-controller-manager

Component that runs controller processes. It continuously compares the desired state of the cluster (given by the objects specifications) with the current one (read from `etcd`). Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process. These controllers include:

- Node Controller: responsible for noticing and reacting when nodes go down.
- Replication Controller: in charge of maintaining the correct number of pods for every replica object in the system.
- Endpoints Controller: populates the Endpoint objects (which links Services and Pods).
- Service Account & Token Controllers: create default accounts and API access tokens for new namespaces.

cloud-controller-manager

This component runs controllers that interact with the underlying cloud providers. The `cloud-controller-manager` binary is a beta feature introduced in Kubernetes 1.6. It only runs cloud-provider-specific controller loops. You can disable these controller loops in the `kube-controller-manager`.

`cloud-controller-manager` allows the cloud vendor's code and the Kubernetes code to evolve independently of each other. In prior releases, the core Kubernetes code was dependent upon cloud-provider-specific code for functionality. In future releases, code specific to cloud vendors should be maintained by the cloud vendor themselves, and linked to `cloud-controller-manager` while running Kubernetes. Some examples of controllers with cloud provider dependencies are:

- Node Controller: checks the cloud provider to update or delete Kubernetes nodes using cloud APIs.

- Route Controller: responsible for setting up network routes in the cloud infrastructure.
- Service Controller: for creating, updating and deleting cloud provider load balancers.
- Volume Controller: creates, attaches, and mounts volumes, interacting with the cloud provider to orchestrate them.

2.4.2 Node components

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

Container Runtime

The `container runtime` is the software that is responsible for running containers. Kubernetes supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

kubelet

An agent that runs on each node in the cluster, making sure that containers are running in a pod. The `kubelet` receives from the API server the specifications of the Pods and interacts with the `container runtime` to run them, monitoring their state and assuring that the containers are running and healthy. The connection with the `container runtime` is established through the Container Runtime Interface and is based on gRPC.

kube-proxy

`kube-proxy` is a network agent that runs on each node in your cluster, implementing part of the Kubernetes Service concept. It maintains network rules on nodes, which allow network communication to your Pods from inside or outside of the cluster. If the operating system is providing a packet filtering layer, `kube-proxy` uses it, otherwise it forwards the traffic itself.

Addons

Features and functionalities not yet available natively in Kubernetes, but implemented by third parties pods. Some examples are DNS, dashboard (a web gui), monitoring and logging.

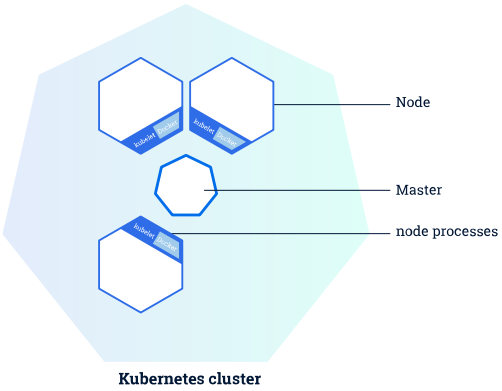

Figure 2.4: Kubernetes master and worker nodes [1].

2.5 Kubernetes objects

Kubernetes defines several types of objects, which constitutes its building blocks. Usually, a K8s resource object contains the following fields [**online:k8s_api_doc**]:

- `apiVersion`: the versioned schema of this representation of the object;
- `kind`: a string value representing the REST resource this object represents;
- `ObjectMeta`: metadata about the object, such as its name, annotations, labels etc.;
- `ResourceSpec`: defined by the user, it describes the desired state of the object;
- `ResourceStatus`: filled in by the server, it reports the current state of the resource.

The allowed operations on these resources are the typical CRUD actions:

- **Create**: create the resource in the storage backend; once a resource is created, the system applies the desired state.
- **Read**: comes with 3 variants
  - **Get**: retrieve a specific resource object by name;
  - **List**: retrieve all resource objects of a specific type within a namespace, and the results can be restricted to resources matching a selector query;
  - **Watch**: stream results for an object(s) as it is updated.

- **Update**: comes with 2 forms
  - **Replace**: replace the existing spec with the provided one;
  - **Patch**: apply a change to a specific field.
- **Delete**: delete a resource; depending on the specific resource, child objects may or may not be garbage collected by the server.

In the following we illustrate the main objects needed in the next chapters.

2.5.1 Label & Selector

Labels are key-value pairs attached to a K8s object and used to organize and mark a subset of objects. Selectors are the grouping primitives which allow to select a set of objects with the same label.

2.5.2 Namespace

Namespaces are virtual partitions of the cluster. By default, Kubernetes creates 4 Namespaces:

- **kube-system**: it contains objects created by K8s system, mainly control-plane agents;
- **default**: it contains objects and resources created by users and it is the one used by default;
- **kube-public**: readable by everyone (even not authenticated users), it is used for special purposes like exposing cluster public information;
- **kube-node-lease**: it maintains objects for heartbeat data from nodes.

It is a good practice to split the cluster into many Namespaces in order to better virtualize the cluster.

2.5.3 Pod

Pods are the basic processing units in Kubernetes. A pod is a logic collection of one or more containers which share the same network and storage, and are scheduled together on the same pod. Pods are ephemeral and have no auto-repair capacities: for this reason they are usually managed by a controller which handles replication, fault-tolerance, self-healing etc.

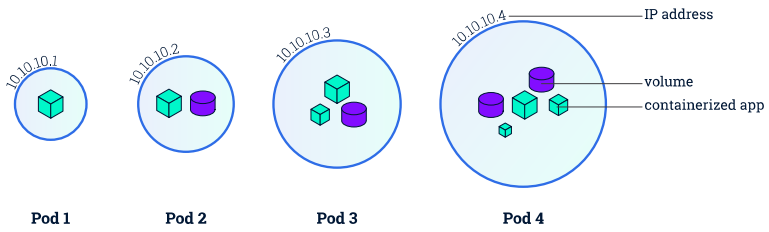

Figure 2.5: Kubernetes pods [1]

2.5.4 ReplicaSet

ReplicaSets control a set of pods allowing to scale the number of pods currently in execution. If a pod in the set is deleted, the ReplicaSet notices that the current number of replicas (read from the `Status`) is different from the desired one (specified in the `Spec`) and creates a new pod. Usually ReplicaSets are not used directly: a higher-level concept is provided by Kubernetes, called **Deployment**.

2.5.5 Deployment

Deployments manage the creation, update and deletion of pods. A Deployment automatically creates a ReplicaSet, which then creates the desired number of pods. For this reason an application is typically executed within a Deployment and not in a single pod. The listing is an example of deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
```

```
15          app: nginx
16      spec:
17        containers:
18        - name: nginx
19          image: nginx:1.7.9
20          ports:
21          - containerPort: 80
```

The code above allows to create a Deployment with name `nginx-deployment` and a label `app`, with value `nginx`. It creates three replicated pods and, as defined in the `selector` field, manages all the pods labelled as `app:nginx`. The template field shows the information of the created pods: they are labelled `app:nginx` and launch one container which runs the nginx DockerHub image at version 1.7.9 on port 80.

2.5.6 Service

A Service is an abstract way to expose an application running on a set of Pods as a network service. It can have different access scopes depending on its ServiceType:

- **ClusterIP**: Service accessible only from within the cluster, it is the default type;
- **NodePort**: exposes the Service on a static port of each Node's IP; the NodePort Service can be accessed, from outside the cluster, by contacting `<NodeIP>:<NodePort>`;
- **LoadBalancer**: exposes the Service externally using a cloud provider's load balancer;
- **ExternalName**: maps the Service to an external one so that local apps can access it.

The following Service is named `my-service` and redirects requests coming from TCP port 80 to port 9376 of any Pod with the `app=MyApp` label.

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: my-service
5 spec:
6   selector:
```

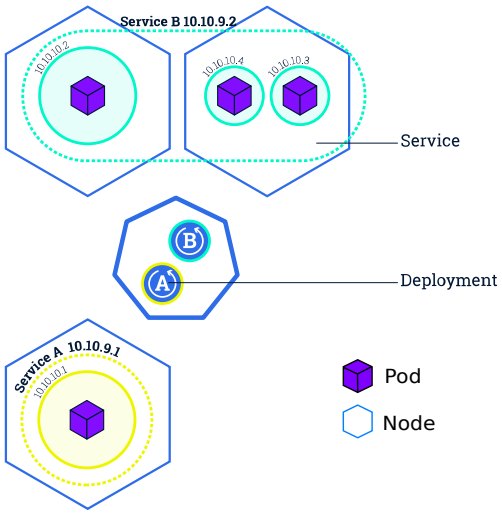

Figure 2.6: Kubernetes Services [1]

```
7       app: myApp
8     ports:
9       - protocol: TCP
10        port: 80
11        targetPort: 9376
```

2.6 RBAC

Kubernetes defines several APIs for the management of accesses. The Role-based access control (RBAC) is a method of regulating access to compute or network resources based on the roles of individual users.

The API group `rbac.authorization.k8s.io` defines four object types to define these permissions:

- **Role**: define rules valid for a specific namespace
- **ClusterRole**: define rules valid for all namespaces
- **RoleBinding**: link an identity to a set of rules in a specific namespace
- **ClusterRoleBinding**: link an identity to a set of roles in all namespaces

2.6.1 ServiceAccount

The ServiceAccount is a Kubernetes object in the `core/v1` API group that provides an identity for processes. When a new object of this kind is created, the API Server provide to it a new client certificate that will be used in all the future authentications.

2.6.2 Role and ClusterRole

The *Role* and the *ClusterRole* contains rules that represent a set of permissions. In these permissions there cannot be "deny" rules.

The only difference between of them is that the first sets the permissions within a particular namespace (the one which contains the resource), while the second is a non-namespaced resource and can be used in all the namespaces.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

In this example [1] we are creating a set of permissions in the *default* namespace that will grant access to get, watch, and list pod resources. We can have a similar example, but cluster-wide scoped, with the following ClusterRole.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not
   namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

2.6.3 RoleBinding and ClusterRoleBinding

The *RoleBinding* and the *ClusterRoleBinding* resources [1] grant the permissions defined in a *Role* or a *ClusterRole* to a given user, set of users or to a ServiceAccount. A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide.

```
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" namespace.
# You need to already have a Role named "pod-reader" in that namespace.
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

2.7 Virtual-Kubelet

Two Kubernetes-based tools which have been used during the development of this project are Virtual-Kubelet and Kubebuilder. Virtual Kubelet is an open source Kubernetes kubelet implementation that masquerades a cluster as a kubelet for the purposes of connecting Kubernetes to other APIs [2]. Virtual Kubelet is a Cloud Native Computing Foundation sandbox project.

The project offers a provider interface that developers need to implement in order to use it. The official documentation [2] says that “providers must provide the following functionality to be considered a supported integration with Virtual Kubelet:

1. Provides the back-end plumbing necessary to support the lifecycle management of pods, containers and supporting resources in the context of Kubernetes.

2. Conforms to the current API provided by Virtual Kubelet.

3. Does not have access to the Kubernetes API Server and has a well-defined callback mechanism for getting data like secrets or configmaps".

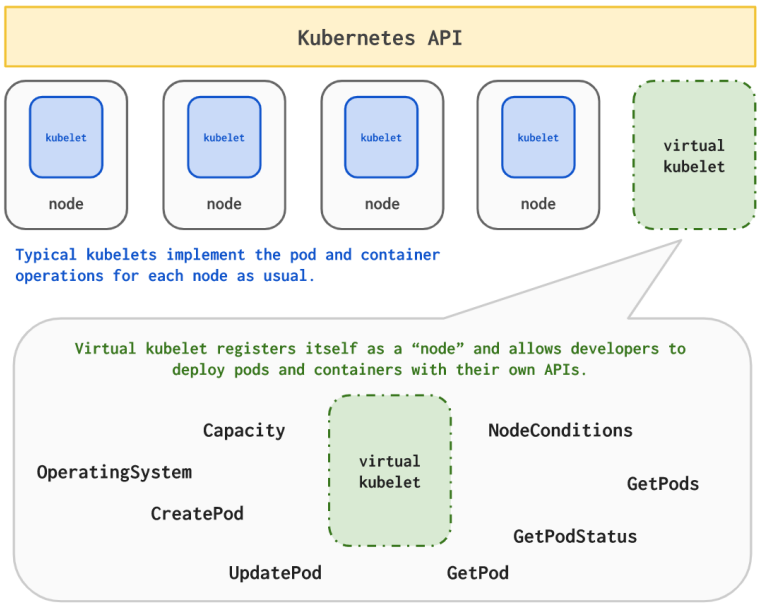

Figure 2.7: Virtual-Kubelet concept [2]

2.8 Kubebuilder

Kubebuilder is a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs) [3].

CustomResourceDefinition is an API resource offered by Kubernetes which allows to define Custom Resources (CRs) with a name and schema specified by the user. When a new CustomResourceDefinition is created, the Kubernetes API server creates a new RESTful resource path; the CRD can be either namespaced or cluster-scoped. The name of a CRD object must be a valid DNS subdomain name.

A **Custom Resource** is an endpoint in the Kubernetes API that is not available in a default Kubernetes installation and which frees users from writing their own API server to handle them [1]. On their own, custom resources simply let you store and retrieve structured data. In order to have a more powerful management, you

also need to provide a custom controller which executes a control loop over the custom resource it watches: this behaviour is called Operator pattern [11].

Kubebuilder helps a developer in defining his Custom Resource, taking automatically basic decisions and writing a lot of boilerplate code. These are the main actions operated by Kubebuilder [3]:

1. Create a new project directory.
2. Create one or more resource APIs as CRDs and then add fields to the resources.
3. Implement reconcile loops in controllers and watch additional resources.
4. Test by running against a cluster (self-installs CRDs and starts controllers automatically).
5. Update bootstrapped integration tests to test new fields and business logic.
6. Build and publish a container from the provided Dockerfile.

Chapter 3

Liqo

In this chapter we analyze Liqo architecture, starting from the idea behind it. Then we describe the main concepts of this open-source project in which this work is involved.

3.1 Liqo Idea

Liqo aims to create an opportunistic interconnection of multiple Kubernetes clusters allowing seamless resource and service sharing among them, creating an "endless Kubernetes ocean" where the user applications can be scheduled.

We can have a multiple cluster environment in a lot of different scenarios, both owned by the same entity or owned by different entities, These cluster may have underutilized resources because all these clusters have to have enough resources to deal with a peak of load by their own, but during the day they have moments of low load. In these moments they are wasting a part of their resources that can be available to be shared.

Liqo aims to extend the resources present in an already existent cluster using the ones currently non-occupied in neighbor clusters in an opportunistic way, so no peering and no sharing are definitive or not reversible, and it's always possible unpeer the two clusters in a simple way and return to the original state. When we extend a cluster with Liqo there is no change in the standard Kubernetes APIs, the ones described in Chapter 2 are still valid in the new environment, and the user applications have not to be changed in order to wirk with Liqo.

Liqo extends the cluster by adding a new virtual node for each remote peered cluster, creating in that way a "virtual big node" where the pods can be scheduled by the default Kubernetes scheduler with no change. The Kubernetes Pods that will be scheduled on this virtual node will be took by the Virtual Kubelet and offloaded to the remote cluster.

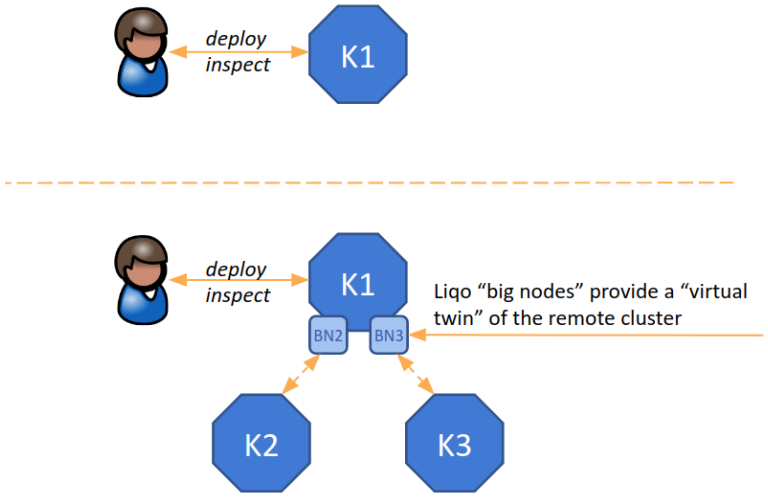

Figure 3.1: No Change in Kubernetes API

3.2 Concepts

Liqo enables resource sharing across Kubernetes clusters. In this section is described the architecture of Liqo, composed by four main concepts:

1. **Discovery:** the process in which Liqo detects other clusters to peer with;
2. **Peering:** how Liqo exchanges information with the other cluster and eventually establish an administrative interconnection with it.
3. **Networking:** how works the network interconnection with other clusters established by Liqo.
4. **Offloading:** propagation of resources and services from one cluster to peered ones, using the the big cluster / big node model based on the Virtual Kubelet.

More information can be found on the Liqo official documentation [12].

3.2.1 Discovery

The idea behind Discovery process is to have a way Liqo can discover other clusters, to obtain information about them and eventually to start a peering. Liqo can dynamically discover and add new clusters to the "Big Cluster" abstraction. These clusters can be discovered in a lot of different ways:

- **Manually**, for testing or not-yet-configured domains.
- **Automatically**, with DNS on selected domains or with mDNS on local area network.

Figure 3.2 represents some ways in which discovery process may occur. This process concludes with the creation of a `ForeignCluster` CR in the local cluster, after retrieving all the needed information from different data sources.

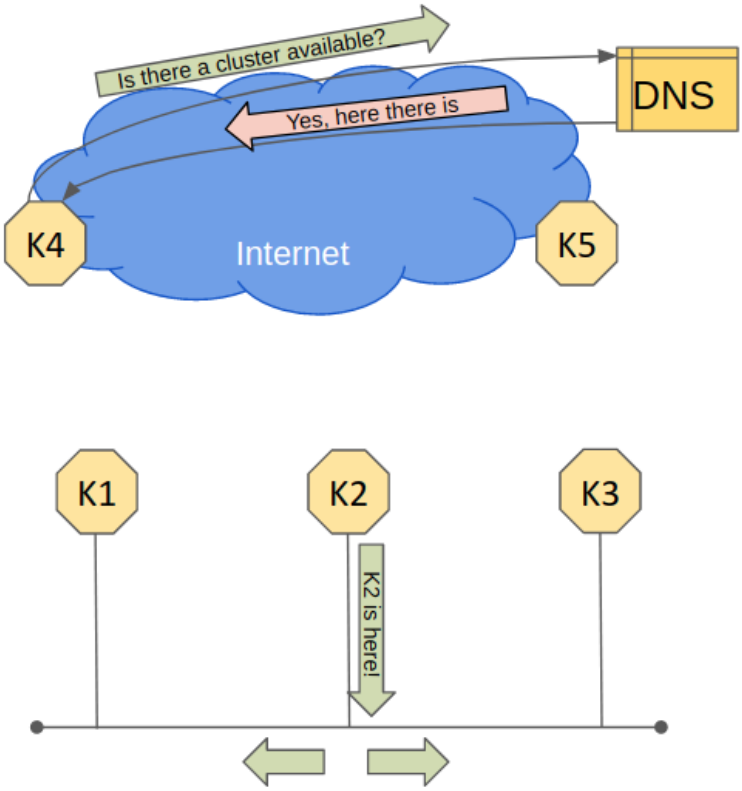

Figure 3.2: Discovery

3.2.2 Peering

The peering process allows to manage the control plane of the shared resources among different clusters. Liqo can dynamically peer different and administratively separate clusters with a policy-driven, voluntary, and direct relationship. This connection has to be established before sharing any resources. Periodic Advertisement messages embedding cluster capabilities are periodically sent to other peers; these messages are then used to build a local virtual-node where jobs can be scheduled: if

a job is assigned to a virtual-node, it will be actually sent to the respective foreign cluster. It has a peer-to-peer architecture, so no *master* cluster is involved.

The Liqo peering uses the information collected during the discovery phase to contact the remote cluster and checks that both clusters that will be part of the peering are available and have accepted the interconnection.

In Figure 3.3 are represented two peering scenario: in the first there are two clusters peered each other, while in the second we have a cluster which is peered with two other clusters. Notice that K3 and K5 do not have a connection between them: since Liqo is based on peer-to-peer connection, K4 is not a master cluster and so K3 does not know K5. Keep in mind this topology because we will return on it later on.

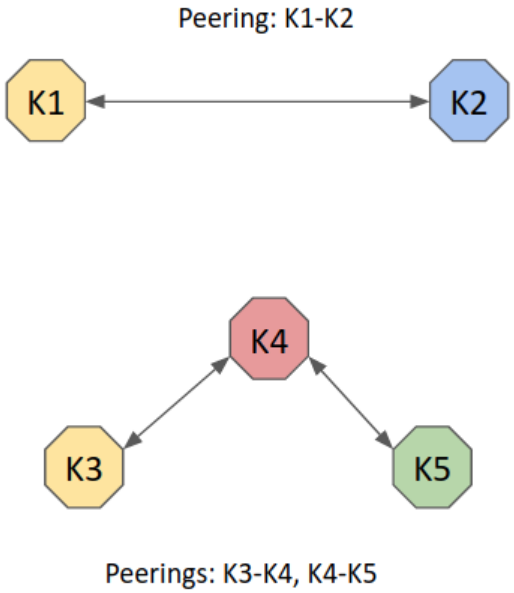

Figure 3.3: Peering

3.2.3 Networking

In Liqo the networking module is needed to connect the networks of the peered Kubernetes clusters. The goal is to extend the pod-to-pod communication to multiple clusters, using the peering information. The interconnection between clusters is dynamic, secure and done on top their existing network configuration. To avoid changes on them, Liqo network configuration is isolated as much as possible exploiting overlay networks, custom network namespaces, custom routing tables, and policy routing rules. All the process is done automatically, so no additional user input is required in the interconnection then those required at install time.

As you can see in the diagram in figure 3.4, the basic architecture of Liqo networking consists of several components needed to enable the connection across multiple clusters:

- **Liqo-Network-Manager:** manages the exchange of network configuration with remote clusters.
- **Liqo-Gateway:** manages life-cycle of secure tunnels to remote clusters.
- **Liqo-Route:** configures routes for cross-cluster traffic from the nodes to the active Liqo-Gateway.

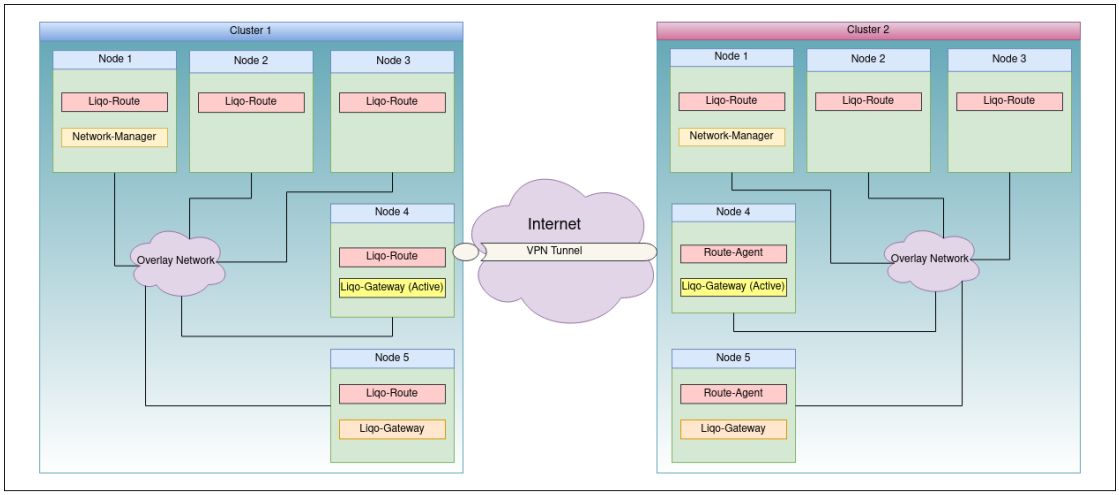

Figure 3.4: Network Architecture

The **Liqo Network Manager** has several tasks in enabling communication between peered clusters. First of all, it handles the creation of the `networkconfigs` CR, which contains the network configuration of the home cluster. This CR will be sent to remote clusters. Moreover, the Network Manager process the received NetworkConfigs, remappping addresses if necessary and creating the `tunnelendpoint` CR, which models the network interconnection between the two clusters.

The Network Manager embeds the **IPAM**, IP Address Management, which is the module in charge of:

1. Manage the networks used in the home cluster.
2. Translate IP addresses of offloaded pods in the corresponding ones visible in the home cluster.

3. Translate Endpoints IP addresses during reflection.

Liqo Gateway is the component responsible of the creation of secure tunnels from the home cluster to peered ones. It also inserts NAT rules for remote pods and External CIDR, which is the network used for communications between pods offloaded on different clusters. It is composed by several operators.

The **Liqo Route** component runs on each node of the cluster. It too is made up of several operators. Its tasks are:

- create a VXLAN interface on the host and add it to the overlay network.
- configure routes and policy routing rules to send cross cluster traffic to the active Liqo Gateway, if they are not running on the same node.
- configure route and policy routing rules to send cross cluster traffic to the `liqo-netns` namespace, if it is running on the same node of the active Liqo Gateway.

3.2.4 Offloading

After the peering a new virtual node is created in the home cluster. It is called *big node*, because it represents the resources shared by the remote cluster, while the home cluster became the *big cluster*, as it is a cluster whose resources are actually span across different physical clusters. The big nodes are equivalent to physical nodes, but instead of be manages by a kubelet, they are managed by the Virtual Kubelet, technology depicted in the previous chapter. When also network interconnection is done, it will set the big node as ready.

At this point, it is possible to offload pods on the remote cluster, which corresponds to schedule pod on the virtual node. When this occurs, the virtual kubelet creates a new pod on the remote cluster, which will be scheduled on a remote physical node managed by a physical kubelet. On the home cluster a shadow pod is kept, which status is aligned to the remote one.

In figure 3.5 is showed a scenario in which the yellow marked pod is offloaded on the foreign cluster and so there is a shadow pod on the home one. You can see also that all other endpoints are reflected on foreign cluster, so that yellow pod can contact pods on the home cluster.

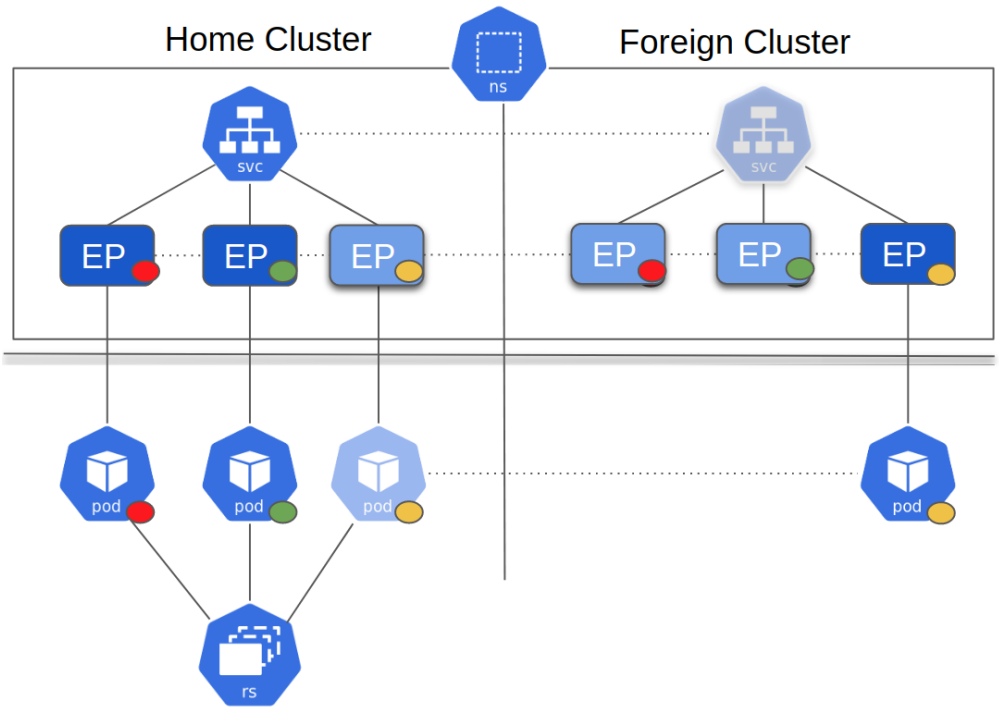

Figure 3.5: Pod offloaded on the foreign cluster

When offloading pods, the Virtual Kubelet needs to know in which remote namespace create the pods. So it maps each local namespace to a remote one, with a one-to-one correspondence, and reflection of resources can start.

The virtual node is created with a taint to avoid the offloading of all pods, but only ones we specify. In fact, there is a `MutatingWebhook` which add a toleration to pods whose namespace has the label `liqo.io/enabled="true"`. This label is the *quick offloading* approach, in which are used default parameters. There is also a *fine-grained* approach which rely on the `NamespaceOffloading` CR. More details can be found on the Liqo official documentation. [12].

3.3 Liqoctl

With the release of Liqo `0.3.0`, it was introduced `liqoctl`, he swiss-knife CLI tool to install and manage Liqo clusters. It speeds up all the procedures to install Liqo and peer clusters each other which otherwise would be complex: it automatically handles the required customizations for each supported providers (e.g., AWS, EKS, etc.) and generates the command to run on other clusters to peer with the local one.

Since `liqoctl` configures and installs Liqo using Helm3, it is possible for users

who need a custom configuration use the CLI as a a provider-specific values file generator and then install Liqo with Helm as usual.

Chapter 4

Service Mesh

This chapter presents Service Mesh technology, showing its concepts and features. This technology is at the base of this work, since we want to integrate them in the multi-cloud environment depicted in Chapter 3. Then, we presents also two different Service Mesh, Linkerd and Istio, which are the most popular and the ones we will analyze later on.

4.1 Issues of micro-services approach

In the last years we seen the transition from monolithic applications to micro-services ones, a new approach in which services are independent from each other and runs on separate containers. This approach brings lots of advantages because micro-services can be modified without involving the entire application: it becomes easy to re-deploy failed services, scale them or introduce a new version of a single single service.

However, this approach presents some problems in communications: an application can be made by hundreds of services, each of which may have more instances. It is easy to understand it could became challenging to monitor communications between them, guaranteeing security, reliability and correct functioning of the application. [13]

A Service Mesh is what we need to solve this issue and more. In the next section we will see how it works and which features provides.

4.2 Features

Service Mesh technology born prior to Kubernetes, but when it became de-facto standard for deploying micro-services applications interest on Service Mesh increased. As said in previous section, the main concern about micro-service approach

is network traffic between services and Service Mesh manages it in a graceful and scalable way, which cannot be obtained with a manual work in the long-run. It makes the communication between service over the network safe and reliable. [14]

The key feature of a service mesh are:

- **Reliability:** it manages the communication between services improving efficiency e reliability automating retries and backoff for failed requests.
- **Observability:** it enables observability for distributed micro-services system, providing an out-of-the-box monitoring and tracing tools, such as Prometheus or Jaeger in Kubernetes, which permit to visualize metrics and traffic flows of the application.
- **Security:** it automatically encrypts communications between services. Moreover, it can manage authentication and authorization, providing the possibility to configure policies which allow or deny communication between certain services.
- **Load Balancing:** it provides a smarter load balancing than the Kubernetes default one, which can balance also communication technology like gRPC.
- **Service Discovery:** it enables services to discover each other.
- **Simplify Deployment:** with service mesh it is possible to use more advanced deployment strategies like rolling updates, canary release, blue-green deployments and so on.

4.3 How it works

Service Mesh does not introduce its features in the business logic of the application, but it abstract the logic which rules the service-to-service communication at infrastructure level. This means observability, network and security policy are separated from the application logic.

To achieve this result it leverages on proxies injected at infrastructure level on the side of services, which is the reason why they are called *sidecar*. In Kubernetes, they are injected into pods as sidecar container. They intercepts all the traffic from and to the pod and routes it applying the configured policies. All proxies make up the **data plane**.

There is a second important part of the Service Mesh which is the **control plane**. It is the core of the mesh: it provides API and/or GUI to users, collects metrics, manages service discovery and communicates policies to proxies.

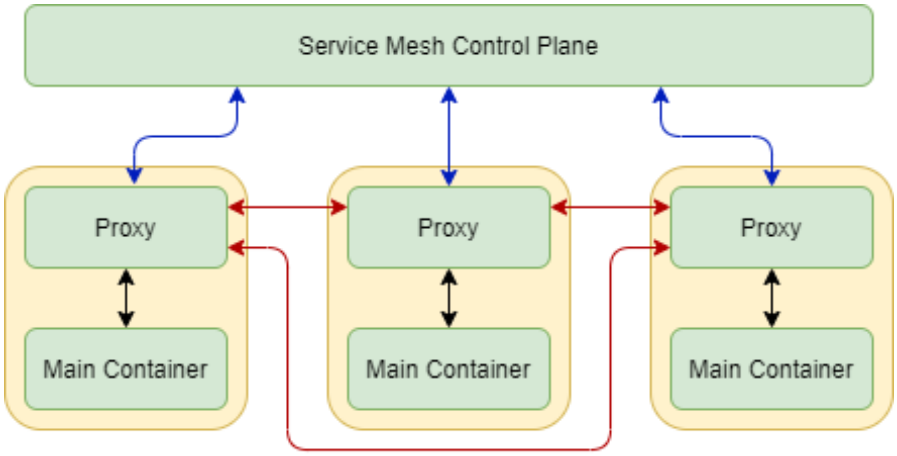

Figure 4.1: Service Mesh Architecture

In figure 4.1 is represented the architecture of a service mesh. In each pod there is a proxy which communicates with the control plane (blue lines) and with the other proxies (red lines). As you can see, the application containers communicate only with their sidecar proxy (black lines).

4.4 Linkerd

Linkerd is an ultralight, security-first service mesh for Kubernetes. It provides runtime debugging, observability, reliability and security without code changes in the application.

4.4.1 A bit of history

Figure 4.2: Linkerd Timeline [15]

In 2013 Twitter switched to a micro-services architecture and it had problems we discuss at the beginning of the chapter. So they started to develop a solution to

implement the features they needed.

Later this solution became open source and in February 2016 Linkerd 0.1 was released. In the same year its creators coined the *service mesh* term.

In January 2017 Linkerd became project a **Cloud Native Computing Foundation** (CNCF) project. In April of the same year **Linkerd v1.0** was released: it is JVM-based, written in Scala, and its main characteristics are:

- Highly configurable.
- Powerful and complex.
- Multi-platform.

However, this project presents some issues and limitation: in particular, Linkerd v1.0 has an high resource consumption and an high complexity, due to the many configurations possible. For these reasons it was not so adopted.

To solve these issues, developers decided to completely rewrite Linkerd using Rust language for the proxy, to maximize performance and efficiency, and Go language for the control plane, because it integrates well with Kubernetes. In February 2018 **Linkerd v2.0** was announced and released some month after. The design of this new version is focused on:

- Zero-config.
- Lightweight and simple.
- Kubernetes-first.

4.4.2 Architecture

As depicted before, service mesh consists of a control plane and a data plane. Linkerd is no exception.

The **control plane** is composed by a set of services running in a dedicated namespace. These services perform various tasks and drive the behaviour of the data plane.

The **data plane** is composed by transparent proxies running as sidecar containers next to each service instance. They handle all the traffic from and to the service and communicate with the control plane, sending telemetry data and receiving control signals. [16]

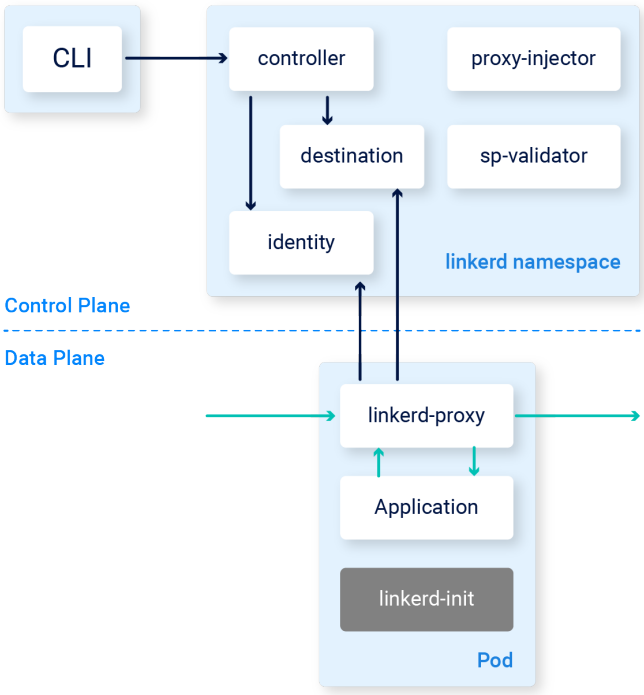

Figure 4.3: Linkerd Architecture [16]

In figure 4.3 you can see all the components of the Linkerd service mesh. Let's see them one by one.

CLI

Linkerd CLI runs outside the cluster and provides a way for users to interact with the control plane.

Controller

The `controller` is one of the control plane components, which run in a dedicated namespace (by default `linkerd`). The controller provides an API for the CLI to interface with.

Destination

The `destination` component provides to proxy two main information: service profile information, used for per-route metrics, retries and timeouts, and where to send requests.

Identity

The `identity` component is the TLS Certification Authority: it accepts CRSs from proxies and issues certificates, which are used for mTLS in proxy-to-proxy communications.

Proxy Injector

This components receives a webhook request every time a pod is created. If the pod has the annotation `linkerd.io/inject: enabled`, it modifies pod's specification adding everything needed for the service mesh.

Service Profile Validator

Shortened as `sp-validator`, this component validates new service profiles before they are saved.

Proxy

Linkerd does not use Envoy proxy, but relies on an its own one, written in Rust. It is an ultralight transparent micro-proxy which handles all the incoming and outgoing TCP traffic of the pod. This model adds the need functionality without any code change in the application. The proxy supports service discovery via DNS and the destination gRPC API.

The main features of the proxy are:

- Proxying for HTTP, HTTP/2 and arbitrary TCP protocols.
- Prometheus metrics export for HTTP and TCP traffic.
- WebSocket proxying.
- Latency-aware, layer-7 load balancing
- Layer-4 load balancing for non-HTTP traffic
- Mutual TLS
- Diagnostic tap API.

Linkerd Init Container

The `init-container` runs before all other containers in the pod and adds two `iptables` rules for intercepting traffic:

- Traffic sent to pod external IP address is forwarded to the port 4143 of the proxy.

- Traffic originated from the pod and directed to an external IP address is forwarded to the port 4140 of the proxy.

4.4.3 Extensions

Starting from version 2.10, Linkerd supports extensions which add new features to the base installations. Some of them are built-in, developed by Linkerd itself, while others are from third-parties [17].

Jaeger

This is a built-in extension which adds distributed tracing to Linkerd service mesh, using OpenTelemetry collector and Jaeger. Other than this extension, distributed tracing requires also code changes in the application and some configurations. However, many features often related to distributed tracing are available in Linkerd without changes, by installing the Viz extension [18].

Viz

Viz is the built-in extension which provides observability in Linkerd service mesh. Telemetry and monitoring features are automatic, they require only the installation of the `viz` extension, without any code change. These features include:

- Recording of top-line metrics for HTTP, HTTP/2 and gRPC traffic.

- Recording of TCP-level metrics for other TCP traffic.

- Reporting metrics per service, caller, route.

- Generating topology graphs.

- Live and on-demand request sampling.

This data can be visualized using the CLI or some tools provided by the extension itself. In fact, it includes the Linkerd dashboard, coupled with some pre-build Grafana dashboards and a Prometheus instance, which collects metrics and can be queried directly [19].

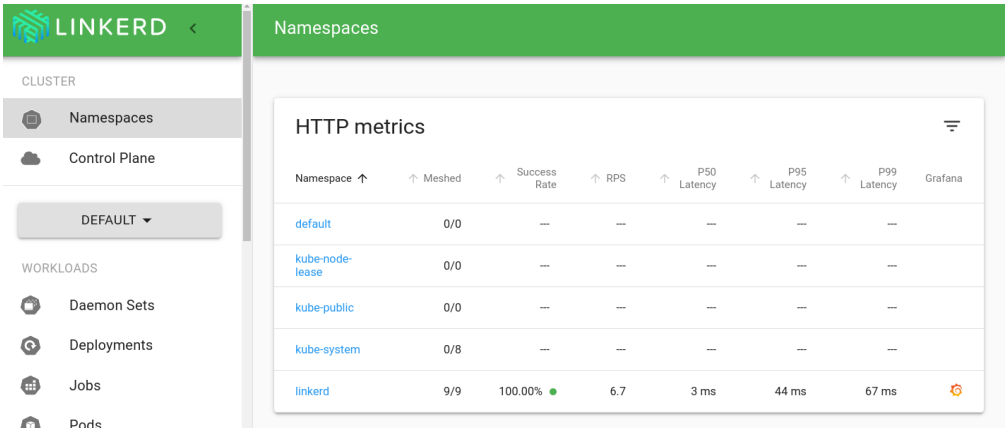

Figure 4.4: Linkerd Dashboard [20]

Multicluster

This is a built-in extension which enables multi-cluster support for Linkerd service mesh. The cross-cluster connection is completely transparent to the application. It works by *mirroring* service information between clusters. Remote services are represented as Kubernetes services, so application does not need to distinguish between local and remote services. Moreover, all Linkerd features apply uniformly in both cases.

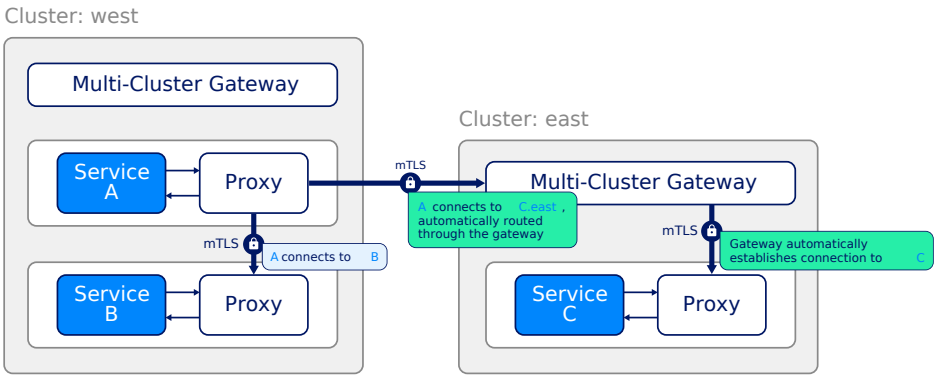

Figure 4.5: Linkerd Multicluster Overview [21]

Linkerd multi-cluster has two components:

- **Service Mirror:** it watches the a target clusters and mirrors services locally on a source cluster.
- **Gateway:** it provides a way for target clusters to receive traffic from source clusters.

Once the extension is installed, you need to label the services that have to been exported to other clusters [21].

SMI

The SMI extension adds the SMI (Service Mesh Interface) functionality in Linkerd-enabled Kubernetes clusters.

Tapshark

The `tapshark` extension provides a Wireshark inspired CLI for Linkerd Tap.

Buoyant

This extension connects a Linkerd-enabled cluster to Buoyant Cloud, which provides a global health dashboard for Linkerd.

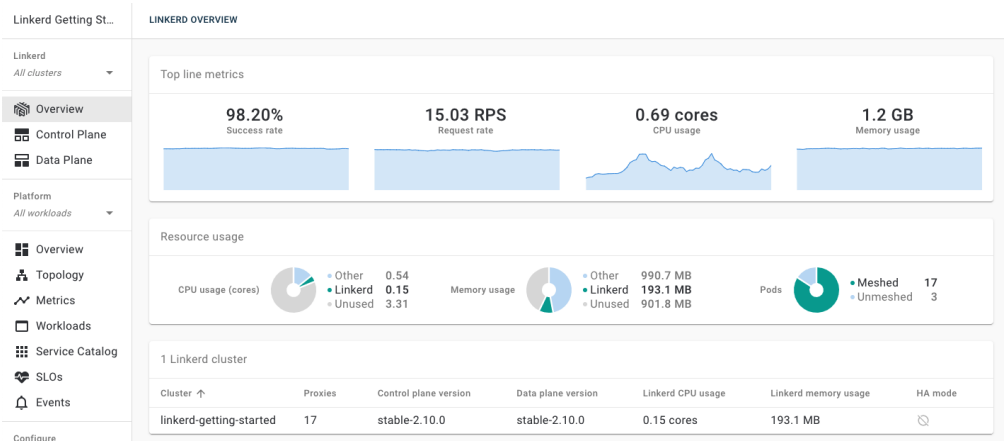

Figure 4.6: Linkerd Buoyant Dashboard [20]

4.5 Istio

Istio is an open-source service mesh, launched in 2016 by Google, alongside IBM, Lyft (the Envoy developers) and others. Its architecture is based on trusted service mesh software used internally by Google for years and then, as for Kubernetes, it was made public to reach many users as possible [22].

Istio name is a Greek word which means *sail*. In a first time it supported only Kubernetes-based deployment, but it was rapidly extended to other environments and now it is platform-independent [23].

Istio service mesh is transparent to applications and provides a uniform and efficient way to secure, connect and monitor services. Its features includes:

- Secure service-to-service communication.
- Load balancing for HTTP, gRPC, WebSocket and TCP traffic.
- Fine-grained traffic control.
- A pluggable policy layer and configuration API.
- Automatic metrics, logs and traces for all the traffic in the cluster.

Istio supports a wide range of deployment needs. Its control plane runs on Kubernetes and you can add in your mesh both applications deployed in Kubernetes or destinations outside of it, like VMs or other endpoints. You can install and configure Istio by yourself or leverage on products which manages Istio for you [24].

4.5.1 Concepts

Security

Using a micro-services approach leads to particular security needs, as protection against man-in-the-middle attacks, flexible access controls, auditing tools and mutual TLS. Istio provides a comprehensive security solution which mitigates both internal and external threats against your data endpoints, communication and platform. Its features includes strong identity, powerful policy, transparent TLS encryption and authentication, authorization and audit (AAA) [25].

Istio security is based on:

- **Security-by-default:** it does not require any changes to application code or infrastructure.
- **Defense in-depth:** it integrates with the other existing security systems, providing multiple defense layers.
- **Zero-trust networks:** security solutions are suited for distributed networks.

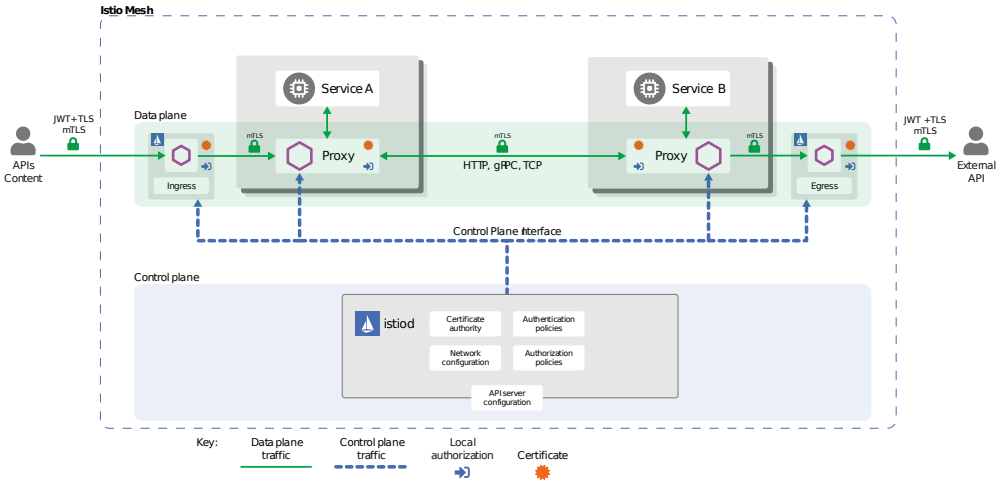

Figure 4.7: Istio Security Architecture [25]

Traffic Management

Since routing traffic affects performance, Istio provides traffic routing rules that let user easily controls the flow of traffic and API calls between services. With Istio is easier configure service-level properties (e.g. circuit breakers, timeouts, retires, etc) and complex deployment strategies, like A/B testing, canary deployment, etc. This traffic management model relies on Envoy proxies which direct and control traffic in your application without any code change [24].

Before direct traffic in the mesh, Istio needs to discover services connecting to a service discovery system. For example, in it runs on Kubernetes, it automatically detects services and their endpoints. In micro-services applications services often has more than one endpoint, so Istio provides a load balancing. By default, the Envoy proxy distributes traffic with a round-robin model. However, the default behaviour is far from what Istio can offer [26].

You can add your own configuration using the traffic management API. This can be done setting some CRDs:

- **Virtual services:** configure an ordered list of routing rule to control how Envoy proxies route requests for a service into the service mesh.
- **Destination rules:** configure policies that will be applied to a request after the routing rules in the virtual service are applied.
- **Gateways:** configure load balancing for Envoy proxies.
- **Service entries:** insert external dependencies of the mesh.

- **Sidecars:** configure the scope of the Envoy proxy in which apply features (e.g. namespace isolation)

Observability

Istio provides observability through a detailed telemetry generated for all communications within the mesh. This telemetry gives to operators the possibility to troubleshoot, maintain, optimize their applications and understand how services are interacting, both with other services and Istio components. All these features are available without requiring application changes [27].

Istio generates various types of telemetry to offer observability for the service mesh:

- **Metrics:** Istio generates metrics based on the four "golden signals", which are latency, traffic, errors and saturation. It also provides metrics for the control plane and a set of dashboards to visualize them.

- **Distributed Traces:** Istio generates distributed traces for each service so that can be understood call flows and service dependencies within the mesh.

- **Access Logs:** Istio can generate a full record of each request, including source and destination metadata.

4.5.2 Architecture

The Istio service mesh is composed by two logical part:

- **Data plane:** it consists in a set of Envoy proxies deployed as sidecars in Kubernetes or run alongside each service in other cases. They mediate and control all the communications between services and report metrics to the control plane.

- **Control plane:** it dynamically programs proxies to route traffic according the desired configuration

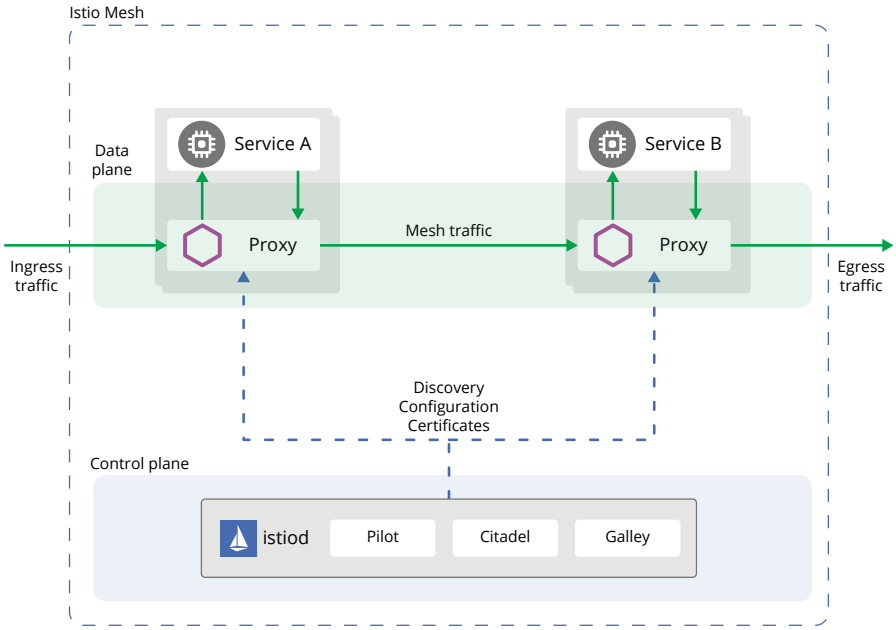

Figure 4.8: Istio Architecture [28]

In figure 4.8 there is a diagram that shows the different components of which each plane is made up. Let's describe them one by one.

Envoy

Envoy is an high-performance, written in C++, which mediate all the traffic from and to the service. Istio uses an extended version of it. Proxies are deployed as sidecars to services, adding them all the Envoy built-in features.

This model allows to add Istio features to existing deployments without requiring changes. Istio features enabled by Envoy proxies include:

- **Traffic control:** fine-grained traffic control for HTTP, gRPC, WebSockets and TCP traffic.
- **Network resiliency:** retries, failovers, circuit breakers and fault injection.
- **Security and authentication:** security policies, access control and rate limiting.

Istiod

Istiod is the combination of what used to be three components: Pilot, Gallery and Citadel. In fact, it provides all their features: service discovery, configuration and certificate management.

Istiod translates the high level routing rules into configuration for Envoy and propagates them to proxies at runtime. Pilot, embedded into it, abstracts platform-specific service discovery by synthesizing them into a standard format for Envoy proxies.

Istiod has a built-in identity and credential management which permit to havea strong service-to-service and end-user authentication. Using Istio operators can encrypt non protected traffic in the mash and enforce policies based on service identity, which are better than the one based on unstable layer 3/4 network identifiers. Moreover, Istio can be used to manage access control to services.

Istiod is also a **Certificate Authority** (CA) and generates certificates to allow secure mTLS communication in the data plane.

4.5.3 Multi-cluster

Istio implements a multi-cluster feature which allows to create a service mesh which spans on more than one cluster. It is quite flexible and supports several scenarios:

1. **Multi-Primary:** the mesh extends on clusters which are called Primary: this means they have a control plane. In this scenario, clusters are considered on the same network and so communication is pod-to-pod across cluster boundaries.

2. **Primary-Remote:** in this scenario one cluster is a Primary while the others are Remote, which means they connects to an external control plane, in this case the primary cluster one. Even in this scenario clusters are on the same networks and so communication is pod-to-pod across cluster boundaries.

3. **Multi-Primary on different networks:** it is the same case of the first scenario, but the clusters are on different networks. This means communication between pods is indirect and passes through a gateway. Each cluster must have one of them and it has to be reachable from the other clusters.

4. **Primary-Remote on different networks:** this scenario is like the second one, but clusters are on different networks and so it is required the indirect communication describe in the third scenario. Since the remote cluster does not have a control plane, pods on it has pass through the gateway to communicate with it.

However, Istio multi-cluster has some limitations: first of all you have to establish a trust between clusters. There are several options to do it. Additionally, you must ensure API server of each cluster is reachable from the others. Many providers expose it via a network load balancer, but in case of on-premise clusters you may

need a public IP address. Gateways can be an option to enable access to the API server.

More information can be found on the official Istio documentation [29].

Chapter 5

Service Mesh on Liqo architecture: state of art

After presenting the technologies involved, the next step is to put them together and analyze the behaviour in order to discover why service meshes do not work on Liqo architecture. In this chapter we will analyze both service meshes presented previously, Linkerd and Istio, discover all their limitations on the Liqo architecture and understand if one of them can bring to better results than the other.

The environment used to test the technologies is composed by two cluster both with Liqo installed and peered each other. From now we will call them:

- **Home Cluster:** the cluster on which the service mesh control plane is installed.
- **Remote Cluster:** cluster without service mesh control plane on which we will offload some pods.

On these clusters is deployed a microservices application, with some pods running on the home cluster and others offloaded on the remote one.

5.1 Linkerd Analysis

In this section we will focus on which are issues and limitations of a Linkerd service mesh running in a scenario like the one described at the beginning of the chapter. First of all, it is necessary to specify that the Linkerd version used is `2.10.x`, because results of this analysis can change with future releases of the software.

The analysis highlights four issue, two of which are greater, and we will see how to avoid them using some workarounds. However, these are not solutions because

they introduce serious limitations that impact on the flexibility that both Liqo and Linkerd pursue.

5.1.1 Namespace Reflection

Offloaded pods on the remote cluster needs a way to reach the control plane on the home cluster. Based on Liqo features, it is necessary to reflect the Linkerd namespace on the remote cluster using the `NamespaceOffloading` CRD. As seen before, reflection replicates some resources that may be useful to offloaded pods, such as services and endpoints.

However, it should be noted that the reflection must be done keeping the same namespace name of the home cluster. To understand why let imagine a scenario in which reflection is done with a different name, so that the Linkerd namespace name on the home cluster is different from to the one reflected on the remote cluster. Linkerd injects into meshed pods several environmental variables and some of them contain service addresses which are used by the proxy to contact the control plane. Those addresses are in the format `{service}.{namespace}.svc.{trust-domain}`. Now, it is clear that if the namespace changes proxy has a wrong address of Linkerd services and will not be able to contact them.

In addition, also the other meshed namespaces must be offloaded using the same name. In fact, the namespace is used in some other environmental variables injected by Linkerd that define the identity name of the pod. These variables includes the pod namespace, so if it changes authentication will fail.

5.1.2 mTLS Certificates

As explain in the dedicated chapter, Linkerd implements an automatic **mutual TLS** (mTLS) on communications between proxies and this implies that they need a certificate. One of the issues when offloading meshed pods with Liqo is about obtaining this certificate. Before explain that issue, we need to better understand the process by which the proxy requests and receives them.

In the Linkerd control plane there is a component called `identity` that is a **Certification Authority** (CA). This CA issues certificates to each proxy that expires in 24 hours and are automatically rotated. At startup, the proxy connects to this component and issues a **Certificate Signing Request** (CSR) which contains an initial certificate containing the pod's Service Account as identity and the actual service account token. The control plane validates the CSR checking the token and returns to the proxy the signed certificate which is used as both client and server one. When it expires, it is renewed with the same procedure. Proxy became ready only when has its own certificate.

Now it is clear that the service account token has a main role in this process.

However, this can be problematic when an offloaded pod tries to obtain a certificate: in our scenario, a pod offloaded on the remote cluster needs to use the token associated to its service account on the home cluster. Fortunately, Liqo reflects all secrets in the offloaded namespaces, including service account tokens. It also mounts them in the pods with the exception of the ones associated to the default service account. This means pods that use the default service account cannot receive their certificates, while there are not problems with pods that uses other service accounts.

Until Kubernetes 1.20 it is possible offload meshed pods assigning them a service account instead of use the default one. This behaviour introduces a minor limitation in the user experience. Since Kubernetes 1.21 this workaround is no longer available: in fact, the way in which service account token are mounted changes and Liqo does not support it.

5.1.3 Service Profiles

Major limitations in running a service mesh in our scenario are due to the service discovery process. This process involves the `destination` component of the Linkerd control plane and the proxy, which has the goal of discovering the target to which to forward the request. Service Discovery process is divided in two steps:

1. **Resolve Service Profile:** proxy obtains metadata about a service, like what is its authority, if it has traffic splits, how to manage retries and so on;
2. **Resolve Endpoints:** proxy obtains endpoints of the service, based on the resolved profile at the previous step;

We analyze the implications of the first step in our scenario in this subsection, while we will look at ones of the second step in the next subsection.

The first main limitation with Service Discovery is related to the way Service Profiles are resolved. Let follow the processing of a request by the proxy: first of all, the main container resolves the address and sends the request that is intercepted by the proxy. Now it needs the Service Profile, so it calls an API, exposed by `destination`, that given the IP address returns the profile.

If we imagine this process in our scenario, we can notice immediately a big problem: it is true that Liqo replicates services, but each cluster has its own Service CIDR and this means a service almost certainly have different addresses on the two clusters. So, when the proxy on the remote cluster asks for a Service Profile, it sends an IP address of a service that `destination`, which is on the home cluster, does not know or coincides with another service.

In Linkerd if a proxy cannot resolve the Service Profile of a destination, it assumes that it is out of the mesh and contact it without the features of the service mesh (observability, reliability, security and so on).

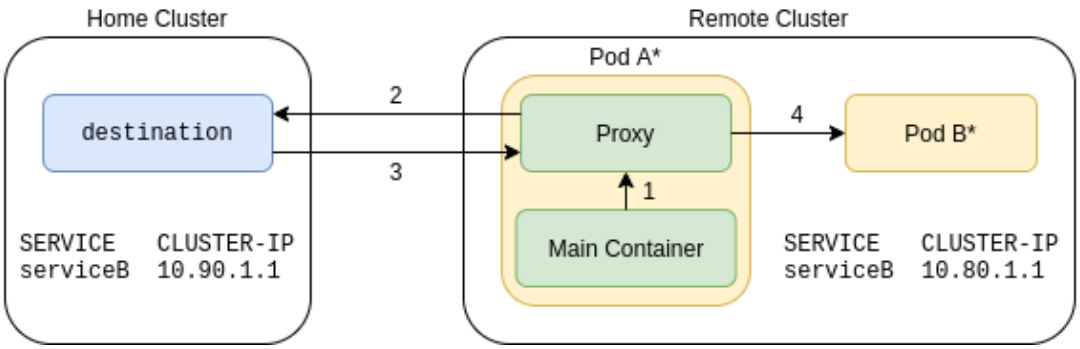

Figure 5.1: Service Profile Example

To better understand the problem, we can take a look at the example in Figure 5.1, that shows the fist part of service discovery process in our scenario. As you can see, `serviceB` has different IP addresses in the two clusters: `10.90.1.1` on the home cluster and `10.80.1.1` on the remote one. Let's analyze all the steps:

1. The main container of Pod A wants to send a request to `serviceB`, so resolves the name into an IP address. Since Pod A is offloaded, it gets `10.80.1.1` from the DNS.
2. Proxy intercepts the request and asks to `destination` for the ServiceProfile of `10.80.1.1`.
3. `destination` looks up into services on its cluster, but it does not find any results. So it replies with and empty Service Profile.
4. Proxy receives the response and understands that the target is out of the mesh, so forward the request in the standard Kubernetes way which arrives to and endpoint of `serviceB`, in this case Pod B.

This issue introduces a major limitation: in fact, to use Linker Service Mesh on Liqo the reflected services on the remote cluster must have the same address of the original ones on the home cluster. This is not always possible because the clusters may have different Service CIDRs or the address may be already assigned to another service.

5.1.4 Endpoints

In this subsection we discuss about issues related to the second step of the Linkerd service discovery on the Liqo architecture. As we saw before, with the first step the proxy receives the Service Profile of the target, while this second step it resolves the endpoints of the desired service.

To obtain endpoints the proxy makes a second call to `destination`: it sends the Service Profile authority, resolved before, and receives a list of weighted endpoints as a stream of updates. Based on the information contained in the profile, the proxy chooses the endpoint to contact and forwards the request to it.

On Liqo architecture there can be problems with addresses of endpoints. In fact, if there is a collision between cluster Pod CIDRs, Liqo automatically remaps the CIDR of the remote cluster into a free one. In this case, the same endpoint has a different address on each cluster. When proxy asks for endpoints, `destination` replies with the addresses that it sees on its cluster, the home one; but in certain cases proxy does not recognize them in its cluster, the remote one, and so it is unable to forward the message.

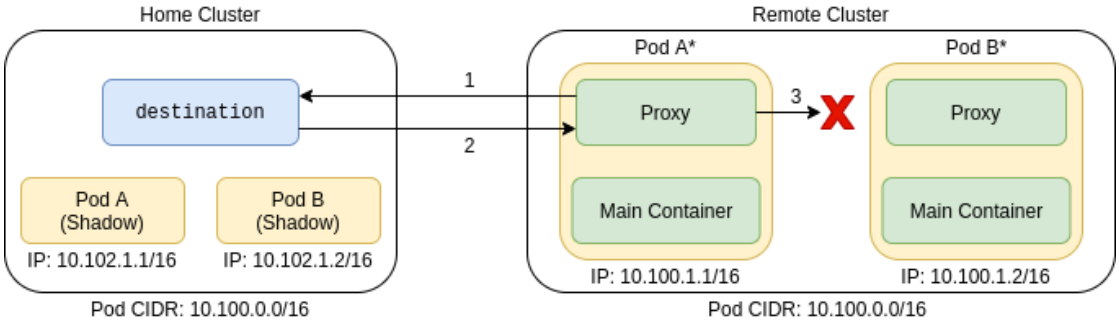

Figure 5.2: Offloaded proxy tries to contact offloaded endpoint

Let's consider the example of our testing scenario, as depicted in Figure 5.2. Both clusters have the same Pod CIDR, `10.100.0.0/16`, so it is need a remapping. In this case the network of the remote cluster is remapped on `10.102.0.0/16` in the home one, which is free. We have two offloaded pods on the remote cluster and their respective shadows pods on the home one, which have the remapped address. Now we imagine that Pod A wants to contact Service B and it has already resolved the Service Profile. The next step is to obtain the endpoints, so:

1. Proxy sends a request for endpoints to `destination` sending the authority of Service Profile received before.
2. `destination` starts a watcher on the endpoints of the desired service and when there are updates sends a message to the proxy. In this case, it sends `10.102.1.2`.
3. Proxy does not recognize this endpoint, because Pod B in its context has `10.100.1.2` as IP address.

It is clear that the service mesh cannot work with two cluster if an offloaded pod tries to contact another offloaded pod. But since the remote cluster remapped in

turn the home cluster addresses, offloaded pods can not either contact endpoints on the home cluster.

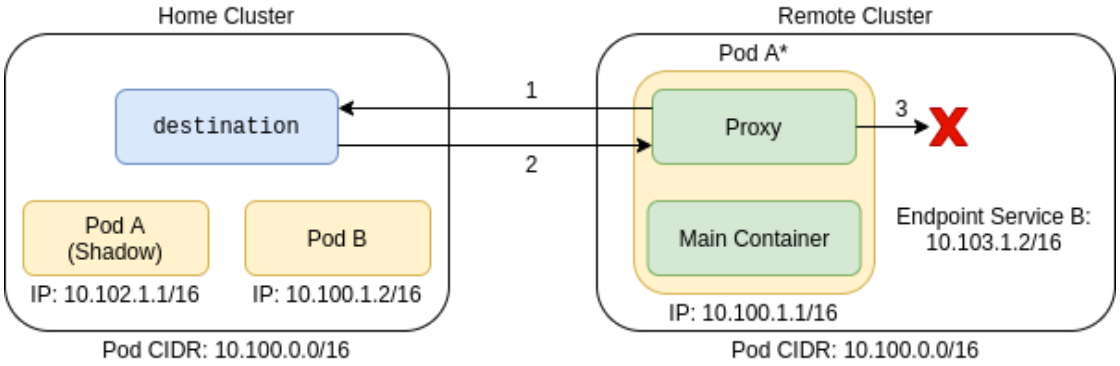

Figure 5.3: Offloaded proxy tries to contact home endpoint

In figure 5.3 there is a representation of this scenario: we have Pod A, offloaded on the remote cluster, that asks for endpoints of service B. `destination` sends it a message containing the endpoint `10.100.1.2`, that it finds in its context. Proxy does not recognize this endpoint, because on the remote server Liqo remapped it to `10.103.1.2` and so message cannot be forwarded.

In order to make the mesh working, we have to use clusters that do not have collisions between their Pod CIDRs, so that they do not remap each other. This introduces a new major limitation in using Linkerd service mesh on Liqo architecture.

For now, we limited our analysis to two cluster only, but if we extend to an indefinite number of them the workaround explained before does not work anymore. In fact, even if we have not remapping between clusters, an offloaded proxy cannot contact another offloaded proxy if they are on different clusters.

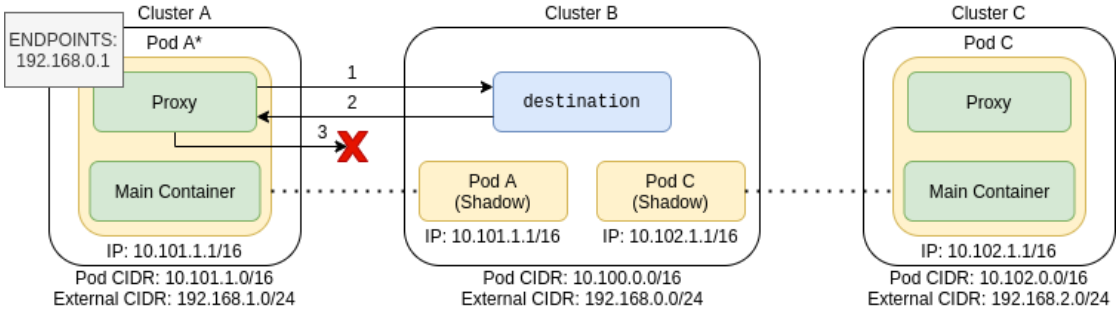

Figure 5.4: Endpoint resolution failure in a three cluster scenario

Since the complexity of the topic, we leverage on the example in figure 5.4 to

understand the problem. In this scenario we consider three clusters, without any remapping neither on the Pod CIDR nor on the External CIDR. Pod A and Pod C are offloaded on cluster A and cluster C respectively, while cluster B is the home cluster, in which there is the Linkerd control plane. As explained in chapter 3, cluster B reflects the endpoint Pod C on cluster B remapping it on an address taken from its External CIDR, in this case `192.168.0.1`. If Pod A wants to send a request to Pod C it has to contact this address. Let's follow again the process step by step:

1. Proxy requests endpoints for service C to `destination`;
2. `destination` is unaware of the remapping on the External CIDR, so replies to the proxy with the address `10.102.1.1`;
3. Proxy receives the response, but it is unable to contact Pod C because the address is not known in its context since Pod C is reachable by contacting `192.168.0.1`.

This means the Linkerd service mesh can work on Liqo only with applications offloaded on at most one cluster and there are no workarounds to avoid this limitation.

5.2 Istio Analysis

In this section we will focus on issues and limitation of an Istio service mesh on the Liqo architecture. We will refer to the scenario depicted in the introduction of the chapter.

The analysis was conducted on Istio version 1.10 and it brings to light four issues of varying degrees: as we will see, some of them required little code modifications while others impose some limitations. In any case, they impact in a negative way on the flexibility of the solution. Obviously, results can change with newer versions of the software.

5.2.1 Namespace Reflection

As for Linkerd, Istio needs to expose its services on the remote cluster. This means that we have to reflect its namespace, using the `NamespaceOffloading` CRD. However, we cannot reflect with an arbitrary name, but we must keep the orginal name, used on the home cluster. In fact, Istio sets some environmental variables in meshed pods, one of which contains the Istio namespace name: it is the variable that contains the **Certification Authority** (CA) address, needed for authentication.

Listing 5.1: Environmental variable containing the Istio namespace name

```
...
env:
  - name: CA_ADDR
    value: istiod.istio-system.svc:15012
...
```

Unlike Linkerd, other meshed namespaces can be offloaded with other names because environmental variables containing them are initialized by the Downward API, that keeps the value on the remote cluster when pod is offloaded.

5.2.2 Downward API

When Istio injects Envoy proxy into a pod, it sets several environmental variables. Some of them are populated using the **Downward API**, which exposes pod information into containers. Data can be taken from any field of the pod, including the status. Normally, this API is implemented by the kubelet, but in case of virtual nodes, like the ones used by Liqo, it is implemented by the **Virtual Kubelet**. However, it has only a partial implementation of the Downward API, in particular values taken from the status are not supported. Istio uses two values of this type in its environmental variables:

Listing 5.2: Istio environmental variables populated with values from the status

```
...
env:
  - name: INSTANCE_IP
    valueFrom:
      fieldRef:
        apiVersion: v1
        fieldPath: status.podIP
  - name: HOST_IP
    valueFrom:
      fieldRef:
        apiVersion: v1
        fieldPath: status.hostIP
...
```

When a pod with this kind of variables is scheduled on a virtual node, it cannot became ready because the virtual kubelet cannot populate them. This introduces an issue when we try to offload a pod of an Istio service mesh. There are no ways to get around the problem without modify the code.

Introduce a complete support for the Downward API in the virtual kubelet is a long and complex work and it is not the focus of this thesis. In addition, it is not necessary in Liqo because pods on virtual nodes are actually running on a real cluster, the remote one, that has a complete support to Downward API on its kubelet. So, we decided to make a temporary little modification in the code of the virtual kubelet:

```
1  // podFieldSelectorRuntimeValue returns the runtime value of the
       given selector for a pod.
2  func podFieldSelectorRuntimeValue(fs *corev1.ObjectFieldSelector, pod
       *corev1.Pod) (string, error) {
3      internalFieldPath, _, err := podshelper.
       ConvertDownwardAPIFieldLabel(fs.APIVersion, fs.FieldPath, "")
4      if err != nil {
5          return "", err
6      }
7      switch internalFieldPath {
8      case "spec.nodeName":
9          return pod.Spec.NodeName, nil
10     case "spec.serviceAccountName":
11         return pod.Spec.ServiceAccountName, nil
12     // return an empty string in case of unsupported fields.
13     case "status.podIP":
14         return "", nil
15     case "status.hostIP":
16         return "", nil
17     }
18     return fieldpath.ExtractFieldPathAsString(pod, internalFieldPath)
19 }
```

In this function are introduced two new cases, corresponding to the values needed by Istio environmental variables. As you can see in lines between 13 and 16, these are populated not with the real values but with an empty string. When a pod is offloaded on the remote cluster, its kubelet will populate them with proper values. This modification is thought for testing purpose only, so that we can continue our analysis about issues and limitations of an Istio service mesh on the Liqo architecture.

5.2.3 Authentication

Kubernetes supports two forms of service account tokens:

- **First party tokens:** default service account tokens stored in secrets, mounted in all pods without expiration or audience.

- **Third party tokens:** tokens projected by the kubelet into pods with configurable properties like expiration and audience. The kubelet will request and store them into the pod and automatically rotate them when they approach expiration. The application is responsible for reloading tokens after rotation.

Listing 5.3: Example of a pod with a third party token

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /var/run/secrets/tokens
      name: vault-token
  serviceAccountName: build-robot
  volumes:
  - name: vault-token
    projected:
      sources:
      - serviceAccountToken:
          path: vault-token
          expirationSeconds: 7200
          audience: vault
```

Istio uses service account tokens for authentication between the proxy and the control plane. By default it uses third party token, since they are more secure than first party ones. However these are not supported by all clusters, specially ones with Kubernetes version 1.19 or below, so Istio gives also the possibility to use the first party tokens. If you install Istio using `istioctl`, it automatically detects if you cluster supports this featues, otherwise it downgrade to first party ones. This selection can also be done manually by setting the value `values.global.jwtPolicy` to `third-party-jwt` or `first-party-jwt`.

In our testing scenario, when a pod that mounts a third party token is offloaded receives a new token on the remote cluster, that is not valid in the context of the home cluster. Liqo does not support the replication of third party tokens: first they are not mounted in secrets and this complicates the operations; in addition, they are linked to the local kubelet that manages their rotation. This implies that we have to downgrade to first party tokens for our testing purposes, introducing a

new limitation: uses a less secure authentication system.

Finally, if we use first-party tokens for authentication, we will have the problem described before when talking about Linkerd certificates: in fact, Liqo does not mount the default token in offloaded Pods. So also in this case we have to avoid the default service account. We will discuss the solution of this problem in the next chapter.

5.2.4 Endpoints

In an Istio service mesh the control plane sends to the proxy the endpoints of a specific service. This means we suffer of a problem similar to the one with Linkerd service discovery: Istio control plane sends IP addresses taken from its context that might have been remapped by Liqo. As seen for Linkerd, problems may appear when an offloaded pods tries to contact another endpoint. In fact, it receives an address from the control plane that is not assigned to an endpoint in its context or, even worst, it overlaps with other services.

The provisional solution is the same of Linkerd: use two clusters with Pod CIDRs that do not collide, so that there will be no remapping of the endpoints on both sides, home and remote cluster. This means also that is not possible to use more than two clusters, because in that case remapping is necessary.

Chapter 6

Integrating Linkerd and Liqo

In this chapter we proceed to the next step of the work, which is to develop a service mesh prototype that works in the Liqo environment by solving issues described before. The prototype is based on one of the service mesh presented before. Between them we choose to focus on Linkerd because of the issue related to third-party service account tokens: in fact, as seen in the previous chapter, this kind of tokens are managed by the kubelet and stored directly into the pods. This implies we are not sure that they can be replicated and, even if we succeed, it is not sure that they will work. In addition, it is to be considered the fact they are automatically rotated at expiration time. From this point of view Linkerd does not presents unsolvable issues and so guarantees a better starting point.

In the following sections we take one by one issue of Linkerd with Liqo, presented in Chapter 5, and we provide the design of a solution and its implementation. We discuss the issues staring from the major ones.

6.1 Service Profiles

We see Linkerd proxy requires Service Profiles by sending the IP address of the service, but if the pod is offloaded it will send an address that control plane does not know or matches with the wrong service. This causes the major limitation we have which forces us to use clusters with the same Service CIDR so that we can reflect services with the same IP address, hoping they are free.

To solve this issue we reply on a feature of Linkerd control plane: in Service Discovery process it can accepts both IP addresses or DNS names. The latter are the same on both clusters if namespaces are reflected with the same name, as we discuss later.

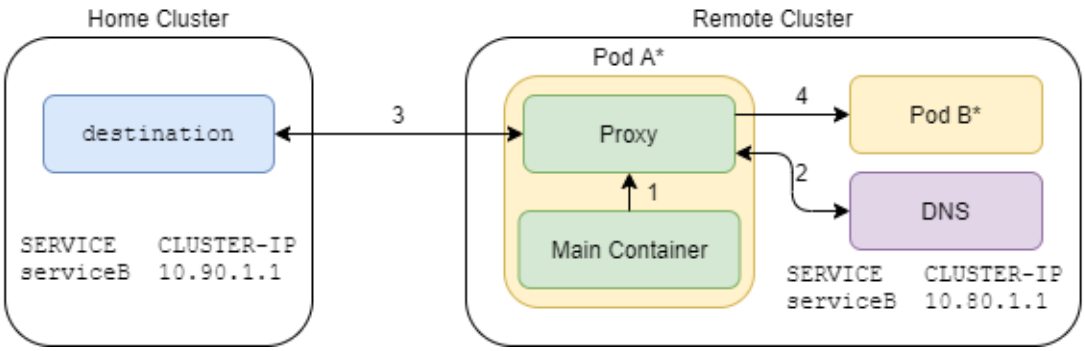

Figure 6.1: Solution to Service Profiles issue

The main idea behind the solution is convert the IP address into a name by performing a reverse query to the cluster DNS. In figure 6.1 there is a representation of the solution in our testing context:

1. The main containter of Pod A sends a request to the IP of service B, which in its cluster is `10.80.1.1`.

2. Proxy intercepts the request and performs a reverse query to the cluster DNS. Supposing that service B is in the default namespace, it obtains `serviceB.default.svc.cluster.local`.

3. Proxy uses the name instead of the address into the request and receives the correct Service Profile.

4. After completing service discovery, proxy is now able to contact the endpoint of service B with all the features of service mesh.

Explained the idea, now let's focus on its implementation. This requires some modifications into the Linkerd proxy, written in Rust. First of all we need a function that given an IP address return the corresponding name:

Listing 6.1: Function to get DNS name from IP address

```
fn get_name_from_ip(addr: &String) -> Result<String,String> {
    // Parse String to IpAddr
    let ip: std::net::IpAddr = match addr.parse() {
        Ok(ip) => ip,
        Err(_) => return Err("Cannot parse address".to_string())
    };

    // Retrieve hostname
    match lookup_addr(&ip) {
        Ok(hostname) => return Ok(hostname),
```

```
11          Err(_) => return Err("Cannot retrieve hostname".to_string())
12
13      };
14  }
```

In this function we receive the address as a string, so first we have to parse it into a `std::net::IpAddr` object that we will use to query the DNS. Then we select a suitable library to perform that query and the choice fell on `dns-lookup`. We use the `lookup_addr` function of this library that given an IP return the DNS name.

Secondly we have to integrate it into the proxy, precisely before that it sends the request to the control plane:

Listing 6.2: Integration of the function into the proxy

```
1  ...
2      let LookupAddr(addr) = t.param();
3
4      // Split addr in name and port
5      let mut host = addr.to_string();
6      let v: Vec<_> = (&host).split(":").collect();
7      let name_addr = v[0].to_string();
8      let port = v[1];
9
10     // Translate IP into name
11     match get_name_from_ip(&name_addr) {
12         Ok(name) => host = format!("{}:{}", name, port),
13         Err(e) => warn!("{}", e)
14     }
15
16     info!("Host is {}", host);
17
18     let request = api::GetDestination {
19         path: host,
20         context_token: self.context_token.clone(),
21         ..Default::default()
22     };
23 ...
```

Looking at the code above, extracted from the method which call the gRPC API, you can see that we have to get the IP address from the parameter `addr`, because the latter contains also the port. Then we try to convert it into a name: in case of success we will use the name in the request, otherwise we continue using the address, which is the default behaviour of the proxy.

6.2 Endpoints

The second major limitation we have is related to the second part of the service discovery process: as explained in section 5.1.4 when there is a remapping of endpoint addresses between clusters the mesh does not work because the control plane is unaware about that. What we have to do for solving this issue is communicate to the control plane which are these remappings.

The main idea to achieve the goal is create a communication between the `destination` component of Linkerd control plane and the **Liqo IPAM**, which is the component of Liqo that manages the remappings. In this way `destination` can convert the address it knows into the one valid in the context of the requesting pod. The new process is depicted in Figure 6.2:

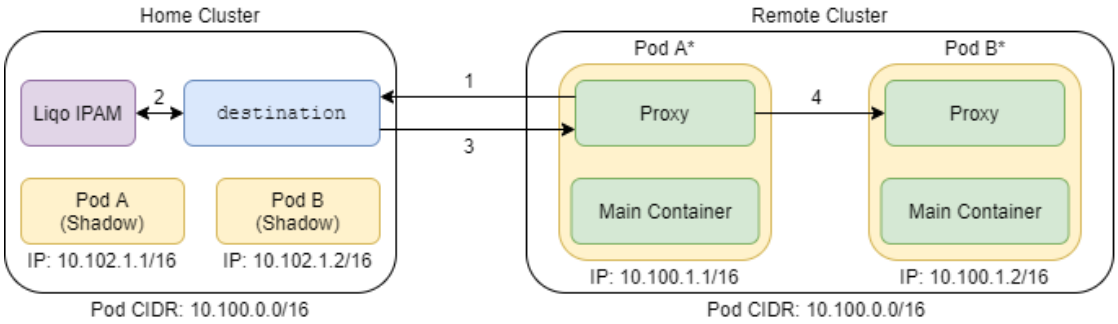

Figure 6.2: Solution to Endpoints issue

As example we consider the testing scenario of the previous chapter in which we have two clusters, home and remote, and an offloaded pod, Pod A, wants to contact an offloaded endpoint of a service:

1. Proxy sends to `destination` a request for endpoints of Service B, using information contained in the Service Profile received before.

2. `destination` get the IP address of Pod B, which in its context is `10.102.1.2`. Then it understand Pod A is offloaded and so it asks to Liqo IPAM to convert the address for the context of Pod A. IPAM returns `10.100.1.2`.

3. `destination` responses to Pod A, sending it the address `10.100.1.2`.

4. Pod A can now contact directly Pod B, applying the policies contained in the Service Profile.

The connection between Linkerd and Liqo can be easily because Liqo IPAM already integrates a gRPC server. We will create a new API that will be called

by `destination` before sending the response to the source. However, there is an unsolved question: how does destination understand the source is offloaded? To solve this question, the first thought is to look at source IP of the request coming from the pod. However, this is not possible because `destination` itself is in the mesh an its proxy masquerade the source IP with localhost address `127.0.0.1`. In addition, even if we can obtain that IP, probably it will be the address the pod has on its cluster and so we cannot recognize where it is because more clusters in the federation can have the same Pod CIDR.

The definitive solution we proposes is to modify the gRPC API used by the proxy to require endpoints by adding a new parameter: we choose to use **Cluster ID** because, as we will see later, it simplify the code in the function that will translate addresses into the IPAM.

The last part of the design is about how to retrieve the Cluster ID into the proxy. The simplest way is something like Linkerd does with its parameters, which is to inject it into an environmental variable. This process will be done by the Virtual Kubelet when forges the pod to be offloaded.

The implementation starts from this last point. In particular, we need to modify the function `forgeContainers` in the package `forge` of the Virtual Kubelet.

Listing 6.3: Function that adds the Cluster ID env var

```
func (f *apiForger) forgeContainers (
    inputContainers []corev1.Container,
    inputVolumes    []corev1.Volume ) []corev1.Container
{
    containers := make([]corev1.Container, 0)

    for _, container := range inputContainers {
        volumeMounts := filterVolumeMounts(inputVolumes,
                                           container.VolumeMounts)
        env := corev1.EnvVar{
            Name:  "LIQO_CLUSTER_ID",
            Value: strings.TrimPrefix(
                    f.virtualNodeName.Value().ToString(),
                    virtualKubelet.VirtualNodePrefix),
        }
        envs := append(container.Env, env)
        containers = append(containers,
                  translateContainer(container, volumeMounts, envs))
    }
    return containers
}
```

We uses an existing loop, where `volumeMounts` are filtered, to add in each container an environmental variable named `LIQO_CLUSTER_ID` with the value taken from the node name. In fact, virtual nodes created by Liqo contains the Cluster ID

in the name. Then we append our new variable to the existing ones. This function adds the Cluster ID variable in all containers even if it is not necessary simply because is the easiest way to implement this feature and it does not add so much overhead.

Secondly, we need to modify the gRPC API in order to pass the Cluster ID value to the control plane. The first step is adapt the protocol buffer to our need:

Listing 6.4: Modified `destionation.proto`

```
service Destination {
  // Given a destination, return all addresses in that destination
  // as a long-running stream of updates.
  rpc Get(GetEndpoints) returns(stream Update) {}
...
}

message GetEndpoints {
  string scheme = 1;
  string path = 2;
  string context_token = 3;
  string cluster_id = 4;
}
...
```

Starting from that, we have to regenerate Go client and server, which are used by the control plane. We have also to adjust the Rust client on the proxy, to extract the Cluster ID from the variable and to add it into the request, which is the `GetEndpoints` message depicted in the protocol buffer.

On control plane side, modifications are required into `destination` component that needs to receive the new parameter and then use the gRPC API exposed by Liqo IPAM. The part of the code in which Linkerd sends updates on endpoints to proxies is in the **Endpoint Translator**. Here there are two methods, `sendClientAdd` and `sendClientRemove`, which needs the address translation. So first of all we create a method that will be called by them if necessary:

Listing 6.5: Function to contact IPAM from `destination`

```
func initIpamClient() (liqonetIpam.IpamClient, error)
{
    conn, err := grpc.Dial(
        "liqo-network-manager.liqo.svc.cluster.local:6000",
        grpc.WithInsecure(),
        grpc.WithBlock())
    if err != nil {
        return nil, err
    }
    return liqonetIpam.NewIpamClient(conn), nil
}
```

```
12
13 func (et *endpointTranslator)
14     translateEndpointIP(address *net.TcpAddress)
15     (*net.TcpAddress, error)
16 {
17     // Convert endpointIP to string
18     endpointIP := addr.ProxyIPToString(address.GetIp())
19
20     // Create request
21     request := &liqonetIpam.GetRemotePodIPRequest{
22         Ip:        endpointIP,
23         ClusterID: et.srcClusterId,
24     }
25
26     // Initialize IPAM client
27     liqoIPAM, err := initIpamClient()
28     if err != nil {
29         return nil, fmt.Errorf("Cannot connect to Liqo IPAM: %s",err)
30     } else {
31         et.log.Infof("Connected to Liqo IPAM")
32     }
33
34     // Send request to IPAM
35     response, err := liqoIPAM.GetRemotePodIP(context.TODO(), request)
36     if err != nil {
37         return nil, err
38     }
39
40     // Get translated IP from response
41     ip := response.GetRemoteIP()
42     et.log.Infof("Converted %s into %s", endpointIP, ip)
43
44     // Convert translated IP into the correct type
45     ipv4, err := addr.ParseProxyIPV4(ip)
46     if err != nil {
47         return nil, err
48     }
49     return &net.TcpAddress{
50         Ip:   ipv4,
51         Port: address.GetPort(),
52     }, nil
53 }
```

Before sending the request of translation to the IPAM, we need to format the address into a string. Then we put it into the request with the Cluster ID sent by the proxy, which is saved as parameter in the endpoint translator. When the response arrives from the IPAM, we need to reconvert the received address from string to the type needed by Linkerd.

This function is then integrated in `sendClientAdd` and `sendClientRemove` with some small differences because in this first case there are weighted addresses, a struct containing the address and its weight, while in the second case there are regular addresses.

Listing 6.6: Integration of IPAM call into `sendClientAdd`

```
func (et *endpointTranslator) sendClientAdd(set watcher.AddressSet)
{
    addrs := []*pb.WeightedAddr{}
    for _, address := range set.Addresses {
        var (
            wa  *pb.WeightedAddr
            err error
        )
        ...
        if et.srcClusterId != "" {
            endpointAddr, err := et.translateEndpointIP(wa.Addr)
            if err != nil {
                et.log.Errorf("IP translation failed: %s", err)
            } else {
                wa.Addr = endpointAddr
            }
        }
        addrs = append(addrs, wa)
    }
    ...
}
```

Listing 6.7: Integration of IPAM call into `sendClientRemove`

```
func(et *endpointTranslator) sendClientRemove(set watcher.AddressSet)
{
    addrs := []*net.TcpAddress{}
    for _, address := range set.Addresses {
        tcpAddr, err := toAddr(address)
        ...
        if et.srcClusterId != "" {
            endpointAddr, err := et.translateEndpointIP(tcpAddr)
            if err != nil {
                et.log.Errorf("IP translation failed: %s", err)
            } else {
                tcpAddr = endpointAddr
            }
        }
        addrs = append(addrs, tcpAddr)
    }
    ...
}
```

These two functions iterate on the addresses returned by a watcher and, after some checks, they add the address to a slice. Before adding them to the slice, we put our code. First of all we check if the received Cluster ID is not empty string:

- If it is empty string it means the pod is on the same cluster of Linkerd control plane, so it does not need the translation. In fact, the `LIQO_CLUSTER_ID` environmental variable is present only in offloaded pods because is set by the Virtual Kubelet.
- In case we have a value, we invoke the `translateEndpointIP` method seen before, and if all gone well we set the address to the translated one.

The last and major part of implementation regards the Liqo IPAM. It uses data stored into the `IpamStorage` CRD, which among other data contains subnets inherent to remapping for each Cluster ID. At state of art this subnets are not sufficient to perform the translations we need, so we have to add new ones. For keeping coherence in their names we have also to change some of those already present. After modifications, stored subnets are:

- **LocalNATPodCIDR:** network used in the remote cluster for local Pods. The default value is "None" that means remote cluster uses local cluster PodCIDR.
- **LocalNATExternalCIDR:** network used in remote cluster for local service endpoints. The default is "None" that means remote cluster uses local cluster ExternalCIDR.
- **RemotePodCIDR:** the PodCIDR of the remote cluster.
- **RemoteExternalCIDR:** the ExternalCIDR of the remote cluster.
- **RemoteNATPodCIDR:** Network used in the local cluster to remap Pods offloaded on the remote cluster.
- **RemoteNATExternalCIDR:** Network used in local cluster for remote service endpoints.

The values we add are the RemotePodCIDR and the RemoteExternalCIDR. Now we have all the data needed for the translations and so we implement our API. First of all we have to change the protocol buffer and regenerate Go client and server, as done before with Linkerd API. Then we write the function called when a request arrives. Due to its complexity, we will analyze the method one part at a time.

Listing 6.8: Translation function in IPAM (Part 1)

```
func (liqoIPAM *IPAM) getRemotePodIPInternal(clusterID, ip string)
    (string, error)
{
    if clusterID == "" {
        return "", &liqoneterrors.WrongParameter{
            Parameter: consts.ClusterIDLabelName,
            Reason:    liqoneterrors.StringNotEmpty,
        }
    }
    if parsedIP := net.ParseIP(ip); parsedIP == nil {
        return "", &liqoneterrors.WrongParameter{
            Reason:    liqoneterrors.ValidIP,
            Parameter: ip,
        }
    }

    // Get cluster subnets
    clusterSubnets, err := liqoIPAM.ipamStorage.getClusterSubnets()
    if err != nil {
        return "", fmt.Errorf("cannot get cluster subnets:%w", err)
    }

    subnets, exists := clusterSubnets[clusterID]
    if !exists {
        return "", fmt.Errorf(
            "cluster %s subnets are not set", clusterID)
    }
```

In this part, we check if received parameters are valid and we get the subnets relative to the Cluster ID from the IPAM storage.

Listing 6.9: Translation function in IPAM (Part 2)

```
    // Get PodCIDR
    podCIDR, err := liqoIPAM.ipamStorage.getPodCIDR()
    if err != nil || podCIDR == emptyCIDR {
        return "", fmt.Errorf("cannot get cluster PodCIDR: %w", err)
    }

    belongs, err := ipBelongsToNetwork(ip, podCIDR)
    if err != nil {
        return "", fmt.Errorf(
            "cannot establish if IP %s belongs to PodCIDR:%w",
            ip, err)
    }
    if belongs {
        if subnets.LocalNATPodCIDR == "None" {
            return ip, nil
        }
```

```
17          /** If ip is in the podCIDR means that source in remote
18          while destination is local, so ip must be remapped
19          on the localNATPodCIDR */
20          return utils.MapIPToNetwork(subnets.LocalNATPodCIDR, ip)
21      }
```

In this second part we check if the endpoint we want to translate is in the local Pod CIDR. If true, this means that an offloaded pod wants to contact a local one and so we have to return the IP this endpoint has on the remote cluster. We look at the value of LocalNATPodCIDR: if it is `None` remote cluster does not remap local pods, so we return the address as is. Otherwise we remap the IP on the LocalNATPodCIDR and then we return it.

Listing 6.10: Translation function in IPAM (Part 3)

```
1       belongs, err = ipBelongsToNetwork(ip, subnets.RemoteNATPodCIDR)
2       if err != nil {
3           return "", fmt.Errorf(
4               "cannot establish if IP %s belongs to PodCIDR:%w",
5               ip, err)
6       }
7       if belongs {
8           // Check if RemotePodCIDR is set
9           if subnets.RemotePodCIDR == "" {
10              return "", &liqoneterrors.WrongParameter{
11                  Reason: liqoneterrors.StringNotEmpty,
12              }
13          }
14          /** If ip is in the remoteNATPodCIDR means that source
15          and destination are on the same remote cluster,
16          so the ip must be remapped on the remotePodCIDR */
17          return utils.MapIPToNetwork(subnets.RemotePodCIDR, ip)
18      }
```

In this part there is the second case of remapping: if the IP belongs the RemoteNATPodCIDR it means that a remote pod wants to contact a pod on its own cluster. So we remap the endpoint to RemotePodCIDR, which is the Pod CIDR of the remote cluster.

Listing 6.11: Translation function in IPAM (Part 4)

```
1       endpointMappings, err := liqoIPAM.ipamStorage
2                                        .getEndpointMappings()
3       if err != nil {
4           return "", fmt.Errorf("cannot get Endpoint IPs: %w", err)
5       }
6
7       mapping, exists := endpointMappings[ip]
8
```

```
9       if !exists {
10          return "", fmt.Errorf("mapping for %s does not exist", ip)
11      }
12
13      /** Source and destination are on different remote clusters,
14      so ip must be remapped on the externalCIDR of the local cluster.
15      If it was remapped by the remote cluster, the ip must be
16      remapped on the localNATExternalCIDR  */
17      if subnets.LocalNATExternalCIDR != "None" {
18          return utils.MapIPToNetwork(subnets.LocalNATExternalCIDR,
19                                      mapping.IP)
20      }
21
22      return mapping.IP, nil
23  }
```

Here we have the last part of the function, where we manage the scenario in which a remote pod wants to contact another remote pod, but they are offloaded on different clusters. First of all we need to know how the endpoint is mapped on the local ExternalCIDR or map it if it is not yet. Then we have two possibilities:

- If localNATExternalCIDR is set, this means remote cluster remaps the local ExternalCIDR, so we have to remap again the address from the ExternalCIDR to the localNATExternalCIDR.
- If it is not set, we return directly the address from the local External CIDR.

Summarizing, we resolved all remapping cases:

1. Source on remote cluster, destination on local cluster: we have to remap the destination address on the localNATPodCIDR, if it is different from `None`.
2. Source and destination on the same remote cluster: we have to remap the destination address on the RemotePodCIDR.
3. Source and destination offloaded on different clusters: we have to return the address from ExternalCIDR corresponding to the destination. We have to remap it on localNATExternalCIDR,if it is different from `None`.

6.3 mTLS Certificate

In the previous chapter we saw the mesh does not work if offloaded pods uses default service account token, because they are not mounted in offloaded pods, even though they are correctly reflected on the remote cluster. The reason of this issue is a condition which prevents the mount of default tokens in the Virtual Kubelet

when forging the pod that will be offloaded. The solution is very simple: we have to eliminate this condition.

Listing 6.12: Function that forges volumes in Liqo Virtual Kubelet

```
1  func forgeVolumes(volumesIn []corev1.Volume) []corev1.Volume {
2      volumesOut := make([]corev1.Volume, 0)
3      for _, v := range volumesIn {
4          if v.ConfigMap != nil
5              || v.EmptyDir != nil
6              || v.DownwardAPI != nil {
7              volumesOut = append(volumesOut, v)
8          }
9          if v.Secret != nil
10             /* && !strings.Contains(v.Secret.SecretName,
11                                     "default-token") */ {
12             volumesOut = append(volumesOut, v)
13         }
14     }
15     return volumesOut
16 }
```

Above there is the code of `forgeVolumes`, the function which creates the volumes of offloaded pods. At line 10 there is the condition mentioned before: it checks if in the name of the Secret there is `default-token`. If is true, it does not create the volume and so it will not be mounted later. Eliminating this condition, reported here as comment, allows the forge of the volume, which will be mounted into the pod and used by the proxy for authentication with Linkerd control plane.

6.4 Namespace Reflection

The problem we have with namespace reflection regards names: at state of art we cannot reflect meshed namespaces with a different name than the original one. Solving this limitation requires two different approaches: one for Linkerd namespace and one for others namespaces.

The first case can be easier to solve than the second, because addresses of Linkerd services, which include the namespace, are stored in environmental variables. Changing these values will probably make the service mesh work.

The second case is more difficult because due to our modification, described in section 6.1, services names, which include namespace ones, are involved in service discovery process, so we need a translation system for the addresses.

Since the issue is quite complex to solve and reflection of namespaces with the same name is a scenario supported by Liqo, we decide to skip this fix for focusing on ones described in previous sections, which are more impacting on the usage of Linkerd service mesh on Liqo.

Chapter 7

Evaluation

In this chapter we evaluate the prototype build in chapter 6. The evaluation consists in a simulation of a real usage of our solution by deploying an application and create traffic using a load generator. In addition, we compare it to other solutions, by replicating the same scenarios using them instead of our prototype.

7.1 Performance analysis

In this section we take a look to performance of the prototype. To make a better evaluation, we consider three test cases, to see how our solution reacts to different number of active users and different user spawn rate. In particular, testing cases executed are:

1. 500 users with a spawn rate of 5 users/second.
2. 1000 users with a spawn rate of 5 users/second.
3. 1000 users with a spawn rate of 10 users/second.

As said before, we do not limit to test our solution, but we want compare it with other existing ones. So we execute tests in different multi-cluster scenarios:

- **Linkerd on Liqo:** this is the scenario of our prototype, where the clusters are connect with a custom version of Liqo and meshed with a custom version of Linkerd.
- **Liqo:** in this scenario cluster are connected with Liqo, version `0.3.0`, without a service mesh.
- **Linkerd Multi-cluster:** in this scenario clusters are connected with the multi-cluster solution provided by Linkerd, version `2.10.2`.

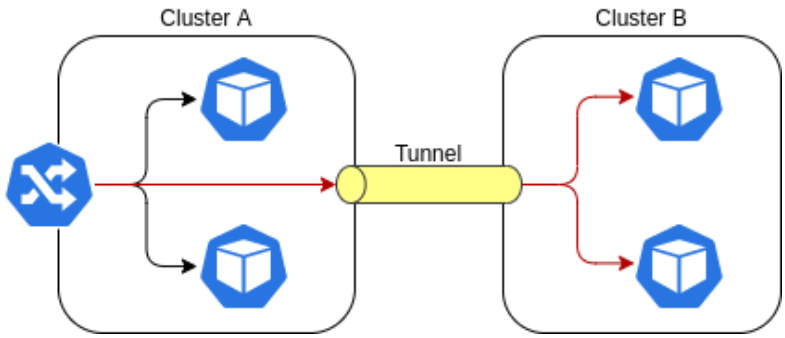

Figure 7.1: Testing environment

In figure 7.1 is represented the testing environment: we have two Kubernetes clusters, connected each time in a different way. We consider Cluster A our home cluster, while cluster B is the remote one. For each scenario, we deploy a micro-services application with an `Horizontal Pod Autoscaler`, which scales the number of replicas for each micro-service depending on the load. We expose this application through an ingress on Cluster A. Then, we create traffic, with the desired users and spawn rate, with a load generator targeting the ingress and the application starts scale up. In the scenarios with Liqo, some of these new replicas are offloaded on another cluster while in Linkerd Multi-cluster we have to deploy manually some replicas on the other cluster and set a `TrafficSplit` to address some traffic on them. Since the home cluster is bigger than the remote one, we set that 70% of the traffic remain on the local cluster, while 30% goes on the remote one. In any case we are in the situation depicted in figure 7.1, with some pods on the home cluster and some pods on the remote one. Traffic can be directed to local pods (black lines) or to remote pods through a tunnel (red lines).

Clusters used in this test are both on-premise and on public clouds. In particular, we use:

- An on-premise cluster composed by two nodes which have Intel(R) Xeon(R) Silver 4116 as CPU, with 24 processors.

- A cluster on Microsoft Azure, with a two nodes of dimension DS2_v2 and three spot nodes of dimension B2s.

We expect that performance of Liqo vanilla gets worse as the load increases, until the application becomes unusable, while the other two solutions keeps good performance in all the three testing scenarios. Moreover, we expect scenarios of Linkerd on Liqo and Linkerd Multi-cluster show similar results. In fact, the goal of the prototype is to have the dynamism and simplicity of Liqo with the good performance given by a service mesh, in this case Linkerd.

Performance are evaluated considering **P95 Latency** and **Error Rate** for each technology in each the testing case. Data reported are from the first ten minutes of test, which is the time the application stabilizes after starting from one replica for each deployment. Data shown in the following charts are average values, since tests are repeated more times.

First test case

In the charts below are reported data of the first testing case, with a regular number of users and a low spawn rate.

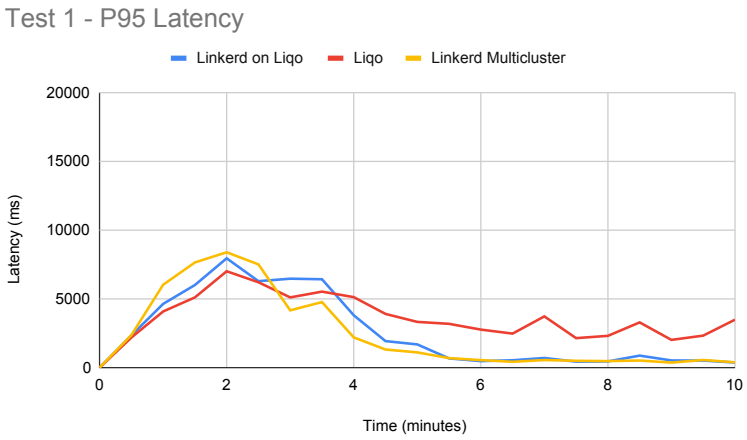

Figure 7.2: P95 Latency in first testing case

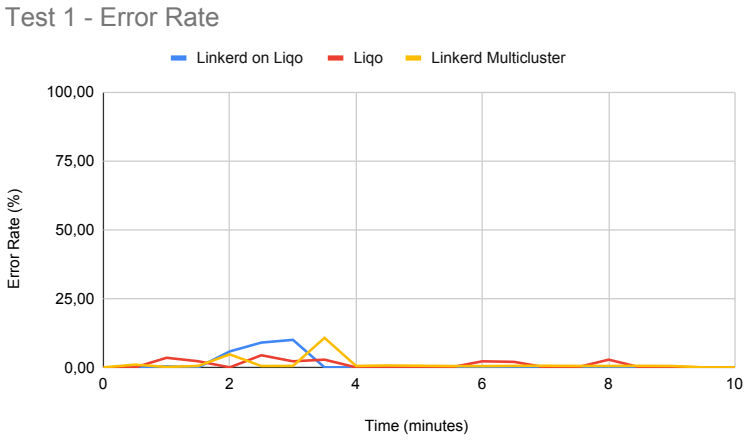

Figure 7.3: Error Rate in first testing case

In this case the behaviour of three solution is quite the same: in the first few minutes application scales up to manage the increasing number of users and, after a transitional period, latency stabilizes on low values and error rate became zero. In fact, with few users and low spawn rate the gRPC load balancing issue is mitigated.

Second test case

In the next case, spawn rate remains the same, while the total number of users is doubled. This implies the application has to manage twice the requests at the end of the transitional period.

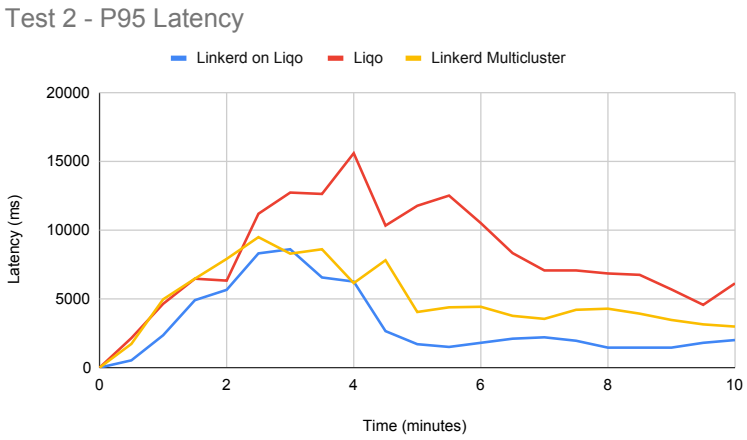

Figure 7.4: P95 Latency in second testing case

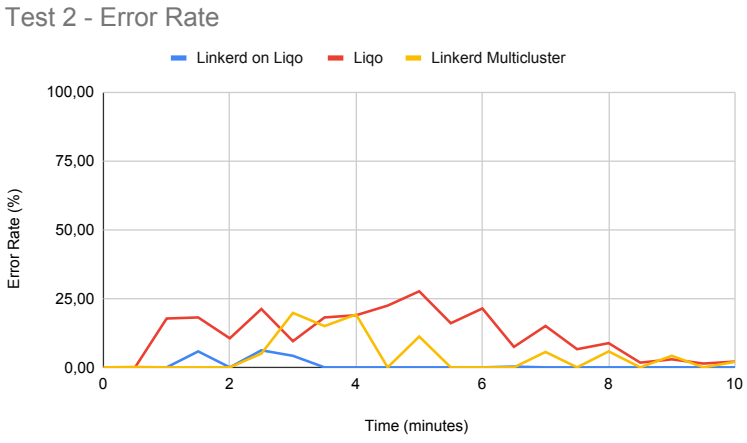

Figure 7.5: Error Rate in second testing case

As you can see in the charts above, the behaviour of Linkerd on Liqo and Linkerd Multi-cluster remains quite the same, but this time they stabilize of higher latency than before due to the increased number of requests. On the other side, vanilla Liqo present a greater transitional period than the others and stabilizes on latency values even more higher than the other two solutions. Even looking at the error rate, we can notice that it has the higher percentage of error. However, with this load vanilla Liqo still manages to converge towards acceptable values.

Third test case

If we double also the spawn rate, the application has to manage the same number of requests than before, but it has less time to react to load changes. This causes a deterioration in performance on Liqo without service mesh:

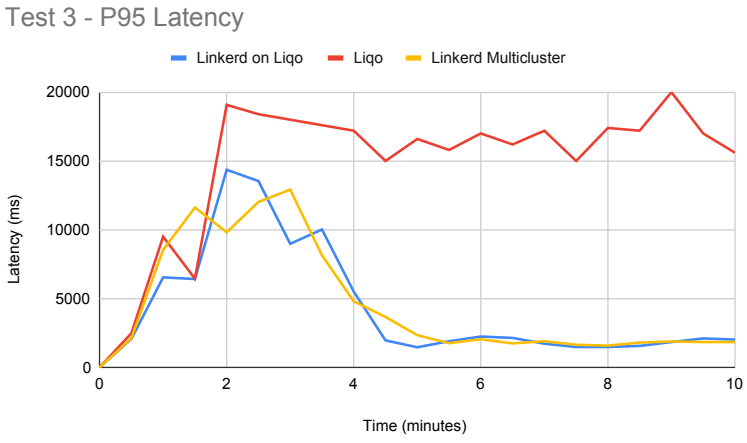

Figure 7.6: P95 Latency in third testing case

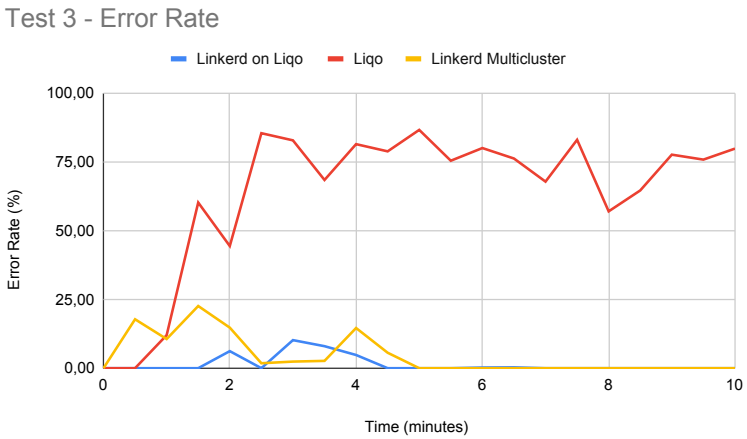

Figure 7.7: Error Rate in third testing case

In fact, while the behaviour of the other two solutions is pretty the same of the previous test cases, with only an higher peak in latency, the one on Liqo presents a very high error rate for the entire test duration, which translates in a very high latency. This time the application on Liqo vanilla is completely unusable. The question is why Liqo has this behaviour. This is due to the gRPC load balancing issue: standard Kubernetes load balancer is unable to redirect gRPC traffic, so it concentrates on a single replica even if the application scales up. With so much requests to handle, the pod fails and so latency and error rate increases. After the transitional period, values remains high because traffic concentrates only on few replicas which will not fail, but they will be overloaded.

Chapter 8

Conclusion and future work

From the tests showed in chapter 7 we can conclude that our solution perform well in scenarios with heavy load where its performance are very close to Linkerd Multi-cluster. At this point, the question can be what this prototype adds respect to Linkerd Multi-cluster, since their performance are similar. The added value of using Liqo as multicluster environment is its dynamism: it manages many clusters as one, so if you unplug a cluster pods running on it, and so on a virtual node, will be rescheduled on another node; or if you add a cluster there will a new node on which new replicas of the services can be scheduled without operator intervention.

On the contrary, with the Linkerd solution you have a static configuration: services running on a cluster are put in communication with services running on another one. If the connection broke there will not be a rescheduling or the application on a cluster cannot scale its services on the other one. Moreover, Liqo has a simple installation compared to Linkerd Multi-cluster, which needs lots of steps.

In the end, we can affirm that this prototype puts together the advantages of Liqo multi-cluster environment with the features of the Linkerd service mesh: one one side we obtain dynamism and simplicity of usage, on the other we can leverage on load balancing, metrics export, secure communications offered by a service mesh.

8.1 Future works

Starting from this thesis work there can be some future developments. A first possible work is to improve resilience of the mesh when there is a loss of connection between clusters. Imagine a scenario in which an application has replicas offloaded on various cluster and some of them has a loss of communication with the control plane. Pods running on them would be unable to communicate each other at the moment. This means if we have a complete replica of the app offloaded it will not

work. In this way, we could imagine to develop a way in which communication can continue between these pods.

Another possible work involves Istio. These technology starting from the analysis done in chapter 5 it will be possible design solutions to overcame its limitations and create another prototype. Then it can be compared with the Linkerd one developed in this work.

Bibliography

[1] *Kubernetes official documentation*. URL: https://kubernetes.io/docs/home/ (cit. on pp. 4, 11, 13, 15–18).

[2] *Virtual-kubelet git repository*. URL: https://github.com/virtual-kubelet/virtual-kubelet (cit. on pp. 4, 17, 18).

[3] *Kubebuilder git repository*. URL: https://github.com/kubernetes-sigs/kubebuilder (cit. on pp. 4, 18, 19).

[4] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015 (cit. on p. 4).

[5] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. «Omega: flexible, scalable schedulers for large compute clusters». In: *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 351–364. URL: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf (cit. on p. 4).

[6] Ferenc Hámori. *The History of Kubernetes on a Timeline*. June 2018. URL: https://blog.risingstack.com/the-history-of-kubernetes/ (cit. on p. 5).

[7] Steven J. Vaughan-Nichols. *The five reasons Kubernetes won the container orchestration wars*. Jan. 2019. URL: https://blogs.dxc.technology/2019/01/28/the-five-reasons-kubernetes-won-the-container-orchestration-wars/ (cit. on p. 5).

[8] Kalyan Ramanathan. *5 business reasons why every CIO should consider Kubernetes*. Oct. 2019. URL: https://www.sumologic.com/blog/why-use-kubernetes/ (cit. on p. 5).

[9] Eric Carter. *Sysdig 2019 Container Usage Report: New Kubernetes and security insights*. Oct. 2019. URL: https://sysdig.com/blog/sysdig-2019-container-usage-report/ (cit. on p. 7).

[10] Diego Ongaro and John Ousterhout. «In search of an understandable consensus algorithm». In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC}| 14)*. 2014, pp. 305–319 (cit. on p. 8).

[11] *Kubernetes Operator pattern*. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (cit. on p. 19).

[12] *Liqo official documentation*. URL: https://doc.liqo.io/ (cit. on pp. 21, 26).

[13] Nikla Lazzari. *Cos'è un Service Mesh*. Dec. 2020. URL: https://www.kiratech.it/blog/cosa-e-il-service-mesh (cit. on p. 28).

[14] Sachin Manpathak. *Kubernetes Service Mesh: A Comparison of Istio, Linkerd, and Consul*. Oct. 2019. URL: https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/ (cit. on p. 29).

[15] Nuno Rodriguez. *Introduction to Linkerd - A lightweight Kubernetes service mesh*. July 2019. URL: https://medium.com/ki-labs-engineering/introduction-to-linkerd-3bfc76d92dc0 (cit. on p. 30).

[16] *Linkerd 2.10 Architecture*. URL: https://linkerd.io/2.10/reference/architecture/ (cit. on pp. 31, 32).

[17] *Linkerd 2.10 Extension list*. URL: https://linkerd.io/2.10/reference/extension-list/ (cit. on p. 34).

[18] *Linkerd 2.10 Distributed Tracing*. URL: https://linkerd.io/2.10/features/distributed-tracing/ (cit. on p. 34).

[19] *Linkerd 2.10 Telemetry and Monitoring*. URL: https://linkerd.io/2.10/features/telemetry/ (cit. on p. 34).

[20] *Linkerd 2.10 Getting Started*. URL: https://linkerd.io/2.10/getting-started/ (cit. on pp. 35, 36).

[21] *Linkerd 2.10 Multi-cluster Communication*. URL: https://linkerd.io/2.10/features/multicluster/ (cit. on pp. 35, 36).

[22] Megan O'Keefe. *Welcome to the service mesh era: Introducing a new Istio blog post series*. Jan. 2019. URL: https://cloud.google.com/blog/products/networking/welcome-to-the-service-mesh-era-introducing-a-new-istio-blog-post-series (cit. on p. 36).

[23] *Istio FAQ*. URL: https://istio.io/latest/about/faq/ (cit. on p. 36).

[24] *The Istio service mesh*. URL: https://istio.io/latest/about/service-mesh/ (cit. on pp. 37, 38).

[25] *Istio Security*. URL: https://istio.io/latest/docs/concepts/security/ (cit. on pp. 37, 38).

[26] *Istio Traffic Management*. URL: https://istio.io/latest/docs/concepts/traffic-management/ (cit. on p. 38).

[27] *Istio Observability*. URL: https://istio.io/latest/docs/concepts/observability/ (cit. on p. 39).

[28] *Istio Architecture*. URL: https://istio.io/latest/docs/ops/deployment/architecture/ (cit. on p. 40).

[29] *Istio Multicluster*. URL: https://istio.io/latest/docs/setup/install/multicluster/ (cit. on p. 42).