Politecnico di Torino

Politecnico di Torino

Master's degree in Computer Engineering

A.a. 2024/2025

Graduation Session October 2025

Optimizing Peer-to-Peer Communication Among Multiple Kubernetes Federated Clusters

Supervisors:

Prof. Fulvio RISSO

Dott. Davide MIOLA

Candidate:

Gabriele SANTI

Summary

The growing adoption of Kubernetes in multi-cluster and heterogeneous environments has led to the development of frameworks such as Liqo, an open-source add-on that enables dynamic federation of Kubernetes clusters, resource sharing, and transparent workload offloading. While Liqo provides robust primitives for inter-cluster communication, its networking model traditionally enforces a consumer-centric topology, where all traffic exchanged between provider clusters must transit through the consumer. This approach introduces redundant hops, increased latency, and potential bottlenecks, thereby limiting efficiency and scalability.

This thesis proposes and implements an architectural enhancement to Liqo's overlay network to address these limitations in scenarios where a consumer offloads workloads to multiple providers simultaneously. The new mechanism allows, when a direct inter-provider connection is available, traffic to be routed directly between providers, bypassing the consumer cluster. The functionality is configurable on a per-Service basis, offering fine-grained control and improved flexibility. In addition to shortening network paths and reducing potential single points of failure, this design lays the groundwork for the future integration of a service mesh capable of dynamically optimizing inter-cluster routing decisions.

Although no quantitative benchmarks have been conducted yet, the architectural benefits are evident: inter-provider communication is simplified, routing overhead is reduced, and network responsiveness to the direct connection usage is not compromised. The proposed enhancement is currently under review and is planned for inclusion in a future release of Liqo, contributing to the evolution of efficient and resilient networking in federated Kubernetes environments.

Acknowledgements

Vorrei aprire questa sezione ringraziando il professor Fulvio Risso, Davide Miola e tutti i ragazzi di Data plane per avermi supportato durante tutti questi mesi. Ricordo con un sorriso il primo meeting a cui ho partecipato, nonostante non avessi capito pressoché niente, nessuno mi ha mai giudicato e sono sempre stati tutti disponibili a spiegare tutto quello che potevano.

Un grazie enorme va alla mia famiglia: babbo, mamma e Viola. Nel mio lunghissimo percorso hanno sempre dimostrato di credere in me, anche quando i risultati faticavano ad arrivare.

Grazie mille a Chiara, che mi ha accompagnato in questo percorso. In lei ho trovato una compagna, un'amica e una confidente, sempre in prima linea per aiutarmi quando le cose si facevano difficili. Grazie di avermi supportato e sopportato, altri mille giorni così.

Grazie alla famiglia Cuccaro, per avermi accolto quando la mia famiglia vera era lontana.

Grazie a tutti gli amici di una vita: Giacomo, Fede, Gabri, Cecca e tutti gli altri ed altre; mi hanno dimostrato di esserci sempre, non dimenticherò mai le risate fatte su Discord quando era l'unico modo per vedersi.

Ed infine, grazie anche a tutte le persone con cui condividerò i festeggiamenti per questo traguardo, anche se non avete il nome qui sopra, avrete sempre tutto il mio affetto.

Table of Contents

List of Figures

Chapter 1

Introduction

The widespread adoption of cloud computing and containerization technologies has profoundly transformed the way modern applications are designed, deployed, and maintained. In this containerized context, Kubernetes emerged as the *de facto* standard for container orchestration, providing a powerful and **extensible** platform to manage workloads at scale. Its native support for declarative management, self-healing mechanisms, and extensibility has made Kubernetes the cornerstone of today's cloud-native ecosystem.
However, as organizations increasingly embrace microservices-based architectures, applications no longer reside within the boundaries of a single cluster. While Kubernetes effectively manages workloads inside an individual cluster, it offers limited support for scenarios where multiple clusters must collaborate seamlessly. This limitation becomes critical in contexts such as multi-cloud deployments, edge computing environments, and geo-distributed infrastructures, where scalability, resource efficiency, and fault tolerance require the federation of diverse and heterogeneous clusters.
To address this gap, **Liqo**, an open-source project started at Politecnico di Torino, extends Kubernetes beyond the single-cluster model, enabling the federation of multiple clusters as if they were a single, unified system. Through Liqo, workloads can be transparently offloaded across clusters, resources can be shared dynamically, and inter-cluster communication can be managed efficiently.

1.1 Goals

The primary objective of this thesis is to **enhance Liqo's networking capabilities** by introducing a feature that enables direct communication between provider clusters. In the current implementation, inter-cluster traffic follows a hub-and-spoke model, where the consumer cluster acts as an intermediary for all traffic

between providers. (Figure 1.1, all the components shown will be discussed in detail in the following chapters.)

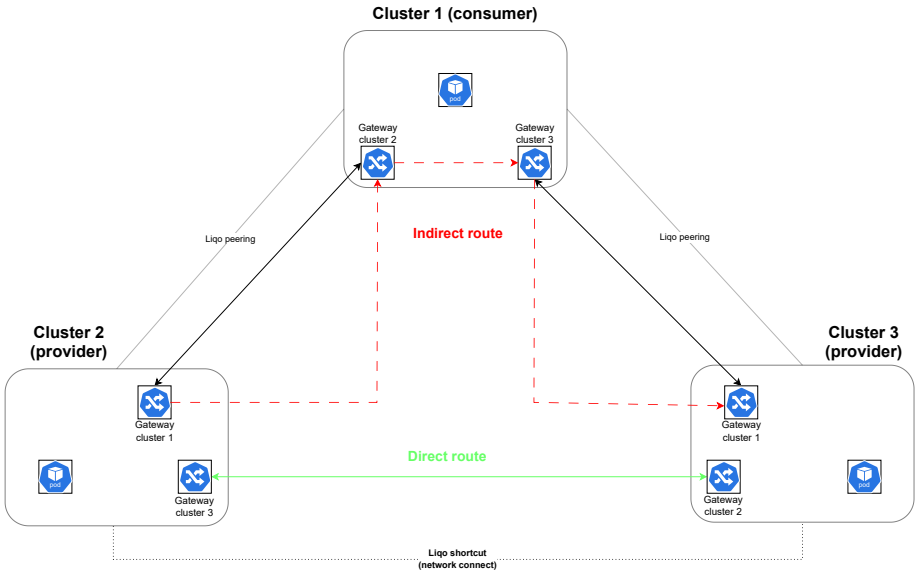

Figure 1.1: Direct and indirect pod communication between peered clusters in Liqo.

While this approach ensures centralized control and uniform traffic management, it also introduces several drawbacks:

- It creates a potential bottleneck that limits available bandwidth,
- It increases the networking overhead on the consumer cluster, and
- It may degrade performance for distributed workloads that rely on high-volume east–west communication.

To address these limitations, the thesis proposes the design and implementation of an optional feature allowing traffic to flow directly between provider clusters, bypassing the consumer when centralized routing is unnecessary. This optimization aims to:

- Improve bandwidth utilization by leveraging direct network paths,

- Reduce the workload on the consumer cluster, freeing it from unnecessary networking duties, and
- Increase the efficiency of cross-cluster communication for distributed, high-throughput applications.

At the same time, the feature is designed to **remain configurable and optional**, as in some scenarios it is desirable or even necessary to enforce routing through the consumer cluster (e.g., for monitoring, auditing, or policy enforcement). By providing this flexibility, the work ensures that Liqo can better adapt to heterogeneous use cases.

1.2 Collaboration With Eng

This thesis has been developed in collaboration with the company Engineering Ingegneria Informatica S.p.A.[1], as part of the European project IPCEI-CIS AVANT[2], funded by the European Union with program NextGenerationEU.

1.2.1 What is IPCEI-CIS AVANT

IPCEI stands for Important Project of Common European Interest. These are large-scale, strategic projects co-funded by multiple EU Member States and approved under EU state-aid rules, to support innovation, infrastructure, or technologies of major EU importance.

CIS means Cloud Infrastructure and Services. IPCEI-CIS is the IPCEI focused on developing next-generation cloud and edge computing infrastructures and services in Europe.

AVANT (an acronym from dAta and infrastructural serVices for the digitAl coNTinuum) is one of the projects under the umbrella of IPCEI-CIS. It is led by Engineering and aims to deliver advanced cloud-to-edge technologies, flexible infrastructures, interoperability, and open source components.

1.3 Structure of the Thesis

The thesis is organized as follows:

- Chapter 2 introduces the fundamental concepts of Kubernetes, with a particular focus on its networking architecture
- Chapter 3 presents Liqo, detailing its architecture, the Kubernetes resources it introduces, peering mechanism, and networking model.

- Chapter 4 briefly presents the work on this subject carried out by a colleague, on which this thesis partly builds.
- Chapter 5 describes the first implementation of the proposed feature, highlighting the design choices and integrations with the `foreign_cluster_connector` solution.
- Chapter 6 discusses the final implementation, outlining improvements, optimizations, and the ideas behind architectural and technical decisions.
- Chapter 7 concludes the thesis, summarizing the results and proposing directions for future work.

Chapter 2

Kubernetes

This chapter provides a historical and technical overview of Kubernetes, focusing on its architecture, core resources, and networking model. It establishes the foundational concepts necessary to understand the extensions and advanced orchestration patterns —such as multi-cluster federation— addressed in the following chapters.

2.1 Introduction to Kubernetes

"Kubernetes is a *portable*, *extensible*, *open source* platform for managing containerized workloads and Services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes Services, support, and tools are widely available." [3]

The origins of Kubernetes can be traced back to *Google*'s internal infrastructure expertise, particularly its pioneering work on container management systems. In the early 2000s, Google developed *Borg*, its first unified system for managing containerized workloads across its vast data centers. Borg was crucial for running services like Gmail and Google Search, enabling high resource utilization, fault tolerance, and scalability for Google's large-scale operations.

A significant development in the container ecosystem occurred in 2013 with the release of *Docker*. Developed by dotCloud, Docker is an open-source software tool that popularized lightweight container technology, simplifying the packaging, distribution, and deployment of applications. While Docker revolutionized cloud-native infrastructure, its limitation of primarily running on a single node highlighted a crucial need for a system capable of orchestrating multiple containers across numerous machines. Recognizing this, Google engineers Craig McLuckie, Joe Beda, and Brendan Burns, who had worked on Borg, conceived an open-source container orchestrator that would become Kubernetes.

In 2015, with the release of Kubernetes 1.0, Google further cemented its commitment by donating the project to the **Cloud Native Computing Foundation** (CNCF)[4], an initiative aimed at making cloud-native computing ubiquitous. Since then, Kubernetes has rapidly grown, becoming the CNCF's first graduated project by 2018 and surpassing competitors to become the industry standard for container orchestration. Its profound impact aids in the development of cloud-native microservices, enables faster application development, and provides the automation and observability essential for modern application management.

2.2 Architecture

Kubernetes adopts a modular architecture that clearly separates the *control plane* from the *data plane*, a design pattern inherited from the networking domain to promote scalability, and manageability.

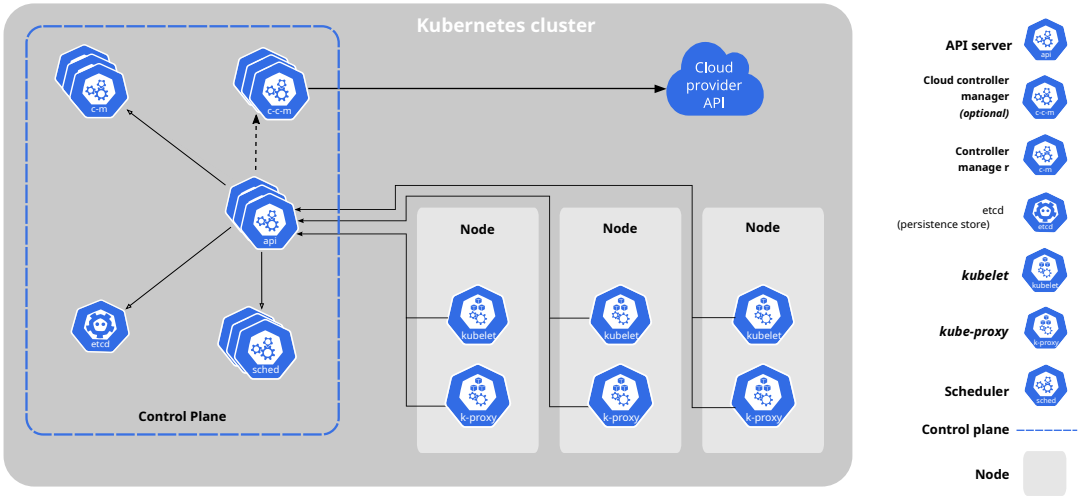

Figure 2.1: Overview of the architecture of a Kubernetes cluster. Source: [3]

2.2.1 Control Plane Components

The control plane is responsible for the global management and orchestration of the cluster: it makes decisions about scheduling, scaling, and the overall desired state of the system. Key components such as the `kube-apiserver`, `kube-controller-manager`, `kube-scheduler`, and `etcd` (the cluster's backing store) reside in the control plane, collectively ensuring that the cluster operates according to user specifications and policies.

A defining feature of Kubernetes, which is implemented by the control plane, is its **declarative approach** to cluster management. Users specify the desired state of the system —such as which applications should be running, their configurations, resource requirements, and much more— using manifest files (typically in YAML or JSON format).

The control plane continuously monitors the actual state of the cluster and takes automated actions to reconcile any differences, ensuring that the current state matches the user-declared specification. This model abstracts away the complexity of manual operations, enabling robust automation and self-healing capabilities.

All the following components can run on any machine in the cluster, but for simplicity, they are often located on a single node.

`etcd`

`etcd` is a distributed key-value store that provides a reliable way to store the cluster's state data. It is used by the control plane to persist all configuration data, state information, and metadata about the cluster. `etcd` is designed to be **highly available** and resilient, ensuring that the cluster can recover from failures and maintain consistency across all nodes.

It is based on the Raft consensus algorithm, which allows different nodes to work as a coherent group, ensuring data consistency and fault tolerance. Only the API server interacts directly with `etcd`, abstracting its complexity from other components and users.

`kube-apiserver`

At the core of Kubernetes is its API server (implemented by `kube-apiserver`), which represents the central entry point for all interactions with the cluster. It exposes an *HTTP interface* through which both internal components and external clients communicate with the control plane. Every operation in Kubernetes—whether deploying a workload, scaling an application, or modifying configuration—is expressed as an HTTP request handled by `kube-apiserver` which processes them, validates them, and updates the cluster's state in `etcd`.

Users typically interact with `kube-apiserver` through client tools rather than forging direct HTTP requests. The most common utility is `kubectl`, a command-line interface that translates user-friendly commands into API requests. Beyond `kubectl`, users can also leverage various client libraries in languages like Go, Python, or JavaScript.

`kube-scheduler`

The scheduler is the component that assigns the workload (represented by *Pods*) to specific nodes in the cluster based on resource availability, constraints, and policies.

`kube-controller-manager`

The `kube-controller-manager` is responsible for running controller processes. It continuously monitors the state of the cluster and makes or requests changes as needed to reconcile the actual state with the desired state defined in `etcd`.

Each controller operates on a specific resource of the cluster, but to reduce complexity, they are all compiled into a single binary and run in a single process.

Some examples of controllers are:

- **Node Controller**: Monitors the health of nodes and takes action when they become unreachable.
- **Replication Controller**: Ensures that the desired number of Pods are running at all times.
- **EndpointSlice Controller**: Populates EndpointSlice objects, which link Services and Pods.
- **Service Account and Token Controllers**: Manage service accounts and their associated tokens, which are used for authentication within the cluster.

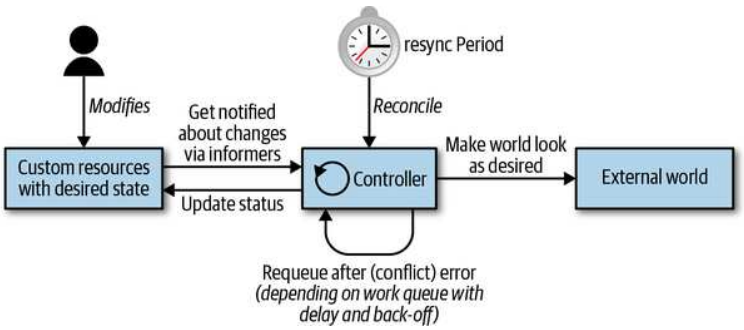

Figure 2.2: Overview of the how the operator pattern works. Source: [5]

`cloud-controller-manager`

This component allows the integration of Kubernetes with various cloud service providers. It manages cloud-specific resources, such as load balancers, storage volumes, and networking components.

By decoupling cloud-specific logic from the core Kubernetes components, it enables a more modular and flexible architecture, allowing it to run seamlessly across different environments.

2.2.2 Data Plane Components

The data plane comprises the worker nodes and their local agents (primarily the *kubelet* and *kube-proxy*), which form the execution environment for containerized applications.

Unlike the control plane which makes orchestration decisions, worker nodes focus on workload execution and resource provision. Each worker node runs critical components—primarily the *kubelet* and *kube-proxy*—that translate the control plane's instructions into action.

These components are responsible for running the actual application workloads (encapsulated in *Pods*), managing local container lifecycle, and handling network traffic according to the rules established by the control plane.

Container Runtime

The container runtime is the software component in charge of running containers on each node. Kubernetes supports multiple container runtimes, including Docker, `containerd`, and `CRI-O` and any other implementation of the *Kubernetes Container Runtime Interface* (CRI)[6].

`kubelet`

The `kubelet` makes sure that the containers are running in a Pod and are healthy. The `kubelet` acts as a bridge between the API server and the container runtime, receiving Pod specifications, monitoring execution, and reporting the status back to the control plane so that action can be taken.

`kube-proxy`

`kube-proxy` is a network agent that runs on each node in the Kubernetes cluster. It maintains network rules that enable network communication to Pods from both inside and outside the cluster. If present, it leverages the operating system's native packet filtering capabilities (`nftables` or `IPVS`) to implement Service abstractions; otherwise it can also use a user-space proxy mode.

Add-ons

Features and functionalities that are not part of the core Kubernetes components but implemented by third parties to provide additional features and functionalities.

Some examples are: Istio[7] for Service mesh, Prometheus[8] for metrics collection, Grafana[9] for visualization or **Liqo**[10] for multi-cluster orchestration.

2.3 Kubernetes Resources

Kubernetes resources are persistent entities in the cluster that represent the **desired** state of various aspects of the system. Specifically, they describe:

- What applications or workloads are running and on which nodes (e.g., Pods, DaemonSets).
- How these applications should be configured (e.g., ConfigMaps, Secrets).
- The policies governing their behavior such as restart policies or fault-tolerance. (e.g., Deployments, ReplicaSets).
- The resources available to those applications in terms of CPU or memory.
- How they should be exposed to the network (e.g., Services, Ingress).

Once a resource is created, the control plane will constantly work to ensure that that object exists and its specifications are met. Usually, a resource object contains the following fields:

- `apiVersion`: Specifies the version of the Kubernetes API that the object uses;
- `kind`: Indicates the type of resource (e.g. Pod, Service, Deployment);
- `ObjectMeta`: Contains data that helps uniquely identify the object, including a name, namespace, labels, and annotations;
- `Spec`: Defined by the user, it represents the desired state of the resource, including configurations and settings specific to the resource type;
- `Status`: Populated by the server, it reports the current state of the resource as observed by the system.

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

- **Create**: Creates the resource in the backend; once created, the system applies the desired state;
- **Read**: Retrieve information about the resource, comes in three variants;
  - **Get**: Fetch a specific resource by name;

  - **List**: Fetch a collection of resources of a specific type within a single namespace. The result can be filtered using labels and field selectors;
  - **Watch**: Establish a streaming connection to the API server that receives notifications about changes to resources. This allows clients to react to resource modifications in real-time without constant polling.
- **Update**: Modify the specifications of an existing resource, two modes are supported;
  - **Patch**: Apply a partial update to a resource;
  - **Replace**: Replace an existing resource with a new one.
- **Delete**: Remove a resource from the cluster.

In the following, some of the most relevant resources for this work are described in more depth.

2.3.1 Pods

Pods are the smallest deployable units of computing on Kubernetes. A Pod is a group of one or more containers with shared storage and network resources. They are *ephemeral*, meaning they are created, destroyed, and re-created on demand. Pods are designed to be lightweight and transient, making them ideal for running microservices[1].

ReplicaSet

A ReplicaSet is a higher-level abstraction that manages a set of identical Pods, ensuring that a specified number of replicas are running at any given time. If a Pod fails or is deleted, the ReplicaSet automatically creates a new one to maintain the desired state.

Deployment

Deployments provide a higher-level abstraction for managing Pods and ReplicaSets. While ReplicaSets ensure a specified number of identical Pods are running, Deployments add more lifecycle management capabilities that simplify application operations. They enable crucial features such as rolling updates (gradually replacing old Pods with new ones), version rollbacks, and controlled scaling.

[1]Microservices are an architectural approach in which an application is composed of small, independent services that communicate over well-defined APIs. Each service focuses on a specific business capability and can be developed, deployed, and scaled independently.

Namespaces

Namespaces are the mechanism Kubernetes uses to provide logical isolation between resources within a single cluster. They create virtual boundaries that divide cluster resources, allowing multiple teams, projects, or applications to share the same physical infrastructure without interfering with each other. Names of resources need to be unique within a namespace, but not across namespaces, enabling resource naming simplification.

A real-world example

The following Listing 2.1 is an example of a Kubernetes Deployment manifest. It's written in YAML format, a human-readable serialization standard commonly used for configuration files. YAML is often preferred to JSON thanks to its improved readability, though the Kubernetes API supports both.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: demo-namespace
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.19
        ports:
        - containerPort: 80
```

Listing 2.1: Example Kubernetes Deployment.

In the `metadata` section, the user specifies the name of the deployment and the namespace where it will be created.

The `spec` field represent the specification the user desires, it's up to the system to achieve the described situation.

The `replicas` field is self explanatory, under the hood it creates a ReplicaSet resource, which effectively manages the Pod replicas.

`selectors` are used to identify the Pods that belong to this deployment, in this case, all Pods with the label `app:nginx` will be managed by this deployment.

The `template` field contains the template to be used for the created Pods. Since Pods are resources as well, they are again populated with `metadata` and `spec`. This last field has info on the Pod itself, in this case the container image to be used and the network ports to be exposed.

This example could be easily put in operation using the `kubectl` tool, in this case with the command:

```
kubectl apply -f deploymentsample.yaml
```

2.3.2 Services

Services are fundamental Kubernetes resources that provide network connectivity to a set of Pods. They abstract away the ephemeral nature of Pods by providing a **stable endpoint** that remains constant even as the underlying Pods are created, terminated, or replaced. This abstraction is crucial for microservice architectures where components need to reliably communicate with each other.

A Service works by defining a logical set of Pods using label selectors and exposing them through a single DNS name and network port. When a request is made to a Service, it routes traffic to one of the backing Pods using a load-balancing algorithm.

Kubernetes supports several types of Services to accommodate different networking requirements:

- **ClusterIP**: The default type that exposes the Service on an internal IP accessible only within the cluster. This is useful for internal communication between microservices and will be the one on which we are focusing on this thesis as it is the one used intra-cluster.
- **NodePort**: Extends ClusterIP by exposing the Service on a static port on each node's IP. This makes the Service accessible from outside the cluster by reaching any node at `<NodeIP>:<NodePort>`.
- **LoadBalancer**: Extends NodePort by provisioning an external load balancer (when supported by the cloud provider) that routes traffic to the Service.
- **ExternalName**: Maps the Service to a DNS name rather than selecting Pods, primarily used for accessing external services from within the cluster.

2.3.3 EndpointSlices

EndpointSlices represent a collection of network endpoints that provide addresses and ports that Services use to route traffic to Pods. When a Pod is deployed, the EndpointSlice controller automatically creates or updates the corresponding

EndpointSlice objects to include the Pod's IP address and port information and populates the associated Service with those values.

2.3.4 Legacy: Endpoints

In Kubernetes versions prior to 1.16, the Endpoint resource was used in place of the EndpointSlice. The main difference is that Endpoints were monolithic objects, containing all the endpoints for a Service. This caused update inefficiency and scalability issues, especially for Services with a large number of endpoints since an update to a single endpoint required updating the entire Endpoints object.

This led to the gradual introduction of EndpointSlices, which store endpoints in sets of maximum 100 endpoints ("slices"), making updates more efficient.

2.4 Kubernetes Network Architecture

After discussing how Kubernetes works and its core resources, it's important to understand how the network is handled to make communication between Pods and resources possible.

This is defined by the **Kubernetes networking model** [11], which establishes some principles for understanding how networking should work in a cluster:

- Every Pod has its own IP address.
- All containers within a Pod share the same IP address and port space.
- All Pods can communicate with all other Pods in the cluster **without NAT**.
- Agents on a node (e.g., `kubelet`, `kube-proxy`) can communicate with all Pods on that node.
- Isolation is enforced through particular resources called *network policies*, not with addressing management.

The result of these principles is that Pods can be used as if they were hosts, and the containers inside, as processes. This simplifies the development of distributed applications and the migration from systems based on VMs. Moreover, since isolation is provided by network policies, the network structure can be kept simple.

However, these are only principles and Kubernetes does not provide a default implementation. In facts, it relies on third-party solutions, called *Container Network Interface* (CNI)[12] plugins, to actually implement the networking model. Some examples of CNI plugins are *Calico*[13], *Flannel*[14], and *Cilium*[15].

Two main models are typically employed for cluster networking plugins: **flat** and **overlay** network architectures.

2.4.1 Flat Network

This model aligns closely with the networking principles by providing direct routability between Pods without encapsulation or NAT. In a flat network, Pod IP addresses are part of the physical network's routing domain, making them directly accessible across the cluster.

The benefits of this architecture are:

- No need for packet encapsulation or decapsulation between nodes.
- Lower CPU overhead, as no additional tunnel processing is required.
- Reduced bandwidth overhead, since fewer protocol headers are transmitted over the network.

2.4.2 Overlay Network

Instead, overlay networks create a virtual network on top of the existing physical network infrastructure. They use encapsulation techniques (like VXLAN[16] or GRE[17]) to enable Pod-to-Pod communication across different nodes, even if those nodes are on different subnets.

This approach provides several advantages:

- Works across almost any network infrastructure without special configuration (particularly useful in cloud or multi-tenant environments)
- Isolates Pod IP space from the physical network, preventing address conflicts.
- Enables deployment across networks where direct Pod-to-Pod routing would otherwise be impossible.

2.4.3 Linux Network Namespaces

Before describing the scenarios of communication, it's important to understand what Linux namespaces are, in particular the *network namespace*.

Namespaces are a mechanism offered by Linux to isolate and virtualize kernel resources for a set of processes. There are different types of namespaces, each isolating specific resources, such as process IDs, mount points, or the network stack.

The **network namespace** (also known as *netns*) provides isolation for network resources, including interfaces, IP addresses, routing tables, and firewall rules. Each network namespace has its own set of these resources, allowing processes within a namespace to have their own independent network stack.

When a Pod is created, the CNI assigns it its own network namespace.

2.4.4 Pod Networking

Container-to-Container

Inside a Pod, containers share the same network namespace, meaning they can communicate directly through the *loopback* interface and the respective ports.

Pod-to-Pod on the same node

Pods on the same node leverage the root network namespace of the node to communicate with each other. As shown in Figure 2.3, when a Pod is deployed on a node, the CNI creates a pair of **virtual interfaces**. One end is placed inside the Pod's network namespace, while the other end is the node's root network namespace.[2]

Together with a **virtual bridge**, this allows the communication between Pods using only the Linux networking stack.

Pod-to-Pod on different nodes

This is the scenario where the CNI matters the most, in fact, when the traffic needs to go outside the node, it needs to know how to reach the destination Pod.

In case a CNI implementing a flat network is used, routes are pushed in the node routing table to reach the Pod IPs through the corresponding node IPs. Remember that in a flat network, nodes do not have overlapping IPs, so the routing becomes simpler.

In case an overlay network is used, the traffic is encapsulated and sent to the destination node, where it is decapsulated and delivered to the Pod. The encapsulation method depends on the CNI.

Pod-to-Service

The main difference that comes into play when communicating with a Service is that the Pod does not need to know the IP addresses of the Pods it wants to communicate with, but only the IP address (or DNS name) of the Service.

This traffic is intercepted by the `kube-proxy` component running on the node where the Pod is located which will take care of routing the traffic to one of the backing Pods by applying a load balancing algorithm with the help of *iptables* or *IPVS*.

[2]This behavior is common but may vary depending on the CNI implementation.

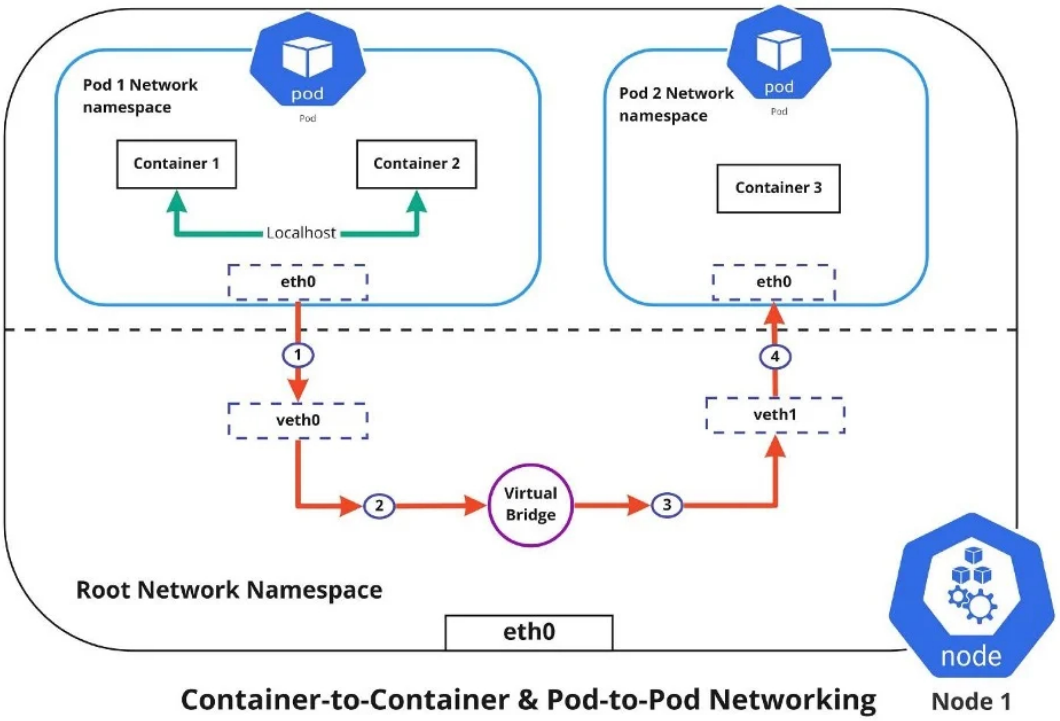

Figure 2.3: A visual representation of the container-to-container and Pod-to-Pod networking. Source: [11]

2.5 Note on Security and Network Policies

While security is a critical aspect of Kubernetes operations, it is not the primary focus of this thesis. Topics such as authentication, authorization, and network policies are only briefly mentioned to provide context for the architectural and networking discussions.

Chapter 3

Liqo

This chapter introduces the concepts and architecture of **Liqo**, which is "an open-source project that enables dynamic and seamless Kubernetes **multi-cluster** topologies, supporting **heterogeneous** on-premise, cloud and edge infrastructures." [18].

3.1 Overview

While Kubernetes supports the orchestration of workloads within a single cluster, Liqo extends this capability by allowing the federation of multiple clusters by leveraging the extensibility of Kubernetes. Also, Liqo is designed to be deployable on any Kubernetes cluster, regardless of the underlying infrastructure or cloud provider.

By federating —or "peering", to use its jargon— clusters, Liqo allows **workloads to be offloaded** across clusters, and resources to be shared dynamically.

To achieve this, Liqo creates on the local cluster a **virtual node** which represents the resources of the remote peered cluster which are available to the local cluster. This virtual node is managed by a component called **Virtual Kubelet**[19], which acts as a bridge between the two clusters, handling the scheduling and the lifecycle of Pods that are offloaded to the remote cluster. On the other side, it connects to the remote cluster's API server to manage the synchronization of resources and workloads.

Some practical use cases of Liqo include:

- **Hybrid Cloud Deployments**: Extend on-premise clusters to public clouds, enabling dynamic resource scaling and workload migration.
- **Edge Computing**: Connect edge clusters to central data centers, allowing workloads to be offloaded to or from edge locations based on resource

availability and latency requirements.

- **Resource Optimization**: Share resources between clusters to maximize utilization, reduce costs, and balance workloads.
- **Bursting Workloads**: Offload peak workloads to remote clusters during high demand periods, avoiding resource shortages.

Liqo comes with a set of components and custom resources (which will be presented in the following sections) together with a CLI, called `liqoctl`, which is used to install, configure, and monitor Liqo on the clusters. Another way to install and manage Liqo is through Helm[20] charts.

3.2 Peering

Peering is a procedure that allows two clusters to connect and share resources. This process creates a **unidirectional** relationship: the *consumer* cluster is the one that offloads resources (also called *local cluster*), and the *provider* cluster (also called *remote cluster*), which provides resources. Note that it can be performed by both clusters, making possible bidirectional peerings and more complex topologies when more peerings are done on multiple clusters.

This connection is established through a handshake mechanism that involves the exchange of **cryptographically signed nonces** and **network configuration details**. This is necessary because clusters communicate over **VPN tunnels** using **WireGuard**[21], ensuring secure and encrypted inter-cluster communication.

These tunnels are entirely created and managed by Liqo, which also handles the routing of all the traffic involved by using some components that will be presented in the next sections.

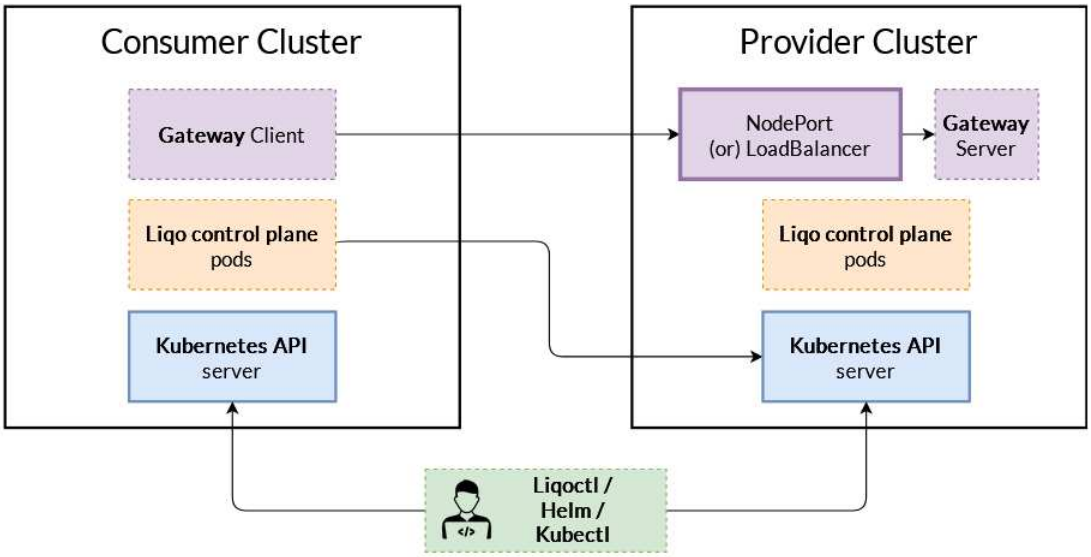

Figure 3.1: Overview of two Liqo peered clusters.

To initiate a peering, the `liqoctl` CLI tool is used. It requires both the kubeconfigs of the involved clusters plus some additional parameters. For more details, refer to the official documentation [18].

Liqo allows also to create connection between clusters without fully peering them, in this case we have a gateway server and client but there is no consumer-provider relationship. In fact, this kind of connection cannot be used for offloading, but only for inter-cluster communications. This kind of connection is named after the command to create it, which is `liqoctl network connect`.

3.3 Offloading

Offloading is the process of transferring workloads from one cluster to another. In Kubernetes jargon, it translates to scheduling Pods on a remote cluster instead of the local one.

Offloading is enabled by the **virtual node** created at the end of the peering process; it is created in the local cluster (the consumer) and represents the resources of the remote cluster (the provider) that are available to the local cluster. The starting amount is negotiated during the peering process and if the consumer needs more resources, it can request them in the form of additional `ResourceSlice` which need to be accepted by the provider in order to be used.

The virtual node acts as a physical node in the consumer cluster, this way

Kubernetes can schedule Pods on it. When this happens, instead of the normal `kubelet`, it runs the **Virtual Kubelet**, which takes care of translating the Pod specifications and sending them to the remote cluster's API server.

This way, the consumer cluster can see the deployed Pods as if they were running locally, while in reality, they are executed remotely.

3.4 Virtual Kubelet

The **Virtual Kubelet** is an open-source `kubelet` implementation that masquerades as a kubelet but does not manage any real node. Instead, it connects to other APIs or services to manage the lifecycle of Pods.

Whenever a peering is established, a Virtual Kubelet is deployed on the consumer cluster, associated with a virtual node to a provider cluster.

In Liqo, a custom version is employed which is responsible for:

- forwarding the Pod specifications to the remote cluster's API server, so that Pods are effectively deployed on a physical node,
- monitoring the status of the offloaded Pods and updating their status in the local cluster.

This way, the Virtual Kubelet can be seen as a bridge between the two clusters.

3.5 Resource Reflection

The reflection is another mechanism that Liqo uses to effectively offload workloads. It consists of synchronizing certain resources between the two clusters, ensuring that both clusters have a consistent view of the resources and workloads.

This is accomplished by leveraging **Kubernetes namespaces**: the user chooses the namespace to offload Pods from, and Liqo creates a corresponding "twin" namespace in the provider clusters.

These twin namespaces will then host the offloaded Pods, together with other resources that need to be synchronized, such as Services or ConfigMaps.

3.6 Network Fabric

The network fabric is the Liqo subsystem used to extend the Kubernetes network model across multiple clusters. It ensures that Pods running in different clusters can communicate with each other with or without NAT translation. The figure 3.2 shows an high-level overview of the Liqo network fabric.

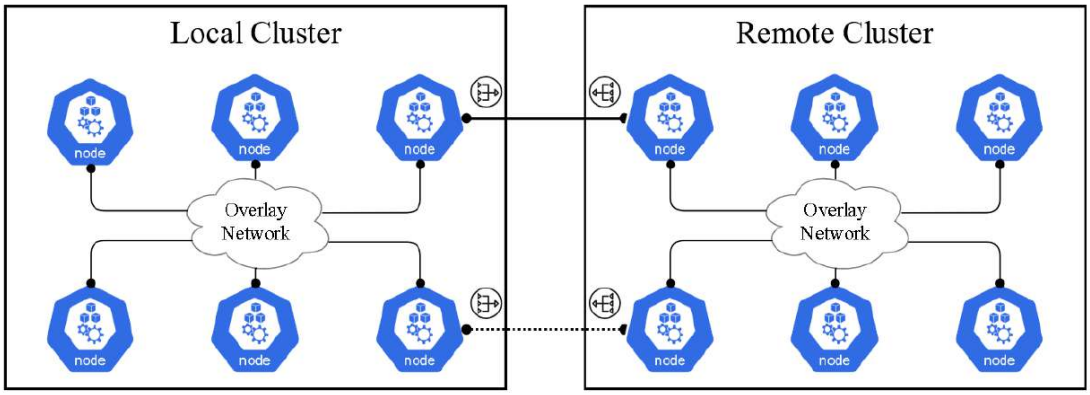

Figure 3.2: Liqo network fabric.

Practically, the network fabric is implemented as a network agent running on every node (deployed via a DaemonSet). Each agent programs local routing so that traffic destined for remote Pods is forwarded to the appropriate Liqo Gateway and sent over WireGuard tunnels. The controller-manager provisions these forwarding and translation rules when a peering is established.

3.6.1 CIDR Remapping

The support that Liqo offers for multiple CNIs and network configurations makes it **impossible to guarantee non-overlapping Pod CIDRs among clusters**. To overcome this problem, Liqo employs a feature called **remapping**, which consists of translating Pod IPs according to the cluster in which they are running.

Referring to Figure 3.3, consider two clusters, A and B with the same local Pod CIDR. Remapping an address means changing the remote Pod CIDRs while keeping the same host part: this way A sees all the addresses of the Pods deployed on B as belonging to a different Pod CIDR, and the same on B.

Of course, this remapping **is negotiated** during the connection setup, as both clusters need to agree on the chosen CIDRs to route the traffic correctly.

Figure 3.3 below shows an example:

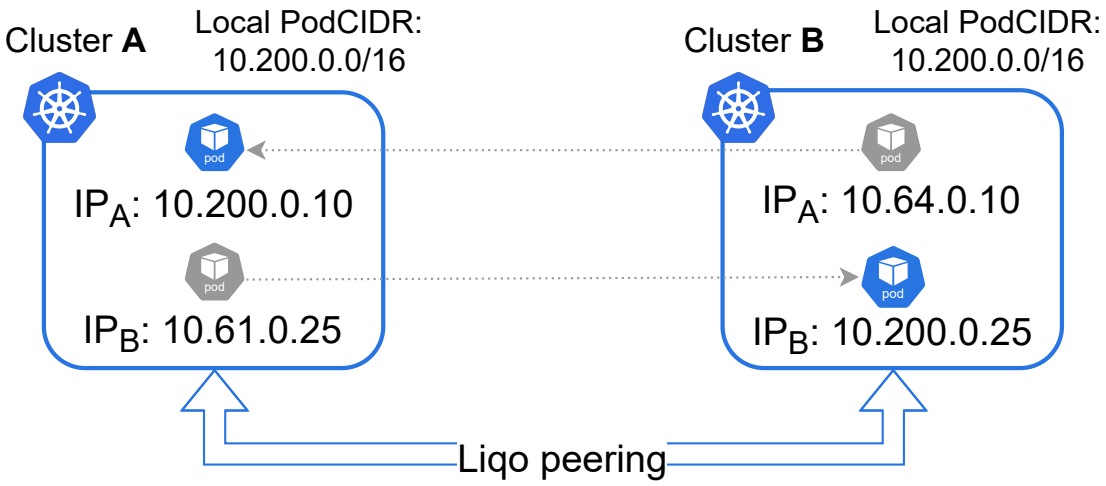

Figure 3.3: Liqo remapping example: A remaps all the addresses in cluster B with 10.61.0.0/16, while B remaps cluster A with 10.64.0.0/16.
Note that, despite the CIDR remapping, the host part is maintained.

Thus, remapping enables clusters to **"have their own point of view on their neighbors"**, in networking terms.

Moreover, it affects only Pod IPs, that is, the endpoints written in the EndpointSlices (2.3.3) of each cluster. This means that the native behavior of Services is not altered.

Remapping is not always necessary: if the Pod CIDRs of the two clusters do not overlap, Liqo can be configured to avoid doing it and allow clusters to communicate using their own CIDRs.

3.7 Liqo Custom Resources (CRDs)

Liqo extends the Kubernetes API by introducing several Custom Resource Definitions (CRDs) to manage its components and functionalities. Here is a list of the most relevant ones for this thesis:

3.7.1 VirtualNode

VirtualNode was already mentioned. It represents a remote cluster in the local cluster which is masqueraded as a node. It contains information about the remote cluster's resources, status, and configuration.

3.7.2 Connection

Connection represents the Liqo connections made with other clusters. It comprises both the peerings and the network connections, informing about their status, with a measure of the latency. It is used to check if the connection is up and running.

3.7.3 Configuration

Configuration describes the remappings performed by the cluster to the others. It contains all the CIDRs used by the cluster, both the original and the remapped ones. It is used to configure the network fabric.

The following is an example of the content of a Configuration CR. The wide output shows also the ExternalCIDRs but it was omitted for space reasons.

```
$ kubectl get configurations -n liqo-tenant-cluster2
NAME       DESIRED POD CIDR    REMAPPED POD CIDR   AGE
cluster2   ["10.200.0.0/16"]   ["10.60.0.0/16"]    11d
```

3.7.4 ShadowEndpointSlice

ShadowEndpointSlice is a resource used to reflect the endpoints of an EndpointSlice. It is created and populated in the local cluster and sent to the remote cluster with the endpoints **as seen by the local cluster**.

Once received by the remote cluster, it is handled by the ShadowEndpointSlice controller (in the `liqo-controller-manager`), which creates its own the EndpointSlice performing a remapping if necessary.

The creation of the ShadowEndpointSlice is managed by the **virtual kubelet**, which transmits only the endpoints of Pods running outside the cluster that it is reflecting to; the endpoints of the Pods deployed locally are already managed by the native EndpointSlice controller.

The following Figure 3.4 shows a representation of the EndpointSlice reflection, highlighting the role of the ForeignCluster CR, which provides the status of the peered cluster.

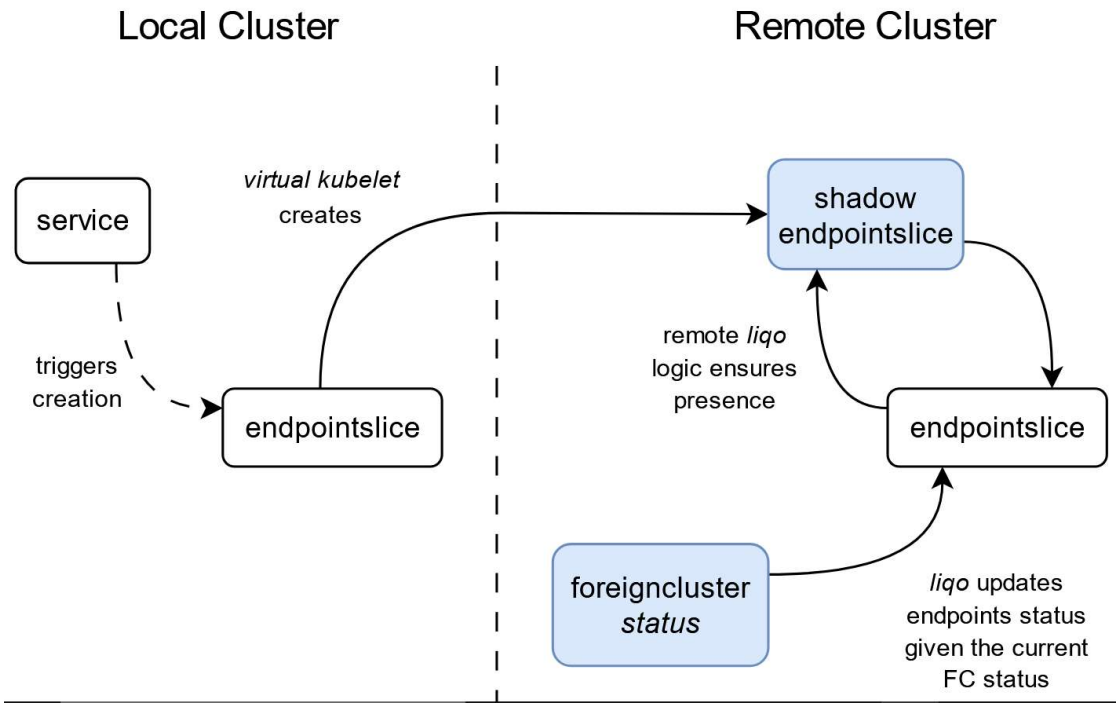

Figure 3.4: Schematic representation of the endpointslice reflection workflow. Solid lines refer to liqo-related tasks, while dashed ones to standard Kubernetes logic. Blue rectangles refer to liqo-related resources.

3.7.5 IP

IP CR represents a single IP address and specifies what is its remapping on the external CIDR of the local cluster.

Offloaded Pods trigger the creation of an IP CR: local ones do not because their address is already managed by the native Kubernetes components. Manually created IP CRs can be used to make external IPs reachable from other clusters.

The following section describes the purpose of the external CIDR in Liqo.

Example and notes on External CIDR

Addresses in the external CIDR are used as proxy destinations when the cluster has no direct route to a remote Pod. Traffic sent to an external-CIDR address is recognized by the network fabric and routed across the inter-cluster tunnels to the provider cluster that actually hosts the Pod.

Here is an example of the content of an IP CR (the command was launched on the consumer cluster):

```
    $ kubectl get ips -n liqo-demo
NAME                LOCAL IP      REMAPPED IP   REMAPPED IP CIDR
nginx-demo-1        10.60.0.173   10.70.0.6     10.70.0.0/16
nginx-demo-2        10.60.0.66    10.70.0.7     10.70.0.0/16
nginx-demo-3        10.63.0.86    10.70.0.5     10.70.0.0/16
nginx-demo-4        10.63.0.103   10.70.0.8     10.70.0.0/16
```

As we can see by the different Pod CIDRs of the local IPs (10.60.0.0/16 and 10.63.0.0/16), the Pods named `nginx-demo-1` and `nginx-demo-2` are offloaded to a cluster, while `nginx-demo-3` and `nginx-demo-4` are offloaded to another cluster.

Also, note that the host part of the forged External IPs (marked with Remapped IP in the example) differs from the original Pod IPs (i.e. x.x.0.170 vs x.x.0.6). This is a big difference from the remapping example (Figure 3.3), where the host part is preserved.

In fact, this kind of remapping is more similar to a NAT translation rather than a remapping.

Chapter 4

State of the Art: `foreign_cluster_connector`

This chapter briefly presents the `foreign_cluster_connector` [22] contribution from a previous thesis project, which served as the foundation for this work.

It consists of a Kubernetes controller running in a consumer cluster, which is capable of creating a direct connection between provider clusters without directly accessing them. After doing this, it creates a CR that stores the details of the newly created connection.

4.1 CR & Controller

The main purpose of `foreign_cluster_connector`[1] is to simplify the deployment and management of the connections among remote clusters in Liqo.

The idea follows the typical Kubernetes declarative approach: the user writes a manifest containing the data needed for a peering or a network connection, the controller intercepts it, and forwards the connection request to the involved clusters. The big improvement is that the manifest is applied directly in the consumer cluster, there is no need to switch contexts. Once received, the connection is created with the provided parameters using `liqoctl`.

Whenever a connection is created this way, the controller creates **in the local cluster** a resource of kind `ForeignClusterConnection`, which stores the status of the connection, the parameters used for the peering and the Pod CIDRs (both the local and the remapped) used by the clusters involved in the peering.

[1] `foreign_cluster_connector` is the name of the implementation, whereas `ForeignClusterConnection` is the name of the CR that it is defined.

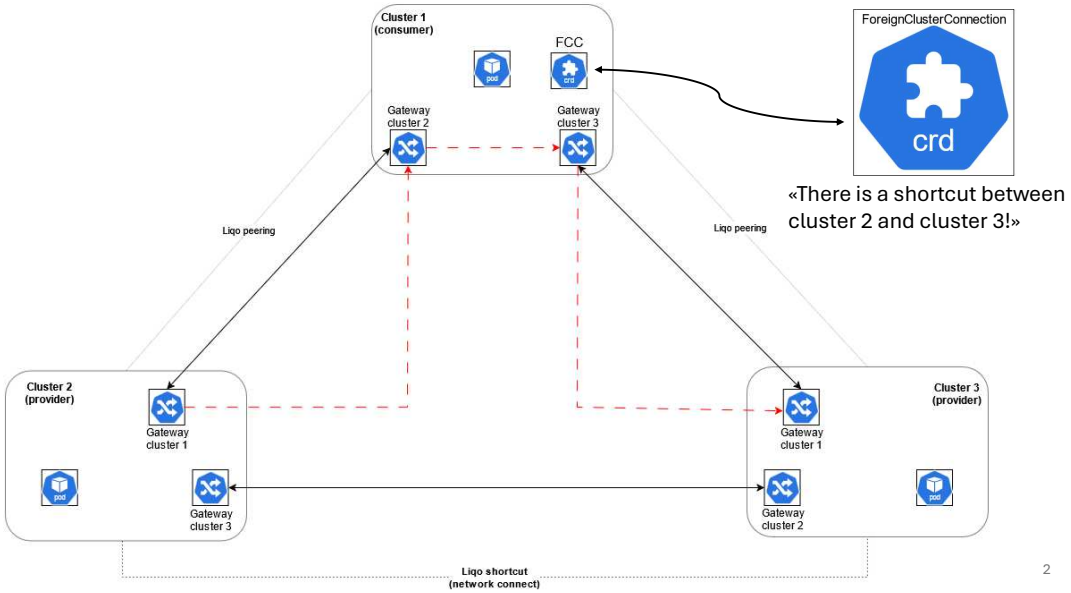

Figure 4.1: An application of the ForeignClusterConnection solution. Through the CLI or applying a manifest in `cluster 1`, the controller starts and creates the connection between `cluster 2` and `cluster 3`. When it is established, the controller updates the CR reporting its status.

This is important because the Pod CIDRs negotiated during the connection are not known by clusters not involved in the peering, so the central cluster gets to know data that it would not have otherwise (unless forging a specific request to the API servers of the remote clusters). This information will be useful in the next chapter, where a more precise explanation of the optimization will be provided.

Below is the content of a `ForeignClusterConnection` CR: in the `spec` field are the parameters used for the connection, while in the `status` field is the current status of the connection, together with the negotiated Pod CIDR of both cluster, under the name of "remapped PodCIDR".

```
Name: cluster2-cluster3
Namespace: default
API Version: networking.liqo.io/v1beta1
Kind: ForeignClusterConnection
Spec:
  Foreign Cluster A:  cluster2
  Foreign Cluster B:  cluster3
  Networking:
    Client Gateway Type:       networking.liqo.io/v1beta1/
   wggatewayclienttemplates
    Client Template Name:      wireguard-client
    Client Template Namespace: liqo
    Mtu:                       1450
    Server Gateway Type:       networking.liqo.io/v1beta1/
   wggatewayservertemplates
    Server Service Port:       51840
    Server Service Type:       NodePort
    Server Template Name:      wireguard-server
    Server Template Namespace: liqo
    Timeout Seconds:           120
    Wait:                      true
Status:
  Foreign Cluster A Networking:
    Pod CIDR:            10.63.0.0/16
    Remapped Pod CIDR:   10.61.0.0/16
  Foreign Cluster B Networking:
    Pod CIDR:            10.60.0.0/16
    Remapped Pod CIDR:   10.61.0.0/16
  Is Connected:          true
  Last Updated:          2025-06-19T23:16:18Z
  Phase:                 Connected
```

Listing 4.1: Example of a ForeignClusterConnector CR.

For better user experience, the solution comes with a CLI to manage directly the connections from the terminal.

4.2 Reflection and Inter-Cluster Communication

The features implemented by this work are used only for the remote connection management, in fact, the behavior of Liqo is not altered and the traffic between Pods in provider clusters is still flowing through the central cluster (as shown in Figure 4.1).

More in depth, here is a description of the actual state of the system, regarding the peer-to-peer communication among provider clusters.

4.2.1 Peer-to-Peer Communication: an Example

The topology is the following (shown in Figure 4.1):

- There are three clusters: `C1`, `C2` and `C3`. C1 acts as a consumer, while `C2` and `C3` are provider clusters peered with `C1`.
- Between the two providers, a network connection (often called "shortcut") is established.
- A Pod is running on each cluster, and they are exposed with a **Service of type ClusterIP**.

  Note that Services are reflected by Liqo, so every cluster has their own version of the Service, the only difference among those, are their endpoints, which are subject to remapping (described in Section 3.6.1).

Now, consider this scenario: **the Pod in `C2` wants to communicate with the Pod in `C3`**, either using a Service or directly using the target Pod IP.

The address of the Pod in `C3` is not known to `C2`, because it is running remotely and providers do not have full knowledge of the topology, as they are only used to provide resources.

This address is known only by `C1`, which has a full view of the topology, and it is responsible of advertising it by reflecting its EndpointSlices.

4.2.2 Reflection of EndpointSlices

The described procedure is fundamental because the endpoints served by Services are fetched from the EndpointSlices, and most importantly, **the CIDRs of these endpoints are used by Liqo to route the traffic correctly**.

This is what happens during the reflection of the EndpointSlice from `C1` to `C2` (the same happens for `C3`):

1. The trigger for this procedure is the exposure of the Pod deployment with a Service in `C1`. Since it involves Pods offloaded to remote clusters (`C2` and `C3`), the local Endpointslice containing those endpoints must be reflected to the remote clusters, so that the Service is functional for all the Pods.
2. Then `C1` forges a ShadowEndpointSlice with the endpoints **as seen by the central cluster** of the Pods involved in the Service, without the endpoints of the Pod running in `C2` (its address is known because it is running locally).
3. In case of endpoints belonging to a different cluster than the one receiving the ShadowEndpointSlice, **an address belonging to `C1` external CIDR is provided**.

This is done because `C1` detects that the endpoint is neither local nor on that cluster, so it must handle the routing itself leveraging the external CIDR addresses. Note that this happens even if the `ForeignClusterConnection` resource is present.

4. `C2` receives the ShadowEndpointSlice and remaps the endpoints according to their remapping rules (Section 3.6.1) represented by the Configuration CRs (Section 3.7.3) it owns.

5. Done that, an EndpointSlice is allocated in `C2`, and the Service is functional.

6. Now, when the Pod in `C2` tries to reach the Pod in `C3`, it uses the address of `C1` external CIDR (remapped), and the traffic flows through `C1`, which routes it to `C3`.

Figure 4.2 is a visual representation of the described procedure.

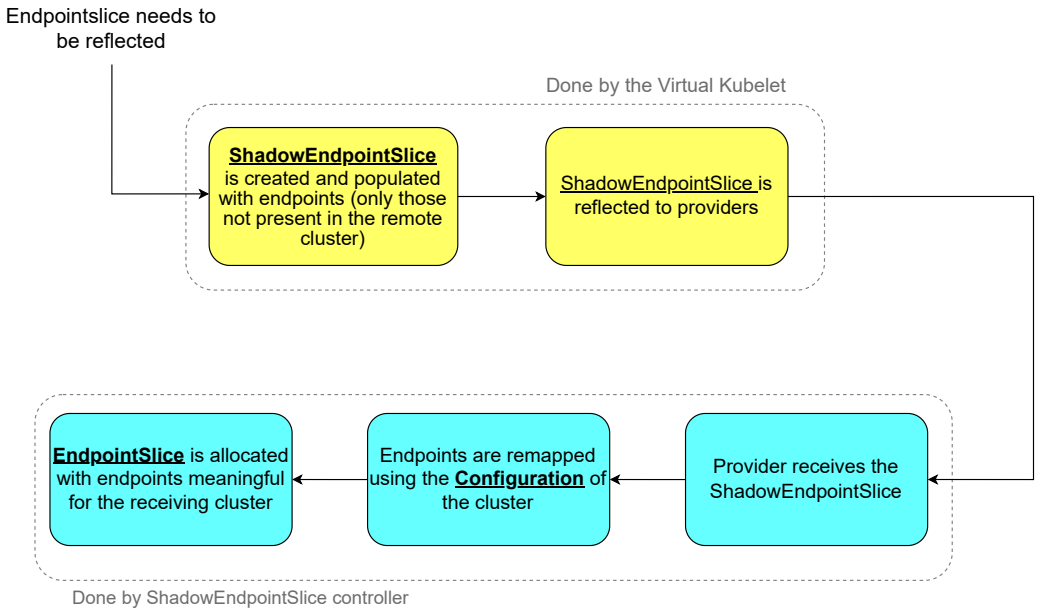

Figure 4.2: Reflection of EndpointSlices from the consumer to a provider. The yellow rectangles represent actions performed by the consumer, while blue ones represent actions performed by the provider.

Chapter 5

The "Automatic" Implementation

This work implements two alternative solutions to the same problem (1.1). This chapter presents the first implementation, an "automatic approach" built on top of the technology described in the previous chapter.

Figure 5.1 is a recap of the most basic topology which benefits this solution: one consumer peered with two providers which have a direct connection. In this case addresses are also specified so the reader can better follow the explanations. Lastly, CIDRs are not shown explicitly in Figure 5.1; Liqo's default network prefix length is /16, so the CIDR for each address can be inferred from the first two octets.

Note that the addresses shown are already remapped: close to the gateways of each cluster are represented the actual addresses used to reach remote pods, located where the arrow is directed. For example, the pod on `C3` has local address **10.200.0.244** but is reached from `C1` using the address **10.63.0.244**. Section 3.6.1 explains how remapping works in Liqo.

5.1 Overview – "Anticipated" Remapping

As presented in Section 4.2.2, the reflection of EndpointSlices is the mechanism that needs to be improved in order to enable direct connections between provider clusters.[1]

In broad terms, the idea behind this implementation is that the consumer cluster

[1]It is the main focus because the **routing rules that allow the direct communication between providers are already present in both the network fabrics involved**. It can be tested by manually pinging the addresses remapped on the correct CIDR.

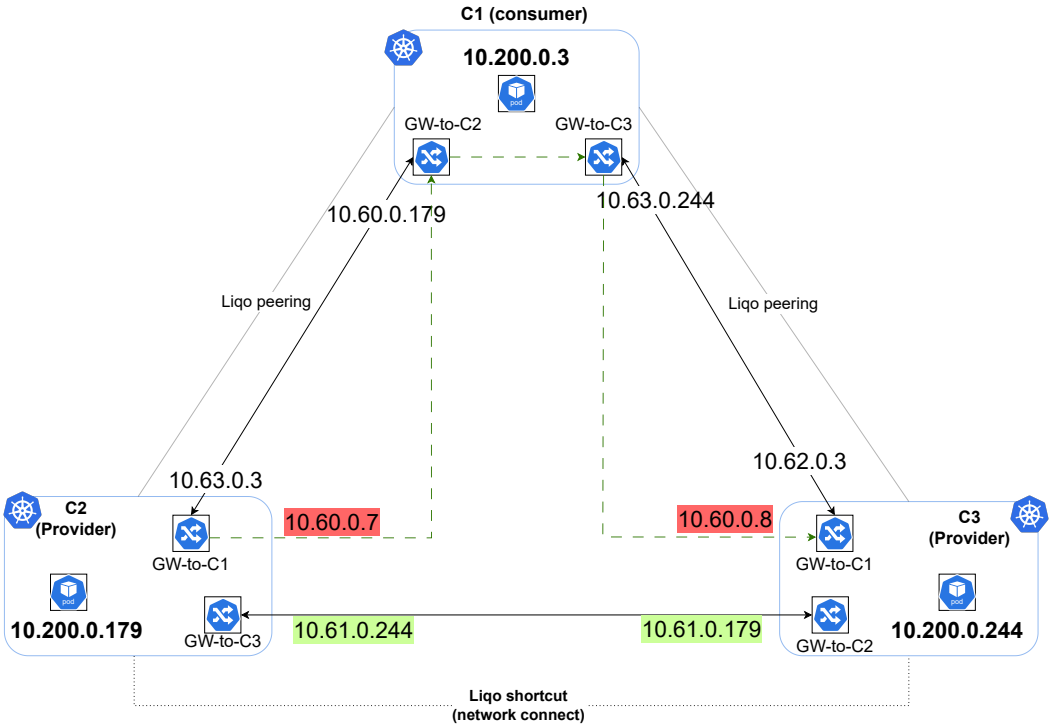

Figure 5.1: Sample topology with addresses.
In red are shown the addresses used by the providers to reach each other through the consumer, in green the addresses used to reach the pods directly.

detects which are the endpoints that can benefit from a direct connection, then "anticipates the remapping" before sending the ShadowEndpointSlice and flags them so that when they are received by the provider, it can understand that the regular remapping must not be applied for them.

The anticipated remapping consists in performing the remapping for the provider cluster **directly in the consumer cluster** using a CIDR meaningful for the receiving cluster: in this case the CIDR is the one used for the the direct connection by the provider (in the example of Figure 5.1 is 10.61.0.0/16 on both ends).

This way, the provider cluster receives the "direct" endpoints already remapped and ready to be used.

5.1.1 Role of the `ForeignClusterConnection` CR

The importance of the `ForeignClusterConnection` CR, is that it makes available to the consumer cluster two CIDRs: one is used by the consumer cluster to reach

the provider on the other end of the connection (`Pod CIDR` in listing 5.1) and the other one is the CIDR used by the provider to reach the other provider (`Remapped Pod CIDR`).

The listing below shows details about the content of the `ForeignCluster Connection` CR, allocated to represent the direct connection between `C1` and `C2`.

```
1  Kind: ForeignClusterConnection
2  Spec:
3    Foreign Cluster A:  cluster2
4    Foreign Cluster B:  cluster3
5  Status:
6    Foreign Cluster A Networking:
7      Pod CIDR:            10.63.0.0/16
8      Remapped Pod CIDR:   10.61.0.0/16
9    Foreign Cluster B Networking:
10     Pod CIDR:            10.60.0.0/16
11     Remapped Pod CIDR:   10.61.0.0/16
12   Is Connected:          true
13   Last Updated:          2025-06-19T23:16:18Z
14   Phase:                 Connected
```

Listing 5.1: Focus of the Pod CIDRs in a `ForeignClusterConnection` CR.

So, referring to Figure 5.1 and Listing 5.1, we can see that:

- cluster A is `C2` and B is `C3`,
- `Status.A.PodCIDR` represents **how `C1` reaches `C3`**,
- `Status.A.RemappedPodCIDR` represents **how `C2` reaches `C3`**.

5.1.2 Detailed Workflow

More in depth, let's analyze what happens when an EndpointSlice is reflected to a provider cluster, in order to understand how the idea is implemented. The steps are the following, the reflection is done by the virtual kubelet running towards a provider cluster; the starting point is when an EndpointSlice needs to be reflected to a provider cluster, at the ShadowEndpointSlice creation:

1. The virtual kubelet checks if there is a `ForeignClusterConnection` resource that involves the target cluster, if so, it extracts the CIDRs stored there.
2. The first is the one corresponding to **the other end of the connection, from the consumer point of view**. This is used to detect if an endpoint can benefit from the direct connection.

3. In case an endpoint meets this condition, the other extracted CIDR is used: it is the CIDR that the remote cluster uses to communicate with the other provider cluster.

4. The anticipated remapping is done using the CIDR and the real part, already known by the consumer.

5. If a remapping is done this way, that address is added to an array and included in the label of the ShadowEndpointSlice

6. All the endpoints not meeting the condition in point 1 are remapped regularly.

7. The provider clusters receives the ShadowEndpointSlice and skips the remapping for all the addresses contained in the label (which are already remapped)

This solution does not require user intervention to make traffic flow through the direct connection: it is for this reason that this implementation is called "automatic".

5.1.3 Example of a ShadowEndpointSlice

The listing below is the content of an ShadowEndpointSlice **reflected to `C2`** in the scenario of Figure 5.1.

Some minor fields have been omitted for the sake of clarity and the name of the pods have been changed to better describe the scenario.

```
Name:          nginx-1
Namespace:     liqo-demo
Labels:        app=nginx
               endpointslice.kubernetes.io/managed-by=endpointslice
   .reflection.liqo.io
               kubernetes.io/service-name=nginx-direct
               offloading.liqo.io/destination=cluster2
               offloading.liqo.io/origin=cluster1
               shortcut-addresses=10.61.0.244
API Version:   offloading.liqo.io/v1beta1
Kind:          ShadowEndpointSlice
Spec:
  Template:
    Address Type:  IPv4
    Endpoints:
      - Addresses: 10.61.0.244
        Conditions:
          Ready: true
        Node Name: cluster1
        Target Ref:
          Kind: RemotePod
```

```
21            Name: nginx-on-cluster-3
22            Namespace: liqo-demo
23        - Addresses: 10.200.0.3
24          Conditions:
25            Ready: true
26          Node Name: cluster1
27          Target Ref:
28            Kind: RemotePod
29            Name: nginx-on-cluster-1
30            Namespace: liqo-demo
```

Listing 5.2: Example of the ShadowEndpointSlice reflected to `C2`. The scenario is the same shown in Figure 5.1.

A ShadowEndpointSlice sent before this implementation would have contained the following endpoints:

- **10.200.0.3**, which is the local address of the pod on `C1`. At the EndpointSlice creation, it will be remapped to the address 10.63.0.3.
- **10.70.0.7**, which is the address of the pod on `C3`, remapped on the external CIDR of `C1`. At the EndpointSlice creation, it will be remapped to 10.60.0.7.

The listing below shows the resulting EndpointSlice in `C2` after the remapping is applied. The local pod address (10.200.0.179) is not present because it is stored in a separate EndpointSlice which is created and managed by the native Kubernetes; here are only the remote endpoints, managed by Liqo.

```
1  Name:          nginx-eps-1
2  Namespace:     liqo-demo
3  Labels:        app=nginx
4                 endpointslice.kubernetes.io/managed-by=endpointslice
       .reflection.liqo.io
5                 kubernetes.io/service-name=nginx-direct
6                 liqo.io/managed=true
7                 liqo.io/managed-by=shadowendpointslice
8                 offloading.liqo.io/destination=cluster2
9                 offloading.liqo.io/origin=cluster1
10 AddressType:   IPv4
11 Endpoints:
12   - Addresses:  10.61.0.244
13     Conditions:
14       Ready:    true
15     TargetRef:  RemotePod/nginx-on-cluster-3
16     NodeName:   cluster1
17   - Addresses:  10.63.0.3
18     Conditions:
19       Ready:    true
20     TargetRef:  RemotePod/nginx-on-cluster-1
```

```
21      NodeName:    cluster1
```

Listing 5.3: The EndpointSlice in `C2` after remapping the ShadowEndpointSlice in Listing 5.2.

Chapter 6

The "Semi-Automatic" Implementation

This chapter presents the final implementation, which is the one whose design was approved by the Liqo maintainers and whose code is currently in review [23], waiting to be merged and released in a future Liqo version.

Again, to make things clearer, the same topology used in the previous chapter is considered (Figure 5.1). The main difference is that this implementation is based on the standard Liqo (version 1.0.1) without the `ForeignClusterConnection` prototype. No other CR is used, only the ones already defined by Liqo.

6.1 Why Another Implementation

When the automatic implementation was presented to the Liqo community, some issues were raised: it was not flexible (users may want their traffic to be routed through the central cluster for many reasons) and does not scale by design.

In fact, the number of `ForeignClusterConnection`s in the consumer cluster grows proportionally with that of the connections between **pairs of providers**[1], leading to a situation where it becomes so big that it negatively impacts the performance of the system when reflecting EndpointSlices. The described scenario is not uncommon, as a central cluster with many edge providers is a typical use case for Liqo.

This led to the development of another solution: a more flexible one, which only relies on the native Liqo CRs.

[1] In a topology with n providers, the maximum number of `ForeignClusterConnection`s is $n(n-1)/2$, thus growing with the square of the number of providers.

6.2 Overview

The core idea revolves around one key difference with respect to the previously discussed implementation: in the automatic version, the remapping was anticipated by the consumer and sent together with the endpoint to replace; this time, **the remapping is performed by the provider**, which receives some additional data from the consumer in order to be able to perform the remapping to the correct address. The remapping is hence considered "forced", because the provider *must* remap an address which would not normally undergo remapping.

Moreover, to increase flexibility, a per-Service approach has been chosen: the traffic which should leverage the direct connection is decided by the user applying a specific label on the Service, which is why this implementation is said to be "semi-automatic".

This way Liqo's default behavior remains unchanged, whereas the choice of using direct connections is left to the user, and the scalability issue is resolved as no `ForeignClusterConnection` CRs are needed.

To make this possible, without the help of the `ForeignClusterConnection` CR, the consumer cluster needs the data which was previously stored there.

6.2.1 Data Requirements

More in detail, referring to Figure 6.1, the data needed to perform this new remapping is:

1. The clusterID of the provider on the other end of the connection,
2. The CIDR negotiated by the provider for the direct connection,
3. The host part of the pod's IP address.

This data is almost the same that was used in the previous implementation, except for point 1, which is now needed by the recipient provider to identify which is the direct connection use.

The reason why the anticipated remapping (that is, remapping directly in the consumer) is not a possible solution without the `ForeignClusterConnection` CR, is because the 3 values above are stored in different places: while the clusterID and the host part of the endpoint can be easily obtained by the consumer cluster through its informers[2], the problem is about the CIDR.

[2]A tool offered by native Kubernetes, it acts like a cache for resources frequently accessed, so not to wait for responses from the API server.

This information can be retrieved only making a request to the API server of the provider cluster, which is a very slow operation, and should be avoided as much as possible[3].

The request to the provider's API server is therefore inevitable — the same lookup is required by the standard remapping —, but its overhead is mitigated by a local cache, which reduces latency for repeated accesses. More importantly, performing the lookup on the provider is conceptually correct: the CIDR and configuration data are authoritative at the provider, so delegating the remapping to the provider ensures the operation uses locally consistent information and avoids exchanging useless data between clusters.

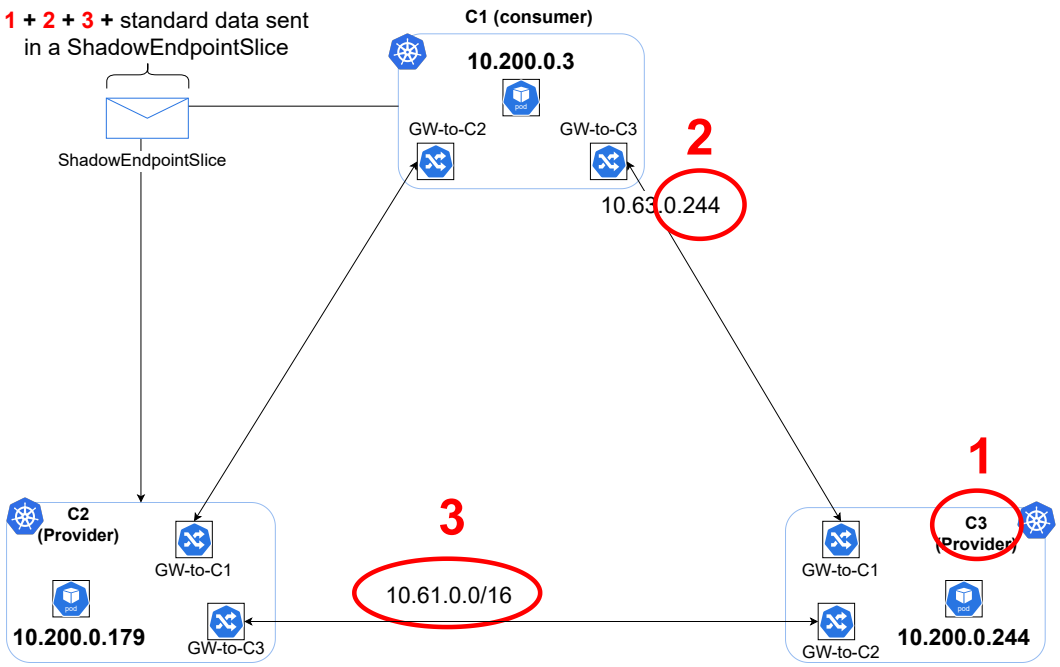

Figure 6.1: Schema of the data **required by `C2`** to perform the forced remapping: 1 and 2 are available in the consumer cluster and sent in the ShadowEndpointSlice, 3 is known by the provider.

[3]The "automatic" implementation makes this request and stores the CIDRs in the CR, so it's only done once, after the connection has been established. In this case the request should be performed everytime an EndpointSlice is reflected.

6.2.2 Detailed workflow

This section describes what happens when an EndpointSlice is reflected to a provider cluster, in order to understand how the idea is implemented. The steps are the following:

1. The starting point is the creation of a ShadowEndpointSlice by the virtual kubelet running towards a provider cluster,
2. The virtual kubelet checks if the Service corresponding to the EndpointSlice has the label which enables the direct connection feature, if so it proceeds with the following steps, otherwise it reflects the EndpointSlice normally,
3. For each endpoint in the EndpointSlice:
   (a) The virtual kubelet checks if the endpoint can benefit from a direct connection, that is, if the pod is running in a provider cluster and is not the one receiving the EndpointSlice,
   (b) In case the endpoint meets this condition, the clusterID of the provider where the pod is running is extracted,
   (c) It retrieves **the local IP** of the pod, that is the address as seen by the consumer cluster. It will be later used to extract the host part,
   (d) It retrieves also **the remapped IP** of the pod, which is the address sent normally in the ShadowEndpointSlice, it will be used to understand which is the address to replace,
   (e) For each clusterID, a pair (local IP, remapped IP) is stored in a data structure
4. After all the endpoints have been processed, the above data structure is encoded to JSON and added to an Annotation[4] of the ShadowEndpointSlice,
5. The ShadowEndpointSlice creation proceeds as usual, and is finally sent to the provider cluster.
6. The provider cluster receives the ShadowEndpointSlice and checks if it has the Annotation containing the data for the forced remapping,
7. In case it is present, data is decoded and stored in a data structure,
8. At this point every received endpoint is processed:

[4]The maximum size of Kubernetes annotations is 256kB, in case the encoded data exceeds this limit, no direct connection data is attached to the ShadowEndpointSlice.

9. If it is not found in the data structure it is remapped normally,

10. Otherwise, the other two values are extracted (clusterID and local IP),

11. The ClusterID is used to retrieve the `Configuration` CR, which contains the CIDR used by the provider to reache the cluster identified with that ID,

12. Using this CIDR and the host part extracted from the local IP, the remapping is performed,

13. The endpoint is updated with the new remapped IP.

Figure 6.2 summarizes the additional data sent in the ShadowEndpointSlice to enable the forced remapping.

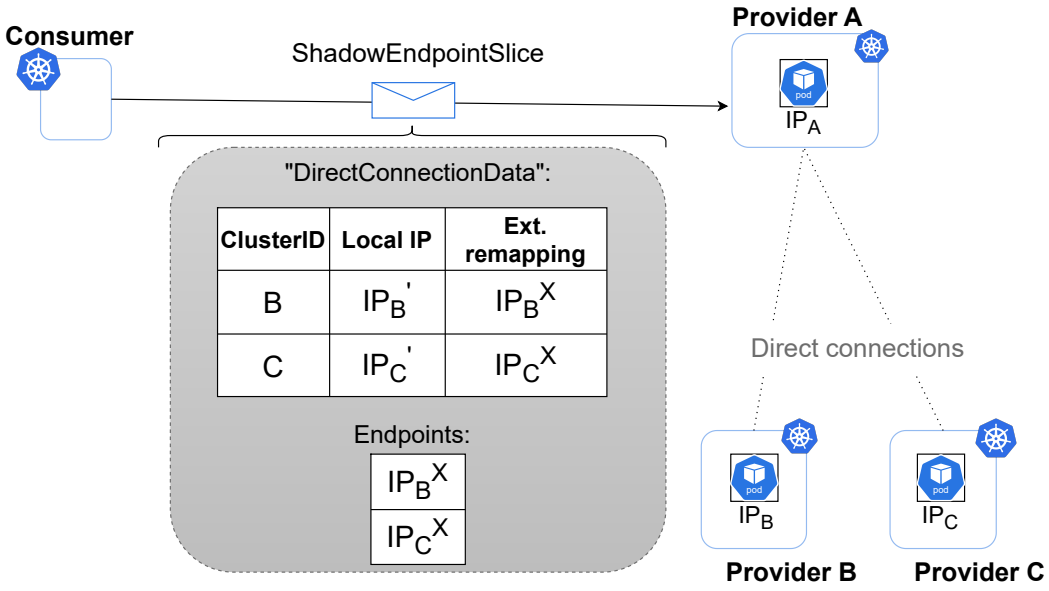

Figure 6.2: Schema of the data sent in the ShadowEndpointSlice. **DirectConnectionData** is the additional payload sent in the Annotations and used for forced remapping. Superscript X denotes IPs remapped into the consumer's External CIDR (these are the addresses normally carried in the ShadowEndpointSlice); superscript prime (') denotes the local IPs as observed by the consumer. Subscripts indicate the clusterID of the pod's home cluster.

6.3 Code Snippets

Some code snippets are presented to show how the implementation is done in practice. Note that some parts (error handling for instance) are avoided for space reasons. The entire code, including future reviews, is available in the author's Pull Request to the main repository [23].

6.3.1 Forging the ShadowEndpointSlice with the Additional Data

Listing 6.1 shows the part of code during the forging process, in charge of retrieving the data needed for the forced remapping. The `Handle()` function is the one called when an EndpointSlice needs to be reflected to a provider cluster. `local` is the EndpointSlice to reflect, while `remote` is the corresponding ShadowEndpointSlice in the provider cluster. Note that this code is executed both in case the ShadowEndpointSlice is created for the first time and in case it is updated. The logic is the same in both cases.

Several small utility functions were added to centralize common operations (label checks, resource field extraction and annotation handling), reducing duplication and clarifying the main flow.

All the Get operations (6.1, lines 13 and 32, aimed at retrieving the Node and the IP resources) in this code snippet are not requested directly to the API server but instead on the local cache of the informers, which is much faster.

```
1  func (ner *NamespacedEndpointSliceReflector) Handle(ctx context.
       Context, name string) error {
2
3  // ...existing code...
4
5  var marshaledData []byte
6    if ner.ShouldProvideDirectConnectionData(local) {
7      var remoteConnectionsData directconnectioninfo.InfoList
8
9      for _, endpoint := range local.Endpoints {
10       if endpoint.NodeName == nil {
11         continue
12       }
13       node, err := ner.localNodeClient.Get(*endpoint.NodeName)
14
15       if !directconnectioninfo.ShouldIncludeDataFromNode(node,
       string(forge.RemoteCluster)) {
16         continue
17       }
18
```

```
19        clusterID, err := getters.RetrieveRemoteClusterIDFromNode(
      node)
20        objectName := endpoint.TargetRef.Name
21        ipsObj, err := ner.localIPs.Get(objectName)
22        localIPs, err := ipamutils.GetLocalIPFromObject(ipsObj)
23        remappedIPs, err := ipamutils.GetRemappedIPFromObject(ipsObj
      )
24        remoteConnectionsData.Add(clusterID, []string{localIPs}, []
      string{remappedIPs})
25      }
26      if len(remoteConnectionsData.Items) == 0 {
27        /*no direct connection data*/
28      } else {
29        marshaledData, err = remoteConnectionsData.ToJSON()
30        if len(marshaledData)+totalAnnotationsSize(local) >=
      maxAnnotationSize {
31          // Max size of data in annotations is 256KB
32          marshaledData = nil
33        } else {
34          /*success*/
35        }
36      }
37    }
38  // ...further code...
```

Listing 6.1: Code snippet of the ShadowEndpointSlice forging process.

6.3.2 ShadowEndpointSlice Controller – Remapping Process

Listing 6.2 below shows the part of code executed by **in the provider cluster**, when a ShadowEndpointSlice is received. The `Reconcile()` function is the one called when a ShadowEndpointSlice is created or updated in the provider cluster.

Its logic is very simple and straightforward, as it just checks if the Annotation containing the data about direct connections is present, and if so it decodes it and remaps the endpoints with the function `MapEndpointsWithConfiguration()`.

```
1  // Reconcile ShadowEndpointSlices objects.
2  func (r *Reconciler) Reconcile(ctx context.Context, req ctrl.
       Request) (ctrl.Result, error) {
3    // ...existing code...
4
5    // Check if direct connections data is provided
6    var remoteConnectionsData directconnectioninfo.InfoList
7    if val, ok := shadowEps.Annotations[
       directConnectionAnnotationLabel]; ok {
8      err := remoteConnectionsData.FromJSON([]byte(val))
```

```
9
10      if err != nil {/*error logging*/}
11      // JSON is not propagated to the EndpointSlice
12      delete(shadowEps.Annotations, directConnectionAnnotationLabel)
13    }
14
15    remappedEndpoints := shadowEps.Spec.Template.Endpoints
16    if foreigncluster.IsNetworkingModuleEnabled(fc) {
17      // remap the endpoints if the network configuration of the
       remote cluster overlaps with the local one
18      if err := MapEndpointsWithConfiguration(ctx, r.Client,
       clusterID, remappedEndpoints, remoteConnectionsData); err !=
       nil {
19        /*error logging*/
20      }
21    }
22
23    // Forge the endpointslice given the shadowendpointslice
24    newEps := discoveryv1.EndpointSlice{
25      ObjectMeta: metav1.ObjectMeta{
26        Name:      shadowEps.Name,
27        Namespace: shadowEps.Namespace,
28        Labels: labels.Merge(shadowEps.Labels, labels.Set{
29          consts.ManagedByLabelKey: consts.
       ManagedByShadowEndpointSliceValue}),
30        Annotations: shadowEps.Annotations,
31      },
32      AddressType: shadowEps.Spec.Template.AddressType,
33      Endpoints:   remappedEndpoints,
34      Ports:       shadowEps.Spec.Template.Ports,
35    }
36  // ... further code ...
```

Listing 6.2: Code snippet of the ShadowEndpointSlice remapping process.

6.3.3 The Forced Remapping

Listing 6.3 and Listing 6.4 show the functions in charge of performing the forced remapping, using the data sent in the Annotation of the ShadowEndpointSlice.

`MapEndpointsWithConfiguration()` iterates over all the endpoints in the ShadowEndpointSlice, and for each of them it checks if it is present in the data structure containing the data for the forced remapping. If so, it extracts the clusterID and the local IP, retrieves the CIDR from the `Configuration` CR and performs the remapping using the utility function `ForceMapAddressWithConfiguration()`.

This code is better described in steps 8 to 13 of the workflow presented above.

```
1  func MapEndpointsWithConfiguration(ctx context.Context, cl client.
       Client,
2    clusterID liqov1beta1.ClusterID, endpoints []discoveryv1.
       Endpoint,
3    list directconnectioninfo.InfoList,
4  ) error {
5    for i := range endpoints {
6      for j := range endpoints[i].Addresses {
7        addr := endpoints[i].Addresses[j]
8
9        addrHasBeenRemapped := false
10
11       // Check if mapping should be forced
12       if len(list.Items) != 0 {
13         clusterID, ip, addressFound := list.GetConnectionDataByIP(
       addr)
14
15           if addressFound {
16             rAddr, err := ipamips.ForceMapAddressWithConfiguration(
       ctx, cl, liqov1beta1.ClusterID(clusterID), ip)
17
18             if err == nil {
19               endpoints[i].Addresses[j] = rAddr.String()
20               addrHasBeenRemapped = true
21             } else {/*error logging*/}
22           }
23           if addrHasBeenRemapped {
24             break
25           }
26       }
27
28       // Regular mapping is performed
29       if !addrHasBeenRemapped {
30         rAddr, err := ipamips.MapAddress(ctx, cl, clusterID, addr)
31         if err != nil {
32           return err
33         }
34         endpoints[i].Addresses[j] = rAddr
35       }
36     }
37   }
38
39   return nil
40 }
```

Listing 6.3: Code snippet of the mapping using MapEndpointsWithConfiguration.

`ForceMapAddressWithConfiguration()` is a utility function which performs the actual remapping, given the CIDR and the local IP. It extracts the host part

of the local IP and combines it with the CIDR to create the new remapped IP.

This function makes use of a function called `GetConfigurationByClusterID()`, which retrieves the `Configuration` CR corresponding to the clusterID passed as argument.

This is the only case in which a request to the API server is made without using the local informers cache but this would happen anyway, since also the standard remapping needs to retrieve a `Configuration` CR. This problem is easily solved by the standard Liqo caching mechanism: it stores the remappings in a local cache, including the ones made with the forced approach.

```
1  func ForceMapAddressWithConfiguration(ctx context.Context, cl
       client.Client,
2    clusterID liqov1beta1.ClusterID, address string) (net.IP, error)
       {
3    // This address is used only to get its host part!
4    addr := net.ParseIP(address)
5
6    cfg, err := getters.GetConfigurationByClusterID(ctx, cl,
       clusterID, corev1.NamespaceAll)
7    if err != nil {
8      return addr, err
9    }
10
11   podCidr := cidrutils.GetPrimary(cfg.Status.Remote.CIDR.Pod).
       String()
12   _, podnet, err := net.ParseCIDR(podCidr)
13   if err != nil {
14     return addr, err
15   }
16
17   return RemapMask(addr, *podnet), nil
18 }
```

Listing 6.4: Code snippet of the forced remapping function.

Chapter 7

Conclusions and Future Work

This thesis presents two approaches to enable direct communication between provider clusters in Liqo. Both solutions address the problem by modifying how EndpointSlices are reflected to provider clusters, with slight differences discussed in Section 5 and Section 6. The first, fully automatic approach requires the `ForeignClusterConnection` CR, which has not yet been accepted by the Liqo team. The second, semi-automatic approach relies solely on Liqo, offering a more scalable and flexible solution; it only requires annotating a Service, a task that can be easily automated in future implementations.

7.1 Results

The observed results are straightforward: by shortening the path between providers, latency is certainly reduced, the workload on the consumer cluster is decreased, and the overall network efficiency is improved.

Moreover, the standard Liqo behavior is not altered unless specified, so the upgrade to a future Liqo version can be seamless and non-desruptive for all those systems which require the inter-provider traffic to be routed through the consumer.

Only a few quantitative notes are worth mentioning about the semi-automatic implementation:

- The overhead introduced by the additional data sent in the ShadowEndpointSlice is negligible, as it reaches ~130 bytes per endpoint in the worst case[1]. This is a very small amount of data, considering that an EndpointSlice

[1]This case is: clusterID of 63 characters, and addresses that use 3 digits per octet.

can contain up to 100 endpoints, so the overhead per endpoint is very low, making this solution efficient and scalable.

- Some operations are required to perform the remapping: among those, the most computationally expensive are the requests to the API servers. However, these requests are backed either by informers or a cache internal to Liqo so that the overhead is mitigated. Anyway, these are needed also in the standard remapping, so the overhead introduced by this implementation is negligible in this case.
- The runtime overhead is limited to the cases where endpoints must be checked against the consumer-supplied DirectConnectionData. In our experiments no measurable degradation was observed because the data structures remained small. If this becomes a bottleneck, easy mitigations are available (for example compressing the encoded payload, adding an index, or using a hash set). The current design intentionally exposes the reflected data so operators can inspect and trace EndpointSlice creation and updates.

7.2 Future Work

Although the semi-automatic approach was chosen to be merged into the main project, the automatic one can be useful as a proof of concept and a starting point for future developments in the direction of a fully automatic solution.

Otherwise, the semi-automatic implementation can be further improved in the future, for example by:

- Developing a controller that's capable of automatically applying the label to the Services which can benefit from the direct connection, based on some criteria (e.g. the amount of traffic exchanged between two clusters).
- Implementing a mechanism to automatically create direct connections between clusters, based on the network topology and latency measurements.
- Extending the implementation to support more complex network topologies, such as a Service Mesh among different remote clusters.

In conclusion, this thesis represents a step forward in optimizing inter-cluster communications in Liqo, and opens the door to further enhancements in this field.

Bibliography

[1] Accessed: 2025-10-10. 2025. URL: https://www.eng.it/it (cit. on p. 3).

[2] Accessed: 2025-10-10. 2025. URL: https://www.eng.it/it/insights/stories/research-projects/ipcei-cis-avant (cit. on p. 3).

[3] The Kubernetes Authors. *Kubernetes Documentation*. Accessed: 2025-09-04. 2025. URL: https://kubernetes.io/docs/ (cit. on pp. 5, 6).

[4] The Linux Foundation®. Accessed: 2025-10-10. 2025. URL: https://www.cncf.io/ (cit. on p. 6).

[5] Michael Hausenblas and Stefan Schimanski. *Programming Kubernetes: Developing Cloud-Native Applications*. Ed. by Inc. O'Reilly Media. O'Reilly Media, Inc., 2019 (cit. on p. 8).

[6] Accessed: 2025-10-10. 2025. URL: https://kubernetes.io/docs/concepts/architecture/cri/ (cit. on p. 9).

[7] Accessed: 2025-10-10. 2025. URL: https://istio.io/ (cit. on p. 10).

[8] Accessed: 2025-10-10. 2025. URL: https://prometheus.io/ (cit. on p. 10).

[9] Accessed: 2025-10-10. 2025. URL: https://grafana.com/ (cit. on p. 10).

[10] The Liqo Authors. *Liqo Website*. Accessed: 2025-09-12. 2025. URL: https://liqo.io/ (cit. on p. 10).

[11] The Kubernetes Authors. *Kubernetes Networking Model*. Accessed: 2025-09-12. 2025. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/ (cit. on pp. 14, 17).

[12] Accessed: 2025-10-10. 2025. URL: https://github.com/containernetworking/cni (cit. on p. 14).

[13] Accessed: 2025-10-10. 2025. URL: https://docs.tigera.io/calico/latest/about (cit. on p. 14).

[14] Accessed: 2025-10-10. 2025. URL: https://github.com/flannel-io/flannel (cit. on p. 14).

[15] Accessed: 2025-10-10. 2025. URL: https://cilium.io/ (cit. on p. 14).

[16] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. http://www.rfc-editor.org/rfc/rfc7348.txt. RFC Editor, Aug. 2014. URL: http://www.rfc-editor.org/rfc/rfc7348.txt (cit. on p. 15).

[17] Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. *Generic Routing Encapsulation (GRE)*. RFC 2784. http://www.rfc-editor.org/rfc/rfc2784.txt. RFC Editor, Mar. 2000. URL: http://www.rfc-editor.org/rfc/rfc2784.txt (cit. on p. 15).

[18] The Liqo Authors. *Liqo Documentation*. Accessed: 2025-09-12. 2025. URL: https://docs.liqo.io/en/v1.0.1/ (cit. on pp. 18, 20).

[19] Virtual Kubelet Authors. *Virtual Kubelet Documentation*. Accessed: 2025-09-16. 2025. URL: https://virtual-kubelet.io (cit. on p. 18).

[20] Accessed: 2025-10-10. 2025. URL: https://helm.sh/ (cit. on p. 19).

[21] Accessed: 2025-10-10. 2025. URL: https://www.wireguard.com/ (cit. on p. 19).

[22] Santo Calderone. «Optimizing peer-to-peer multi-cluster communications for Liqo-based multi-cloud deployments». MA thesis. Politecnico di Torino, 2025. URL: https://webthesis.biblio.polito.it/36364/ (cit. on p. 27).

[23] Gabriele Santi. 2025. URL: https://github.com/liqotech/liqo/pull/3115 (cit. on pp. 38, 43).