POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Enabling Fine-Grained Security for Liquid Computing in Multi-Cluster Kubernetes Environments

Supervisor	Candidate
Prof. Fulvio RISSO	Francesco D'ANZI

October 2023

Summary

Cloud computing has revolutionized the way of deploying and managing applications. Among the numerous technologies that have emerged to facilitate cloud-native application deployment, Kubernetes stands out as a cornerstone for container orchestration, simplifying application scaling and management and providing organizations with the agility required to thrive in the cloud-native era.

While Kubernetes serves as a powerful foundation for cloud-native applications, the need for multi-cluster architectures enabling the creation of federated clusters that act as a single entity has grown, driven by requirements for geographic distribution, redundancy and diverse infrastructure resources. "Liquid computing" is a paradigm that proposes to realize multi-cluster environments, creating a continuum of computing resources. This concept is followed by Liqo, an open-source project started at Politecnico di Torino, that allows the building of multi-cluster topologies within Kubernetes.

The goal of this thesis is to enable fine-grained security for connectivity in Liqo. The current model of full pod-to-pod connectivity lacks granularity and control: to address this limitation, the thesis presents a solution that allows a single cluster in a Liqo environment to selectively contact its pods offloaded in other clusters and the endpoints of offloaded services hosted by it. This approach enhances security while maintaining the flexibility and scalability benefits of liquid computing. The implementation of this solution involves the development of two custom controllers responsible for enforcing connectivity restrictions, which manage Iptables firewall rules for each cluster, ensuring that communication occurs only within the defined constraints. Lastly, it is presented a practical use case achievable thanks to the new feature: the creation of data spaces, realized offloading workloads in the cluster that hosts data of interest.

Table of Contents

List of Figures

Chapter 1

Introduction

In the contemporary landscape of computing, the rapid evolution of technology has ushered in a paradigm shift, moving away from traditional on-premises infrastructure towards more agile, scalable, and flexible cloud-based solutions. Cloud computing, as a foundational concept, has played a pivotal role in this transition. It has redefined the way we conceive and manage computational resources, offering businesses and organizations unparalleled opportunities for innovation, cost-efficiency, and scalability.

At the heart of cloud-native computing stands Kubernetes, an open-source container orchestration platform that has gained widespread adoption. Kubernetes, often referred to as K8s, revolutionizes the deployment and management of containerized applications, providing a unified framework for automating the deployment, scaling, and operation of applications. This powerful tool has accelerated the development and deployment of software, enabling organizations to harness the true potential of the cloud.

In recent years, there has been a notable growth of cloud-native solutions to address the substantial volume of requests that large companies routinely handle in order to deliver services to millions of users. This approach, when compared to the traditional monolithic software architecture, has gained prominence. The fundamental idea behind it is to construct applications by breaking them down into numerous small, closely related and loosely connected components. These components can be managed independently, simplifying the overall application development process. When combined with containerization techniques and the capabilities provided by container orchestrators, this approach offers unparalleled ease of management, which has become widespread in the software industry.

Among the various orchestrating systems, Kubernetes has assumed a central role in the realm of cloud computing: its features, user-friendliness and robust declarative API allow users to effectively manage the dynamic nature of modern

workloads. Its open-source nature and the availability of developer tools have contributed significantly to its adoption and, more importantly, to the growth of the cloud community as a whole. Consequently, cloud-native principles are no longer limited to large corporations but are increasingly embraced by small and medium-sized businesses.

As a result, clusters are being employed across various facets of the software industry, necessitating their interconnection to fully harness their capabilities.

1.1 The Need for Multi-Cluster Environments

The prevalence of Kubernetes is closely tied to its adoption in the cloud: increasingly, more providers are constructing and offering managed clusters as a service. On the other hand, Kubernetes also enjoys popularity in on-premise environments, where it leverages its rich ecosystem to reduce the gap with public clouds. Additionally, there is a growing interest in edge setups, with a rising number of projects focused on implementing Kubernetes in lightweight, geographically dispersed infrastructures.

Following these trends, organizations no longer require isolated clusters; instead, they seek clusters that can communicate and collaborate, creating a multi-cluster environment. This approach transforms clusters into entities similar to nodes within a larger system. A fundamental requirement in this context is the ability to share resources and dynamically exchange workloads responsively.

Kubernetes, however, does not inherently support this approach. Its highest level of abstraction is the "node" entity, typically representing a single physical machine within a cluster. The essence of a multi-cluster approach in Kubernetes revolves around enabling individual clusters to share their nodes with other clusters. This idea allows for the existence of distinct clusters, as perceived by Kubernetes, while effectively creating a logical, unified cluster for enhanced cooperation and resource utilization.

Despite the increased complexity, the widespread use of multiple cluster topologies introduces new and exciting possibilities. These possibilities extend beyond the basic concept of static application orchestration across multiple clusters that has been explored thus far. In fact, multi-cluster topologies can be useful to orchestrate applications across various locations and unify access to the infrastructure. Among the others, this introduces the intriguing option of migrating an application from cluster to cluster, transparently and quickly.

1.2 Multi-Cluster and Liqo

The concept of "liquid computing" refers to the dynamic allocation of computing resources based on the needs of the user. This approach has become crucial for companies searching for quick adaptability in their operations. Liqo, an open-source project initiated at Politecnico di Torino, capitalizes on this concept to enable the creation of dynamic multi-cluster configurations within Kubernetes.

Liqo approach facilitates the connection of multiple independent clusters, allowing them to pool resources and workloads while being managed as a unified entity. By extending the Kubernetes API, Liqo enables the amalgamation of different clusters into a multi-cluster network of computing nodes. This is accomplished by establishing automatic peer-to-peer relationships that facilitate resource and service sharing among disparate and diverse clusters.

A significant advantage of Liqo is its seamless ability to offload workloads to remote peers without necessitating modifications to Kubernetes or the applications. This makes multi-cluster computing a natural and transparent process, where remote clusters are regarded as additional nodes alongside the local cluster. To facilitate communication between remote pods, Liqo provides a network framework that enables pod-to-pod connectivity across multiple clusters.

1.3 Goal of the thesis

Liqo currently adopts a full pod-to-pod connectivity model which may not always align with the security demands of the multi-cluster topology that is desired to be implemented. As a consequence the need for finer granularity and control arises: this work aims to address this gap in connectivity security, enabling fine-grained traffic control while preserving the innate flexibility and scalability benefits of Liqo, leveraging also this new feature in a PoC that shows the possibility to create data spaces.

1.4 Structure of the work

This thesis is structured as follows:

- **Chapter 2** provides an overview of Kubernetes, its architecture, and concepts;
- **Chapter 3** provides an overview of Liqo, its architecture features, and components;

- **Chapter 4** provides a presentation of advanced network concepts used in the next chapters;
- **Chapter 5** provides a description of the architectural changes to Liqo and its implementation details to enable fine-grained security in a Multi-Cluster Kubernetes environment
- **Chapter 6** describes the possibility of creating data spaces through Liqo, presenting a proof-of-concept;
- **Chapter 7** provides a performance analysis of the proposals in this thesis;
- **Chapter 8** provides conclusions about the thesis and future perspectives.

Chapter 2

Kubernetes

In this chapter, we delve into the architecture of Kubernetes, providing insight into its historical development to establish the groundwork for subsequent discussions. Kubernetes, often abbreviated as K8s, is an extensive framework, and a comprehensive examination would require substantial time and discussion. Therefore, we offer here a description of its core concepts and components, with further details available in the official documentation[1].

2.1 Kubernetes: a bit of history

The **Borg** [2] system was developed by Google in 2004: it was a small project that at first just had less than 5 people working on it and 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" [2]. In 2013 Google announced **Omega** [3], 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 2014, Google introduced Kubernetes as an open-source version of Borg. Joe Beda, Brendan Burns, Craig McLuckie, and other Google engineers developed Kubernetes. Borg had a significant impact on its creation and design, and many of its original contributors had previously worked on it. While the original Borg project was coded in C++, Kubernetes was built using the Go programming language. In the year 2015, Kubernetes version 1.0 was officially released. Concurrently, Google entered into a strategic partnership with the Linux Foundation to establish the Cloud Native Computing Foundation (CNCF) [4]. Since then,

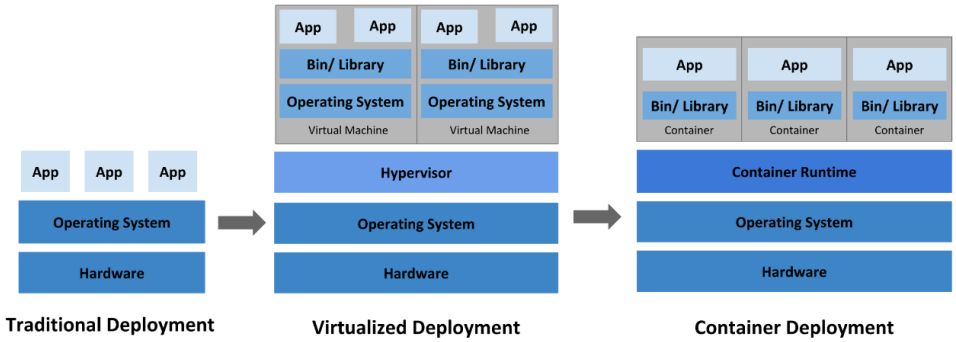


Figure 2.1: Evolution of applications deployments

Kubernetes has experienced substantial growth, achieving the CNCF graduated status and gaining widespread adoption within nearly all major corporations. At the moment, it is the de-facto dtandard for conatiner orchestration [5]

2.2 Evolution of workloads management

Traditional deployment era In the traditional deployment era, organizations ran applications on physical servers. There was no way to define application constraints to limit resource usage, and some applications would end up taking most of the resources available, making the remaining applications starve. This led system managers to deploy one server per application, increasing costs and maintenance work. At this point, the community rediscovered the abandoned concept of virtualization.
Virtualized deployment era In the virtualized deployment era, developers had the capability to operate multiple Virtual Machines (VMs) on a single physical server. This approach ensured that applications remained isolated from one another, with each application running within its dedicated VM. Virtualization provided the flexibility to define specific resource constraints for each VM, establishing robust boundaries that prevented software in one VM from interfering with the broader system or other VMs. This isolation contributed significantly to a more stable and secure environment, as applications could neither disrupt each other's operation nor freely access confidential application data. Furthermore, virtualization facilitated enhanced scalability since application instances could be readily expanded or reduced by creating or deleting VMs as needed. Each VM encapsulated a complete operating system and could be tailored to include precisely the required versioned dependencies, resulting in well-defined compartments that were straightforward to manage,

maintain, and troubleshoot. In summary, this approach reduced the deployment of physical servers, resulting in cost savings, and allowed organizations to optimize the utilization of their existing servers, thereby preventing them from sitting idle or underused

Containerized deployment era The next phase in the progression of workload deployment marked the emergence of containerization. Containers operate in a manner akin to VMs, albeit with less stringent isolation properties that enable different applications to coexist within the same Operating System. This reduced isolation grants them the "lightweight" classification. Much like VMs, containers possess their distinct file systems, CPU allocation, memory allocation, process space, and more. Containers are designed to be decoupled from the underlying infrastructure, granting them the crucial advantage of portability across various cloud environments and OS distributions.The popularity of containers stems from their set of additional benefits, such as:

- Facilitating agile application creation and deployment due to the ease of generating container images compared to VM images.
- Enabling continuous development, integration, and deployment through dependable and frequent container image builds and deployments.
- Supporting application health checks and observability.
- Offering portability across different cloud platforms and OS distributions.
- Emphasizing application-centric management, elevating the abstraction level to focus primarily on running the application.
- Optimizing resource utilization, leading to higher efficiency and density.

Concurrently, there has been a notable evolution in workload management methods. Initially, VMs were treated as individual entities, transitioning to a more generalized "cattle" model, albeit with a significant degree of coupling to their lifecycles. The evolution continued to a decoupled approach, exemplified by Kubernetes. Kubernetes employs a declarative approach that articulates general intentions for the system to apply to all relevant resources, eliminating the need to manage individual instances. This shift presents a detached perspective where resources are viewed as commodities that can be created, terminated, and replaced as necessary

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 we can see in figure 2.2, Kubernetes is by far the most used container orchestrator.

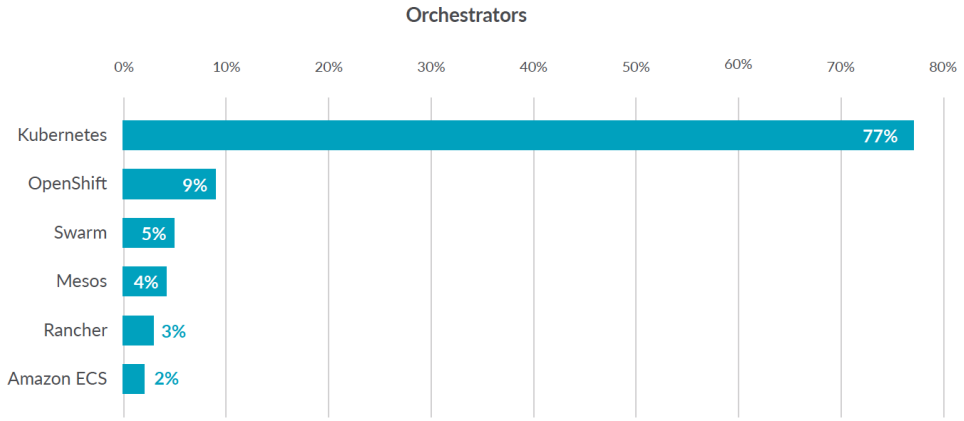


Figure 2.2: Container orchestrators use [6].

In the following, we provide a description of such system
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 creation of new containers of a deployment, remove existing containers and adopt all their resources to the new container
- **Storage orchestration** 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 **worker** 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.

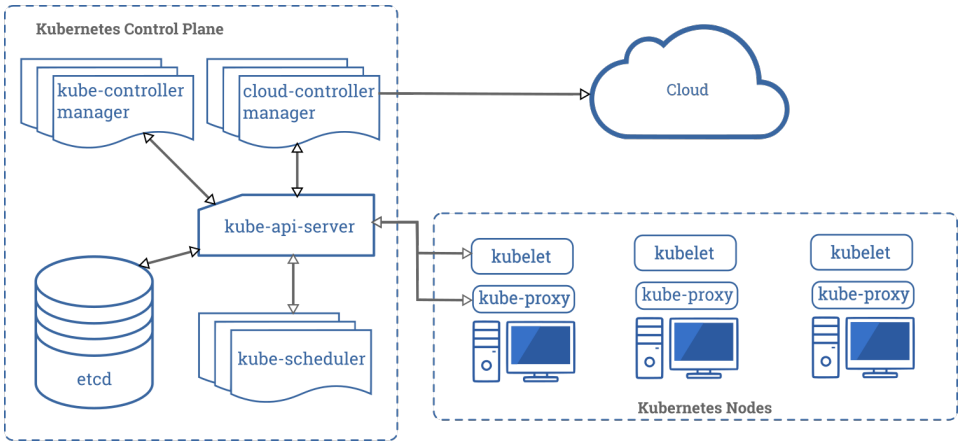


Figure 2.3: Kubernetes architecture

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, 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.
- Endpoint Controller: populates Endpoint objects (which link Services and Pods) [deprecated and substituted by the EndpointSlice API].
- EndpointSlice Controller: populates EndpointSlice objects (which link 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.

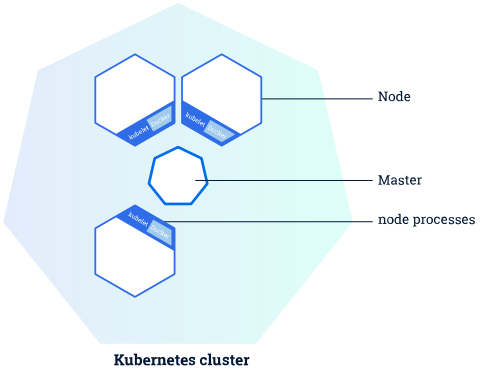


Figure 2.4: Kubernetes master and worker nodes [1].

2.5 Kubernetes objects

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

- `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 variants
  - **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 Labels and Selectors

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.

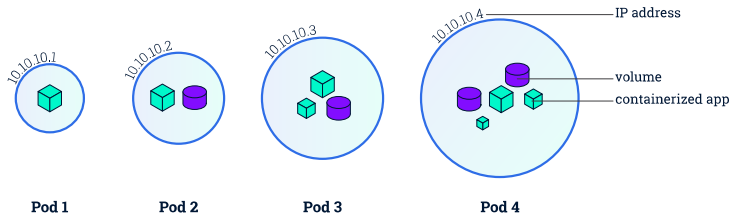


Figure 2.5: Kubernetes pods [1].

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 2.1 is an example of deployment.

Listing 2.1: Basic example of Kubernetes Deployment [1].

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - 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 DaemonSet

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created. Some typical uses of a DaemonSet are:

- running a cluster storage daemon on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

2.5.7 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

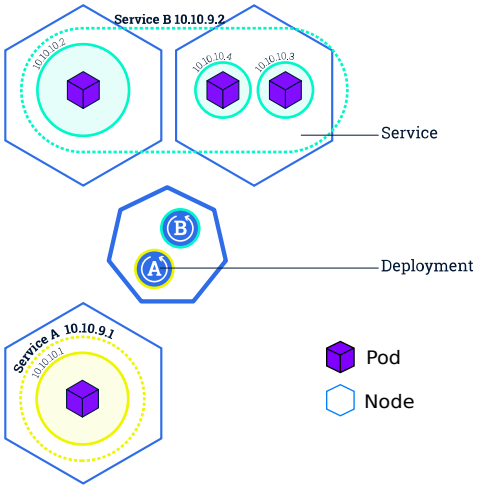


Figure 2.6: Kubernetes Services [1].

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.

2.5.8 EndpointSlice

An EndpointSlice is an abstraction that contains references to a set of network endpoints of a service. It is created by the kube-controller-manager and contains a list of IP addresses and ports for each pod that backs the service. The EndpointSlice provides an alternative that is more scalable and extensible

than the original and deprecated Endpoint resource. It tracks IP addresses, ports, readiness, and topology information for pods backing a service. Listing 2.2 shows an example of an EndpointSlice resource:

Listing 2.2: Basic example of Kubernetes Service [1].

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

2.6 Kubernetes network architecture

Kubernetes defines a network model that helps provide simplicity and consistency across a range of networking environments and network implementations. The Kubernetes network model provides the foundation for understanding how containers, pods, and services within Kubernetes communicate with each other [7]. The Kubernetes network model specifies:

1. Every pod gets its own IP address
2. Containers within a pod share the pod IP address and can communicate with each other freely
3. Pods can communicate with all other pods in the cluster using pod IP addresses (without NAT)
4. Agents on a node (e.g., system daemons, kubelet) can communicate with all pods on that node
5. Pods on a node's host network can communicate with all pods on all nodes (without NAT)
6. Isolation (restriction of what each pod can communicate with) is defined using network policies

As a result, pods can be treated much like VMs or hosts (they all have unique IP addresses), and the containers within pods very much like processes running

within a VM or host (they run in the same network namespace and share an IP address). This model makes it easier for applications to be migrated from VMs and hosts to pods managed by Kubernetes. In addition, because isolation is defined using network policies rather than the structure of the network, the network remains simple to understand. This style of network is sometimes referred to as a "flat network"

2.6.1 Container communication within same pod

Containers in a Pod are accessible via `localhost`, they use the same network namespace. For containers, the observable host name is a Pod's name. Since containers share the same IP address and port space, different ports in containers for incoming connections must be used. Because of this, applications in a Pod must coordinate their usage of ports.

2.6.2 Pod communication within the same node

Before the infrastructure container is started, a virtual Ethernet interface pair (a `veth` pair) is created for the container. One interface of the `veth` pair stays in the host's namespace (it tagged with `vethxxx`) while the other interface is moved into the container's network namespace and renamed to `eth0`. These two virtual interfaces are like two ends of a pipe that everything goes in one side, comes out on the other.The interface in the host's network namespace is attached to a network bridge that container runtime is configured to use. The `eth0` interface in the container is assigned an IP address from the bridge's address range. Anything that application running inside the container sends to the `eth0` network interface and comes out at the other `veth` Interface in host's namespace and is sent to bridge. So, any network connected to the bridge can receive it.

2.6.3 Pod communication on different nodes

Pod IP addresses must be unique across the whole cluster, so the bridges across the nodes must use non-overlapping address ranges to prevent pods from different nodes from receiving the same IP address. There are many methods to connect bridges on different nodes. This can be done with overlay or underlay networks, or through regular Layer 3 routing (direct routing).

2.6.4 CNI (Container Network Interface)

CNI (Container Network Interface) represents a project within the Cloud Native Computing Foundation (CNCF). This project comprises a specification and

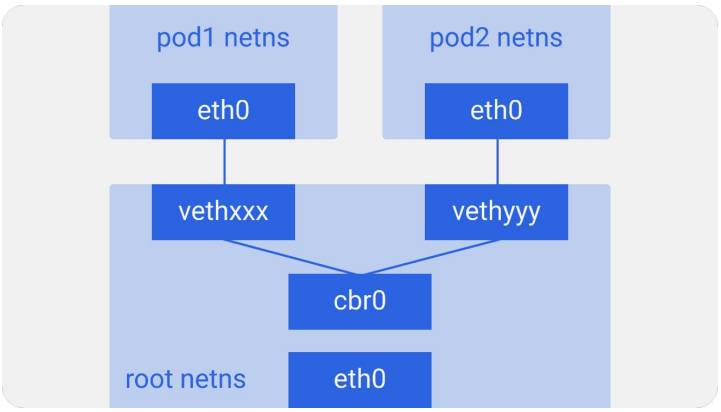


Figure 2.7: Pod to pod communication within same node.

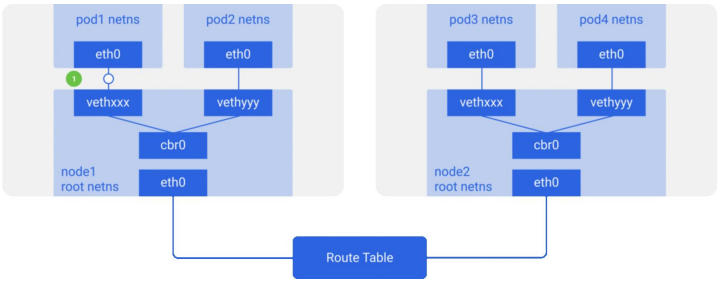


Figure 2.8: Pod to pod communication across different nodes.

libraries that enable the creation of plugins responsible for configuring network interfaces within Linux containers. CNI's primary focus is on managing the network connectivity of containers and ensuring that allocated resources are properly reclaimed when a container is deleted. In the context of Kubernetes, CNI specifications and plugins are utilized to orchestrate networking. Kubernetes, through CNI, can communicate with other containers' IP addresses without relying on Network Address Translation (NAT). Whenever a Pod

is initialized or removed, the default CNI plugin is invoked with its default configuration. This default CNI plugin creates a pseudo interface, attaches it to the underlying network, assigns an IP address, configures routes, and links it to the Pod's namespace. When launching the Kubelet, it is important to specify the use of the CNI plugin by including the flag `-networkplugin=cni`. If the environment does not utilize the default configuration directory located at `/etc/cni.net.d`, the CNI plugin can receive the appropriate configuration directory as a value using the `-cni-conf-dir` flag. Additionally, the Kubelet expects to find the CNI plugin binary at `/opt/cni/bin`, but an alternative location can be designated using the `-cni-bin-dir` flag.

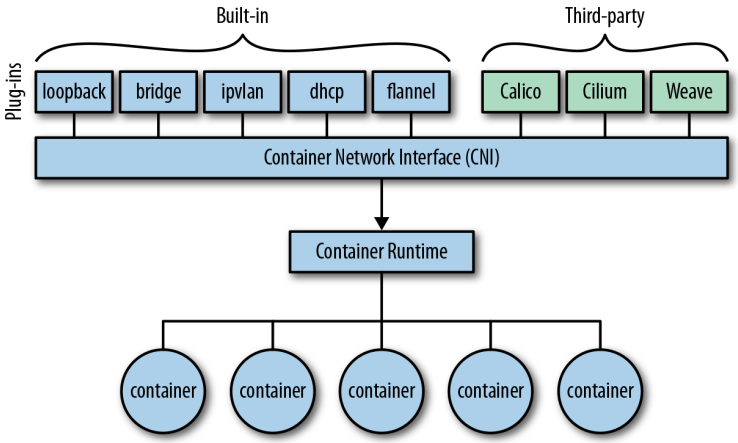


Figure 2.9: Container network interface [8]

2.6.5 Pod to service networking

Pod IP addresses are inherently transient and can fluctuate unpredictably in response to various events such as scaling operations, application failures, or node reboots. These events have the potential to cause Pod IP addresses to change without any warning. To tackle this challenge, a Kubernetes Service effectively manages the state of Pods, enabling the monitoring of a dynamic set of Pod IP addresses that may evolve over time. Services serve as an abstraction layer above Pods and assign a single virtual IP address to a collection of Pod IP addresses. Any traffic directed towards the virtual IP address associated with a service will be automatically routed to the group of Pods that are linked to that virtual IP. This architectural design permits the composition of Pods associated with a service to undergo modifications at any given moment, all while ensuring that clients only need to be aware of the unchanging virtual IP address of the service [9].

2.7 Kubebuilder

Kubebuilder [10] is a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs)

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. 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:

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

This chapter presents the fundamnetal concepts of **Liqo** [12] and the core components that constitute its architecture.

3.1 An overview of Liqo

Kubernetes technology plays a prominent role in managing cloud-related tasks. Cluster configurations are typically designed to provide an abundance of resources, including significant computing power, ample memory, and extensive storage capacity, often surpassing the immediate requirements for handling temporary spikes in workload. This surplus capacity represents an untapped potential that can be harnessed by other clusters experiencing lower resource demands during specific timeframes. Liqo's primary objective is to unlock this latent power by establishing connections between clusters, enabling them to collaborate effectively in pursuit of their objectives.

To achieve this objective, clusters initiate peering sessions that result in the formation of a larger virtual cluster that collectively aggregates the resources exposed by each participating cluster involved in the peering process.

The advantage of Liqo stands in its ability to build upon the core concepts well-known within the Kubernetes ecosystem, thus extending its capabilities. In essence, a cluster views its peers as virtual nodes that seamlessly add up to its physical ones, permitting task scheduling on these nodes without regard to their actual origins or nature.

The next sections will provide a more in-depth exploration of these concepts, commencing with a focus on a core element and its setup: the Liqo peering mechanism.

3.2 Liqo Peering

When two or more Kubernetes clusters are ready to host workloads, they can participate in a multi-cluster setup by initiating a peering session between them. This marks the initiation of the Liqo experience. A Liqo peering operation combines distinct entities, uniting them into a broader environment with the capacity to manage more extensive workloads effectively. Consequently, each participating cluster becomes aware of the presence of other remote peers, represented by the **ForeignCluster** Custom Resource (CR). This process involves the exchange of network parameters and other cluster-related information, establishing a secure VPN that pods will utilize for intercommunication within a large, distributed cross-cluster application.

Cluster peerings are not required to be symmetric. Their flexibility allows a cluster to establish:

- an **outgoing peering**, so that the cluster can offload its workloads, but won't receive any by its pee
- an **incoming peering**, so that the cluster hosts remote workloads, but won't offload any to its peer
- a **bidirectional peering**, the union of the two above

When an outgoing peering is active, it is of absolut importance to control what can and cannot be offloaded. This is done by leveraging some native Kubernetes concepts, namely Namespaces and label selectors, as well as some logic provided by Liqo to select which namespaces to offload, which pods within those namespaces to offload, and even which remote peers are the target of this offloading mechanism: the possibilities are endless. The basic requirement for starting a peering session is to have access to the remote Kubernetes API server. This allows clusters to exchange information and create resources remotely. The result is a VPN that remote pods use to communicate as if they were all in the same Kubernetes cluster.

3.3 Liqo Reflection

After the establishment of a peering connection, the capacity for workload offloading is activated through the utilization of the virtual node concept and namespace extension.

A virtual node serves as a representation of a remote cluster, encapsulating all its shared resources like CPU and memory. This mechanism allows for a seamless expansion of the local cluster's available resources. When a virtual

node is integrated into the cluster, it is automatically considered by the standard Kubernetes scheduler when determining the optimal location for executing workloads.

Furthermore, Liqo facilitates the extension of Kubernetes namespaces beyond cluster boundaries. When a specific namespace is chosen for offloading, Liqo automatically generates corresponding twin namespaces within the chosen subset of remote clusters. These remote twin namespaces serve as hosts for the offloaded pods and any other resources that belong to the local namespace being extended remotely. This encompasses elements related to service exposure, such as Ingress, Service, and Endpoints resources, as well as the storage of configuration data, including ConfigMaps and Secrets, among others.

3.4 Network Fabric

The network fabric within Liqo is a subsystem that seamlessly extends the Kubernetes network model across multiple autonomous clusters. This extension enables offloaded pods to interact with one another as if they were all executing within the same local cluster.

In particular, the network fabric ensures that all pods within a specific cluster can communicate with every pod across all the remotely peered clusters. This communication can occur with or without the need for Network Address Translation (NAT) translation. Since Liqo accommodates arbitrary clusters with varying parameters and components (e.g., CNI plugins), it becomes impossible to guarantee non-overlapping pod IP address ranges (i.e.,PodCIDR). Therefore, when address ranges do not overlap, it may be necessary to employ address translation mechanisms to enable NAT-less communication, which is preferred.

Figure 3.1 provides a high-level illustration of the network fabric established between two clusters, with the key components elaborated upon in the next sections.

3.4.1 Cross-cluster VPN tunnels

The linkage between peered clusters is established using secure VPN tunnels created with **WireGuard**. These tunnels are dynamically established once the peering process is complete, based on the negotiated parameters.

The responsibility for setting up these tunnels falls upon the **Liqo gateway**, a crucial component of the network fabric. This gateway runs as a privileged pod on one of the nodes within the cluster. Additionally, it takes care of populating the routing table and configuring NAT rules, leveraging **iptables**, as necessary to address address conflicts.

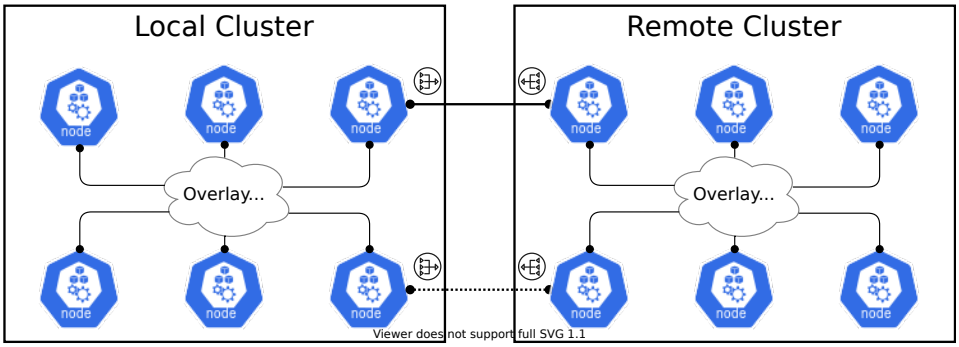


Figure 3.1: Network Fabric

Despite executing within the host network, this component relies on a separate network namespace and policy routing to ensure isolation and prevent conflicts with the existing Kubernetes CNI plugin. Furthermore, it supports an active/standby high-availability configuration to minimize downtime in the event that the primary replica needs to be restarted.

3.4.2 In-cluster overlay network

The overlay network plays a crucial role in routing all traffic that originates from local pods or nodes and is destined for a remote cluster. This traffic is directed to the gateway, where it enters the VPN tunnel. Conversely, on the other end, traffic exiting the VPN tunnel enters the overlay network to reach the node hosting the target pod.

Liqo employs a VXLAN-based configuration for this purpose, which is set up by a network fabric component executed on all physical nodes within the cluster. This component functions as a DaemonSet, ensuring uniform configuration across all nodes, and is also responsible for managing the necessary routing entries to ensure accurate traffic forwarding.

3.5 Liqo CRDs

The following subsections present some of the Custom Resources that allow the peering and reflection features in Liqo.

3.5.1 NetworkConfig CR

This Custom Resource serves as a representation of a set of network parameters, primarily IP addresses, that are used by clusters to understand how a remote

peer has reconfigured the local PodCIDR and to ascertain the remote peer's PodCIDR. The `spec` section contains information pertaining to the local cluster, while the `status` section records any modifications made to the specifications. The concept involves one cluster creating this CR and transmitting it to the remote cluster with which it intends to establish a peering relationship. The remote cluster processes this CR and annotates the `status` section with all the adjustments made to IP address ranges to prevent conflicts. These updates are then relayed back to the originating cluster.

Simultaneously, a similar process occurs in the reverse direction. The remote cluster generates a NetworkConfig, fills in its `spec` section, and transmits it to the local cluster. The local cluster, in turn, annotates any changes in the `status` section to inform the remote cluster of any modifications to the original specifications.

After both CRs have been processed, a control loop within Liqo reconciles them to generate the TunnelEndpoint CR.

3.5.2 TunnelEndpoint CR

This CR contains the relevant network configuration to establish a VPN tunnel with the remote cluster. Thanks to this, pods are able to reach other remote pods as if they were on the same network.

3.5.3 ForeignCluster CR

This CR represents a remote cluster. It contains details about the peering session that is in place between two clusters, such as whether the peering was successfully established and in which direction it is going (outgoing, incoming, or both). A ForeignCluster is created starting from the NetworkConfig that the two parties have exchanged and processed.

3.5.4 ShadowPod CR

When a pod is scheduled on a virtual node, in parallel a pod is created in the remote cluster for the actual workload execution. In the remote cluster, a new object paired with the remote pod is created: the ShadowPod. This resource, combined with its controller, guarantees the presence of the pod in the remote cluster, even in case of connection failures.

3.6 Liqo components

3.6.1 CRD Replicator

This component is dedicated to the reflection of some Liqo CRs just presented. To do so, it requires access to the remote API Server. It is a core element as it implements the network parameter exchange between clusters to set up the TunnelEndpoint CRs which will later be used respectively to keep track of the active peering sessions and to ensure remote pod-to-pod communications. The replicated CRDs are:

- NetworkConfig
- ResourceRequest
- ResourceOffer
- NamespaceMapping

The architecture of the CRD Replicator is complex, but essentially it is reflector: a data structure that contains the necessary objects and data required to detect changes within both local and remote namespaces. It accomplishes this by utilizing local and remote informers. Additionally, the reflector is responsible for executing the conventional CRUD (Create, Read, Update, Delete) operations within these namespaces, employing local and remote clients for this purpose. To elaborate further, when an object, such as a NetworkConfig, is generated within a namespace that has been enabled for reflection and is equipped with the appropriate metadata labels, the local reflector (associated with the cluster that initiated the object's creation) follows these steps:

1. it detects a new object to be reflected
2. it creates a copy of that object in the remote namespace by using a preconfigured client to access the remote API serve
3. it listens to any changes occurring in the reflected object, which usually boils down to a status update performed by the remote cluster controllers, as happens with NetworkConfig to let the sender cluster know about possible remappings
4. it listens to any changes occurring in the local original copy, such as a deletion that needs to propagate to the remote cluster's namespace so that the remote copy gets deleted as well

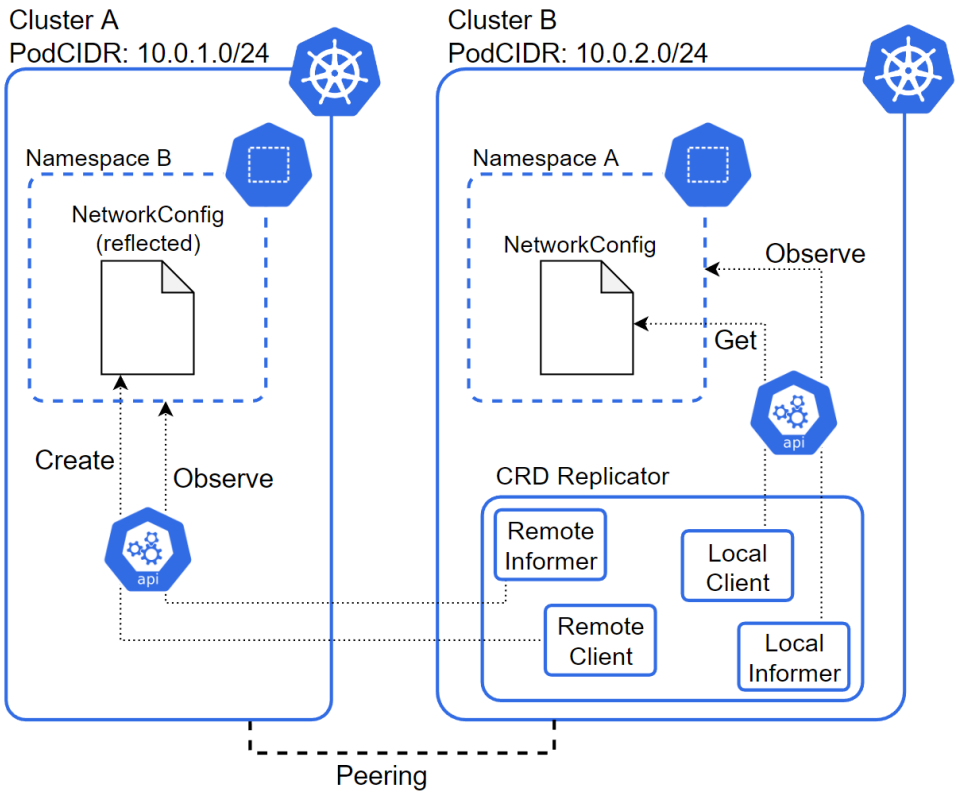


Figure 3.2: CRD Replicator

3.6.2 Virtual Kubelet

This component is a tailored adaptation of the Virtual Kubelet project [13]. Whenever a peering session is initiated with a remote cluster, a distinct instance of this component is generated. Once instantiated, it serves a dual purpose: first, it facilitates the offloading of pods to remote clusters, which are perceived by the Kubernetes control plane as standard cluster nodes suitable for scheduling regular tasks. Additionally, this component plays a role in reflecting essential Kubernetes resources, including Services and Endpoints. When deployed within a Liqo-enabled namespace, specifically one extended to a remote location, these resources are continually mirrored to the designated remote peers.

3.6.3 IPAM

This component contains the logic that translates IP addresses back and forth and keeps track of all the possible remappings between the local cluster and the remote peers. It is fundamental within Liqo as it knows all the NAT rules that are used to avoid address conflicts.

3.6.4 Network manager

The network manager serves as the control plane of the Liqo network fabric. It operates as a pod and assumes responsibility for negotiating connection parameters with each remote cluster during the peering procedure. Within its framework, it incorporates an IP Address Management (IPAM) plugin, designed to address potential network conflicts by defining high-level NAT rules (which are enforced by data plane components). Furthermore, it provides an interface that the reflection logic utilizes to manage the remapping of IP addresses. More specifically, this functionality is employed for the translation of pod IPs (for instance, during the synchronization process from the remote cluster to the local cluster) and also for EndpointSlices reflection (which involves propagation from the local to the remote cluster).

3.6.5 Liqo Gateway

This component takes on the role of managing connections with other clusters. All traffic flowing between two clusters engaged in peering must traverse through this component. While it is possible to have multiple Liqo Gateways, only one can be active at a time, with the others serving as backup options in case of failures. The establishment and management of connections between clusters are executed through VPN tunnels, and this component is responsible for their administration. Liqo offers support for various VPN drivers (such as Wireguard, OpenVPN, IPSec), providing an interface for implementing the necessary logic. However, currently, the only implemented driver is Wireguard.

Chapter 4

Advanced Networking Concepts and Tools

In this chapter are explained some networking concepts and technologies used for the implementation of the thesis work.

4.1 Linux Namespaces

Namespaces are a feauture within the Linux kernel designed to segment kernel resources in a way that one group of processes perceives a specific set of resources, while another group of processes perceives an different set of resources. This functionality operates by employing the same namespace for a given set of resources and processes, yet these namespaces point to distinct and separate resources. It is important to note that resources can exist in multiple namespaces simultaneously. Examples of such resources encompass process IDs, hostnames, user IDs, file names, certain identifiers linked to network access, and mechanisms for interprocess communication. In the realm of Linux containers, namespaces constitute a foundational element.

4.1.1 Namespace kinds

Since kernel version 5.6, there are 8 kinds of namespaces. Namespace functionality is the same across all kinds: each process is associated with a namespace and can only see or use the resources associated with that namespace, and descendant namespaces where applicable. This way each process (or process group thereof) can have a unique view of the resources. Which resource is isolated depends on the kind of namespace that has been created for a given process group. Namespace kinds are:

- **Mount namespace**: controls mount points
- **PID namespace**: provides processes with an independent set of process IDs
- **Network namespace**: allows Linux network stack to behave in isolated groups
- **IPC namespace**: allows processes to have separated IPC
- **UTS namespace**: allows a single system to appear to have different host and domain names for different processes
- **User namespace**: related to user privileges
- **Control group namespace**: hides the identity of the cgroup of which process is a member
- **Time namespace**: allows processes to see different system times

4.2 Linux Network Stack

4.2.1 Netfilter

The Netfilter framework within the Linux kernel is the basic building block on which packet selection systems like Iptables or the newer Nftables are built upon. It provides a bunch of hooks inside the Linux kernel, which are being traversed by network packets as those flow through the kernel (see figure 4.1). Other kernel components can register callback functions with those hooks, which enables them to examine the packets and make decisions on whether packets shall be dropped, accepted, or modified.

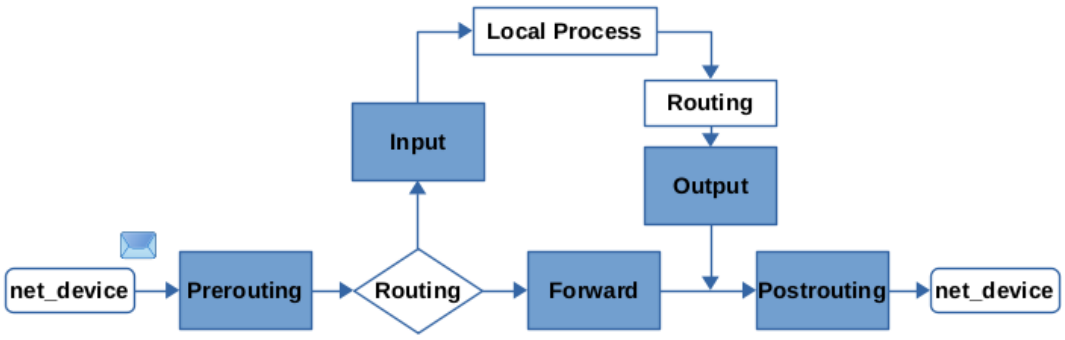


Figure 4.1: Netfilter stack overview

A network packet received on a network device first traverses the Prerouting hook. Then the routing decision happens and thereby the kernel determines whether this packet is destined at a local process (e.g. socket of a server listening on the system) or whether the packet shall be forwarded (in that case the system works as a router). In the first case, the packet then traverses the Input hook and is then given to the local process. In the second case, the packet traverses the Forward hook and finally the Postrouting hook, before being sent out on a network device. A packet that has been generated by a local process (e.g. a client or server software that likes to send something out on the network), first traverses the Output hook and then also the Postrouting hook, before it is sent out on a network device.

4.2.2 Iptables

Iptables is a user-space utility program that allows a system administrator to configure the IP packet filter rules of the Linux kernel firewall. Iptables uses tables to organize its rules. These tables classify rules according to the type of decisions they are used to make. For instance, if a rule deals with network address translation (nat), it will be put into the nat table. Within each Iptables table, rules are further organized within separate "chains". While tables are defined by the general aim of the rules they hold, the built-in chains represent the netfilter hooks which trigger them, determining when rules will be evaluated in a packet's delivery path (4.1).

A fixed set of tables exists, each table containing a fixed set of chains. When a packet traverses a netfilter hook, the sequence in which each table's chain related to the specific hook is called (i.e. priority) is also already fixed. Table 4.1 shows an overview of these information.

Table	**Contains chains**
raw	PREROUTING, OUTPUT
mangle	PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING
nat	PREROUTING, (INPUT), OUTPUT, POSTROUTING
filter	INPUT,FORWARD,OUTPUT

Table 4.1: Iptables: tables in priority descending order and related chain

Iptables rules are placed within a specific chain of a specific table. As each chain is called, the packet in question will be checked against each rule

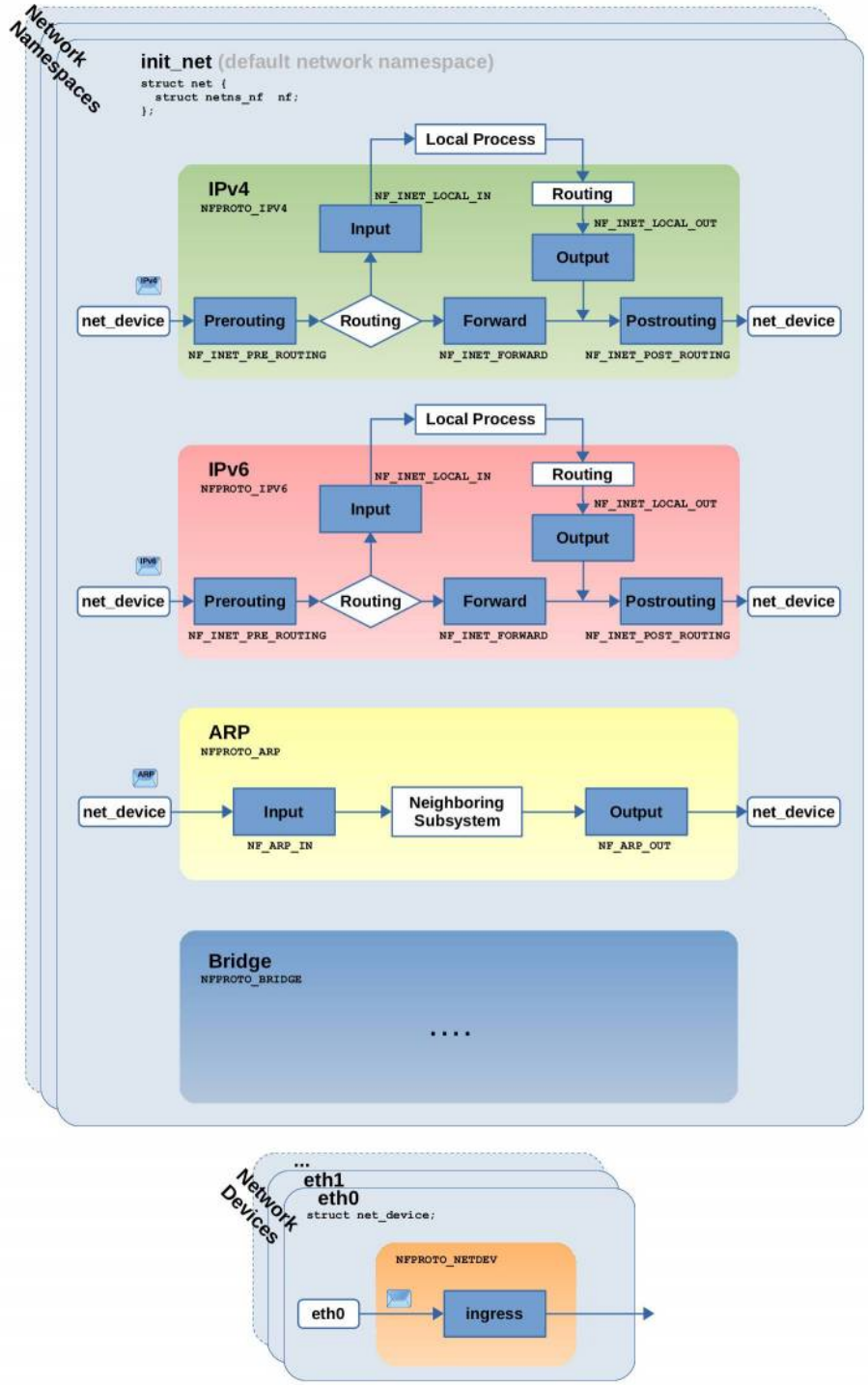


Figure 4.2: Netfilter stack details overview

within the chain in order. Each rule has a matching component and an action component. The matching portion of a rule specifies the criteria that a packet must meet in order for the associated action (or "target") to be executed. Rules can be constructed to match by protocol type, destination or source address, destination or source port, destination or source network, input or output interface, headers, or connection state among other criteria. These can be combined to create complex rule sets to distinguish between different traffic. Instead, the action portion of a rule refers to the actions that are triggered when a packet meets the matching criteria of a rule. Targets are generally divided into two categories:

- **Terminating targets**: perform an action which terminates evaluation within the chain and returns control to the netfilter hook. Depending on the return value provided, the hook might drop the packet or allow the packet to continue to the next stage of processing.

- **Non-terminating targets**: perform an action and continue evaluation within the chain. Although each chain must eventually pass back a final terminating decision, any number of non-terminating targets can be executed beforehand.

There is also a special class of non-terminating target, the jump target, actions that result in moving to a different chain for additional processing: Iptables allows to create user-defined chains, possible to reach only by "jumping" to them from a rule (they are not registered with a netfilter hook). Rules can be placed in user-defined chains in the same way that they can be placed into built-in ones. This construct allows for greater organization and provides the framework necessary for more robust branching.

4.2.3 Ipset

Ipset is a user-space utility program that is used to set up, maintain and inspect so called IP sets in the Linux kernel. Depending on the type of the set, an IP set may store IP(v4/v6) addresses, (TCP/UDP) port numbers, IP and MAC address pairs, IP address and port number pairs, etc. IP sets can be used via the *set* match in iptables rules, specifying between source or destination which IP address or port to use from the packet to match the given set. One of the standout features of IPset is its ability to efficiently perform this match operation: by utilizing data structures that optimize searches, such as hash tables and bitmap maps, IPset ensures that the lookup process is both fast and resource-efficient.

4.2.4 Connection tracking

Connection tracking, often referred to as "conntrack," is a core functionality of the Linux kernel. It is a method for monitoring and maintaining state information about active network connections. Connection tracking uses a table containing information about each connection, such as source and destination IP addresses, port numbers, and the current state; in fact, it assigns a state to each network connection, with reference to the nature of the traffic. The connection states are:

- **NEW**: represents the start of a new connection
- **ESTABLISHED**: denotes a connection with an existing flow of traffic
- **RELATED**: used for connections related to an established one
- **INVALID**: marks packets that are invalid or cannot be classified
- **UNTRACKED**: packets that are not subject to connection tracking

Connection tracking is closely integrated with Iptables and it is possible to create more sophisticated rules that take into account the state of connections, using it as a condition for allowing or denying access.

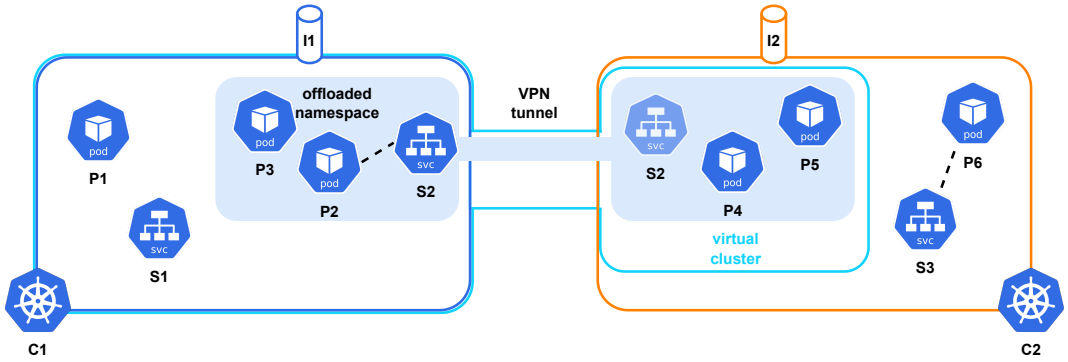
Chapter 5

Fine-Grained Security for Intra-Cluster Connectivity in Liqo

This chapter presents in details the proposed solution to enable fine-grained security for connectivity in Liqo and its implementation.

5.1 The Problem

As described in section 3.4, the Liqo network fabric is in charge of transparently extending the Kubernetes network model across multiple independent clusters, such that offloaded pods can communicate with each other as if they were all executed locally. Traditionally, Kubernetes guarantees that pods on a node can communicate with all pods on any node without NAT translation. Liqo broadens this requirement, ensuring all pods in a given cluster can communicate with all pods on all remote peered clusters, either with or without NAT translation. Indeed, the transparent support for arbitrary clusters, with completely uncoordinated parameters and components (e.g., CNI) makes impossible to guarantee non-overlapping pod IP address ranges (i.e., PodCIDR). This requires the support for IP translation mechanisms, provided that NAT-less communication is preferred whenever address ranges are disjointed. The figure 5.1 shows an example of a possible 2-cluster topology created by Liqo, resuming also standard connectivity behaviour in the related table.



	P1	P2	P3	P4	P5	P6	S1	S2	S3	I1	I2
P1		YES	YES	YES	YES	YES	YES	YES	NO	YES	NO
P2	YES		YES	YES	YES	YES	YES	YES	NO	YES	NO
P3	YES	YES		YES	YES	YES	YES	YES	NO	YES	NO
P4	YES	YES	YES		YES	YES	NO	YES	YES	NO	YES
P5	YES	YES	YES	YES		YES	NO	YES	YES	NO	YES
P6	YES	YES	YES	YES	YES		NO	YES	YES	NO	YES

Figure 5.1: Liqo full pod-to-pod connectivity between 2 clusters; S1 expose P1, S2 expose P2, S3 expose P6

5.1.1 Full pod-to-pod connectivity

This **full pod-to-pod** connectivity provided by Liqo network fabric has a drawback: it enables to seamlessly contact offloaded pods or multi-cluster service, but in the meanwhile it also compels clusters engaged in an active peering session to allow all other possible traffic between them. Such condition might not be desirable in some cases. For example, looking at the topology in figure 5.1:

- cluster C2 may wish to share its computational resources with Cluster C1, thus enabling the offloading of the namespace containing pods P4 and P5 and granting C1 the capability to access these two pods, while preventing C1's local pods, such as P3, from reaching their own pods like P6, which are not taking part of this multi-cluster environment

- conversely, on the other hand, C1 might want to allow pods from C2, such as P6, to access its own service S2 (and thus its associated endpoint P2) through offloading, but without granting P6 the ability to contact its own

pods like P1.

5.2 Architecture

This section provides a detailed presentation of the developed architecture: it is based on filtering through Iptables (see section 4.2.2) and Ipset (see section 4.2.3 and the focus will be on how to use these tools to enable fine-grained security for intra-cluster connectivity

5.2.1 Problem Area

The developed architecture will be about the **cross-cluster area** of the Liqo network (see figure 5.2). The main component is the **gateway**(see section 3.6.5), a pod where all the traffic from the other peered clusters arrives and where it could potentially be filtered. The connections between gateways are called **peers**, each connection between two clusters has its own peer and each peer is independent from the others. They can be created and destroyed without affecting the connectivity towards other clusters. This is one of the fundamental concepts which stands behind the creation and deletion of **peerings**.

5.2.2 Intra-cluster traffic segregation

The design involves establishing a dynamic management of Iptables filtering rules within the cluster gateway, following a **whitelist** approach which allows traffic exclusively related to pods and services that are engaged in the multi-cluster topology. To implement this functionality, we can focus on two separate operations that must be carried out by the gateway:

- allowing a remote cluster to contact exclusively its **pods offloaded** on the local one
- enabling a remote cluster to consume **services offloaded** on it from the local cluster

Offloaded pods

Looking at the example in the figure 5.3, C2 allows C1 to contact, through its local pods P1, P2, and P3, only the offloaded pods P4 and P5 while preventing traffic towards P6. The gateway of C2 should, therefore, have an Iptables rule for each offloaded pod, accepting packets with the destination IP address of the offloaded pod and the source IP address within C1 PodCIDR. In practice, for optimization purposes, a single rule is used. This rule, based on the packet's

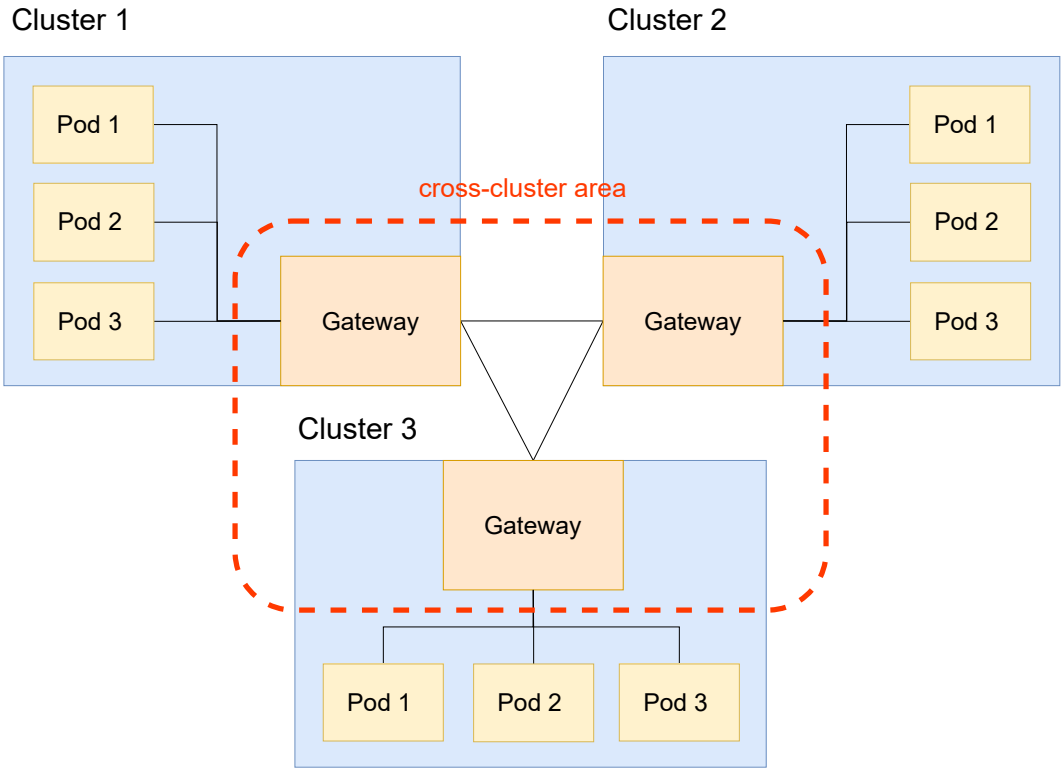
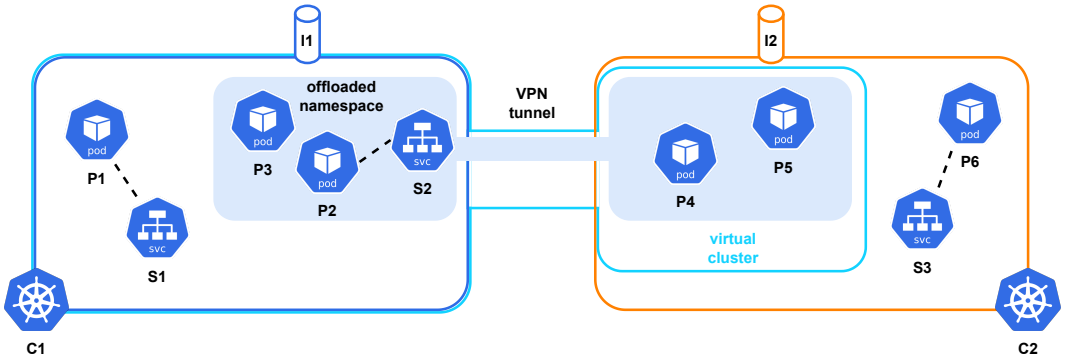


Figure 5.2: Liqo network cross-cluster area in a 3 clusters setup

destination IP address, searches for a match within an Ipset created following the offloading of the first pod. At the moment, this Ipset contains the IP addresses of P4 and P5. The Ipset is then updated in accordance with the offloading or unoffloading of pods by C1 on C2.

It is worth noting that, from the reverse perspective, P4 and P5 cannot initiate a connection to C1 for security reasons, as they are physically scheduled on another cluster, even though they logically belong to C1. However, the passage of response traffic is ensured after they have been contacted by C1, made possible thanks to an Iptables rule that match the **ESTABLISHED** connection tracking state (see section 4.2.4) which, in fact, identify incoming packets that are part of an established connection, allowing response traffic; furthermore, this rule match also **RELATED** conntrack state, useful to allow possible traffic from a new connection, but associated with an existing one, e.g. an FTP data transfer, or an ICMP error.



	P1	P2	P3	P4	P5	P6	S1	S2	S3	I1	I2
P1		YES	YES	YES	YES	NO	YES	YES	NO	YES	NO
P2	YES		YES	YES	YES	NO	YES	YES	NO	YES	NO
P3	YES	YES		YES	YES	NO	YES	YES	NO	YES	NO
P4	NO	NO	NO		YES	YES	NO	NO	YES	NO	YES
P5	NO	NO	NO	YES		YES	NO	NO	YES	NO	YES
P6	NO	NO	NO	YES	YES		NO	NO	YES	NO	YES

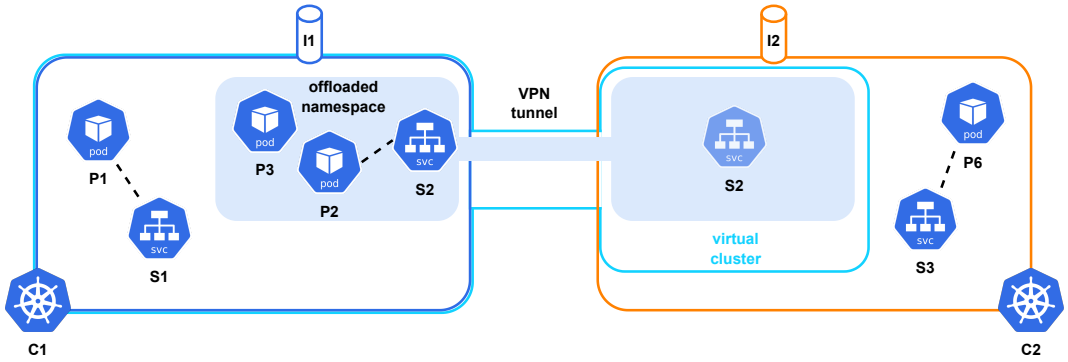
Figure 5.3: Intra-cluster traffic segregation with offloaded pods; S1 expose P1, S2 expose P2, S3 expose P6; yellow cells in the table highlights the differences compared to standard Liqo behaviour

Offloaded services

To ensure that a remote cluster can consistently access offloaded services from the local cluster, the gateway must intervene when the pods serving as endpoints for such services are not located on that remote cluster. In light of this condition, we encounter two potential scenarios for the placement of these pods:

- **On the local cluster**: in the example shown in the figure 5.4, remote pod P6 can contact exclusively P2 on C1
- **Offloaded on another remote cluster**: the figure 5.5 shows an example in which C1 gateway prevents connection with local pods P1 and P3, instead allowing traffic towards P2, which is offloaded on C3, as it serves as an endpoint for S2.

To achieve these behaviours, C1 gateway has a rule that accepts packets with the source IP address within C2 PodCIDR and destination IP address that



	P1	P2	P3	P6	S1	S2	S3	I1	I2
P1		YES	YES	NO	YES	YES	NO	YES	NO
P2	YES		YES	NO	YES	YES	NO	YES	NO
P3	YES	YES		NO	YES	YES	NO	YES	NO
P6	NO	YES	NO		NO	YES	YES	NO	YES

Figure 5.4: Intra-cluster traffic segregation with offloaded service and local endpoint; S1 expose P1, S2 expose P2, S3 expose P6; yellow cells in the table highlights the differences compared to standard Liqo behaviour

match within an Ipset, collecting all the endpoints of service S2 that a C2 pod, like P6, needs to consume S2.

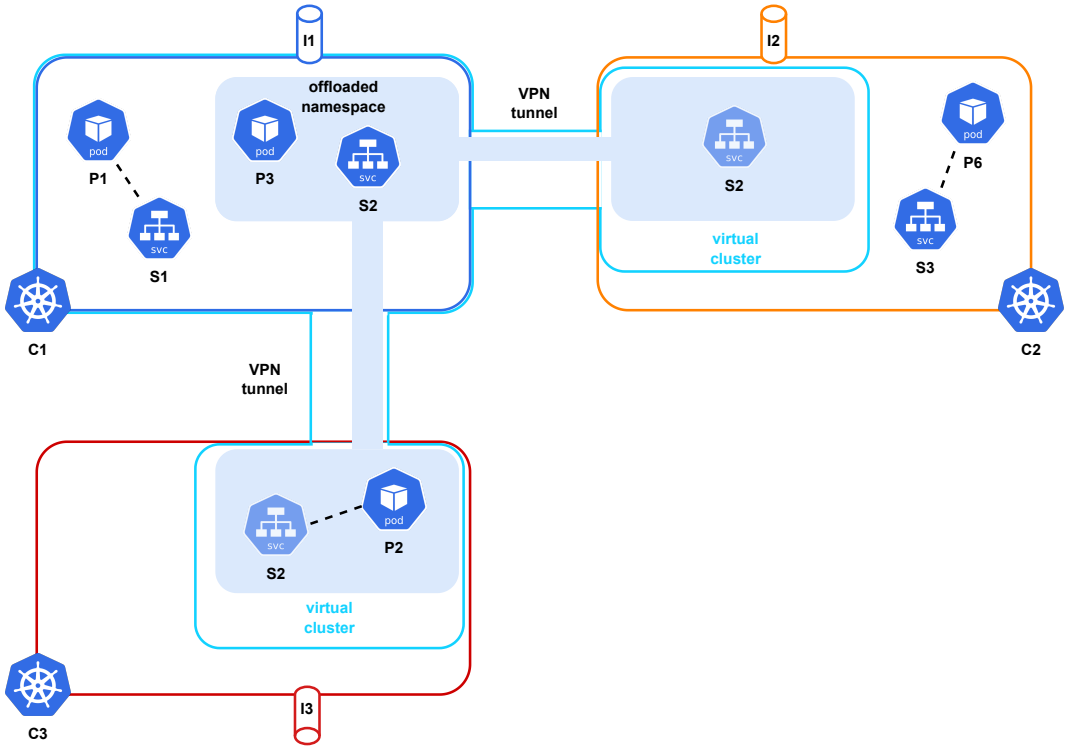
5.3 Implementation

This section delves into the implementation of the proposed architecture, describing how the gateway is capable of achieving the behaviors previously outlined.

5.3.1 Liqo NetNS

The **gateway** is implemented utilizing **Linux network namespaces** (see section 4.1) and is contained within the **liqo-gateway** component, which operates as a pod in the Liqo namespace.

Typically, when a pod is instantiated, Kubernetes generates a **network namespace** specifically dedicated for that pod. However, Kubernetes also allows for the utilization of the **host** network namespace if specified within the pod resource configuration. In the context of Liqo, this approach is



	P1	P2	P3	P6	S1	S2	S3	I1	I2	I3
P1		YES	YES	NO	YES	YES	NO	YES	NO	NO
P2	NO		NO	NO	NO	YES	NO	YES	NO	YES
P3	YES	YES		NO	YES	YES	NO	YES	NO	NO
P6	NO	YES	NO		NO	YES	YES	NO	YES	NO

Figure 5.5: Intra-cluster traffic segregation with offloaded service and endpoint on a third cluster; S1 expose P1, S2 expose P2, S3 expose P6; yellow cells in the table highlights the differences compared to standard Liqo behaviour

advantageous as it enables the application of routing rules between the **vxlan** (internal network segment of Liqo) and the **cross-cluster** section of Liqo network. Nonetheless, this configuration introduces a problem. The **cross-cluster** part of Liqo necessitates the implementation of a series of **iptables rules** for each peer, which cannot be aggregated. To prevent the insertion of a large number of **iptables rules** into the host network namespace, the **liqo-gateway** component creates a dedicated **network namespace**. As a result, the **liqo-gateway** component functions as a pod utilizing the host **network**

namespace, autonomously creating an additional network namespace, used to apply Liqo Iptables rules. This configuration is visualized in figure 5.6.

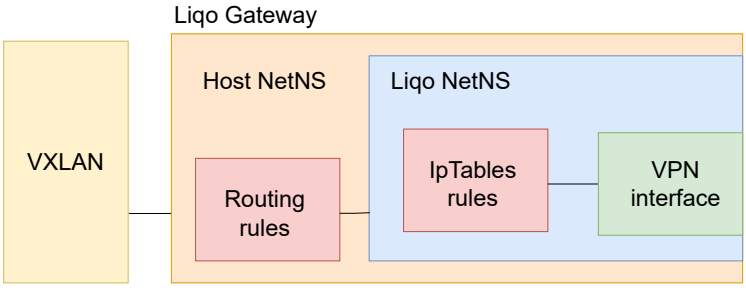


Figure 5.6: Liqo Gateway overview

5.3.2 OffloadedPod controller

Part of this thesis work has been dedicated to implementing a custom controller responsible for managing the Iptables rules that enable remote clusters to reach their offloaded pods. This controller operates within the `liqo-gateway` pod and is named the **OffloadedPod controller**. It reconciles objects of the type Pod characterized by the `liqo.io/managed-by: shadowpod` label in response to create/delete/update events. Starting from the information of the reconciled Pod, it generates a **PodInfo** object (5.1), containing the following data:

- **PodIP**: the IP address of the pod
- **RemoteClusterID**: the Liqo unique identifier of the remote cluster that offloaded the pod
- **Deleting**: a boolean that is true if the pod does not longer exists

The controller stores them within a `sync.Map` (concurrent data structure provided by the Go standard library for safely storing and accessing key-value pairs), which works as a local cache within the gateway, retaining essential information about offloaded pods for the management of Iptables rules.

Reconciliation logic

After the controller fetches the Pod object with a GET request to the API server, the reconciliation logic is as follows (5.2):

- If the Pod object is not found, but there exists a podInfo object in the cache related to it, a soft delete of the podInfo is performed, thus Deleting

Listing 5.1: PodInfo struct

```
// PodInfo contains informations useful to create rules allowing
// traffic towards offloaded pods.
type PodInfo struct {
    PodIP           string
    RemoteClusterID string
    Deleting        bool
}
```

field is set to true to indicate the non-existence of the Pod during the update of Iptables rules that is initiated. Upon completing the update, the podInfo object is removed from the cache.

- If the Pod object is found, the corresponding podInfo object is created. After confirming that the pod is not in the process of being deleted and that the IP address is not null, it is stored in the cache. Subsequently, the update of Iptables rules is initiated.

Listing 5.2: Reconcile() function of the OffloadedPod controller

```
// Reconcile pods offloaded from other clusters to the local one.
func (r *OffloadedPodController) Reconcile(ctx context.Context,
    req ctrl.Request) (ctrl.Result, error) {
    var ensureIptablesRules = func(netns ns.NetNS) error {
        return r.EnsureRulesForClustersForwarding(r.podsInfo, r.
    endpointslicesInfo, r.IPSHandler)
    }
    nsName := req.NamespacedName
    klog.Infof("Reconcile Pod %q", nsName)

    pod := corev1.Pod{}
    if err := r.Get(ctx, nsName, &pod); err != nil {
        if client.IgnoreNotFound(err) == nil {
            // Pod not found, podInfo object found: delete podInfo
     object
            if value, ok := r.podsInfo.LoadAndDelete(nsName); ok {
                klog.Infof("Pod %q not found: ensuring updated
    iptables rules", nsName)

                // Soft delete object
                podInfo := value.(liqoiptables.PodInfo)
                podInfo.Deleting = true
                r.podsInfo.Store(nsName, podInfo)

                if err := r.gatewayNetns.Do(ensureIptablesRules);
    err != nil {
```

```
22                      return ctrl.Result{}, fmt.Errorf("error while
    ensuring iptables rules: %w", err)
23                  }
24
25                  // Hard delete object
26                  r.podsInfo.Delete(nsName)
27              }
28          }
29          return ctrl.Result{}, err
30      }
31
32      // Build podInfo object
33      podInfo := liqoiptables.PodInfo{
34          PodIP:           pod.Status.PodIP,
35          RemoteClusterID: pod.Labels[liqovk.LiqoOriginClusterIDKey
    ],
36      }
37
38      // Check if the object is under deletion
39      if !pod.ObjectMeta.DeletionTimestamp.IsZero() {
40          // Pod under deletion: skip creation of iptables rules and
     return no error
41          klog.Infof("Pod %q under deletion: skipping iptables rules
     update", nsName)
42          return ctrl.Result{}, nil
43      }
44
45      // Check if the pod IP is set
46      if podInfo.PodIP == "" {
47          // Pod IP address not yet set: skip creation of iptables
    rules and return no error
48          klog.Infof("Pod %q IP address not yet set: skipping
    iptables rules update", nsName)
49          return ctrl.Result{}, nil
50      }
51
52      // Store podInfo object
53      r.podsInfo.Store(nsName, podInfo)
54
55      // Ensure iptables rules
56      klog.Infof("Ensuring updated iptables rules")
57      if err := r.gatewayNetns.Do(ensureIptablesRules); err != nil {
58          klog.Errorf("Error while ensuring iptables rules: %w", err
    )
59          return ctrl.Result{}, err
60      }
61
62      return ctrl.Result{}, nil
63  }
```

5.3.3 ReflectedEndpointslice controller

To manage the iptables rules allowing remote clusters to utilize offloaded services in the scenarios described in Section 5.2.2, another custom controller, the **ReflectedEndpointslice controller**, is implemented within the `liqo-gateway` pod. This controller reconciles objects of the type EndpointSlice (see section 2.5.8 labeled with `endpointslice.kubernetes.io/managed-by: endpointslice-controller.k8s.io`, i.e. only those managed by the vanilla Kubernetes control plane.

The controller also watches events related to `NamespaceOffloading` objects, which are Liqo Custom Resources representing the process of offloading a local namespace. An event handler has been configured to add the EndpointSlices from the namespace associated with the NamespaceOffloading to the controller workqueue for reconciliation. This is necessary because without this mechanism, EndpointSlices created before the offloading of a namespace or those that continue to exist following unoffloading would not be reconciled in response to these events.

The controller stores within a sync.Map, which works as a local cache in the gateway to retain essential information for the management of Iptables rules, a map of **EndpointInfo** objects (5.3) for each reconciled EndpointSlice object. An EndpointInfo object contains the following data:

- **Address**: the IP address of an EndpointSlice endpoint
- **SrcClusterIDs**: the unique identifiers of all the remote clusters that must be able to contact the endpoint through the gateway
- **Deleting**: a boolean that is true if the pod does not longer exists

Listing 5.3: EndpointInfo struct

```
// EndpointInfo contains informations useful to create rules allowing
// traffic towards service endpoints.
type EndpointInfo struct {
    Address       string
    SrcClusterIDs []string
    Deleting      bool
}
```

Reconciliation logic

The reconciliation logic starts fetching the EndpointSlice object through a GET request to the API server.

Then, if the Endpointslice object is not found, but exists an entry in the cache related to it, a soft delete of all the endpointInfo related to it is performed, therefore, for each one, the Deleting field is set to true to indicate the non-existence of the EndpointSlice during the update of iptables rules that is initiated. Upon completing the update, the the entry is removed from the cache (5.4).

Listing 5.4: Reconcile() function of the ReflectedEndpointslice controller

```
// Reconcile local endpointslices that are also reflected on remote clusters as a result of offloading.
func (r *ReflectedEndpointsliceController) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    var ensureIptablesRules = func(netns ns.NetNS) error {
        return r.EnsureRulesForClustersForwarding(r.podsInfo, r.endpointslicesInfo, r.IPSHandler)
    }
    nsName := req.NamespacedName
    klog.Infof("Reconcile Endpointslice %q", nsName)

    endpointslice := discoveryv1.EndpointSlice{}
    if err := r.Get(ctx, nsName, &endpointslice); err != nil {
        if client.IgnoreNotFound(err) == nil {
            // Endpointslice not found, endpointsliceInfo object found: delete endpointInfo objects.
            if value, ok := r.endpointslicesInfo.LoadAndDelete(nsName); ok {
                klog.Infof("Endpointslice %q not found: ensuring updated iptables rules", nsName)

                // Soft delete object
                endpointsInfo := value.(map[string]liqoiptables.EndpointInfo)
                for endpoint, endpointInfo := range endpointsInfo {
                    endpointInfo.Deleting = true
                    endpointsInfo[endpoint] = endpointInfo
                }
                r.endpointslicesInfo.Store(nsName, endpointsInfo)

                if err := r.gatewayNetns.Do(ensureIptablesRules); err != nil {
                    return ctrl.Result{}, fmt.Errorf("error while ensuring iptables rules: %w", err)
                }

                // Hard delete object
                r.endpointslicesInfo.Delete(nsName)
            }
```

```
31          }
32          return ctrl.Result{}, err
33      }
34      /*...*/
35  }
```

If the EndpointSlice object is found, we proceed to check the existence of the NamespaceOffloading. If it is not found, it means that the namespace, and consequently, the service, are not offloaded. Therefore, the endpoints in this EndpointSlice should not be reachable, and we follow the same deletion procedure described previously (5.5).

Listing 5.5: Reconcile() function of the OffloadedPod controller

```
1  // Reconcile local endpointslices that are also reflected on
       remote clusters as a result of offloading.
2  func (r *ReflectedEndpointsliceController) Reconcile(ctx context.
       Context, req ctrl.Request) (ctrl.Result, error) {
3      var ensureIptablesRules = func(netns ns.NetNS) error {
4          return r.EnsureRulesForClustersForwarding(r.podsInfo, r.
       endpointslicesInfo, r.IPSHandler)
5      }
6      nsName := req.NamespacedName
7      klog.Infof("Reconcile Endpointslice %q", nsName)
8
9      endpointslice := discoveryv1.EndpointSlice{}
10     if err := r.Get(ctx, nsName, &endpointslice); err != nil {
11         /*...*/
12     }
13
14     // Check endpointslice's namespace offloading
15     nsOffloading, err := getters.GetOffloadingByNamespace(ctx, r.
       Client, endpointslice.Namespace)
16     if err != nil {
17         if client.IgnoreNotFound(err) == nil {
18             // Delete endpointInfo objects related to this
       endpointslice
19             if value, ok := r.endpointslicesInfo.LoadAndDelete(
       nsName); ok {
20                 // Endpointslice not found, endpointsliceInfo
       object found: ensure iptables rules
21                 klog.Infof("Endpointslice %q not found: ensuring
       updated iptables rules", nsName)
22
23                 // Soft delete object
24                 endpointsInfo := value.(map[string]liqoiptables.
       EndpointInfo)
25                 for endpoint, endpointInfo := range endpointsInfo
       {
26                     endpointInfo.Deleting = true
```

```
27                      endpointsInfo[endpoint] = endpointInfo
28                  }
29                  r.endpointslicesInfo.Store(nsName, endpointsInfo)
30
31                  if err := r.gatewayNetns.Do(ensureIptablesRules);
    err != nil {
32                      return ctrl.Result{}, fmt.Errorf("error while
    ensuring iptables rules: %w", err)
33                  }
34
35                  // Hard delete object
36                  r.endpointslicesInfo.Delete(nsName)
37              }
38          }
39          return ctrl.Result{}, err
40      }
41
42      /*...*/
43  }
```

If the Endpointslice and the NamespaceOffloading are found, the corresponding EndpointInfo objects are created and stored in a Map: for each endpoint in the EndpointSlice, the ClusterIDs of the clusters from which the endpoint can be reached are then extracted. This includes all clusters where the Service is offloaded, except for the one where the pod serving as the endpoint is running (5.6).

Listing 5.6: Reconcile() function of the OffloadedPod controller

```
1  // Reconcile local endpointslices that are also reflected on
       remote clusters as a result of offloading.
2  func (r *ReflectedEndpointsliceController) Reconcile(ctx context.
       Context, req ctrl.Request) (ctrl.Result, error) {
3      var ensureIptablesRules = func(netns ns.NetNS) error {
4          return r.EnsureRulesForClustersForwarding(r.podsInfo, r.
       endpointslicesInfo, r.IPSHandler)
5      }
6      nsName := req.NamespacedName
7      klog.Infof("Reconcile Endpointslice %q", nsName)
8
9      endpointslice := discoveryv1.EndpointSlice{}
10     if err := r.Get(ctx, nsName, &endpointslice); err != nil {
11         /*...*/
12     }
13
14     /*...*/
15
16     clusterSelector := nsOffloading.Spec.ClusterSelector
17
```

```
18      nodes := virtualkubeletv1alpha1.VirtualNodeList{}
19      if err := r.List(ctx, &nodes); err != nil {
20          return ctrl.Result{}, fmt.Errorf("%w", err)
21      }
22
23      // Build endpointInfo objects
24      endpointsInfo := map[string]liqoiptables.EndpointInfo{}
25      // For each endpoint, find ClusterIDs of clusters that can
    reach that endpoint
26      for _, endpoint := range endpointslice.Endpoints {
27          clusterIDs := []string{}
28          for i := range nodes.Items {
29              if *endpoint.NodeName == nodes.Items[i].Name {
30                  continue
31              }
32
33              matchClusterSelctor, err := nsoffctrl.
    MatchVirtualNodeSelectorTerms(ctx, r.Client, &nodes.Items[i], &
    clusterSelector)
34              if err != nil {
35                  return ctrl.Result{}, fmt.Errorf("%w", err)
36              }
37
38              if matchClusterSelctor {
39                  if clusterID, found := virtualnodeutils.
    GetVirtualNodeClusterID(&nodes.Items[i]); found {
40                      clusterIDs = append(clusterIDs, clusterID)
41                  }
42              }
43          }
44
45          endpointsInfo[endpoint.Addresses[0]] = liqoiptables.
    EndpointInfo{Address: endpoint.Addresses[0], SrcClusterIDs:
    clusterIDs}
46      }
47
48          /*...*/
49  }
```

Finally, after checking the EndpointSlice s not in the process of being deleted, the Map of EndpointInfo objects is stored in an entry of the cache related to the EndpointSlice and Iptables rules updating starts.

Listing 5.7: Reconcile() function of the OffloadedPod controller

```
1  // Reconcile local endpointslices that are also reflected on
       remote clusters as a result of offloading.
2  func (r *ReflectedEndpointsliceController) Reconcile(ctx context.
       Context, req ctrl.Request) (ctrl.Result, error) {
3
```

```
4       /*...*/
5
6       // Check if the object is under deletion
7       if !endpointslice.ObjectMeta.DeletionTimestamp.IsZero() {
8           // Endpointslice under deletion: skip creation of iptables
        rules and return no error
9           klog.Infof("Endpointslice %q under deletion: skipping
        iptables rules update", nsName)
10          return ctrl.Result{}, nil
11      }
12
13      // Check if endpoint(s) are no more part of the endpointslice
14      value, loaded := r.endpointslicesInfo.Load(nsName)
15      if loaded {
16          oldEndpointsInfo := value.(map[string]liqoiptables.
        EndpointInfo)
17          for oldEndpoint, oldEndpointInfo := range oldEndpointsInfo
        {
18              if _, ok := endpointsInfo[oldEndpoint]; !ok {
19                  oldEndpointInfo.Deleting = true
20                  endpointsInfo[oldEndpoint] = oldEndpointInfo
21              }
22          }
23      }
24
25      // Check if there aren't new information: in this case it's
        not necessary ensure iptables rules
26      if len(endpointsInfo) == 0 {
27          // Endpoints fields of Endpoinslice yet empty: skip
        creation of iptables rules and return no error
28          klog.Infof("Endpoints of endpointslice %q not yet set:
        skipping iptables rules update", nsName)
29          return ctrl.Result{}, nil
30      }
31
32      // Store endpointslicesInfo object
33      r.endpointslicesInfo.Store(nsName, endpointsInfo)
34
35      // Ensure iptables rules
36      klog.Infof("Ensuring updated iptables rules")
37      if err := r.gatewayNetns.Do(ensureIptablesRules); err != nil {
38          return ctrl.Result{}, fmt.Errorf("error while ensuring
        iptables rules: %w", err)
39      }
40
41      return ctrl.Result{}, nil
42  }
```

5.3.4 Iptables rules management

A significant portion of this work is dedicated to managing the Iptables rules that implement the previously described behaviors.

Since it is possible to create custom chains in Iptables (as described in the dedicated section 4.2.2), Liqo creates one when installed, named LIQO-FORWARD, in the Filter table. Traffic is redirected to this chain from the standard FORWARD chain, and it contains the rules necessary for the correct operation of Liqo.

When the first peering is activated, we add the following rule to this chain:

```
iptables -m conntrack --ctstate ESTABLISHED, RELATED -j ACCEPT
```

The rule accepting packets marked by connection tracking as:

- ESTABLISHED, necessary to allow response traffic from addresses that cannot start connection
- RELATED, useful to allow possible traffic from a new connection, but associated with an existing one,e.g. an FTP data transfer, or an ICMP error

For each activated peering, and thus for each remote cluster, is then added a redirect rule to a new custom chain, named `LIQO-FRWD-CLS-`*remoteClusterID*. In this chain are inserted all the rules, dedicated to the specific remote cluster, that contributes to the functioning of this thesis work, and that we are going to describe.

OffloadedPod controller rules

Under the hood, the OffloadedPod controller manages Iptables rules through the buildRulesPerClusterForOffloadedPods() function (5.8). Starting from the local cache of PodInfo objects, IP addresses are grouped by the cluster from which the respective pods are offloaded. Then, for each group, an Ipset is created, the IP addresses are inserted into it, and it is used to create a rule of the following type:

```
iptables -m set --match-set [Ipset name] dst -j ACCEPT
```

In practice, this logic creates a rule for each cluster that has offloaded one or more pods, allowing contact with those pods. This rule is then inserted into the custom chain corresponding to the cluster.

Listing 5.8: buildRulesPerClusterForOffloadedPods() function

```
// buildRulesPerClusterForOffloadedPods builds rules allowing traffic from remote clusters towards their pods offloaded on this cluster.
func buildRulesPerClusterForOffloadedPods(podsInfo *sync.Map, ipSetHandler *liqoipset.IPSHandler) (map[string][]IPTableRule, error) {
    // Map of Pod IPs per cluster
    ipsPerCluster := map[string][]string{}
    // Populate Pod IPs per cluster
    podsInfo.Range(func(key, value any) bool {
        podInfo := value.(PodInfo)
        klog.Infof("buildIPSetPerClusterForOffloadedPods: %s", podInfo)
        if _, ok := ipsPerCluster[podInfo.RemoteClusterID]; !ok {
            // Add remote cluster ID key (regardless of pod being deleted or not)
            ipsPerCluster[podInfo.RemoteClusterID] = []string{}
        }
        if !podInfo.Deleting {
            ipsPerCluster[podInfo.RemoteClusterID] = append(ipsPerCluster[podInfo.RemoteClusterID], podInfo.PodIP)
        }
        return true
    })
    // Map of IPTables rules and IP sets per cluster
    rulesPerCluster := map[string][]IPTableRule{}
    // Populate IPTables rules and IP set per cluster
    for clusterID, ips := range ipsPerCluster {
        rulesPerCluster[clusterID] = []IPTableRule{}
        // Create IP set
        setName := getClusterIPSetForOffloadedPods(clusterID)
        ipset, err := ipSetHandler.CreateSet(setName, "")
        if err != nil {
            klog.Infof("Error while creating IP set %q: %w", setName, err)
            return nil, err
        }
        // Clear IP set (just in case it already existed)
        if err := ipSetHandler.FlushSet(ipset.Name); err != nil {
            klog.Infof("Error while deleting all entries from IP set %q: %w", setName, err)
            return nil, err
        }
        if len(ips) > 0 {
            for _, podIP := range ips {
                // Add pod's IP entry to IP set
                if err := ipSetHandler.AddEntry(podIP, ipset); err != nil {
```

```
                    klog.Infof("Error while adding entry %q to IP set %q: %w", podIP, ipset.Name, err)
                    return nil, err
                }
            }
            // Add match-set rule
            rulesPerCluster[clusterID] = append(
                rulesPerCluster[clusterID],
                IPTableRule{
                    "-m", "comment", "--comment",
                    // WARNING: Never use double-quotes inside the comment, otherwise IpTableRule parser will fail
                    fmt.Sprintf("Allows traffic from '%s' only to pods offloaded by that remote cluster", clusterID),
                    "-m", setModule,
                    "--match-set", ipset.Name, "dst",
                    "-j", ACCEPT})
        }
    }
    return rulesPerCluster, nil
}
```

ReflectedEndpointslice controller rules

The ReflectedEndpointslice controller internally manages Iptables rules through the buildRulesPerClusterForEndpointslicesReflected() function 5.9). The local cache of the controller stores the respective EndpointInfo objects for each EndpointSlice. The function implements a mapping that, based on the cache, generates as many Iptables rules as there are EndpointSlices for offloaded services on that remote cluster. The rules are of the following type:

```
iptables -m set --match-set [Ipset name] dst -j ACCEPT
```

where Ipset groups the endpoints IP addresses of the specific Endpointslice that a remote cluster must be able to reach.

Each rule is then inserted into the custom chain corresponding to the cluster.

Listing 5.9: buildRulesPerClusterForEndpointslicesReflected() function

```
// buildRulesPerClusterForEndpointslicesReflected builds rules allowing traffic towards endpoints of local services reflected on other clusters.
func buildRulesPerClusterForEndpointslicesReflected(
    endpointslicesInfo *sync.Map,
    ipSetHandler *liqoipset.IPSHandler,
) (map[string][]IPTableRule, error) {
    // Map of endpoint IPs per cluster
```

```
7       endpointSetsPerCluster := map[string]map[string][]string{}
8
9       // Populate endpoint IPs per cluster
10      endpointslicesInfo.Range(func(key, value any) bool {
11          namespacedName := key.(types.NamespacedName)
12          endpointsInfo := value.(map[string]EndpointInfo)
13          for _, endpointInfo := range endpointsInfo {
14              for _, clusterID := range endpointInfo.SrcClusterIDs {
15                  if _, ok := endpointSetsPerCluster[clusterID]; !ok
    {
16                      endpointSetsPerCluster[clusterID] = map[string
    ][]string{}
17                  }
18                  if _, ok := endpointSetsPerCluster[clusterID][
    namespacedName.String()]; !ok {
19                      endpointSetsPerCluster[clusterID][
    namespacedName.String()] = []string{}
20                  }
21                  if !endpointInfo.Deleting {
22                      endpointSetsPerCluster[clusterID][
    namespacedName.String()] = append(
23                          endpointSetsPerCluster[clusterID][
    namespacedName.String()], endpointInfo.Address)
24                  }
25              }
26          }
27          return true
28      })
29
30      // Map of IPTables rules and IP sets per cluster
31      rulesPerCluster := map[string][]IPTableRule{}
32
33      // Populate IP set per endpointslice and cluster, and
    IPTables rules per cluster
34      for clusterID, endpointsSets := range endpointSetsPerCluster {
35          rulesPerCluster[clusterID] = []IPTableRule{}
36          for namespacedName, endpointSet := range endpointsSets {
37              namespacedNameChunks := strings.Split(namespacedName,
    "/")
38              if len(namespacedNameChunks) != 2 {
39                  return nil, fmt.Errorf("invalid value %v",
    namespacedNameChunks)
40              }
41              setName := fmt.Sprintf("%s-%s", strings.ToUpper(
    namespacedNameChunks[1]), strings.Split(clusterID, "-")[0])
42              croppedSetName := k8sstrings.ShortenString(setName,
    IPSetNameMaxLength)
43              // Create IP set
```

```
44              ipset, err := ipSetHandler.CreateSet(croppedSetName,
    setName)
45              if err != nil {
46                  klog.Infof("Error while creating IP set %q: %w",
    setName, err)
47                  return nil, err
48              }
49              // Clear IP set (just in case it already existed)
50              if err := ipSetHandler.FlushSet(ipset.Name); err !=
    nil {
51                  klog.Infof("Error while deleting all entries from
    IP set %q: %w", setName, err)
52                  return nil, err
53              }
54              if len(endpointSet) > 0 {
55                  for _, ip := range endpointSet {
56                      // Add endpoint's IP entry to IP set
57                      if err := ipSetHandler.AddEntry(ip, ipset);
    err != nil {
58                          klog.Infof("Error while adding entry %q to
     IP set %q: %w", ip, ipset.Name, err)
59                          return nil, err
60                      }
61                  }
62                  // Add match-set rule
63                  rulesPerCluster[clusterID] = append(
64                      rulesPerCluster[clusterID],
65                      IPTableRule{
66                          "-m", setModule,
67                          "--match-set", ipset.Name, "dst",
68                          "-j", ACCEPT})
69              }
70          }
71      }
72      return rulesPerCluster, nil
73  }
```

Chapter 6

Use case: data spaces with Liqo

This chapter aims to demonstrate that Liqo, with fine-grained security proposed by this thesis work, can dynamically create flexible data spaces upon request, potentially spanning multiple administrative domains. This enables a data producer to offer its data to potential consumers, without giving up on security and data ownership/sovereignty rules, and without affecting the possibility of consumers to read and process.

6.1 Data spaces

Data represents a fundamental asset in our contemporary digital society. Nonetheless, managing data access and ensuring its security is a complex endeavor, especially given the distinct roles involved in data production and utilization. The primary challenges involve controlling access rights and, more significantly, safeguarding data from unlawful appropriation and duplication, particularly when handling sensitive information. This intricate landscape can be examined through the lenses of two critical concepts: **data sovereignty** and **data gravity**

Data sovereignty pertains to the regulation and management of data flows and the accompanying infrastructure within the jurisdiction of a specific country [14]. This issue becomes especially relevant when governments are concerned about the sovereignty of their data, particularly when it is stored in the cloud. They need to address questions related to data confidentiality and ensure that government data remains under their jurisdiction even when hosted abroad [15]. These concepts are equally applicable to universities and businesses that might have their data stored externally. Furthermore, it is essential to

mention the European General Data Protection Regulation (GDPR) in this context. GDPR imposes various obligations on organizations, irrespective of their location, as long as they process data pertaining to individuals within the European Union [16]. Additionally, Europe has introduced two significant legislative acts, namely the Data Governance Act [17] and the Data Act [18]. The former aims to enhance trust in data sharing, strengthen mechanisms to increase data availability, and address technical barriers to data reuse. The latter complements the Data Governance Act by providing clarity on who can derive value from data and the conditions under which this can occur.

Data gravity refers to the capacity of data to draw in applications, services, and additional data. In this context, individual data components are assigned both mass and density, and as these attributes grow significantly, the process of transferring data between locations via a network becomes increasingly challenging in terms of time, logistics, and cost-effectiveness. This phenomenon mirrors the behavior of a substantial physical mass that exerts a gravitational pull on nearby objects – the greater the mass, the more potent the attraction [19].

Data spaces offer a solution to these issues by establishing a confined environment where third parties can access and utilize data, but only to the extent permitted by the data producer, ensuring that only the intended information is made available to them.

The Open DEI project [20], funded by the EU, released a position paper outlining a data space as a decentralized infrastructure designed for secure and trustworthy data sharing within data ecosystems, built upon universally accepted principles. Users of these data spaces gain the capability to access data in a manner that is secure, transparent, trusted, user-friendly, and unified. The authority to grant access and usage rights to the data is vested exclusively in individuals or organizations with the rightful ownership of the data.

6.2 Use case overview

In a conventional scenario involving two separate administrative entities, one serving as the data producer and the other as the data consumer, each entity manages its own cluster. The transfer of raw data occurs between these parties. Both entities are motivated to uphold control over interactions involving data and processing services within their respective clusters. Particular attention is given to the entity that owns sensitive data, which strives to grant access to this data while concurrently preventing any unauthorized data exfiltration.

To illustrate this use case, let us consider a scenario as depicted in Figure 6.1, involving two clusters. The first cluster, named **Pharma**, is under the

ownership of a pharmaceutical company, which requires the execution of its latest algorithms using patient data. The second cluster, named **Hospital**, is managed by a hospital, which possesses the data to be processed (e.g., medical records). Hospital is interested in running Pharma's algorithm on the patient data but must ensure that this sensitive information remains secure and inaccessible to Pharma. Presently, the only solution is to transfer the data to the Pharma application and rely on non-technical means, such as legal agreements, to prevent data theft. Unfortunately, there are no technical methods available to ensure Pharma's compliance with these agreements.

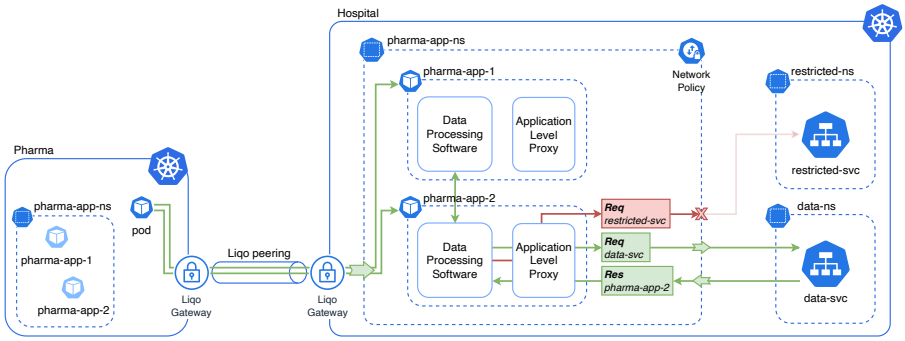


Figure 6.1: Data spaces with Liqo

6.3 Using Liqo for data spaces

In our proposed architecture, Pharma aims to outsource the execution of its workloads to Hospital while still retaining control over the offloading process using the capabilities offered by Liqo. Essentially, Pharma's algorithms are executed within the Hospital cluster, but Pharma maintains authority over this application's lifecycle as if it were running locally. On the other hand, Hospital wishes to review offloading requests before accepting or rejecting them. Upon receiving a valid request, Hospital applies various security measures (e.g., defining how the offloaded service can communicate with the Pharma cluster, which will be elaborated on in the following sections) to ensure the secure execution and monitoring of third-party workloads.

6.3.1 Workflow

This subsection delineates the process of establishing a **secure infrastructure-level data space** connecting Pharma and Hospital clusters, enabling data

consumption through pod offloading while implementing stringent security measures.

The workflow presupposes that both Pharma and Hospital clusters are already configured and operational, with Liqo already installed. These clusters initially remain disconnected, requiring the establishment of a **peering connection**. To initiate this connection, the Pharma cluster, which intends to access data in the Hospital cluster, initiates a Liqo peering with the Hospital cluster. If the Hospital cluster approves the connection, provides Pharma access to Hospital services located in the "data-space" Kubernetes namespace (referred to as data-ns in Figure 3). This namespace houses sensitive data and is external to Pharma's virtual cluster. Within the extension of its cluster on Hospital, Pharma can offload pods in the Hospital cluster to collect data and run its algorithm, facilitating data collection.

Upon initiating the offloading process, Hospital detects it and automatically enforces a series of security measures to securely host these pods within its cluster. These measures include the implementation of Kubernetes **Network Policies**, a mutating operation on the offloaded pods, and, as largely presented in chapter 5, the enforcement of firewall rules on the gateway handling connections. Offloaded pods are granted access to the protected data, allowing them to manipulate and aggregate data, thus introducing an additional layer to the data producer cluster.

The aforementioned security rules ensure that all Pharma pods within the Pharma cluster can only connect to Pharma pods in the Hospital cluster. Specifically, a set of Network Policies ensures that offloaded pods can only communicate with selected services while blocking all other requests within the cluster. This is accomplished through standard Kubernetes APIs. Processed data is then transmitted to Pharma through the Liqo intra-cluster tunnel. In this context, the use of a specially crafted application, running in the pod called **pharma-app-2** in the figure, comes into play. This application receives requests for algorithm results from Pharma, collects the data via the Hospital service that exposes it, processes the data locally on Hospital, and then responds to Pharma's requests with the aggregated results.

The pod offloading process is managed by a Mutating Webhook, which includes an init container and a sidecar alongside the main application container. The init container create the necessary Iptables rules to forward all offloaded pod traffic to the Sidecar container, which acts as a proxy and monitors all communications, allowing only the desired ones. This mechanism establishes a robust barrier safeguarding the Hospital cluster against data exfiltration. The Sidecar intercepts all traffic originating from offloaded pods and directed outside the Hospital cluster, inspecting the data at the application level.

6.3.2 Implementation

Liqo Gateway

The Liqo Gateway serves as the final destination of the intra-cluster tunnel that has successfully completed the peering phase. To ensure the prevention of data exfiltration for Hospital, particular attention is given to this component within our architecture. This presents an opportunity to utilize the custom Liqo Gateway introduced in this work. It monitors pods that have been offloaded from Pharma, gathers their IP addresses, and adds them to a whitelist. This whitelist exclusively permits access to these IP addresses by pods belonging to the Pharma cluster, whether they are local or from a remote cluster. It restricts connections from all other clusters. The whitelisted IP addresses are then translated into Iptables rules and applied to the Liqo Gateway in the Hospital cluster. When offloaded pods are terminated, their respective IP addresses are removed from the whitelist, and the corresponding Iptables rules are deleted. As the Liqo Gateway may also function as a Network Address Translator (NAT) between clusters, the Iptables rules are added outside the secure tunnel. This means they are applied after the traffic has been decapsulated from the tunnel. Consequently, the rules also exist outside the NAT, resulting in NAT transparency in our implementation. The IP addresses used in the rules are local to the Hospital cluster.

NamespaceReconciler controller

In the Hospital cluster, we require a new custom controller, known as the **NamespaceReconciler** controller (A, which oversees the reconciliation of offloaded namespaces. It identifies offloaded namespaces by recognizing the labels applied by Liqo and appends a Network Policy (NetPol) and a ConfigMap to each of the identified namespaces. If a namespace is deleted, the associated NetPol and ConfigMap are also removed. Since NetPol is a resource, its deletion results in the removal of the namespace and all the resources within it. The NetPol is responsible for regulating inbound and outbound traffic within the namespace where it is deployed. In particular, it allows egress destinations and ingress sources, whether they are other namespaces or specific resources, labeled with `data-space/netpol-allow=true`. This label, assigned by the Hospital cluster, helps the Hospital determine which traffic should be allowed or blocked. If the label matches, the traffic is permitted; otherwise, it is blocked. Although the label is contained within the NetPol, the matching process pertains to external resources attempting to exchange traffic with the offloaded namespace. The ConfigMap holds the configuration parameters needed to direct traffic from each offloaded pod through the Sidecar container[21].

Mutating Webhook

Lastly, it is implemented a **Mutating Webhook** performing various tasks when a pod is offloaded. First, an Init container is inserted into the pod before the main container when the offloaded pod is scheduled. This Init container accesses the configuration information stored in the namespace ConfigMap and sets rules that require all traffic from the main container to pass through the Sidecar container. Once the Init container has completed its task, the main container and the Sidecar container are created. This arrangement, enforced by the Init container, ensures that traffic from the main container is directed through the Sidecar. In the current implementation, the Sidecar serves as a traffic monitoring tool without any metrics and is built using the Envoy proxy.

Chapter 7

Experimental validation

This chapter presents some performance analyses conducted by comparing standard Liqo v0.8 with the proof-of-concept introduced in chapter 6. The proof-of-concept is built upon the same version of Liqo, with the additional components proposed in this thesis.

Summarizing the basic scenario from (see figure 6.1) which the results are derived: in the Pharma cluster, a pod is required to access aggregated data provided by the algorithm starting from Hospital data. After completing the previously mentioned workflow (described in section 6.3.1), the Pharma cluster is left with a single pod offloaded on the Hospital cluster that houses the processing logic. When the Pharma cluster pod seeks access to data, it initiates communication with the offloaded pod to request the necessary data. This offloaded pod, in turn, interacts with the service exposed by the Hospital within the data space, where it has been granted access. The offloaded pod collects the required data, performs aggregation or analysis as needed, and then returns the processed information to the requesting pod. All data transmitted back to the Pharma pods undergoes thorough inspection and verification by the sidecar, ensuring that no sensitive data is sent (e.g., by verifying the data against a well-defined data structure). In essence, the offloaded pod functions as an intermediary, bridging the gap between the Pharma pod and the data made available by the Hospital cluster.

7.1 Data space creation time

The standard, or *vanilla*, execution of Liqo follows a two-phase process involving **peering** and **namespace offloading**. In this process, an offloaded namespace is created, and resources deployed within this namespace can take advantage of Liqo offloading feature, while resources in other namespaces, which are not

offloaded, cannot.

Our solution also follows the same two-phase process of peering and namespace offloading. The time duration for the peering phase is approximately (3.8098 ± 0.7780) seconds, while namespace offloading takes around (0.3097 ± 0.0738) seconds. It is evident that the peering phase consumes more time.

The third phase is the **deployment phase**, in which we evaluated the deployment of a variable number of pods in both the standard Liqo setup and our proposed solution. The results, as shown in Table 7.1, indicate that the two scenarios exhibit a striking similarity. This suggests that the inclusion of Init and Sidecar containers in our solution does not significantly impact the deployment time.

#Offloaded Pods	Vanilla (s)	Data space (s)
1	0, 09 ± 0, 023	0.095 ± 0, 021
5	0, 214 ± 0, 077	0, 240 ± 0, 069
10	1, 218 ± 0, 038	1, 214 ± 0, 038
100	31, 945 ± 3, 368	32, 794 ± 6, 346

Table 7.1: Pod deployment time

7.2 Resource consumption

Resource consumption is a crucial metric for assessing the computational load on a system during data transfer operations. In our study, we aim to determine if the fine-grained security measures introduced in chapter 5 have any noticeable impact on the resource consumption of the Liqo Gateway. To evaluate this, we conducted an analysis based on four distinct scenarios, as illustrated in figure 7.1. These scenarios were derived from both the standard Liqo implementation and our proposed solution.

Initially, we collected data during an **idle** period before initiating a **stress test**. This test design allowed us to make a comparative assessment of the four scenarios. We simulated data transfer between the two clusters using the iPerf tool [22]. Upon analyzing the results, we observed that our implementation, referred to as "Data Space" in figure , introduces only a negligible amount of additional resource consumption, whether during idle or stress conditions, compared to the standard Liqo implementation. This slight increase in CPU usage is a reasonable trade-off for the benefits of our proposed solution, which facilitates data provisioning through data spaces without introducing a significant resource overhead. We also monitored RAM consumption during our experiments, and our findings revealed that our solution had no noticeable

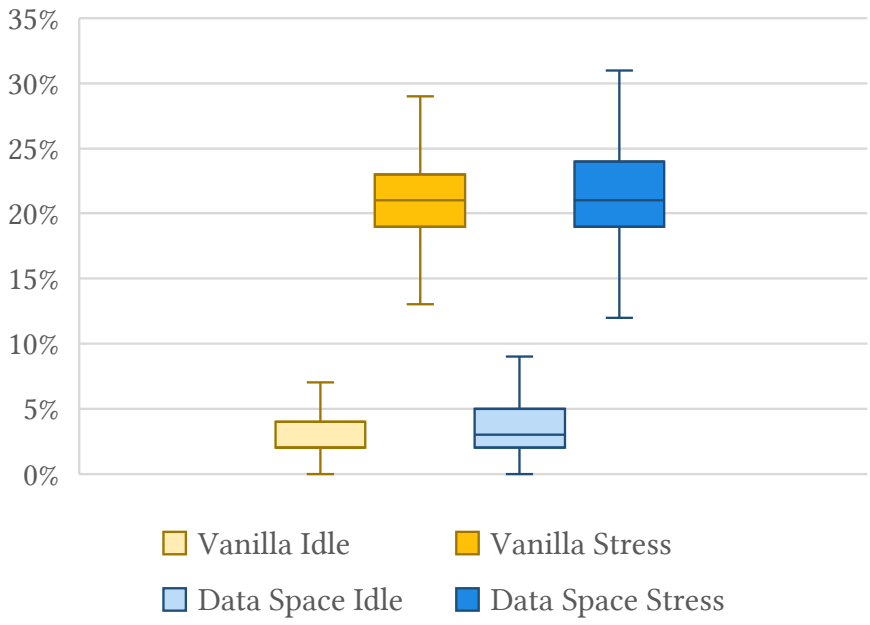


Figure 7.1: CPU consumption

impact on RAM usage compared to standard Liqo. Consequently, we have not included the corresponding graph for the sake of brevity.

7.3 Latency

In this context, **latency** refers to the time duration between a pod's data request and the moment the requested data becomes accessible. Our experiment focuses on scenarios where data is located within a pod, leading us to examine pod-to-pod communication. We considered three distinct scenarios to determine the time required for a pod to access data:

- **Local**: in this scenario, the communicating pods are within the same cluster, either Pharma or Hospital. These pods use local connectivity for data exchange. This scenario simulates communication between a pod seeking data and a pod providing data within the same cluster.

- **Remote**: this scenario involves pods in different clusters, one in Pharma and the other in Hospital. This scenario requires the use of Liqo for connectivity, as data transfer must traverse the intra-cluster tunnel. It reflects communication between pods in separate clusters, offering a comparison to

a situation where Pharma needs to access a remote service in the Hospital cluster and download relevant data for computation in the Pharma cluster

- **Data space**: this is the PoC scenario in which, however, we can have a variable number of offloaded pods acting as proxies, in order to test on multiple parallel connections.

We collected latency benchmarks using Apache Benchmark [18] with 10K requests and a varying number of parallel connections. For clarity, we present results for two instances: one with 10 parallel connections and the other with 100 parallel connections, shown in figure 7.2 and figure 7.3, respectively.

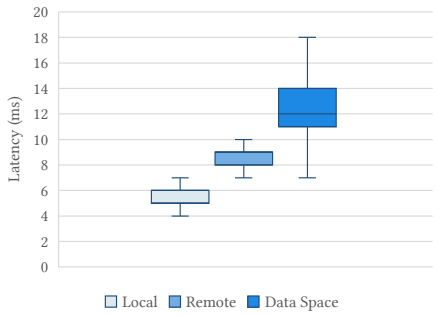


Figure 7.2: Latency with 10 parallel connections

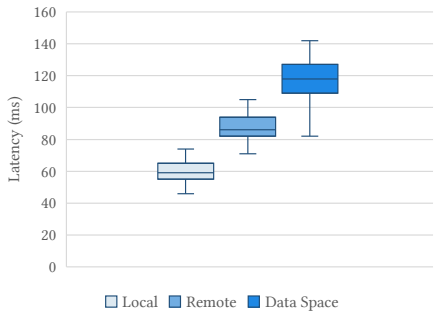


Figure 7.3: Latency with 100 parallel connections

As depicted in these figures, pod-to-pod latency increases when pods are in different clusters compared to when they are within the same cluster. However, the Data Space scenario requires further consideration. During testing, offloaded Pharma pods served as proxies, handling all requests to Hospital data, which introduced noticeable latency. This proxy role enables Hospital to control data flow, as offloaded pods are located within the data space. When deploying our solution in this manner, latency increased by up to 37% (median value) compared to the Remote scenario.

This result prompts a reevaluation of our initial premise. Leveraging the data gravity inherent in our proposed solution allows for more effective management of local and remote latencies. By assigning data aggregation or analysis tasks to the Pharma offloaded pod, the resulting data has a smaller size. In this setup, communication between the offloaded pod and the data-owning Hospital pod remains local to the Hospital cluster, resembling the latency behavior of the Local scenario. However, when the Pharma pod retrieves the aggregated data from the offloaded pod, the latency resembles that of the Remote scenario. Although Data Space latency is higher than Local and Remote latencies, data

aggregation helps mitigate this issue by reducing latency within the data aggregator pod and lowering the total volume of data transferred from Hospital to Pharma. This reduction in data transfer time can significantly impact the total cost of the deployment, especially when clusters are hosted on public cloud providers. While we cannot directly reduce latency, we can achieve a substantial reduction in the total data transfer time.

Chapter 8

Conclusions

The adoption of a multi-cluster approach has evolved into a fundamental requirement in the industry. The capability to enable various clusters to collaborate, share resources, and distribute workloads will be a cornerstone feature in the future of cloud computing.
The thesis has accomplished its initial goal, which was to improve the current full pod-to-pod connectivity model of Liqo, enabling a fine-grained control that allows connectivity exclusively to pods and services engaged in the multi-cluster topology.
Subsequently, a proof-of-concept was presented, which utilizes this new connectivity model and demonstrates the potential to use Liqo for the creation of data spaces.
This work opens up the possibility of identifying other use cases that require multi-cluster solutions. Based on these cases, new connectivity models for Liqo can be designed while maintaining the benefits of flexibility and scalability. The ability to choose from various models that best suit the user's needs, aiming to introduce a certain degree of configurability, could contribute to the increased adoption of Liqo.

Appendix A

NamespaceReconciler

content/code/go/namespace_reconciler_controller.go

```go
/*
Copyright 2022.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/

package controllers

import (
 "context"
 "fmt"

 "gopkg.in/yaml.v3"
 corev1 "k8s.io/api/core/v1"
 netv1 "k8s.io/api/networking/v1"
 metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
 "k8s.io/apimachinery/pkg/runtime"
 "k8s.io/apimachinery/pkg/types"
```

```
29  "k8s.io/klog/v2"
30  ctrl "sigs.k8s.io/controller-runtime"
31  "sigs.k8s.io/controller-runtime/pkg/builder"
32  "sigs.k8s.io/controller-runtime/pkg/client"
33  "sigs.k8s.io/controller-runtime/pkg/log"
34  "sigs.k8s.io/controller-runtime/pkg/predicate"
35
36  "data-space.liqo.io/consts"
37 )
38
39 // NamespaceReconciler reconciles a Namespace object
40 type NamespaceReconciler struct {
41  client.Client
42  Scheme *runtime.Scheme
43 }
44
45 //+kubebuilder:rbac:groups=core,resources=namespaces,verbs=get;
       list;watch;create;update;patch;delete
46 //+kubebuilder:rbac:groups=core,resources=namespaces/status,verbs=
       get;update;patch
47 //+kubebuilder:rbac:groups=core,resources=namespaces/finalizers,
       verbs=update
48
49 //+kubebuilder:rbac:groups=networking.k8s.io,resources=
       networkpolicies,verbs=get;list;watch;create;update;patch;delete
50 //+kubebuilder:rbac:groups=core,resources=configmaps,verbs=get;
       list;watch;create;update;patch;delete
51
52 // Reconcile is part of the main kubernetes reconciliation loop
       which aims to
53 // move the current state of the cluster closer to the desired
       state.
54 // TODO(user): Modify the Reconcile function to compare the state
       specified by
55 // the Namespace object against the actual cluster state, and then
56 // perform operations to make the cluster state reflect the state
       specified by
57 // the user.
58 //
59 // For more details, check Reconcile and its Result here:
60 // - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.13.0/pkg
       /reconcile
61 func (r *NamespaceReconciler) Reconcile(ctx context.Context, req
       ctrl.Request) (ctrl.Result, error) {
62  _ = log.FromContext(ctx)
63
64  nsName := req.NamespacedName
65  klog.Infof("Reconcile Namespace %q", nsName.Name)
66
```

```
67  // Network policy namespaced name
68  npNsName := types.NamespacedName{
69   Namespace: nsName.Name,
70   Name:      consts.NetworkPolicyName,
71  }
72  // Config map namespaced name
73  cmNsName := types.NamespacedName{
74   Namespace: nsName.Name,
75   Name:      consts.ConfigMapName,
76  }
77
78  namespace := corev1.Namespace{}
79  if err := r.Get(ctx, nsName, &namespace); err != nil {
80   err = client.IgnoreNotFound(err)
81   if err == nil {
82    klog.Infof("Namespace %q not found: trying to delete
      NetworkPolicy %q and ConfigMap %q", nsName.Name, npNsName,
      cmNsName)
83    // Delete relevant NetworkPolicy and ConfigMap if found
84    if err := r.deleteNetworkPolicy(ctx, npNsName); err != nil {
85     return ctrl.Result{}, err
86    }
87    if err := r.deleteConfigMap(ctx, cmNsName); err != nil {
88     return ctrl.Result{}, err
89    }
90   }
91   return ctrl.Result{}, err
92  }
93
94  // Intercept if the object is under deletion and return no errors
95  if !namespace.ObjectMeta.DeletionTimestamp.IsZero() {
96   klog.Infof("Namespace %q is under deletion. Relevant resources
      are going to be deleted as well.", nsName.Name)
97   return ctrl.Result{}, nil
98  }
99
100 // Check the "apply network policy" label
101 if v, ok := namespace.Labels[consts.DataSpaceApplyNetpolLabel];
      ok && v == "false" {
102  // Namespace label set to false: delete relevant NetworkPolicy
      if found
103  klog.Infof("Namespace %q contains label %q: trying to delete
      NetworkPolicy %q", nsName.Name, fmt.Sprintf("%s:%s", consts.
      DataSpaceApplyNetpolLabel, "false"), npNsName)
104  if err := r.deleteNetworkPolicy(ctx, npNsName); err != nil {
105   return ctrl.Result{}, err
106  }
107 } else {
```

```
108     // Namespace label not set to false: create NetworkPolicy if not found
109     networkPolicy := forgeNetworkPolicy(nsName.Name)
110     if err := r.createNetworkPolicy(ctx, nsName.Name, networkPolicy); err != nil {
111       return ctrl.Result{}, err
112     }
113   }
114
115   // Check the "apply webhook" label
116   if v, ok := namespace.Labels[consts.DataSpaceApplyWebhookLabel]; ok && v == "false" {
117     // Namespace label set to false: delete relevant ConfigMap if found
118     klog.Infof("Namespace %q contains label %q: trying to delete ConfigMap %q", nsName.Name, fmt.Sprintf("%s:%s", consts.DataSpaceApplyWebhookLabel, "false"), cmNsName)
119     if err := r.deleteConfigMap(ctx, cmNsName); err != nil {
120       return ctrl.Result{}, err
121     }
122   } else {
123     // Namespace label not set to false: create ConfigMap if not found
124     configMap, err := forgeConfigMap(nsName.Name)
125     if err != nil {
126       return ctrl.Result{}, err
127     }
128     if err := r.createConfigMap(ctx, nsName.Name, configMap); err != nil {
129       return ctrl.Result{}, err
130     }
131   }
132
133   return ctrl.Result{}, nil
134 }
135
136 // deleteNetworkPolicy deletes the NetworkPolicy resource in the reconciled namespace
137 func (r *NamespaceReconciler) deleteNetworkPolicy(ctx context.Context, nsName types.NamespacedName) error {
138   var networkPolicy netv1.NetworkPolicy
139   if err := r.Client.Get(ctx, nsName, &networkPolicy); err != nil {
140     err = client.IgnoreNotFound(err)
141     if err == nil {
142       klog.Infof("Skipping not found NetworkPolicy %q in namespace %q", consts.NetworkPolicyName, nsName.Namespace)
143     } else {
144       klog.Errorf("Error while getting NetworkPolicy %q in namespace %q", consts.NetworkPolicyName, nsName.Namespace)
```

```
145     }
146     return err
147   }
148
149   if err := r.Client.Delete(ctx, &networkPolicy); err != nil {
150     klog.Errorf("Error while deleting NetworkPolicy %q in namespace
        %q", consts.NetworkPolicyName, nsName.Namespace)
151   }
152   klog.Infof("Deleted NetworkPolicy %q in namespace %q", consts.
        NetworkPolicyName, nsName.Namespace)
153   return nil
154 }
155
156 // deleteConfigMap deletes the ConfigMap resource in the
      reconciled namespace
157 func (r *NamespaceReconciler) deleteConfigMap(ctx context.Context,
        nsName types.NamespacedName) error {
158   var configMap corev1.ConfigMap
159   if err := r.Client.Get(ctx, nsName, &configMap); err != nil {
160     err = client.IgnoreNotFound(err)
161     if err == nil {
162       klog.Infof("Skipping not found ConfigMap %q in namespace %q",
        consts.ConfigMapName, nsName.Namespace)
163     } else {
164       klog.Errorf("Error while getting ConfigMap %q in namespace %q",
          consts.ConfigMapName, nsName.Namespace)
165     }
166     return err
167   }
168
169   if err := r.Client.Delete(ctx, &configMap); err != nil {
170     klog.Errorf("Error while deleting ConfigMap %q in namespace %q",
        consts.ConfigMapName, nsName.Namespace)
171   }
172   klog.Infof("Deleted ConfigMap %q in namespace %q", consts.
        ConfigMapName, nsName.Namespace)
173   return nil
174 }
175
176 // createNetworkPolicy creates a NetworkPolicy resource in the
      reconciled namespace
177 func (r *NamespaceReconciler) createNetworkPolicy(ctx context.
      Context, namespaceName string, networkPolicy *netv1.
      NetworkPolicy) error {
178   if err := r.Client.Create(ctx, networkPolicy); err != nil {
179     err = client.IgnoreAlreadyExists(err)
180     if err == nil {
181       klog.Infof("NetworkPolicy %q already exists in namespace %q",
        consts.NetworkPolicyName, namespaceName)
```

```
182     } else {
183      klog.Errorf("Error while creating NetworkPolicy %q in namespace
          %q", consts.NetworkPolicyName, namespaceName)
184     }
185     return err
186    }
187    klog.Infof("Created NetworkPolicy %q in namespace %q", consts.
        NetworkPolicyName, namespaceName)
188    return nil
189   }
190
191   // createConfigMap creates a ConfigMap resource in the reconciled
        namespace
192   func (r *NamespaceReconciler) createConfigMap(ctx context.Context,
         namespaceName string, configMap *corev1.ConfigMap) error {
193    if err := r.Client.Create(ctx, configMap); err != nil {
194     err = client.IgnoreAlreadyExists(err)
195     if err == nil {
196      klog.Infof("ConfigMap %q already exists in namespace %q",
         consts.ConfigMapName, namespaceName)
197     } else {
198      klog.Errorf("Error while creating ConfigMap %q in namespace %q"
         , consts.ConfigMapName, namespaceName)
199     }
200     return err
201    }
202    klog.Infof("Created ConfigMap %q in namespace %q", consts.
        ConfigMapName, namespaceName)
203    return nil
204   }
205
206   // forgeNetworkPolicy builds a NetworkPolicy object
207   func forgeNetworkPolicy(namespaceName string) *netv1.NetworkPolicy
         {
208    return &netv1.NetworkPolicy{
209     ObjectMeta: metav1.ObjectMeta{
210      Name:      consts.NetworkPolicyName,
211      Namespace: namespaceName,
212     },
213     Spec: netv1.NetworkPolicySpec{
214      // For matching pods in the NetworkPolicy's namespace
215      PodSelector: metav1.LabelSelector{
216       // Empty: match all
217      },
218      PolicyTypes: []netv1.PolicyType{
219       netv1.PolicyTypeIngress,
220       netv1.PolicyTypeEgress,
221      },
222      Ingress: []netv1.NetworkPolicyIngressRule{{
```

```
223     From: []netv1.NetworkPolicyPeer{
224       // Allow all sources
225     },
226    }},
227    Egress: []netv1.NetworkPolicyEgressRule{{
228     To: []netv1.NetworkPolicyPeer{
229       // OR-ed selectors
230       {
231        // For pods in matching namespaces
232        NamespaceSelector: &metav1.LabelSelector{
233         MatchLabels: map[string]string{
234          consts.DataSpaceNetpolAllowLabel: "true",
235         },
236        },
237       },
238       {
239        // For matching pods in the NetworkPolicy's namespace
240        PodSelector: &metav1.LabelSelector{
241         MatchLabels: map[string]string{
242          consts.DataSpaceNetpolAllowLabel: "true",
243         },
244        },
245       },
246       {
247        // AND-ed selectors
248        // For matching pods in matching namespaces (to allow for
    DNS queries)
249        NamespaceSelector: &metav1.LabelSelector{
250         MatchLabels: map[string]string{
251          corev1.LabelMetadataName: consts.KUBE_SYSTEM,
252         },
253        },
254        PodSelector: &metav1.LabelSelector{
255         MatchLabels: map[string]string{
256          consts.K8sAppLabel: consts.KUBE_DNS,
257         },
258        },
259       },
260     },
261    }},
262   },
263  }
264 }
265
266 // forgeConfigMap builds a ConfigMap object
267 func forgeConfigMap(namespaceName string) (*corev1.ConfigMap,
    error) {
268  envoyConfig := forgeEnvoyConfig()
269
```

```
270  // Encode object into yaml
271  marshaledEnvoyConfig, err := yaml.Marshal(envoyConfig)
272  if err != nil {
273   klog.Fatal("Could not marshal yaml, error: %v", err)
274   return nil, err
275  }
276
277  return &corev1.ConfigMap{
278   ObjectMeta: metav1.ObjectMeta{
279    Name:      consts.ConfigMapName,
280    Namespace: namespaceName,
281   },
282   Data: map[string]string{
283    "keys": string(marshaledEnvoyConfig),
284   },
285  }, nil
286 }
287
288 // forgeEnvoyConfig builds an EnvoyConfig object
289 func forgeEnvoyConfig() *EnvoyConfig {
290  return &EnvoyConfig{
291   Admin: Admin{
292    Address: Address{
293     SocketAddress: SocketAddress{
294      Address:   consts.LOCALHOST_ADDR,
295      PortValue: consts.AdminPort,
296     },
297    },
298   },
299
300   StaticResources: StaticResources{
301    Listeners: []Listener{
302     // Egress TCP listener
303     {
304      Name: "egress_tcp_listener",
305      Address: Address{
306       SocketAddress: SocketAddress{
307        Address:   consts.LOCALHOST_ADDR,
308        PortValue: consts.EgressTcpPort,
309       },
310      },
311      FilterChains: []FilterChain{
312       {
313        Filters: []NameAndConfig{
314         {
315          Name: consts.TcpProxyTypeName,
316          TypedConfig: TypedConfig{
317           Type:       consts.TcpProxyTypeUrl,
318           StatPrefix: consts.EgressTcpStatPrefixName,
```

```
319             Cluster:     consts.EgressClusterName,
320             // Log access to /dev/stdout
321             AccessLog: []NameAndConfig{{
322              Name: consts.AccessLogTypeName,
323              TypedConfig: TypedConfig{
324               Type: consts.AccessLogTypeUrl,
325              },
326             }},
327            },
328           },
329          },
330         },
331        },
332        ListenerFilters: []NameAndConfig{
333         {
334          Name: consts.OriginalDstListenerFilterTypeName,
335          TypedConfig: TypedConfig{
336           Type: consts.OriginalDstListenerFilterTypeUrl,
337          },
338         },
339        },
340       },
341       // Ingress TCP listener
342       {
343        Name: "ingress_tcp_listener",
344        Address: Address{
345         SocketAddress: SocketAddress{
346          Address:    consts.ANY_ADDR,
347          PortValue: consts.IngressTcpPort,
348         },
349        },
350        FilterChains: []FilterChain{
351         {
352          Filters: []NameAndConfig{
353           {
354            Name: consts.TcpProxyTypeName,
355            TypedConfig: TypedConfig{
356             Type:        consts.TcpProxyTypeUrl,
357             StatPrefix: consts.IngressTcpStatPrefixName,
358             Cluster:     consts.IngressClusterName,
359             // Log access to /dev/stdout
360             AccessLog: []NameAndConfig{{
361              Name: consts.AccessLogTypeName,
362              TypedConfig: TypedConfig{
363               Type: consts.AccessLogTypeUrl,
364              },
365             }},
366            },
367           },
```

```
368           },
369          },
370         },
371         ListenerFilters: []NameAndConfig{
372          {
373           Name: consts.OriginalDstListenerFilterTypeName,
374           TypedConfig: TypedConfig{
375            Type: consts.OriginalDstListenerFilterTypeUrl,
376           },
377          },
378         },
379        },
380        // Egress HTTP listener
381        {
382         Name: "egress_http_listener",
383         Address: Address{
384          SocketAddress: SocketAddress{
385           Address:    consts.LOCALHOST_ADDR,
386           PortValue: consts.EgressHttpPort,
387          },
388         },
389         FilterChains: []FilterChain{
390          {
391           Filters: []NameAndConfig{
392            {
393             Name: consts.HttpConnectionManagerTypeName,
394             TypedConfig: TypedConfig{
395              Type:         consts.HttpConnectionManagerTypeUrl,
396              StatPrefix: consts.EgressHttpStatPrefixName,
397              HttpFilters: []NameAndConfig{
398               // Forward
399               {
400                Name: consts.DynamicForwardProxyFilterTypeName,
401                TypedConfig: TypedConfig{
402                 Type: consts.DynamicForwardProxyFilterTypeUrl,
403                 DnsCacheConfig: DnsCacheConfig{
404                  Name:                consts.DnsCacheConfigName,
405                  DnsLookupFamily: consts.Ipv4Only,
406                 },
407                },
408               },
409               // Tap HTTP egress traffic
410               {
411                Name: consts.HttpTapTypeName,
412                TypedConfig: TypedConfig{
413                 Type: consts.HttpTapTypeUrl,
414                 CommonConfig: CommonConfig{
415                  AdminConfig: AdminConfig{
416                   ConfigId: consts.EgressHttpTapConfigId,
```

```
                },
               },
              },
             },
             // Router
             {
              Name: consts.RouterTypeName,
              TypedConfig: TypedConfig{
               Type: consts.RouterTypeUrl,
              },
             },
            },
            RouteConfig: RouteConfig{
             Name: "egress_route_config",
             VirtualHosts: []VirtualHost{
              {
               Name:    "egress_forward_host",
               Domains: []string{"*"},
               Routes: []RouteEntry{
                {
                 Name: "egress_forward_route",
                 Match: Match{
                  Prefix: "/",
                 },
                 Route: Route{
                  Cluster: consts.EgressForwardClusterName,
                 },
                },
               },
              },
             },
            },
            // Log access to /dev/stdout
            AccessLog: []NameAndConfig{{
             Name: consts.AccessLogTypeName,
             TypedConfig: TypedConfig{
              Type: consts.AccessLogTypeUrl,
             },
            }},
           },
          },
         },
        },
       },
      },
      // Ingress HTTP listener
      {
       Name: "ingress_http_listener",
       Address: Address{
```

```
466         SocketAddress: SocketAddress{
467           Address:    consts.ANY_ADDR,
468           PortValue: consts.IngressHttpPort,
469         },
470       },
471       FilterChains: []FilterChain{
472         {
473           Filters: []NameAndConfig{
474             {
475               Name: consts.HttpConnectionManagerTypeName,
476               TypedConfig: TypedConfig{
477                 Type:        consts.HttpConnectionManagerTypeUrl,
478                 StatPrefix: consts.IngressHttpStatPrefixName,
479                 HttpFilters: []NameAndConfig{
480                   // Forward
481                   {
482                     Name: consts.DynamicForwardProxyFilterTypeName,
483                     TypedConfig: TypedConfig{
484                       Type: consts.DynamicForwardProxyFilterTypeUrl,
485                       DnsCacheConfig: DnsCacheConfig{
486                         Name:              consts.DnsCacheConfigName,
487                         DnsLookupFamily: consts.Ipv4Only,
488                       },
489                     },
490                   },
491                   // Tap HTTP ingress traffic
492                   {
493                     Name: consts.HttpTapTypeName,
494                     TypedConfig: TypedConfig{
495                       Type: consts.HttpTapTypeUrl,
496                       CommonConfig: CommonConfig{
497                         AdminConfig: AdminConfig{
498                           ConfigId: consts.IngressHttpTapConfigId,
499                         },
500                       },
501                     },
502                   },
503                   // Router
504                   {
505                     Name: consts.RouterTypeName,
506                     TypedConfig: TypedConfig{
507                       Type: consts.RouterTypeUrl,
508                     },
509                   },
510                 },
511                 RouteConfig: RouteConfig{
512                   Name: "ingress_route_config",
513                   VirtualHosts: []VirtualHost{
514                     {
```

```
                Name:    "ingress_forward_host",
                Domains: []string{"*"},
                Routes: []RouteEntry{
                 {
                  Name: "ingress_forward_route",
                  Match: Match{
                   Prefix: "/",
                  },
                  Route: Route{
                   Cluster: consts.IngressForwardClusterName,
                  },
                 },
                },
               },
              },
             },
             // Log access to /dev/stdout
             AccessLog: []NameAndConfig{{
              Name: consts.AccessLogTypeName,
              TypedConfig: TypedConfig{
               Type: consts.AccessLogTypeUrl,
              },
             }},
            },
           },
          },
         },
        },
       },
      },
      Clusters: []Cluster{
       // Egress cluster
       {
        Name:             consts.EgressClusterName,
        DnsLookupFamily: consts.Ipv4Only,
        Type:             consts.OriginalDstType,
        LbPolicy:         consts.OriginalDstLbPolicy,
        ConnectTimeout:  "6s",
        OriginalDstLbConfig: OriginalDstLbConfig{
         UseHttpHeader: true,
        },
       },
       // Ingress cluster
       {
        Name:             consts.IngressClusterName,
        DnsLookupFamily: consts.Ipv4Only,
        Type:             consts.OriginalDstType,
        LbPolicy:         consts.OriginalDstLbPolicy,
        ConnectTimeout:  "6s",
```

```
564         OriginalDstLbConfig: OriginalDstLbConfig{
565          UseHttpHeader: true,
566         },
567        },
568        // Egress forward cluster
569        {
570         Name:            consts.EgressForwardClusterName,
571         LbPolicy:        consts.OriginalDstLbPolicy,
572         ConnectTimeout: "6s",
573         ClusterType: NameAndConfig{
574          Name: consts.DynamicForwardProxyClusterTypeName,
575          TypedConfig: TypedConfig{
576           Type: consts.DynamicForwardProxyClusterTypeUrl,
577           DnsCacheConfig: DnsCacheConfig{
578            Name:            consts.DnsCacheConfigName,
579            DnsLookupFamily: consts.Ipv4Only,
580           },
581          },
582         },
583        },
584        // Ingress forward cluster
585        {
586         Name:            consts.IngressForwardClusterName,
587         LbPolicy:        consts.OriginalDstLbPolicy,
588         ConnectTimeout: "6s",
589         ClusterType: NameAndConfig{
590          Name: consts.DynamicForwardProxyClusterTypeName,
591          TypedConfig: TypedConfig{
592           Type: consts.DynamicForwardProxyClusterTypeUrl,
593           DnsCacheConfig: DnsCacheConfig{
594            Name:            consts.DnsCacheConfigName,
595            DnsLookupFamily: consts.Ipv4Only,
596           },
597          },
598         },
599        },
600       },
601      },
602     }
603    }
604
605    // SetupWithManager sets up the controller with the Manager.
606    func (r *NamespaceReconciler) SetupWithManager(mgr ctrl.Manager)
         error {
607     // namespacePredicate selects those namespaces matching the
         provided label
608     namespacePredicate, err := predicate.LabelSelectorPredicate(
         metav1.LabelSelector{
609      MatchExpressions: []metav1.LabelSelectorRequirement{{
```

```
610     Key:      consts.RemoteClusterIdLabel,
611     Operator: metav1.LabelSelectorOpExists,
612    }},
613   })
614   if err != nil {
615    klog.Error(err)
616    return err
617   }
618
619   return ctrl.NewControllerManagedBy(mgr).
620    For(&corev1.Namespace{}, builder.WithPredicates(
         namespacePredicate)).
621    Complete(r)
622 }
```

Bibliography

[1] *Kubernetes official documentation*. URL: https://kubernetes.io/docs/home (cit. on pp. 5, 12, 14–17).

[2] 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. 5).

[3] 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. 5).

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

[5] 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. 6).

[6] 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. 8).

[7] *Kubernetes Network Model*. URL: https%20:%20/%20/%20kubernetes%20.%20io%20/%20docs%20/%20concepts%20/%20cluster-administration/networking/#the-%20kubernetes-%20network-%20model (cit. on p. 17).

[8] *k8s CNI*. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/ (cit. on p. 20).

[9] *Kubernetes Services*. URL: https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/ (cit. on p. 20).

[10] *Kubebuilder GitHub repository*. URL: https://github.com/kubernetes-sigs/kubebuilder (cit. on p. 21).

[11] *Kubernetes Operator pattern*. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (cit. on p. 21).

[12] *Liqo documentation*. URL: https://docs.liqo.io/ (cit. on p. 22).

[13] *Virtual Kubelet GitHub repository*. URL: https://github.com/virtual-kubelet/virtual-kubelet (cit. on p. 28).

[14] Patrik Hummel, Matthias Braun, Max Tretter, and Peter Dabrock. "Data sovereignty: A review". In: *Big Data & Society* 8.1 (2021). DOI: 10.1177/2053951720982012. eprint: https://doi.org/10.1177/2053951720982012. URL: https://doi.org/10.1177/2053951720982012 (cit. on p. 57).

[15] Kristina Irion. "Government Cloud Computing and National Data Sovereignty". In: *Policy & Internet* 4.3-4 (2012), pp. 40–71. DOI: https://doi.org/10.1002/poi3.10. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/poi3.10. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/poi3.10 (cit. on p. 57).

[16] Ben Walford. *What is GDPR, the EU's new data protection law?* 2018. URL: https://gdpr.eu/what-is-gdpr/ (cit. on p. 58).

[17] European Commission. *The European Data Governance Act*. 2022. URL: https://digital-strategy.ec.europa.eu/en/policies/data-governance-act (cit. on p. 58).

[18] European Commission. *The European Data Act*. 2022. URL: https://digital-strategy.ec.europa.eu/en/library/data-act-proposal-regulation-harmonised-rules-fair-access-and-use-data (cit. on p. 58).

[19] Coral Walker and Hassan Alrehamy. "Personal Data Lake with Data Gravity Pull". In: *2015 IEEE Fifth International Conference on Big Data and Cloud Computing*. 2015, pp. 160–167. DOI: 10.1109/BDCloud.2015.62 (cit. on p. 58).

[20] Lars Nagel and Douwe Lycklama. *Design Principles for Data Spaces - Position Paper*. Version 1.0. July 2021. DOI: 10.5281/zenodo.5105744. URL: https://doi.org/10.5281/zenodo.5105744 (cit. on p. 58).

[21] *Envoy*. URL: https://digital-strategy.ec.europa.eu/en/policies/data-governance-act (cit. on p. 61).

[22] *iPerf*. URL: `https://iperf.fr/` (cit. on p. 64).