POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico di Torino

Master's Degree Thesis

Optimizing peer-to-peer multi-cluster communications for Liqo-based multi-cloud deployments

Supervisors

Prof. Fulvio RISSO

Dott. Davide MIOLA

Candidate

Santo CALDERONE

July 2025

Summary

The growing complexity of cloud-native infrastructure, particularly in hybrid and multi-cloud contexts, has led to the widespread adoption of Kubernetes as the standard platform for orchestrating containerized workloads. While Kubernetes offers robust mechanisms for managing resources within a single cluster, it lacks native support for seamless multi-cluster deployments. To bridge this gap, the open-source Liqo framework enables dynamic federation of Kubernetes clusters, allowing workload offloading, resource sharing, and secure cross-cluster networking. Despite these capabilities, Liqo's default networking model remains consumer-centric: all traffic between provider clusters must be routed through the consumer, introducing unnecessary latency, limiting effective bandwidth between providers, and creating a potential single point of failure. This architecture limits the scalability and resilience of Liqo-based infrastructures in real-world scenarios. This thesis introduces an architectural enhancement to overcome this limitation. A new Kubernetes Custom Resource Definition (CRD), `ForeignClusterConnection`, is designed and implemented, enabling declarative and automated provisioning of direct tunnels between provider clusters. A dedicated operator orchestrates the lifecycle of these resources, interacting with Liqo's internal APIs and CLI tools to discover cluster metadata, manage network endpoints, and propagate connection status. The implementation, based on the Kubebuilder SDK, follows established Kubernetes operator patterns, ensuring high modularity and seamless integration. Experimental evaluation under simulated WAN conditions demonstrates improvements in latency, throughput, and system resilience when direct inter-provider communication is established. The proposed solution aligns with Kubernetes' declarative model and GitOps principles, paving the way for advanced features such as dynamic endpoint rewriting, policy-driven routing, topology awareness, and monitoring integrations. It enhances the efficiency, robustness, and observability of Liqo-powered multi-cluster deployments, contributing to the evolution of Kubernetes-based federated networking in heterogeneous environments.

Ringraziamenti

Con tutto il cuore ringrazio i miei genitori e mia sorella: so di non essere sempre facile da gestire, eppure mi avete sempre sostenuto, ascoltato e incoraggiato in ogni mia decisione. Questo traguardo è tanto vostro quanto mio.

Un caloroso ringraziamento ai nonni, agli zii, ai cugini e a tutta la famiglia: la vostra vicinanza e i ricordi meravigliosi che abbiamo condiviso hanno illuminato ogni passo di questo percorso, rendendo ogni fatica più leggera e ogni successo più bello.

A Morena: sei arrivata in un momento in cui credevo di bastare a me stesso, e invece mi hai mostrato che con te la vita è migliore, dandomi tutto senza mai farmi pesare o togliermi nulla. Grazie di essere stata e di continuare ad essere sempre al mio fianco.

Un ringraziamento sincero al prof. Risso, a Davide Miola, a Mirco Barone e a tutto il gruppo di data-plane per la vostra competenza, pazienza e disponibilità che mi hanno guidato lungo questo percorso.

Grazie di cuore al gruppo dei "Terroni a Torino": siete stati la mia "casa" lontano da casa, nei momenti di difficoltà e in quelli di festa. Porterò sempre con me le risate, il calore e il vostro affetto; sappiate che io ci sarò sempre per voi.

Ai miei amici di "giù": grazie per aver condiviso con me 27 anni di avventure, per le risate spensierate e gli "insulti" affettuosi che valgono più di mille complimenti. Mi avete permesso di staccare la testa quando ne avevo più bisogno, rendendo ogni momento insieme leggero e memorabile.

Infine, un ringraziamento a chiunque sia entrato o uscito dalla mia vita durante questo percorso: ogni incontro e ogni addio mi hanno aiutato, nel bene e nel male, a crescere e a diventare la persona che sono oggi.

Table of Contents

List of Figures

Acronyms

API
: Application Programming Interface

CLI
: Command Line Interface

CNI
: Container Network Interface

CRD
: Custom Resource Definition

CIDR
: Classless Inter-Domain Routing

IETF
: Internet Engineering Task Force

IPAM
: IP Address Management

RBAC
: Role-Based Access Control

RFC
: Request for Comments

TCP
: Transmission Control Protocol

TLS

Transport Layer Security

TLV

Type–Length–Value

UDP

User Datagram Protocol

VPN

Virtual Private Network

VXLAN

Virtual Extensible LAN

YAML

YAML Ain't Markup Language

Chapter 1

Introduction

The evolution of cloud-native architectures has heightened the demand for infrastructures that are scalable, resilient, and interoperable across heterogeneous cloud and edge environments. Kubernetes has established itself as the *de facto* standard for container orchestration, enabling declarative application deployment and automated lifecycle management. Although Kubernetes provides robust orchestration capabilities within single-cluster environments, extending these capabilities to multi-cluster scenarios introduces several challenges, such as cross-cluster networking, workload portability, and resource federation.

To address these limitations, frameworks such as **Liqo** have emerged, offering dynamic peer-to-peer federation of Kubernetes clusters. Liqo enables transparent workload offloading, automated trust negotiation, and encrypted cross-cluster networking. Despite these features, Liqo-based multi-cluster topologies are limited by a consumer-centric routing model, in which all inter-cluster communications, including those between provider clusters, are relayed through the consumer. This design introduces additional latency, increases egress traffic, and represents a potential single point of failure.

1.1 Goal of the Thesis

The objective of this work is to enhance the network orchestration model provided by Liqo in peer-to-peer multi-cluster topologies. Specifically, a mechanism is proposed to enable direct communication between provider clusters, thereby removing the inefficiencies caused by routing traffic through the consumer cluster.

A new **Custom Resource Definition (CRD)**, named `ForeignCluster Connection`, is introduced, along with a dedicated **Kubernetes Operator** responsible for automating the lifecycle of inter-provider tunnels. Embedding the connection logic within a declarative resource managed by the consumer cluster

ensures consistency with the Liqo architectural model and improves both efficiency and resilience.

1.2 Structure of the Thesis

The remainder of this thesis is organized as follows:

- **Chapter 2 – Background: Kubernetes**
  Provides an overview of the Kubernetes architecture, with particular focus on core resources, scalability mechanisms, and API extensibility through CRDs and the operator pattern.
- **Chapter 3 – Background: Liqo**
  Describes the architecture and key components of Liqo, including peering mechanisms, offloading logic, cross-cluster networking, and the custom resources relevant to multi-cluster orchestration.
- **Chapter 4 – Design: foreign_cluster_connector**
  Presents the design of the mechanism enabling direct provider-to-provider communication, including network topology, routing constraints, and the conceptual model of the `ForeignClusterConnection` CRD.
- **Chapter 5 – Implementation of the foreign_cluster_connector Controller**
  Details the implementation of the controller using the Kubebuilder framework, including CRD definition, reconcile logic, RBAC rules, project scaffolding, build and deployment instructions.
- **Chapter 6 – Command-Line Utility for Managing the Controller and Shortcuts**
  Describes the custom command-line utility developed to manage the controller, including its design goals, command structure, shortcut management features, auxiliary logic, and recommended usage patterns.
- **Chapter 7 – Evaluation**
  Presents the experimental setup and analyzes performance under various network conditions, with metrics such as latency, throughput, and tunnel provisioning times.
- **Chapter 8 – Conclusions and Future Work**
  Summarizes the contributions of this work and outlines directions for future research and development in multi-cloud and multi-cluster orchestration.

Chapter 2

Background: Kubernetes

This chapter provides a technical overview of Kubernetes, the leading open-source platform for container orchestration. It presents the system architecture, core abstractions, networking model, extensibility via custom APIs, access control mechanisms, and the developer tooling offered by Kubebuilder. The goal is to establish a foundation for understanding how Kubernetes can be extended to support advanced orchestration patterns such as multi-cluster federation.

2.1 Introduction to Kubernetes

Kubernetes is an open-source platform that automates deployment, scaling, and management of containerized applications, following principles outlined in its official documentation [1]. It adopts a declarative configuration model and offers strong fault tolerance. By abstracting the underlying infrastructure, it enables consistent deployment across different environments and has become the *de facto* standard for container orchestration. Initially developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes provides a unified API for interacting with cluster resources.

Application specifications—including the number of replicas, resource constraints, and network policies—are defined using YAML manifests. The control plane continuously reconciles the actual state of the cluster with the desired state, supporting features such as self-healing, rolling updates, and autoscaling.

Thanks to its modular and extensible architecture, Kubernetes supports a wide range of cloud-native scenarios across heterogeneous environments, both on-premises and in public clouds.

2.2 Kubernetes Architecture

Kubernetes is logically divided into two main planes. The *Control Plane* maintains the global state of the cluster, while the *Data Plane* runs containerized applications on worker nodes.

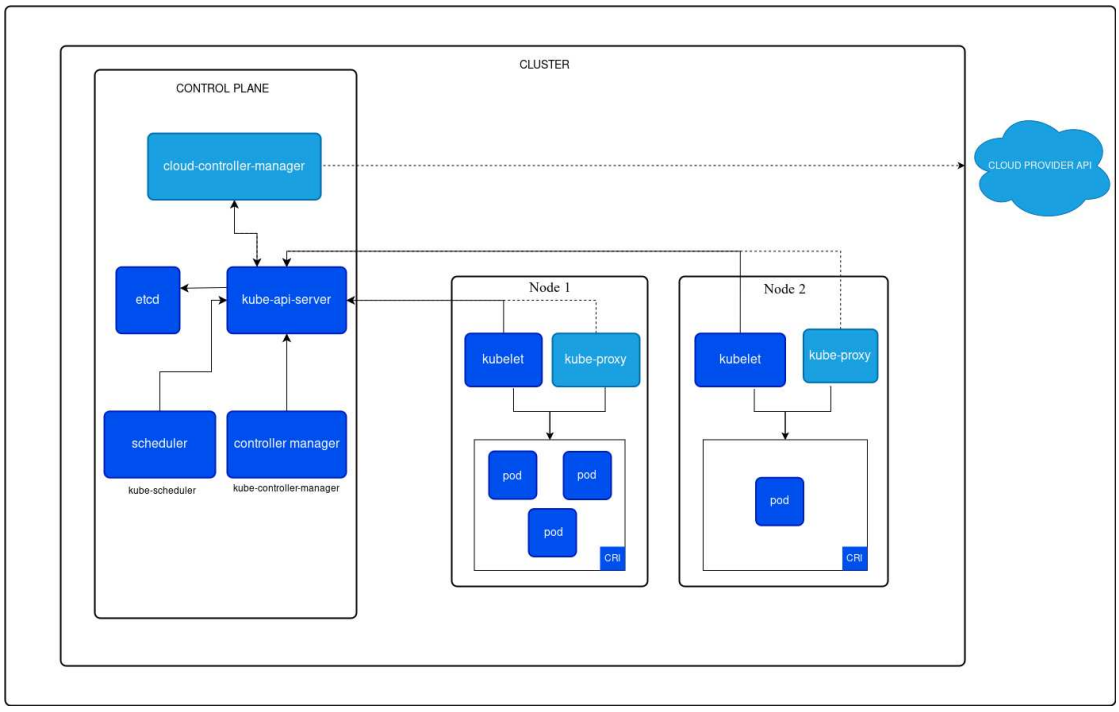

Figure 2.1: High-level architecture of Kubernetes. Source [2]

2.2.1 Control Plane Components

The Control Plane is responsible for maintaining cluster state, processing API requests, and ensuring that the desired configuration is continuously enforced through its core components.

`kube-apiserver`

The API Server exposes a RESTful interface that serves as the central access point to all cluster operations. It validates and processes API requests, performs authentication and authorization, and persists cluster state changes in the data store.

`etcd`

`etcd` is a distributed key–value store used as the single source of truth for the entire cluster state. It stores configuration data, workload definitions, and metadata. The Raft consensus algorithm ensures consistency and availability across all etcd nodes.

`kube-scheduler`

The scheduler is responsible for assigning newly created Pods to suitable nodes. It filters candidate nodes based on resource availability and scheduling constraints, then scores them using customizable policies before binding the Pod to a selected node.

`kube-controller-manager`

This component manages a set of independent controllers that continuously monitor cluster state. Examples include the NodeController (monitors node availability), ReplicaSetController (ensures desired Pod replica count), and EndpointController (maintains service endpoint records).

`cloud-controller-manager`

This controller is specific to cloud environments and decouples cloud-provider logic—such as provisioning volumes or load balancers—from the main Kubernetes control logic. It enables cloud-agnostic operations by abstracting provider-specific APIs.

2.2.2 Data Plane Components (Worker Nodes)

Worker nodes host the components required to run application containers and enforce the desired state defined by the Control Plane. They manage container execution, networking, and report node and Pod status back to the Control Plane.

`kubelet`

`kubelet` is the primary node agent responsible for managing the Pod lifecycle. It ensures that containers are started, monitored, and remain healthy according to the desired Pod specification communicated by the Control Plane. It also enforces resource limits and probes defined in manifests.

kube-proxy

kube-proxy maintains network rules and routes traffic destined for Services across nodes. It leverages iptables, IPVS, or user-space proxying to facilitate Pod-to-Pod and Pod-to-Service communication.

Container Runtime Interface (CRI)

The CRI allows Kubernetes to communicate with container runtimes such as containerd and CRI-O. Through this interface, the kubelet can start, stop, monitor containers, and retrieve logs and metrics.

2.3 Core Kubernetes Resources

Kubernetes defines several fundamental abstractions for managing workloads and resources in a cluster.

2.3.1 Pods

A Pod is the smallest deployable object in Kubernetes. It encapsulates one or more containers that share the same network namespace and volumes. Each Pod receives a unique IP address, enabling intra-Pod communication via localhost. Pods are designed to be ephemeral and disposable; if a Pod fails or is deleted, it is not guaranteed to return in the same state. For this reason, Pods are typically managed indirectly by higher-level resources such as ReplicaSets and Deployments, which ensure availability and desired state.

2.3.2 ReplicaSets

A ReplicaSet ensures that a specified number of identical Pod replicas are running at any given time. It monitors the actual cluster state and creates or deletes Pods as necessary to maintain the desired replica count. ReplicaSets provide the foundation for scaling and redundancy in Kubernetes workloads.

2.3.3 Deployments

A Deployment is a higher-level resource that declaratively manages ReplicaSets to provide updates for Pods. It enables rollout strategies such as rolling updates and supports rollbacks to previous versions in case of failures or misconfigurations. Deployments maintain a revision history, allowing administrators to revert to earlier states easily. They are the most common abstraction for managing the full lifecycle of stateless applications in Kubernetes.

2.3.4 Services

A Service provides a stable abstraction over a dynamic group of Pods. It assigns a consistent virtual IP and DNS name, allowing clients to connect without tracking individual Pod IPs. Services decouple clients from the dynamic nature of Pod scheduling and support load balancing across healthy endpoints. Kubernetes supports multiple Service types, including:

- **ClusterIP**: internal access within the cluster.
- **NodePort**: exposes the Service on each Node's IP at a static port.
- **LoadBalancer**: provisions an external load balancer via a cloud provider.
- **ExternalName**: maps the Service to an external DNS name.

2.3.5 Namespaces

Namespaces provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but the same name can be used in different namespaces. Namespaces are intended for use in environments with many teams or projects, enabling scoped access control and resource quotas.

Kubernetes clusters include four predefined namespaces:

- `default`: The default namespace for user-created resources when no other namespace is specified.
- `kube-system`: Reserved for objects created by the Kubernetes system, including control plane components and system add-ons.
- `kube-public`: Readable by all users (including unauthenticated), typically used for cluster-wide public information.
- `kube-node-lease`: Contains lease objects used for efficient node heartbeat tracking and status reporting.

2.3.6 Labels & Selectors

Labels are key/value pairs attached to objects (e.g., Pods, Services, Deployments) to specify identifying attributes meaningful to users. Selectors allow clients (e.g., a Service) to query and filter objects based on their labels. For example, a Service can select all Pods with `app=frontend` to form its set of endpoints.

2.3.7 EndpointSlices

EndpointSlices are the scalable, extensible successor to the original Endpoints API. For each Service with a selector, the control plane automatically creates one or more EndpointSlice objects, each containing references to a subset of the Service's Pods (grouped by IP family, protocol, port, etc.). This design improves performance and scalability when Services back large numbers of Pods.

2.3.8 ConfigMaps

A ConfigMap is designed to decouple non-confidential configuration data from container images. It allows key–value pairs to be stored separately and consumed by Pods as environment variables, command-line arguments, or mounted files. ConfigMaps facilitate dynamic configuration updates without requiring image rebuilds.

2.3.9 Secrets

A Secret is intended for storing sensitive information such as passwords, tokens, and certificates. Secret data is represented in base64-encoded form and can be provided to Pods through environment variables or volume mounts. Unlike ConfigMaps, Secrets offer additional safeguards for sensitive data and support integration with encryption mechanisms.

2.4 Networking in Kubernetes

Networking is a fundamental component of Kubernetes, enabling reliable communication between Pods, Services, and other resources both within the cluster and with external clients. Kubernetes implements a flat Pod network model at its core, with higher-level abstractions—Services, Ingress, and NetworkPolicies—built on top to support modern, scalable application topologies.

2.4.1 Pod Networking

In Kubernetes, each Pod is assigned a unique IP address from a cluster-wide address space. This flat network model ensures that any Pod can communicate directly with any other Pod across nodes, without requiring network address translation (NAT). This design simplifies service discovery and avoids port conflicts on the host, as each Pod manages its own network namespace.

2.4.2 Service Networking

Pods are ephemeral: they can be created, destroyed, or rescheduled dynamically. To provide a stable communication endpoint, Kubernetes introduces the Service abstraction. A Service selects a set of Pods using label selectors and offers:

- A stable **ClusterIP**, acting as a virtual IP for the lifetime of the Service.
- Automatic DNS resolution within the cluster.

Traffic to a Service's ClusterIP is automatically load-balanced across healthy Pod IPs. Kubernetes achieves this using `kube-proxy`, which programs iptables or IPVS rules, or via EndpointSlices—a scalable alternative that partitions large sets of Pod endpoints for efficient lookups and updates.

2.4.3 Ingress and Egress Traffic

While Services handle internal communication, many applications need to expose APIs or web front-ends to users outside the cluster. Kubernetes manages this with the `Ingress` resource. An Ingress defines HTTP(S) routing rules, including hostnames, paths, and TLS settings, mapping incoming traffic to internal Services. Ingress Controllers implement these rules by configuring reverse proxies or load balancers, enabling SSL termination and advanced routing capabilities.

Egress traffic represents outbound connections from Pods to other services—either internal or external. By default, Pods may initiate outbound connections freely. To restrict or control this, Kubernetes administrators can use:

- **NetworkPolicies** defining egress rules, specifying allowed destinations by IP, namespace, or port.
- CNI plugins that support traffic shaping, such as the `bandwidth` plugin, to enforce rate limits on outbound traffic.

2.4.4 Network Policies

`NetworkPolicy` objects enable declarative control over allowed traffic to and from Pods within a namespace. Policies use `podSelector` and `namespaceSelector` to identify affected Pods and define `ingress` and `egress` rules specifying allowed sources, destinations, ports, and protocols. Enforcement is implemented by the active CNI plugin (e.g., Calico, Cilium) using dataplane technologies like iptables or eBPF, ensuring secure, least-privilege networking.

2.4.5 Container Network Interface (CNI)

Kubernetes delegates low-level Pod network configuration to Container Network Interface (CNI) plugins. These plugins assign IP addresses to Pods, set up virtual interfaces, and configure routing between nodes in the cluster. Popular CNI implementations include Calico, Flannel, Weave, and Cilium, each offering different approaches to routing and policy enforcement. While their internal designs vary, all CNI plugins conform to the CNI specification to ensure consistent, interoperable Pod networking across different environments.

2.4.6 Network Architecture Models

Kubernetes clusters can use different network architectures depending on the chosen CNI plugin:

- **Flat network**: all Pods share a single routable address space, allowing direct communication without encapsulation or NAT.

- **Overlay network**: Pods communicate over a virtual network built atop the underlying infrastructure, using encapsulation (e.g., VXLAN) to isolate cluster traffic and simplify compatibility across diverse networking environments.

These architectural choices involve trade-offs in simplicity, performance, and integration with existing infrastructure.

Overall, Kubernetes networking—from Pod IP assignment to Service load balancing, Ingress routing, egress controls, and NetworkPolicy enforcement—ensures secure, scalable, and efficient communication for containerized workloads at any scale.

2.5 API Extensibility

The Kubernetes API is designed to be extensible via third-party mechanisms without modifying the core source code. Three key tools support extensibility: CRDs, Operators, and Admission Controllers.

2.5.1 Custom Resource Definitions (CRDs)

CRDs allow the creation of new resource types defined by OpenAPI schemas. Once registered with the API server, these resources can be manipulated like native Kubernetes objects using standard tools such as `kubectl`.

2.5.2 Operator Pattern

An Operator is a custom controller built to manage complex applications. It combines a CRD that defines the desired state with a reconciliation loop that ensures the actual cluster state matches the specification. Operators automate tasks such as failover, backups, and upgrades, effectively encoding operational knowledge into software.

2.5.3 Admission Controllers

Admission Controllers are API server plugins that validate or mutate incoming requests before they are persisted. They enforce security policies, inject sidecars, or apply organizational rules. Kubernetes supports both mutating and validating admission webhooks.

2.6 Access Control: Role-Based Access Control (RBAC)

Kubernetes uses Role-Based Access Control (RBAC) to manage access permissions across users, service accounts, and controllers. RBAC policies determine which subjects can perform which actions on which resources, using four main object types:

- **Role**: Defines permissions within a single namespace.
- **ClusterRole**: Grants permissions at the cluster scope or for cluster-wide resources.
- **RoleBinding**: Associates a Role with a user, group, or service account in a namespace.
- **ClusterRoleBinding**: Binds a ClusterRole to a subject across the entire cluster.

Each rule specifies a set of verbs (such as `get`, `create`, `update`, `delete`) and resources it applies to. Controllers typically run under a dedicated service account, and must be explicitly granted permissions via RBAC to read, modify, or reconcile resources such as Custom Resources, Secrets, or ConfigMaps.

RBAC is a critical part of Kubernetes extensibility and security. It ensures that custom controllers operate with the least privilege required, and prevents unauthorized access to sensitive operations. Tools such as Kubebuilder facilitate the definition of RBAC rules through annotations, which generate RBAC manifests automatically as part of the controller scaffolding process.

2.7 Developer Tooling: Kubebuilder

Kubebuilder is a framework for developing Kubernetes APIs and operators. Built on top of controller-runtime, it provides tools for scaffolding CRDs, RBAC rules, webhooks, and controllers with minimal boilerplate [3].

2.7.1 Overview and Capabilities

Kubebuilder simplifies the creation of production-grade Operators by providing:

- Project scaffolding (directory structure, Go modules, Makefiles)
- CRD type definitions with OpenAPI schema validation
- RBAC and controller logic via code generation
- Integration testing support using `envtest`

2.7.2 Scaffolding APIs and Controllers

Using the CLI, developers can initialize a project and scaffold APIs with commands such as:

```
kubebuilder init --domain=example.com \
--repo=github.com/example/operator
kubebuilder create api --group=cache --version=v1alpha1 \
--kind=ExampleResource
```

This process generates Go structs for the custom resource, controller logic, CRD YAMLs, and RBAC manifests.

2.7.3 Reconciliation Loop and Finalizers

Each controller implements a `Reconcile()` function that watches resource changes and applies the necessary actions to reach the desired state. Finalizers are used to define cleanup logic that runs before object deletion, ensuring proper teardown of external resources.

Chapter 3

Background: Liqo

This chapter presents Liqo, an open-source framework that extends Kubernetes to support seamless multi-cluster federation, following principles and workflows outlined in its official documentation [4]. It begins with an overview of Liqo's objectives and feature set—dynamic peering, transparent workload offloading, encrypted cross-cluster networking, and federated storage—demonstrating how it unifies disparate clusters under a single, native Kubernetes API. The chapter then examines the Peering subsystem, detailing both the imperative controller-driven workflow and the declarative, GitOps-compatible Custom Resource approach by which clusters establish trust, negotiate resource quotas, and configure secure tunnels. Finally, the various Core Components are described in depth—node-level agents, the controller manager, Virtual Kubelet integration, IPAM, network and reflection managers, and the CRD replicator—illustrating how these elements collaborate to automate federation, offloading, and state synchronization across cluster boundaries.

3.1 Overview of Liqo

Liqo is an open-source project that enables dynamic, peer-to-peer Kubernetes multi-cluster topologies across heterogeneous infrastructures, from on-premises to cloud and edge. It delivers four main features:

- **Peering.** Automatic negotiation of resource and service consumption relationships between independent clusters, with no manual VPN or certificate management.
- **Offloading.** Native, transparent offloading of workloads onto remote clusters by collapsing each into a *virtual node* visible to the standard Kubernetes scheduler.

- **Network Fabric.** A secure, cross-cluster network layer (built on WireGuard and Geneve) that provides pod-to-pod and pod-to-service connectivity regardless of underlying CNIs.
- **Storage Fabric.** Seamless extension of Kubernetes storage to remote clusters, enabling stateful workloads to follow a data-gravity approach.

3.2 Peering

Peering in Liqo is a *unidirectional* relationship by default: a *consumer* cluster offloads and reflects resources onto a *provider* cluster, while the reverse direction is configured separately if needed.

3.2.1 Peering Lifecycle

Establishing a peering involves four main steps:

1. **Network fabric setup.** Exchange of network configuration and creation of WireGuard gateways (server on provider, client on consumer) to form a secure VPN tunnel.
2. **Authentication.** The consumer obtains a limited kubeconfig identity from the provider to perform Liqo-related API operations.
3. **Resource negotiation.** The consumer requests a `ResourceSlice` defining CPU, memory, and other resources; the provider approves or rejects.
4. **Virtual node setup.** Upon approval, the consumer creates a `VirtualNode` resource, making provider resources schedulable via the standard Kubernetes scheduler.

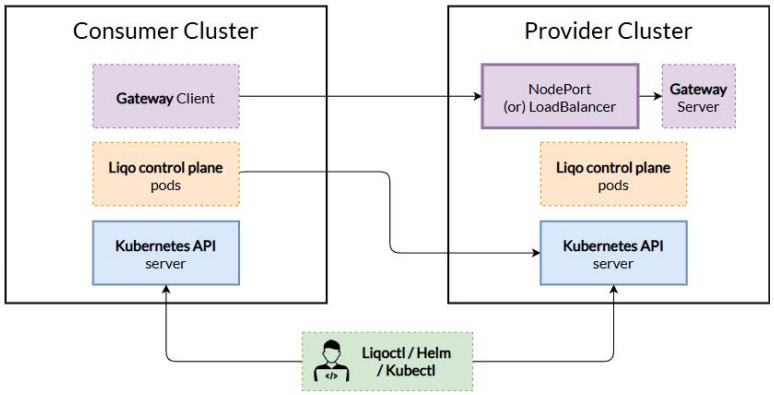

Figure 3.1: Peering architecture between consumer and provider clusters. Source: [5]

3.2.2 Declarative Peering

Peerings can also be managed *declaratively* via Custom Resources (CRs) applied on both clusters. This approach uses:

- Kubernetes `Namespace` labeled for tenancy,
- `Configuration`, `PublicKey`, `GatewayServer`, and `GatewayClient` CRs for networking,
- `Tenant` and `RoleBinding` CRs for authentication,
- `ResourceSlice` CRs for offloading.

By defining these resources declaratively, Liqo automatically negotiates and manages peerings without requiring imperative commands.

3.3 Core Components

Liqo runs as a set of Kubernetes workloads in the `liqo` namespace. Each component encapsulates a subsystem of the multi-cluster topology. These components work collaboratively to manage peering, workload offloading, networking, and resource synchronization.

3.3.1 Liqo Agent

The Liqo Agent comprises lightweight daemon processes running on each node to handle local interactions:

- `liqo-metric-agent`: exposes Prometheus metrics for networking, offloading, and reflection.
- `liqo-proxy` (API server proxy): optionally forwards API traffic when direct access is not possible.
- `liqo-webhook`: injects required tolerations and scheduling constraints into offloaded Pods to ensure compatibility with remote clusters.

3.3.2 Network Manager

The Network Manager is the Liqo subsystem responsible for establishing and maintaining the cross-cluster network fabric. It coordinates control-plane logic, IP address management, and data-plane enforcement to provide secure, seamless connectivity between clusters.

Controller Manager

The `liqo-controller-manager` defines the control-plane logic of the network fabric. It sets up network Custom Resources (CRDs) during the peering process, handles potential network conflicts by defining high-level NAT rules, and manages the translation of Pod IPs during synchronization and EndpointSlice reflection. These rules are enforced by the data-plane components.

IP Address Management (IPAM)

The `liqo-ipam` pod provides an IP address management interface to assign and manage PodCIDRs for remote clusters. It ensures non-overlapping address allocation and exposes an API consumed by the controller-manager to handle IP acquisitions during peering negotiations.

Network Fabric

The `liqo-fabric` pods implement the data-plane of the network fabric. They establish cross-cluster VPN tunnels using WireGuard and configure Geneve overlays on each node. These components enforce routing and nftables NAT rules as defined by the controller-manager, ensuring seamless and secure cross-cluster communication.

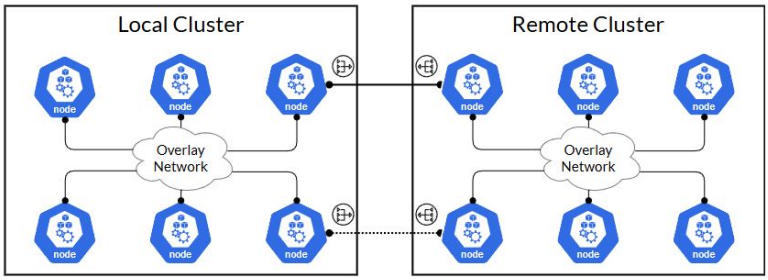

Figure 3.2: High-level network fabric between two clusters. Source: [6]

WireGuard

WireGuard is a modern, open-source VPN protocol designed for simplicity, high performance, and strong security. It employs a minimal codebase and state-of-the-art cryptographic primitives—Curve25519 for key exchange, ChaCha20 for symmetric encryption, and Poly1305 for message authentication—to establish IP tunnels over UDP with low latency and minimal overhead. WireGuard's design

emphasizes ease of deployment, automatic roaming, and a small attack surface, making it well-suited for dynamic multi-cluster networking scenarios [7].

In Liqo, WireGuard underpins the tunnels created by the `GatewayServer` and `GatewayClient` CRs, providing encrypted, authenticated links for all pod-to-pod and pod-to-service traffic across cluster boundaries without requiring operators to manage keys or connections manually.

Geneve

Geneve (Generic Network Virtualization Encapsulation, RFC 8926 [8]) is an IETF-standardized tunneling protocol that encapsulates L2 Ethernet frames in UDP packets and supports extensible metadata via optional TLV fields. In Liqo, `liqo-fabric` pods configure a Geneve interface on each node to carry pod-to-pod and pod-to-service traffic across clusters, decoupling the overlay topology from any specific underlay CNI. This approach ensures interoperability with heterogeneous network fabrics while preserving network isolation and enabling advanced features (e.g. VXLAN compatibility, traffic steering) in a multi-cluster environment.

3.3.3 Virtual Kubelet

The Virtual Kubelet component in Liqo exposes remote clusters as *virtual nodes* within the local Kubernetes cluster. This integration allows the standard scheduler to see and consider resources from peered clusters seamlessly.

It runs as a dedicated pod for each peering, maintaining the `VirtualNode` resource and handling secure, isolated communication with the remote cluster. By doing so, it abstracts the complexity of managing external resources and integrates them natively into the local cluster view.

3.3.4 Reflection Manager

The Reflection Manager ensures that selected Kubernetes resources are kept consistent across clusters involved in a peering. It operates on namespaces marked for offloading, maintaining synchronized copies of key resources in the remote clusters.

It includes the `liqo-crd-replicator` component, which is responsible for ensuring that required Liqo Custom Resources are present and consistent across both the local and remote clusters.

3.4 Custom Resources (CRDs)

Liqo extends the Kubernetes API with a set of Custom Resource Definitions (CRDs) that represent the building blocks for multi-cluster orchestration. These CRDs

model remote clusters, resource allocations, networking endpoints, and offloading policies.

3.4.1 ForeignCluster

The `ForeignCluster` CRD represents a peering relationship with a remote cluster. Its `spec` includes the remote API server address, authentication credentials (a ServiceAccount kubeconfig), and optional resource limits. The `status` subresource reflects the peering phase (e.g. *Bootstrap*, *Accepted*, *Ready*) and conditions such as connectivity health and certificate validity.

3.4.2 ResourceSlice

A `ResourceSlice` object declares the amount of CPU, memory, and other schedulable resources that a provider cluster offers to its peers. It allows fine-grained control over quotas per consumer cluster and is mutually created by both sides during the peering negotiation.

3.4.3 VirtualNode

The `VirtualNode` CRD integrates with the Virtual Kubelet framework, making remote cluster resources appear as a node in the consumer cluster. Its `spec` includes the underlying `ForeignCluster` reference and the `ResourceSlice` to consume. In its `status`, it reports capacity, allocatable resources, and node conditions for scheduling decisions.

3.4.4 NamespaceOffloading

`NamespaceOffloading` enables transparent offloading of an entire namespace's workloads. By applying this CRD to a namespace, Pods are offloaded to the remote cluster, while resources like Services, ConfigMaps, and Secrets are synchronized to maintain consistent application behavior.

3.4.5 ShadowPod & ShadowEndpointSlice

The `ShadowPod` and `ShadowEndpointSlice` CRDs maintain mirrored representations of offloaded resources in the provider cluster. `ShadowPod` tracks the original Pod's specification and status, propagating lifecycle events (such as start, stop, and restart) between clusters. `ShadowEndpointSlice` represents Kubernetes `EndpointSlice` objects, maintaining service discovery data for reflected Services in the remote cluster.

3.4.6 GatewayServer & GatewayClient

Liqo uses two Custom Resources—`GatewayServer` and `GatewayClient`—to abstract the creation of encrypted, cross-cluster tunnels without exposing low-level VPN configuration details.

- **`GatewayServer`** is applied in the provider cluster. It causes Liqo to deploy a gateway pod that listens for incoming WireGuard connections and advertises the provider's Pod network to peers.

- **`GatewayClient`** is applied in the consumer cluster. It instructs Liqo to deploy a gateway pod that dials the provider's gateway and attaches the remote Pod network as a local overlay interface.

These CRDs handle certificate management, port configuration, and network advertisement automatically, so operators interact only with high-level Kubernetes objects.

3.4.7 Connection

The `Connection` CR orchestrates the higher-level peering workflow, binding together the `ForeignCluster`, `GatewayServer/Client`, `ResourceSlice`, and `VirtualNode` resources. It signals when all subcomponents are Healthy, and it drives cleanup when the peering is terminated.

3.4.8 IP

The `IP` Custom Resource records individual IP allocations for cross-cluster networking. It tracks local and remapped IP addresses, ensuring conflict-free, routable communication between clusters. It also supports optional associations with Kubernetes Services and can enable masquerading on nodes when necessary.

3.4.9 Configuration

The `Configuration` Custom Resource defines the overall network mapping between a pair of clusters. It describes both the Pod and External CIDRs for the local and remote clusters, as well as their remapped counterparts. This ensures consistent routing without address conflicts, even in heterogeneous networking environments. The `status` subresource reflects how the declared CIDRs are actually translated during peering.

3.5 Offloading

Offloading in Liqo enables transparent scheduling of Kubernetes Pods onto remote clusters to leverage spare capacity and improve resilience. By marking a namespace for offloading, Liqo intercepts newly created Pods and schedules them onto a `VirtualNode` representing the provider cluster, while maintaining "shadow" instances in the consumer cluster to track their state.

The offloading workflow involves:

1. Annotating a namespace with the `NamespaceOffloading` CR to mark it for offloading.
2. The controller watches for new Pods created in that namespace.
3. For each offloaded Pod, a corresponding `ShadowPod` is created in the consumer cluster to mirror its state.
4. Liqo's Virtual Kubelet schedules the offloaded Pod onto the provider's worker nodes via the `VirtualNode`.
5. Pod status and logs are reflected back to the consumer cluster for seamless monitoring and debugging.

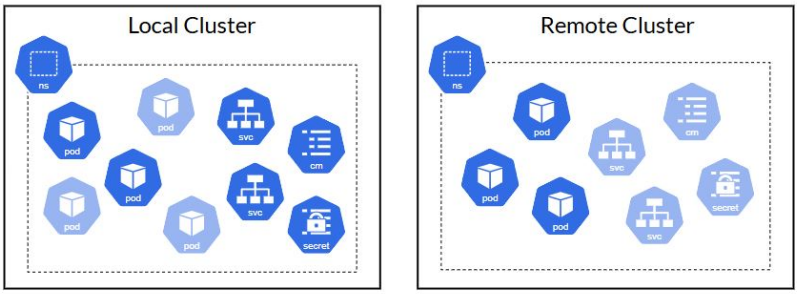

Figure 3.3: Namespace offloading workflow: consumer cluster Pods are mirrored and run on provider resources. Source: [9]

Offloading supports all standard Kubernetes workload types—including Deployments, StatefulSets, and DaemonSets—and synchronizes associated data such as ConfigMaps, Secrets, PersistentVolumeClaims, and volume mounts. Service-related resources are kept in sync across clusters to maintain seamless service discovery and connectivity between consumer and provider. When an offloaded namespace is unmarked, Liqo gracefully cleans up the corresponding remote resources and removes the virtual node binding.

3.6 Resource Reflection

Resource reflection in Liqo ensures that Kubernetes objects remain consistent across clusters by continuously synchronizing their state. It is essential for maintaining service discovery, configuration sharing, and seamless connectivity between consumer and provider clusters during offloading.

Key aspects of reflection:

- Uses resource reflectors to synchronize objects such as Services, EndpointSlices, ConfigMaps, Secrets, and Ingresses between consumer and provider clusters.
- Propagates changes in workload and networking resources from the consumer cluster to the provider cluster to maintain connectivity and functionality for offloaded workloads.
- Maintains consistent metadata and status to ensure seamless integration with Kubernetes-native workflows and tools.

Reflection can be scoped per-namespace or cluster-wide, and supports customization to include or exclude specific resources. This mechanism underpins cross-cluster service discovery, federated configurations, and consistent workload behavior across heterogeneous Kubernetes environments.

3.7 Security and Trust

Security and trust are foundational to Liqo's multi-cluster operations. Liqo employs multiple layers of authentication, authorization, and encryption to ensure secure resource sharing and communication between clusters.

3.7.1 Mutual TLS and WireGuard

All inter-cluster traffic—both API calls and pod-to-pod networking—is secured:

- **API Channel:** The consumer cluster authenticates to the provider using a restricted ServiceAccount kubeconfig. Mutual TLS ensures that only authorized control-plane components can create or modify Liqo resources on the provider cluster.
- **Data Channel:** WireGuard tunnels, automatically configured via Liqo's networking components, provide encrypted and authenticated VPN connections for all pod-to-pod traffic across clusters.

3.7.2 Role-Based Access Control

Liqo's control plane components run under dedicated ServiceAccounts with least-privilege RBAC rules:

- Reconcilers for `ForeignCluster` resources can only create and update Liqo CRDs.
- Namespace offloading logic is scoped to annotated namespaces and cannot affect unrelated resources.
- Network and reflection components operate within the `liqo` namespace, isolating their permissions from application workloads.

These measures establish a zero-trust multi-cluster environment, ensuring only authenticated and authorized peers can share resources and communicate.

Chapter 4

Design: `foreign_cluster_connector`

Liqo enables a **consumer** Kubernetes cluster to offload namespaces and workloads onto geographically distributed **provider** clusters, forming a federated environment where resources can be shared transparently. In this model, each provider cluster retains its own Pod network, while Liqo's networking layer ensures that Pods in different clusters can communicate via remapped IP addresses advertised by the consumer.

By default, all inter-provider traffic is routed through the consumer cluster, which acts as a central hub. This design simplifies routing and service discovery but introduces inefficiencies when direct paths between providers are available but unused.

The goal of the `foreign_cluster_connector` design is to address this limitation by introducing a Kubernetes-native mechanism for declaring and managing direct tunnels between provider clusters. This approach centralizes awareness and lifecycle management in the consumer, while enabling optimized, direct provider-to-provider connectivity without compromising Liqo's existing architecture.

4.1 Baseline Topology and Routing Behavior

The setup comprises three clusters coordinated by Liqo:

- A **Consumer** cluster (`consumer-cluster`), which offloads namespaces to two provider clusters.
- Two **Provider** clusters (`provider-a` and `provider-b`), each running a copy of the offloaded workloads.

Within the offloaded namespace, a Deployment can schedule Pods onto both `provider-a` and `provider-b`. Liqo's Virtual Kubelet on the `consumer-cluster` ensures that Service discovery continues to work seamlessly across clusters by injecting `EndpointSlice` objects into the providers. This lets each provider address remote Pods using a single, advertised IP range. Liqo's overlay network then routes the packets through the `consumer-cluster` to the correct destination, making the `consumer-cluster` the central hub for all cross-provider traffic.

Even if a direct WireGuard tunnel is manually established between the providers, it is unused by default. Liqo's standard configuration does not automatically leverage such direct links, and all inter-provider Pod-to-Pod traffic remains routed via the consumer cluster.

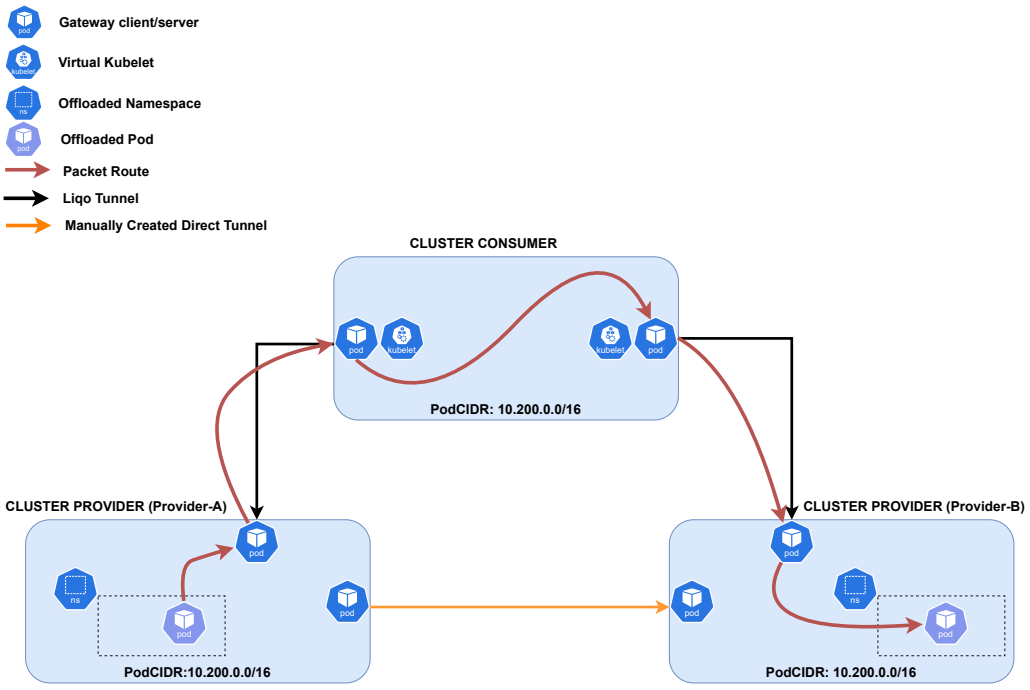

Figure 4.1: Topology with consumer-cluster as the hub. A manual direct tunnel exists between providers, but cross-provider traffic is routed via the consumer.

4.2 Communication Path and Drawbacks

As shown in Figure 4.1, even if a direct tunnel exists between the provider clusters, traffic still traverses the `consumer-cluster`.

The figure illustrates:

- Standard Liqo tunnels between consumer and providers (black lines).
- A manually configured direct tunnel between providers (orange line), which is unused by default.
- Actual Pod-to-Pod traffic paths (red arrows) that flow via the consumer.

This happens because the consumer has no built-in awareness of manually created inter-provider links. It does not advertise the direct-tunnel addresses in the propagated `EndpointSlices`, which list the remapped Pod addresses for Services, allowing providers to discover and route to offloaded workloads. Consequently, providers only learn remapped addresses assigned by the consumer and cannot automatically route Pod-to-Pod traffic directly over the manual tunnel.

This leads to several drawbacks:

1. **Increased Latency:** Traffic takes two hops—provider → consumer → provider—instead of a single direct hop.
2. **Higher Egress Usage:** Traffic unnecessarily exits one provider and enters another via the consumer, consuming bandwidth.
3. **Single Point of Failure:** If the consumer control plane is unavailable, cross-provider traffic fails even if the direct tunnel itself remains functional.

Because routing decisions depend entirely on the addresses advertised by the consumer, any manually established direct tunnel remains invisible to Liqo's default forwarding logic.

4.3 Proposed Solution: Consumer-Side Custom-Resource and Controller

To address the inefficiencies described above, the `consumer-cluster` is extended with a new Kubernetes CustomResource Definition (CRD) called `ForeignCluster Connection`, together with a dedicated controller. The key idea is to let the consumer cluster declaratively define and automatically manage direct tunnels between its provider peers.

Instead of relying on manual configuration, the controller automates the entire lifecycle of these provider-to-provider connections. When a `ForeignCluster Connection` object is applied, the controller retrieves the necessary kubeconfigs for both providers, applies the specified networking parameters to establish the WireGuard tunnel, and finally discovers and stores the resulting remapped Pod

CIDRs in the CR's `status`. This enables the consumer cluster to maintain authoritative awareness of which provider networks can communicate directly, laying the foundation for more efficient routing and simplified network management.

4.3.1 Lightweight Declarative API

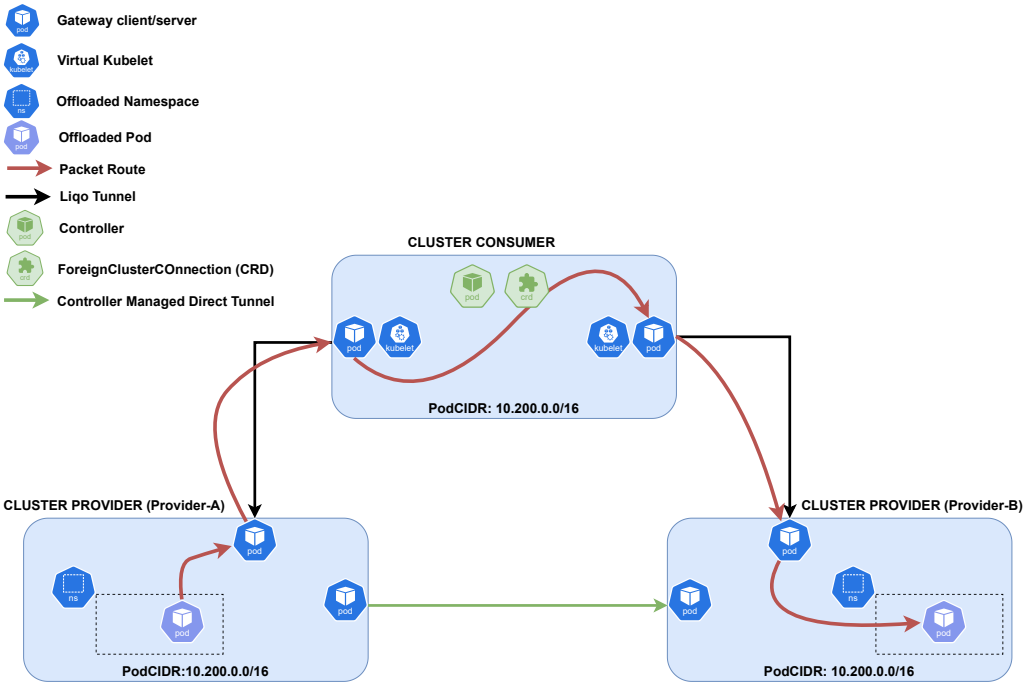

Figure 4.2: Updated topology with the `ForeignClusterConnection` controller. Green elements highlight the new CustomResource, the controller logic in the consumer cluster, and the direct tunnel between providers now automatically managed.

The figure illustrates how the consumer cluster gains centralized management of direct connectivity between providers. The green elements represent the new design features:

- A Kubernetes **CustomResource** (`ForeignClusterConnection`) in the consumer cluster allows users to declaratively specify the intent to establish a direct tunnel between any two provider clusters.

- A dedicated **controller** running in the consumer cluster watches for these resources and automates the creation, management, and teardown of the actual WireGuard tunnels.

- The inter-provider tunnel itself is highlighted in green to emphasize that it is no longer manually configured out-of-band, but is now established and maintained automatically by the controller.

This design solves the previous limitation where the consumer had no built-in awareness of manually created tunnels between providers. By introducing a declarative interface and embedding all tunnel lifecycle logic in the controller, the consumer cluster gains authoritative knowledge of the inter-provider topology. Once a tunnel is created, the controller records the discovered remapped Pod CIDRs in the CR's `status` field. These CIDRs represent the address ranges that provider-a and provider-b will use to communicate directly over the established tunnel.

Declarative Specification of Parameters Each `ForeignClusterConnection` object includes the identifiers of the two provider clusters (e.g., `provider-a` and `provider-b`) and a set of mandatory networking configuration parameters under the `spec.networking` field. These fields define all required tunnel details—such as MTU size, timeouts, gateway templates, and service exposure policies—so that the controller can operate without requiring additional manual steps or prior knowledge of the underlying networking.

No kubeconfig locations or Pod CIDRs need to be manually specified. Instead, the controller automates the entire process:

- Fetches the required kubeconfigs for both provider clusters from Liqo-generated Secrets in the consumer cluster.

- Establishes the WireGuard tunnel between the providers using all configuration parameters defined in the CustomResource.

- Discovers the remapped Pod CIDRs for both providers and records them in the CR's `status`, ensuring the consumer cluster maintains authoritative knowledge of direct routing possibilities.

Example ForeignClusterConnection CR

Below are two example YAML fragments that illustrate how the `ForeignCluster Connection` resource is defined and populated.

Spec section (defined by the user) This snippet shows an example of the `spec` section that an user would apply to request the creation of a direct tunnel. It includes the IDs of the two provider clusters and networking parameters that instruct the controller how to configure the WireGuard connection.

Listing 4.1: Example ForeignClusterConnection CR spec

```
apiVersion: networking.liqo.io/v1beta1
kind: ForeignClusterConnection
metadata:
  name: provider-a-provider-b
  namespace: default
spec:
  foreignClusterA: provider-a
  foreignClusterB: provider-b
  networking:
    mtu: 1450
    timeoutSeconds: 120
    wait: true
    serverGatewayType: networking.liqo.io/v1beta1/
    wggatewayservertemplates
    serverTemplateName: wireguard-server
    serverTemplateNamespace: liqo
    serverServiceType: NodePort
    serverServicePort: 51840
    clientGatewayType: networking.liqo.io/v1beta1/
    wggatewayclienttemplates
    clientTemplateName: wireguard-client
    clientTemplateNamespace: liqo
```

Status section (populated by the controller at runtime) After the controller establishes the tunnel, it automatically fills in the `status` field of the resource. This includes details about the connection state and the remapped Pod CIDRs that the two providers will use to communicate directly over the tunnel. This is an example of what the `status` section might look like after the tunnel is established.

Listing 4.2: Example ForeignClusterConnection CR status

```
status:
  isConnected: true
  phase: Connected
  lastUpdated: "2025-06-18T12:34:56Z"
  foreignClusterANetworking:
    podCIDR:          "10.200.0.0/16"  # provider-a's local range
    remappedPodCIDR: "10.61.0.0/16"   # provider-a's range over the
    direct tunnel
  foreignClusterBNetworking:
    podCIDR:          "10.200.0.0/16"  # provider-b's local range
```

```
10      remappedPodCIDR: "10.63.0.0/16"   # provider-b's range over the
    direct tunnel
```

Controller-Driven Lifecycle

- *On create*: The controller establishes a Liqo-based WireGuard tunnel between the two specified providers and writes the resulting remapped Pod CIDRs into the CR's `status`.
- *On delete*: The controller tears down that specific provider-to-provider tunnel and clears the associated CIDRs, restoring the system to its original state.

By embedding these actions in the reconcile loop, direct links become fully declarative: applying or deleting a `ForeignClusterConnection` object is sufficient to create or remove the tunnel.

Consumer-Centric Awareness Storing the remapped Pod CIDRs in the CR's `status` ensures that the consumer cluster maintains authoritative, persistent knowledge of which provider networks can communicate directly. This centralized awareness enables consistent management of inter-provider shortcuts and provides the foundation for future enhancements such as dynamic route advertisement or policy-driven optimizations.

4.3.2 Key Advantages

- **Declarative and GitOps-Friendly Automation:** Users define direct tunnels simply by applying a `ForeignClusterConnection` resource, without needing manual commands or custom scripts. The controller automates the entire lifecycle—retrieving kubeconfigs, applying all network settings, and creating or tearing down tunnels. This declarative model aligns with Kubernetes best practices and GitOps workflows, ensuring that connectivity remains reproducible, auditable, and consistent across different environments.
- **Centralized, Authoritative Awareness:** By storing remapped Pod CIDRs in each `ForeignClusterConnection`'s `status`, the consumer cluster maintains authoritative knowledge of which provider networks can communicate directly. This centralized approach avoids fragmentation of configuration and enables the consumer to act as the single source of truth for inter-provider connectivity. It lays the groundwork for the consumer to potentially inform providers about direct paths in the future, improving service discovery and routing consistency.

- **Uniform and Scalable Management Across Providers:** Managing tunnels declaratively through Kubernetes manifests ensures consistent behavior even as the federation grows. Adding new connections between any pair of providers becomes as simple as applying a new CR, making the system inherently scalable and reducing the risk of human error when managing multiple connections in complex topologies.
- **Resilience and Improved Network Performance:** Tunnels created through `ForeignClusterConnection` persist independently of the consumer's control plane availability. Even if the consumer cluster experiences downtime, existing direct tunnels between providers continue to operate until the CR is explicitly removed. This ensures higher availability for inter-provider traffic. Moreover, by enabling direct paths, the system reduces latency and avoids unnecessary egress through the consumer cluster, improving both performance and bandwidth efficiency.
- **Foundation for Future Enhancements:** By capturing remapped Pod CIDRs and connection state in the CR's `status`, the design is prepared for future capabilities. This includes the potential for automatic EndpointSlice rewriting to advertise direct addresses to providers, policy-driven routing decisions based on SLA requirements or bandwidth constraints, and integration with observability or auditing dashboards that visualize all active inter-provider tunnels.

4.4 Architectural Impact

This design represents a clear architectural shift for Liqo-based federations: moving from a strictly consumer-mediated routing model to one where the consumer cluster coordinates and manages direct provider-to-provider connectivity in a fully declarative way.

Rather than introducing separate tooling or manual processes, the solution integrates natively with Kubernetes, using Custom Resources and controllers to unify configuration and lifecycle management. This approach ensures consistency across clusters, simplifies operations, and maintains compatibility with GitOps workflows.

By centralizing tunnel awareness and management in the consumer, the architecture improves resilience, reduces unnecessary traffic through the hub, and prepares the ground for future enhancements such as dynamic routing policies or EndpointSlice optimization.

This concludes the design overview, providing the context and motivation for the implementation details described in the next chapter.

Chapter 5

Implementation of the `foreign_cluster_connector` Controller

This chapter provides a technical breakdown of the `foreign_cluster_connector` Operator implementation. It begins by outlining the CustomResourceDefinition schema and its role in driving controller logic, and then describes the project scaffolding created via Kubebuilder along with the resulting file structure. It also details the controller code itself, covering the phases of the Reconcile loop, routines for establishing and tearing down tunnels, helpers for status updates, and functions for CIDR discovery. An end-to-end example is included to demonstrate the entire reconciliation workflow in practice.

The complete source code is publicly available in the official repository [10].

5.1 Core Components

The implementation relies on two fundamental elements: the Operator, responsible for automating the lifecycle of cross-cluster tunnels by reacting to Kubernetes events and using Liqo networking libraries, and the CustomResourceDefinition, which defines the schema through which connections are declared (via spec fields) and status is reported (via status fields) for visibility into network state and errors.

5.1.1 Operator Overview

An Operator in Kubernetes is a specialized controller that extends the API with custom logic. The `foreign_cluster_connector` Operator registers a Reconciler

for the `ForeignClusterConnection` CRD and runs a continuous loop (Reconcile) whenever a CR instance changes. In practice, it performs these steps:

1. **Watch the CRD:** Uses controller-runtime to subscribe to events on `ForeignClusterConnection` objects.
2. **Fetch Kubeconfigs:** Reads Liqo-generated Secrets (`liqo-tenant-<cluster>/kubeconfig-controlplane-<cluster>`) and writes temporary kubeconfig files.
3. **Use Liqo Networking Libraries:** Initializes two `factory.Factory` clients (local and remote), populates a `network.Options` struct from the CR's `spec.networking`, and calls `RunConnect` or `RunReset`.
4. **Patch Status:** Updates `status.phase`, `status.isConnected`, and network CIDRs in `status.foreignCluster?Networking`.
5. **Handle Deletion:** Detects `DeletionTimestamp`, runs teardown logic, removes the finalizer, and lets Kubernetes garbage-collect the CR.

The Operator retrieves Liqo-generated kubeconfig secrets, uses the controller-runtime client to manage CustomResources, initializes networking factories with the CR's networking parameters, runs the connection or teardown routines, and patches the CR status with phase, connectivity flag, and CIDR details—resulting in deterministic, programmatic tunnel provisioning and cleanup.

5.1.2 Custom Resource Definition

The CustomResourceDefinition (CRD) defines the Kubernetes API schema for the new resource type `ForeignClusterConnection`. It specifies the `spec` fields where users declare which clusters to connect and how, and the `status` fields where the Operator records progress, errors, and networking details.

CRD Manifest (YAML)

The full CRD YAML `foreignclusterconnections.yaml` is maintained in the project repository at `config/crd/bases/`. Below is an excerpt highlighting the key `spec` and `status` schema fields that users and the Operator interact with:

Listing 5.1: Excerpt of the ForeignClusterConnection CRD schema

```
spec:
  type: object
  required: [foreignClusterA, foreignClusterB]
  properties:
```

```
    foreignClusterA:
      type: string
    foreignClusterB:
      type: string
    networking:
      type: object
      properties:
        mtu: integer
        serverGatewayType: string
        clientGatewayType: string
        serverTemplateName: string
        clientTemplateName: string
        serverServiceType: string
        serverServicePort: integer
        timeoutSeconds: integer
        wait: boolean

status:
  type: object
  properties:
    isConnected: boolean
    phase: string
    errorMessage: string
    foreignClusterANetworking:
      type: object
      properties:
        podCIDR: string
        remappedPodCIDR: string
    foreignClusterBNetworking:
      type: object
      properties:
        podCIDR: string
        remappedPodCIDR: string
    lastUpdated: string
```

Note: This excerpt omits standard metadata, versioning, subresources, and printer column definitions for clarity. The complete CRD manifest, including full OpenAPI validation details, is available in the project repository.

Spec & Status Fields

Users declare the desired connection in the `spec` block, while the Operator records its observed state in the `status` block. Key fields include:

- **`spec.foreignClusterA` / `spec.foreignClusterB`** Identifiers of the provider clusters to connect. These specify which Liqo-generated kubeconfig Secrets the Operator retrieves to establish the connection.

- **`spec.networking`** Defines the complete set of tunnel configuration options:
  - `mtu`: sets the WireGuard interface MTU.
  - `serverGatewayType` / `clientGatewayType`: specify the transport mechanism (typically `wireguard`).
  - `serverTemplateName` / `clientTemplateName`: optionally reference custom Gateway CR templates for advanced setups.
  - `serverServiceType` / `serverServicePort`: define how the server exposes its service (e.g., ClusterIP, LoadBalancer, NodePort).
  - `timeoutSeconds`: maximum duration before connect or disconnect operations time out.
  - `wait`: if true, instructs the Operator to wait until the tunnel is fully established before completing.
- **`status.isConnected`** A boolean flag indicating whether the `GatewayServer` and `GatewayClient` resources were successfully created and reconciled.
- **`status.phase`** Represents the connection's lifecycle state as managed by the Operator, with values such as `Pending`, `Connecting`, `Connected`, or `Failed`—making reconciliation progress easy to track.
- **`status.errorMessage`** Records any error encountered during connection or teardown, making failures immediately visible via `kubectl describe`.
- **`status.foreignClusterA(B)Networking`** Populated after tunnel creation by reading Liqo's `Network` CRs in the tenant namespaces. Each contains:
  - `podCIDR`: the original Pod network range on that cluster.
  - `remappedPodCIDR`: the translated CIDR range used over the overlay tunnel.
- **`status.lastUpdated`** An RFC3339 timestamp marking when the Operator last updated the status—useful for auditing and troubleshooting.

5.2 Project Generation with Kubebuilder

Scaffolding the `foreign_cluster_connector` starts with Kubebuilder. A few CLI commands and Go-level annotations set up module metadata, API types, controller stubs, CRD schemas, RBAC rules, and optional webhook or metrics plumbing.

5.2.1 Initializing the Module

The `kubebuilder init` command lays the foundation:

```
kubebuilder init \
  --domain networking.liqo.io \
  --repo github.com/youruser/foreign_cluster_connector
```

Behind the scenes, this command:

- Creates `go.mod` and `go.sum` for dependency management.
- Emits a `PROJECT` file defining the API domain (`networking.liqo.io`) and module path.
- Bootstraps folders: `api/`, `internal/controller/`, `config/`, along with stubs for webhooks and metrics servers.
- Pulls in controller-runtime and client-go dependencies, enabling controller development without manual imports.

5.2.2 Creating the API

Next, the `create api` command scaffolds both the type definitions and a reconciler skeleton:

```
kubebuilder create api \
  --group networking \
  --version v1beta1 \
  --kind ForeignClusterConnection
```

This generates:

- `api/v1beta1/foreignclusterconnection_types.go`, pre-populated with empty `Spec` and `Status` structs plus markers for root object and status subresource.
- `internal/controller/foreignclusterconnection_controller.go`, with a stubbed `Reconcile` method and RBAC markers commented in.
- A CRD template under `config/crd/bases/`, ready to be completed by `controller-gen`.
- Updates to `main.go` registering the new API scheme and wiring up the reconciler with the manager.

5.2.3 Generating Code and Manifests

Kubebuilder uses structured Go annotations to automate both code and manifest generation. Two standard Make targets help keep the Go code and manifest definitions consistent:

- `make generate` regenerates Go code derived from the markers in the API types, including deepcopy methods and interface implementations. Typically used whenever the CRD Go structs are modified.
- `make manifests` invokes `controller-gen` to produce the CRD YAML (including OpenAPI validation, printer columns, and subresources) as well as the RBAC Role and RoleBinding manifests. This ensures Kubernetes has the correct resource definitions and permissions.

A typical sequence during development:

```
make generate
make manifests
```

Example Kubebuilder Annotations:

```
// In api/v1beta1/foreignclusterconnection_types.go
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status
// +kubebuilder:resource:singular=foreignclusterconnection,categories=liqo,shortName=fcc;fcconnection
// +kubebuilder:printcolumn:name="ClusterA",type=string,JSONPath=`.spec.foreignClusterA`
// +kubebuilder:printcolumn:name="ClusterB",type=string,JSONPath=`.spec.foreignClusterB`
// +kubebuilder:printcolumn:name="Connected",type=boolean,JSONPath=`.status.isConnected`
// +kubebuilder:printcolumn:name="Phase",type=string,JSONPath=`.status.phase`
// +kubebuilder:printcolumn:name="A-CIDR",type=string,JSONPath=`.status.foreignClusterANetworking.remappedPodCIDR`
// +kubebuilder:printcolumn:name="B-CIDR",type=string,JSONPath=`.status.foreignClusterBNetworking.remappedPodCIDR`

// In internal/controller/foreignclusterconnection_controller.go
// +kubebuilder:rbac:groups=networking.liqo.io,resources=foreignclusterconnections,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=networking.liqo.io,resources=foreignclusterconnections/status,verbs=get;update;patch
```

```
15 // +kubebuilder:rbac:groups=networking.liqo.io,resources=
       foreignclusterconnections/finalizers,verbs=update
16 // +kubebuilder:rbac:groups="",resources=secrets,verbs=get;list;watch
```

These annotations are processed by `controller-gen` during the Make targets to keep Go types, generated code, CRD schemas, and RBAC roles synchronized and up to date.

5.2.4 Project Structure

After running the Kubebuilder commands along with `make generate` and `make manifests`, the directory layout typically appears as follows:

```
foreign_cluster_connector/
├── Makefile
├── PROJECT
├── go.mod
├── go.sum
├── cmd/
│   └── manager/
│       └── main.go
├── api/
│   └── v1beta1/
│       └── foreignclusterconnection_types.go
├── internal/
│   └── controller/
│       └── foreignclusterconnection_controller.go
└── config/
    ├── crd/
    │   └── bases/
    ├── rbac/
    │   └── role*.yaml
    └── manager/
        └── manager.yaml
```

Directory responsibilities:

- **Makefile, go.mod, go.sum, PROJECT:** define build targets, module dependencies, and Kubebuilder configuration.
- **cmd/manager/main.go:** initializes the controller manager, sets up leader election and metrics endpoints, and registers the `foreign_cluster_connector` reconciler.
- **api/v1beta1/foreignclusterconnection_types.go:** declares the `Foreign`

`ClusterConnection` CRD Go type, including validation tags, default values, and printer-column definitions used in the generated CRD YAML.

- **internal/controller/foreignclusterconnection_controller.go:** implements the Reconcile loop and supporting logic for kubeconfig retrieval, networking setup with Liqo libraries, CIDR discovery, and status management.
- **config/crd/bases:** contains the generated CRD YAML manifest, kept synchronized via `controller-gen`.
- **config/rbac:** includes Role and RoleBinding manifests granting the permissions derived from RBAC markers in the Go code.
- **config/manager:** defines the Kubernetes Deployment manifest for the operator, specifying container image, resource requirements, and environment configuration.

5.3 Controller Implementation

This section dives into the core of the controller code. It explains what happens inside the Reconcile loop (fetching resources, handling deletions, finalizers, idempotence checks, status initialization, and the connection flow), and then describes the helper methods for connecting, disconnecting, updating status, and retrieving CIDR information.

5.3.1 Reconcile Loop

The heart of the controller is the `Reconcile` method, where all decision logic resides. The method declaration and initial resource fetch logic are shown below:

Listing 5.2: Reconcile signature and initial fetch

```
1  func (r *ForeignClusterConnectionReconciler) Reconcile(ctx context.
       Context, req ctrl.Request) (ctrl.Result, error) {
2      logger := log.FromContext(ctx)
3      logger.Info("Reconciling ForeignClusterConnection", "namespace",
       req.Namespace, "name", req.Name)
4
5      // 1. Fetch the ForeignClusterConnection instance
6      var connection networkingv1beta1.ForeignClusterConnection
7      if err := r.Get(ctx, req.NamespacedName, &connection); err != nil
       {
8          // Ignore not-found errors (resource deleted)
9          return ctrl.Result{}, client.IgnoreNotFound(err)
10     }
```

```
    // ... proceed with reconcile steps ...
}
```

The reconcile loop is structured into several distinct phases:

Deletion Handling

After retrieving the `ForeignClusterConnection` resource (stored in `connection`), the controller first checks whether the resource has a non-zero `DeletionTimestamp`. This indicates that the user has requested deletion of the resource, requiring appropriate cleanup:

Listing 5.3: Deletion and teardown logic

```
if !connection.ObjectMeta.DeletionTimestamp.IsZero() {
    logger.Info("ForeignClusterConnection is being deleted, starting
    disconnection", "name", req.Name)
    if err := r.disconnectLiqoctl(ctx, &connection); err != nil {
        logger.Error(err, "Error during disconnection")
        return ctrl.Result{}, err
    }
    // Remove finalizer so Kubernetes can garbage-collect
    controllerutil.RemoveFinalizer(&connection, finalizerName)
    if err := r.Update(ctx, &connection); err != nil {
        return ctrl.Result{}, err
    }
    logger.Info("Finalizer removed, ForeignClusterConnection can be
    deleted", "name", req.Name)
    return ctrl.Result{}, nil
}
```

Key points:

- A non-zero `DeletionTimestamp` indicates a delete event.
- `disconnectLiqoctl` performs the tunnel teardown.
- `RemoveFinalizer` ensures Kubernetes can remove the CR after cleanup is completed.

Finalizer Management

If the CR is not being deleted, the next step is to ensure that a finalizer is present so that future deletions will trigger the controller's cleanup routine:

Listing 5.4: Adding a finalizer if it doesn't exist

```
if !controllerutil.ContainsFinalizer(&connection, finalizerName) {
    logger.Info("Adding finalizer", "name", req.Name)
    controllerutil.AddFinalizer(&connection, finalizerName)
    if err := r.Update(ctx, &connection); err != nil {
        return ctrl.Result{}, err
    }
}
```

Key points:

- If the list of finalizers on the CR does not include `finalizerName`, it is appended.
- The CR is then updated to persist the finalizer.
- The presence of the finalizer guarantees that `Reconcile` will be invoked on deletion to perform cleanup.

Idempotence Check

To avoid re-creating a tunnel if it is already active, the controller checks the `Status.IsConnected` flag:

Listing 5.5: Skip if already connected

```
if connection.Status.IsConnected {
    logger.Info("Clusters already connected", "clusterA", connection.
    Spec.ForeignClusterA, "clusterB", connection.Spec.ForeignClusterB)
    return ctrl.Result{}, nil
}
```

If `Status.IsConnected` is `true`, the `Reconcile` invocation terminates immediately, as the desired state is already achieved.

Status Initialization

When a CR is first created, `Status.Phase` is typically empty. It is initialized to `Pending`:

Listing 5.6: Initialize status on first reconcile

```
if connection.Status.Phase == "" {
    connection.Status = networkingv1beta1.
    ForeignClusterConnectionStatus{
        Phase:       "Pending",
        IsConnected: false,
        LastUpdated: time.Now().Format(time.RFC3339),
```

```
6           ErrorMessage: "",
7       }
8       if err := r.Status().Update(ctx, &connection); err != nil {
9           logger.Error(err, "Error initializing status")
10          return ctrl.Result{}, err
11      }
12  }
```

After this update, `connection.Status.Phase` is set to `Pending` and `Status.IsConnected` to `false`.

Connection Flow

Once the CR's status is initialized and no finalizer is blocking deletion, the controller sets the phase to `Connecting` and invokes `executeLiqoctlConnect` to establish the tunnel:

Listing 5.7: Connection flow in `Reconcile`

```
1  logger.Info("Starting connection", "clusterA", connection.Spec.
       ForeignClusterA, "clusterB", connection.Spec.ForeignClusterB)
2
3  // 1. Update status to "Connecting"
4  if err := r.updateStatus(ctx, &connection, "Connecting", ""); err !=
       nil {
5      return ctrl.Result{}, err
6  }
7
8  // 2. Run the connect routine
9  output, err := r.executeLiqoctlConnect(ctx, &connection)
10 if err != nil {
11     // 2a. On error, log and mark status "Failed"
12     logger.Error(err, "Error during liqoctl connect", "output",
       output)
13     _ = r.updateStatus(ctx, &connection, "Failed",
14         fmt.Sprintf("Error: %v, Output: %s", err, output))
15     return ctrl.Result{}, err
16 }
17
18 // 3. On success, patch status to "Connected"
19 logger.Info("Connection succeeded", "clusterA", connection.Spec.
       ForeignClusterA, "clusterB", connection.Spec.ForeignClusterB)
20 if err := r.updateStatus(ctx, &connection, "Connected", ""); err !=
       nil {
21     return ctrl.Result{}, err
22 }
23
24 // 4. Requeue after 30s to catch any health regressions
25 return ctrl.Result{RequeueAfter: 30 * time.Second}, nil
```

In more detail:

1. **Set Phase to Connecting.** The `updateStatus` method patches `status.phase = "Connecting"` and records the current timestamp, indicating that the operator is preparing to establish the tunnel.

2. **Invoke `executeLiqoctlConnect`.** This function performs all necessary steps to create the network tunnel between clusters. It handles credential retrieval, parameter configuration, and the execution of the connection operation.

3. **Handle Errors.** If `executeLiqoctlConnect` fails, the controller logs the error and updates the CR status to `phase = "Failed"`, including diagnostic details in `errorMessage`. This ensures failures are easily visible for troubleshooting via `kubectl describe`.

4. **Mark as Connected.** If the connection is successful, `updateStatus` is called with `"Connected"`, also setting `status.isConnected = true`. This indicates that the overlay tunnel is active and ready for use.

5. **Requeue for Health Checks.** Returning `ctrl.Result{RequeueAfter: 30s}` schedules another reconciliation in 30 seconds. This periodic check helps detect and address connectivity regressions or stale status information.

5.3.2 Connection Logic

The `executeLiqoctlConnect` method implements the full sequence required to establish the network tunnel between clusters. Its implementation is shown below:

Listing 5.8: `executeLiqoctlConnect` implementation

```
1  func (r *ForeignClusterConnectionReconciler) executeLiqoctlConnect(
2      ctx context.Context,
3      connection *networkingv1beta1.ForeignClusterConnection,
4  ) (string, error) {
5      // 1. Load kubeconfigs
6      kubeconfigA, err := r.getKubeconfigFromLiqo(ctx, connection.Spec.
       ForeignClusterA)
7      if err != nil {
8          return "", fmt.Errorf("error retrieving kubeconfig for
       ForeignClusterA: %v", err)
9      }
10     defer os.Remove(kubeconfigA)
11
12     kubeconfigB, err := r.getKubeconfigFromLiqo(ctx, connection.Spec.
       ForeignClusterB)
13     if err != nil {
```

```
14          return "", fmt.Errorf("error retrieving kubeconfig for
    ForeignClusterB: %v", err)
15      }
16      defer os.Remove(kubeconfigB)
17
18      // 2. Set timeout
19      timeout := 120 * time.Second
20      if connection.Spec.Networking.TimeoutSeconds > 0 {
21          timeout = time.Duration(connection.Spec.Networking.
    TimeoutSeconds) * time.Second
22      }
23      ctx, cancel := context.WithTimeout(ctx, timeout)
24      defer cancel()
25
26      // 3. Initialize factories
27      os.Setenv("KUBECONFIG", kubeconfigA)
28      localFactory := factory.NewForLocal()
29      if err := localFactory.Initialize(); err != nil {
30          return "", fmt.Errorf("localFactory initialization error: %v"
    , err)
31      }
32
33      os.Setenv("KUBECONFIG", kubeconfigB)
34      remoteFactory := factory.NewForRemote()
35      if err := remoteFactory.Initialize(); err != nil {
36          return "", fmt.Errorf("remoteFactory initialization error: %v
    ", err)
37      }
38
39      // Reset KUBECONFIG back to cluster A for operations
40      os.Setenv("KUBECONFIG", kubeconfigA)
41      localFactory.Namespace = ""
42      remoteFactory.Namespace = ""
43
44      // 4. Populate network.Options from the CR spec
45      netCfg := connection.Spec.Networking
46      opts := network.NewOptions(localFactory)
47      opts.RemoteFactory = remoteFactory
48
49      opts.MTU = int(netCfg.MTU)
50      opts.ServerGatewayType = netCfg.ServerGatewayType
51      opts.ClientGatewayType = netCfg.ClientGatewayType
52      opts.ServerTemplateName = netCfg.ServerTemplateName
53      opts.ServerTemplateNamespace = netCfg.ServerTemplateNamespace
54      opts.ClientTemplateName = netCfg.ClientTemplateName
55      opts.ClientTemplateNamespace = netCfg.ClientTemplateNamespace
56      opts.ServerServiceType.Set(netCfg.ServerServiceType)
57      opts.ServerServicePort = netCfg.ServerServicePort
58      opts.Timeout = timeout
```

```
59      opts.Wait = netCfg.Wait
60
61      // 5. Configure output printers
62      localFactory.Printer = output.NewLocalPrinter(true, true)
63      remoteFactory.Printer = output.NewRemotePrinter(true, true)
64
65      // 6. Execute connection
66      fmt.Println("Executing 'network connect'...")
67      if err := opts.RunConnect(ctx); err != nil {
68          return "", fmt.Errorf("error during 'network connect': %v",
    err)
69      }
70
71      // 7. After connect, fetch and patch CIDRs into status
72      if err := r.populateCIDRsFromNetworkConfig(ctx, connection, *
    localFactory, *remoteFactory); err != nil {
73          return "", fmt.Errorf("unable to load CIDRs: %v", err)
74      }
75
76      return "Operation 'network connect' completed successfully.", nil
77  }
```

Breaking down the main steps:

1. **Kubeconfig Extraction:** Loads the kubeconfig Secrets for both clusters by calling `getKubeconfigFromLiqo`, which writes each config to a temporary file with any namespace binding removed. These files are then used to authenticate with the respective clusters.

2. **Timeout Setup:** Configures a `context.Context` with a deadline derived from `spec.networking.timeoutSeconds` to ensure the operation has a bounded duration.

3. **Factory Initialization:** Sets the `KUBECONFIG` environment variable to the extracted paths and initializes one factory per cluster. The `Namespace` field in each factory is explicitly cleared to allow Liqo to automatically determine or create the tenant namespace following its standard naming convention (e.g., `liqo-tenant-<foreign-cluster>`).

4. **Options Assembly:** Populates a `network.Options` structure with all fields from `connection.spec.networking`, including MTU, gateway types, template names, service exposure settings, timeout, and wait flag.

5. **Connection Execution:** Invokes `RunConnect`, which creates or updates the required CustomResources: a `GatewayServer` in `ForeignClusterB` and a `GatewayClient` in `ForeignClusterA`. Controllers in each cluster then reconcile these resources to establish the encrypted tunnel.

6. **CIDR Population:** Calls `populateCIDRsFromNetworkConfig` to retrieve the Pod CIDR and remapped CIDR from the newly created `Network` resources in each tenant namespace, updating them in `connection.status`.

Kubeconfig Extraction

The helper `getKubeconfigFromLiqo` is responsible for the following steps:

Listing 5.9: `getKubeconfigFromLiqo` extracts and writes a temp kubeconfig

```
func (r *ForeignClusterConnectionReconciler) getKubeconfigFromLiqo(
    ctx context.Context,
    clusterName string,
) (string, error) {
    namespace := fmt.Sprintf("liqo-tenant-%s", clusterName)
    secretName := fmt.Sprintf("kubeconfig-controlplane-%s",
    clusterName)

    var secret corev1.Secret
    if err := r.Get(ctx, client.ObjectKey{Namespace: namespace, Name:
    secretName}, &secret); err != nil {
        return "", fmt.Errorf("Error retrieving Secret %s in
    namespace %s: %v", secretName, namespace, err)
    }
    data, exists := secret.Data["kubeconfig"]
    if !exists {
        return "", fmt.Errorf("Secret %s missing 'kubeconfig' key",
    secretName)
    }
    config, err := clientcmd.Load(data)
    if err != nil {
        return "", fmt.Errorf("Error parsing kubeconfig: %v", err)
    }
    if config.CurrentContext == "" {
        return "", fmt.Errorf("Kubeconfig has no current context")
    }
    config.Contexts[config.CurrentContext].Namespace = ""

    modified, err := clientcmd.Write(*config)
    if err := os.WriteFile(kubeconfigPath, modified, 0600); err !=
    nil {
        return "", fmt.Errorf("Error writing kubeconfig file: %v",
    err)
    }

    return kubeconfigPath, nil
}
```

Core details:

- Fetches the Secret named `kubeconfig-controlplane-<cluster>` from the `liqo-tenant-<cluster>` namespace.
- Extracts and parses the kubeconfig data from the Secret.
- Removes any namespace binding from the current context to allow Liqo to set or create the tenant namespace following its standard naming convention (e.g., `liqo-tenant-<foreign-cluster>`).
- Writes the modified configuration to a temporary file and returns its path.

5.3.3 Status Management

Patching the status subresource is essential to maintain an up-to-date view of the connection's state. The `updateStatus` helper method centralizes this responsibility:

Listing 5.10: `updateStatus` helper

```
func (r *ForeignClusterConnectionReconciler) updateStatus(
    ctx context.Context,
    connection *networkingv1beta1.ForeignClusterConnection,
    phase, errorMsg string,
) error {
    patch := client.MergeFrom(connection.DeepCopy())

    connection.Status.Phase = phase
    connection.Status.LastUpdated = time.Now().Format(time.RFC3339)
    connection.Status.ErrorMessage = errorMsg
    connection.Status.IsConnected = (phase == "Connected")

    if err := r.Status().Patch(ctx, connection, patch); err != nil {
        log.FromContext(ctx).Error(err, "Error updating status")
        return err
    }
    return nil
}
```

Whenever the phase transitions (`Pending`, `Connecting`, `Connected`, `Failed`) or an error needs to be recorded, this method ensures that only the `status` subresource is updated to reflect the current state of the connection.

5.3.4 CIDR Retrieval

Once the tunnel has been established, it is important to collect and record the remapped Pod CIDRs for both clusters. Storing this information in the central resource ensures that the system has an accurate view of the address mappings used over the direct connection, which can be leveraged for more advanced traffic

management or monitoring in the future. This step is implemented through two helper functions: `populateCIDRsFromNetworkConfig`, which coordinates the status update, and `retrieveCIDRInfoFromFactory`, which retrieves the necessary network configuration from each cluster.

Listing 5.11: `populateCIDRsFromNetworkConfig` and CIDR helper

```
1  func (r *ForeignClusterConnectionReconciler)
       populateCIDRsFromNetworkConfig(
2      ctx context.Context,
3      connection *networkingv1beta1.ForeignClusterConnection,
4      localFactory factory.Factory,
5      remoteFactory factory.Factory,
6  ) error {
7      update := connection.DeepCopy()
8
9      cidrA, err := r.retrieveCIDRInfoFromFactory(ctx, localFactory,
       connection.Spec.ForeignClusterB)
10     if err != nil {
11         return fmt.Errorf("error retrieving CIDR from cluster A: %w",
       err)
12     }
13
14     cidrB, err := r.retrieveCIDRInfoFromFactory(ctx, remoteFactory,
       connection.Spec.ForeignClusterA)
15     if err != nil {
16         return fmt.Errorf("error retrieving CIDR from cluster B: %w",
       err)
17     }
18
19     update.Status.ForeignClusterANetworking = cidrA
20     update.Status.ForeignClusterBNetworking = cidrB
21
22     patch := client.MergeFrom(connection)
23     if err := r.Status().Patch(ctx, update, patch); err != nil {
24         return fmt.Errorf("error updating status with CIDRs: %w", err
       )
25     }
26
27     return nil
28 }
29
30 func (r *ForeignClusterConnectionReconciler)
       retrieveCIDRInfoFromFactory(
31     ctx context.Context,
32     factory factory.Factory,
33     remoteClusterName string,
34 ) (networkingv1beta1.ClusterNetworkingStatus, error) {
35     var result networkingv1beta1.ClusterNetworkingStatus
```

```
36      tenantNs := fmt.Sprintf("liqo-tenant-%s", remoteClusterName)
37      name := fmt.Sprintf("%s-pod", remoteClusterName)
38
39      c, err := client.New(factory.RESTConfig, client.Options{Scheme: r
        .Scheme})
40      if err != nil {
41          return result, fmt.Errorf("error creating client from factory
        : %w", err)
42      }
43
44      var netCfg ipamv1alpha1.Network
45      if err := c.Get(ctx, client.ObjectKey{Namespace: tenantNs, Name:
        name}, &netCfg); err != nil {
46          return result, fmt.Errorf("error retrieving Network CR in
        namespace %q: %w", tenantNs, err)
47      }
48
49      result.PodCIDR = string(netCfg.Spec.CIDR)
50      result.RemappedPodCIDR = string(netCfg.Status.CIDR)
51      return result, nil
52  }
```

Explanation:

- `populateCIDRsFromNetworkConfig` coordinates the retrieval of CIDR information from both clusters and patches it into the `ForeignClusterConnection` resource's status.
- `retrieveCIDRInfoFromFactory` creates a Kubernetes client using the factory's configuration and reads the `Network` CustomResource in the corresponding tenant namespace.
- From the `Network` resource, it extracts both the original Pod CIDR and the remapped CIDR assigned for communication over the tunnel.
- These values are stored in the status fields `ForeignClusterANetworking` and `ForeignClusterBNetworking`, making them visible via standard commands like `kubectl get fcc -o yaml`.

Overall, recording these CIDR values completes the connection setup phase by ensuring that the `status` subresource consistently reflects the actual Pod network mappings between clusters. This supports effective monitoring and facilitates troubleshooting through standard Kubernetes tooling.

5.3.5 Disconnection Logic

When a `ForeignClusterConnection` is deleted (or a new connection attempt fails partway through), the operator must ensure that any established tunnel is correctly torn down. This is handled by the `disconnectLiqoctl` method:

Listing 5.12: `disconnectLiqoctl` implementation

```
func (r *ForeignClusterConnectionReconciler) disconnectLiqoctl(
    ctx context.Context,
    connection *networkingv1beta1.ForeignClusterConnection,
) error {
    logger := log.FromContext(ctx)
    logger.Info("Starting disconnection", "name", connection.Name)

    kubeconfigA, err := r.getKubeconfigFromLiqo(ctx, connection.Spec.ForeignClusterA)
    if err != nil {
        return err
    }
    defer os.Remove(kubeconfigA)

    kubeconfigB, err := r.getKubeconfigFromLiqo(ctx, connection.Spec.ForeignClusterB)
    if err != nil {
        return err
    }
    defer os.Remove(kubeconfigB)

    ctx, cancel := context.WithTimeout(ctx, 30*time.Second)
    defer cancel()

    os.Setenv("KUBECONFIG", kubeconfigA)
    localFactory := factory.NewForLocal()
    if err := localFactory.Initialize(); err != nil {
        return fmt.Errorf("error initializing localFactory: %v", err)
    }

    os.Setenv("KUBECONFIG", kubeconfigB)
    remoteFactory := factory.NewForRemote()
    if err := remoteFactory.Initialize(); err != nil {
        return fmt.Errorf("error initializing remoteFactory: %v", err
    )
    }

    localFactory.Namespace = fmt.Sprintf("liqo-tenant-%s", connection
    .Spec.ForeignClusterB)
    remoteFactory.Namespace = fmt.Sprintf("liqo-tenant-%s",
    connection.Spec.ForeignClusterA)

```

```
38      opts := network.NewOptions(localFactory)
39      opts.RemoteFactory = remoteFactory
40      opts.Timeout = 120 * time.Second
41      opts.Wait = true
42
43      localFactory.Printer = output.NewLocalPrinter(true, true)
44      remoteFactory.Printer = output.NewRemotePrinter(true, true)
45
46      fmt.Println("Executing 'network reset'...")
47      if err := opts.RunReset(ctx); err != nil {
48          return fmt.Errorf("error during 'network reset': %v", err)
49      }
50
51      fmt.Println("Operation 'network reset' completed successfully.")
52      return nil
53  }
```

Explanation:

- The method reuses `getKubeconfigFromLiqo` to obtain the kubeconfigs for both clusters, ensuring authenticated access to their API servers.
- The `Namespace` fields on the factories are explicitly set to the tenant namespace of the *opposite* cluster. This ensures that `RunReset` operates within the correct scope and targets only the resources associated with this specific connection.
- Invoking `RunReset` triggers the deletion of all CustomResources related to the established tunnel (such as `GatewayServer`, `GatewayClient`, and other supporting objects). Liqo's controllers then reconcile these deletions by tearing down the WireGuard tunnel and cleaning up any associated configuration automatically.

This disconnection flow guarantees that resources created for the tunnel are fully removed when the connection is no longer needed, preventing stale state and maintaining the integrity of the system.

5.4 Build & Deployment

This section describes how to build, package, deploy, and operate the controller to manage cross-cluster connections using the `ForeignClusterConnection` resource.

5.4.1 Build Process

The provided `Makefile` defines standard targets to support development and deployment workflows:

- `make build`: Compiles the operator into a local binary for testing.
- `make docker-build`: Builds a Docker image containing the compiled operator.
- `make docker-push`: Uploads the built image to a container registry.
- `make install`: Installs the CRDs and RBAC resources into the cluster.
- `make run`: Runs the operator locally for development.

5.4.2 Deployment

After installing the CRDs and RBAC rules, the operator can be deployed in-cluster using the provided manifests:

- Update the image reference in `config/manager/manager.yaml` if needed.
- Deploy the controller with:

```
kubectl apply -f config/manager/manager.yaml

```

This creates a Deployment in the chosen namespace. The operator Pod will run with the necessary permissions to watch and reconcile `ForeignClusterConnection` resources.

5.4.3 Applying a Sample Resource

Below is an example `ForeignClusterConnection` manifest to connect two Liqo-enabled clusters named `europe-rome-edge` and `europe-milan-edge`:

Listing 5.13: Sample `ForeignClusterConnection` Resource

```
apiVersion: networking.liqo.io/v1beta1
kind: ForeignClusterConnection
metadata:
  name: europe-rome-edge-europe-milan-edge
  namespace: default
spec:
  foreignClusterA: europe-rome-edge
  foreignClusterB: europe-milan-edge
  networking:
    mtu: 1450
    timeoutSeconds: 120
    wait: true
```

```
    serverGatewayType: networking.liqo.io/v1beta1/
    wggatewayservertemplates
    serverTemplateName: wireguard-server
    serverTemplateNamespace: liqo
    serverServiceType: NodePort
    serverServicePort: 51840
    clientGatewayType: networking.liqo.io/v1beta1/
    wggatewayclienttemplates
    clientTemplateName: wireguard-client
    clientTemplateNamespace: liqo
```

Key fields:

- `foreignClusterA` and `foreignClusterB` specify the clusters to connect.
- The `networking` section configures tunnel parameters, including gateway templates, service type, and MTU.

The resource can be applied with:

```
kubectl apply -f sample-fcc.yaml
```

Controller reaction to creation Upon creation of a `ForeignClusterConnection`, the controller executes the following steps:

- Adds a finalizer to ensure proper cleanup on deletion.
- Initializes the status phase to `Pending`, then transitions it to `Connecting`.
- Retrieves kubeconfigs for both clusters from Secrets.
- Initializes factory clients to interact with each cluster.
- Assembles network options from the CR spec.
- Invokes `RunConnect`, creating the `GatewayServer` and `GatewayClient` resources in the respective tenant namespaces.
- Retrieves Pod CIDR and remapped CIDR information from the `Network` resources.
- Updates the CR status to `Connected` with `isConnected=true`.

Example: resulting CR with status After successful reconciliation, the `ForeignClusterConnection` resource will have a populated `status` block, such as:

Listing 5.14: Sample `ForeignClusterConnection` with status

```
apiVersion: networking.liqo.io/v1beta1
kind: ForeignClusterConnection
metadata:
  name: europe-rome-edge-europe-milan-edge
  namespace: default
spec:
  foreignClusterA: europe-rome-edge
  foreignClusterB: europe-milan-edge
  networking:
    mtu: 1450
    ...
status:
  phase: Connected
  isConnected: true
  errorMessage: ""
  lastUpdated: "2025-06-24T15:12:05Z"
  foreignClusterANetworking:
    podCIDR: 10.200.0.0/16
    remappedPodCIDR: 10.61.0.0/16
  foreignClusterBNetworking:
    podCIDR: 10.200.0.0/16
    remappedPodCIDR: 10.63.0.0/16
```

This status provides:

- The current connection phase and health (`Connected`, `isConnected=true`).
- Any error messages (empty in a healthy state).
- The last update timestamp.
- The original and remapped Pod CIDRs for both clusters.

This allows the connection state to be inspected at any time using standard `kubectl` commands.

Controller reaction to deletion When a `ForeignClusterConnection` is no longer needed, it can be deleted with:

```
kubectl delete fcc europe-rome-edge-europe-milan-edge
```

Upon deletion:

- The finalizer prevents immediate garbage collection, ensuring that cleanup completes safely.
- The controller invokes `RunReset`, which removes all associated custom resources (including `GatewayServer`, `GatewayClient`, and other related objects) in both clusters.
- After successful cleanup, the finalizer is removed, allowing Kubernetes to fully delete the resource.

This lifecycle ensures that cross-cluster connections are declaratively created, monitored, and fully cleaned up, with the `status` subresource reflecting the state throughout their lifecycle.

Chapter 6

Command-Line Utility for Managing the Controller and Shortcuts

Managing Kubernetes-native resources such as the `ForeignClusterConnection` Custom Resource directly via `kubectl` is often verbose and error-prone, particularly when applied across multiple clusters. To streamline day-to-day operations and accelerate developer workflows, the `cli-liqo-shortcut` utility introduces a dedicated command-line interface for managing both the lifecycle of the `foreign_cluster_connector` controller and the setup of direct inter-cluster tunnels.

Rather than relying on lengthy `kubectl apply` commands and hand-written YAML, the CLI encapsulates common patterns into composable subcommands with clear syntax, validation, and feedback. This abstraction improves consistency across environments while maintaining compatibility with Kubernetes primitives under the hood.

All commands are intended to be executed from the consumer cluster, which acts as the coordination point for controller deployment and shortcut management.

The full source code is available at the official repository [11].

6.1 Command Structure and Design Goals

The CLI is organized into two primary command groups:

- **`manager`** — Deploys and removes the controller from a cluster.

- **`shortcuts`** — Creates, deletes, and inspects `ForeignClusterConnection` resources.

This separation adheres to the principle of separation of concerns: the controller logic remains decoupled from the shortcut orchestration.

Internally, each command is designed to adhere to the following principles:

- **Simplicity**: All commands are single-purpose and explicitly named (e.g., `create`, `list`).
- **Idempotence**: Repeated invocations are safe and converge to the same state.
- **Validation**: Parameters such as cluster identifiers are checked client-side.
- **Extensibility**: Default values can be overridden via flags, enabling script integration.

6.2 Controller Lifecycle Commands

The `manager` group exposes two commands to install or remove the controller in the target cluster.

- `liqoshortcut manager install`
  - Installs the `foreign_cluster_connector` controller and associated CRDs, RoleBindings, and Deployments.
- `liqoshortcut manager uninstall`
  - Cleans up all installed components, including CRDs and RBAC resources.
  - Ensures that existing CRs are deleted gracefully to avoid orphaned finalizers.

Example

```
$ liqoshortcut manager install
$ liqoshortcut manager uninstall
```

6.3 Shortcut Management Commands

The `shortcuts` command group enables creation, listing, and deletion of `ForeignClusterConnection` CRs, referred to as "shortcuts". These are used to establish peer-to-peer tunnels between provider clusters.

Each shortcut is uniquely parameterized by a pair of provider identifiers, specified using the `-a` and `-b` flags.

- `liqoshortcut shortcuts create -a <foreignClusterA> -b <foreignClusterB>`
  - Creates a new `ForeignClusterConnection` CR in the consumer cluster.
  - Automatically sets the required fields based on internal templates and naming rules.
- `liqoshortcut shortcuts delete -a <foreignClusterA> -b <foreignClusterB>`
  - Removes the corresponding CR and triggers teardown of the tunnel.
- `liqoshortcut shortcuts list`
  - Displays all shortcuts currently present in the cluster, along with their phase, status, and remapped CIDRs.

Example Workflow

```
# Establish a direct tunnel between Rome and Milan clusters
$ liqoshortcut shortcuts create -a europe-rome-edge -b europe-milan-edge

# Inspect active connections
$ liqoshortcut shortcuts list

# Remove the shortcut
$ liqoshortcut shortcuts delete -a europe-rome-edge -b europe-milan-edge
```

6.4 Auxiliary Logic and Operational Guarantees

Although the CLI interacts exclusively with Kubernetes-native APIs, it bundles several internal helpers to enhance robustness:

- Waits for controller resources to reach `Ready` before returning success.

- Patches required RBAC permissions into the controller's service account at install time.
- Validates the shortcut structure to prevent malformed CRs.
- Normalizes cluster naming to ensure consistency across commands.

Each command reuses type definitions from the controller project to guarantee schema compatibility and forward-compatibility. This encapsulation ensures predictable behavior and aligns operational safety with Kubernetes best practices.

6.5 Practical Benefits and Recommended Usage

- **Lower Entry Barrier**: Users without prior experience with Kubernetes CRDs or YAML can establish tunnels through a concise CLI interface.
- **Reduces Errors**: Eliminates the need to manually construct YAML manifests.
- **Encourages GitOps**: CLI invocations can be wrapped in scripts or Makefiles to support repeatable deployments.
- **Integrates with Liqo**: The tool assumes Liqo's naming conventions and Secret placement, making it a natural fit for existing deployments.

Chapter 7

Evaluation

This chapter presents a comparative analysis of indirect (consumer-mediated) and direct (provider-to-provider) overlay tunnels, under both zero-latency and emulated WAN conditions. Metrics include ICMP round-trip time (RTT), HTTP GET latency, TCP throughput and congestion-window behavior, and tunnel provisioning time. Each subsection groups related charts with detailed captions to guide the reader through the results.

7.1 Testbed Description

All clusters are instantiated as Docker containers on a single physical host. This local setup ensures reproducibility and full control over the environment, but lacks intrinsic network delay or packet loss. Consequently, explicit emulation is required to simulate realistic WAN conditions and assess the impact of routing strategies under stress. In this context, the "zero-latency" configuration serves as a reference point representing the ideal, interference-free baseline.

A central cluster (`europe-cloud`) peers with two edge clusters (`europe-rome` and `europe-milan`). In each edge cluster, a single-replica `nginx` Deployment runs in a dedicated namespace offloaded via Liqo. All tests originate from a client `nginx` pod on `europe-rome`, which issues HTTP requests targeting the server `nginx` pod on `europe-milan`. Two routing modes are compared:

- **Indirect mode:** traffic traverses the consumer cluster.
- **Direct mode:** WireGuard tunnel connects providers end-to-end.

Each routing mode is evaluated under two network scenarios:

- *Zero-latency*: a reference configuration where the testbed operates without any injected delay or packet loss, serving as an ideal baseline for comparison.

- *Emulated WAN conditions*: a delay of 30 ms with a 5 ms variance and 0.01 % packet loss uniformly applied to the WireGuard (`liqo-tunnel`) interfaces of all gateway pods.

WAN conditions are emulated using the Linux `tc` utility, which applies delay and loss to each gateway pod's `liqo-tunnel` interface. The following command configures the desired parameters:

```
1 tc qdisc add dev liqo-tunnel root netem delay 30ms 5ms loss 0.01%
```

Here, `liqo-tunnel` refers to the virtual WireGuard interface used for inter-cluster communication. The `netem` queuing discipline provides reproducible WAN-like behavior by introducing latency and controlled packet loss into the overlay path. This setup enables controlled stress testing in an otherwise delay-free local environment, consistent with established practices in Kubernetes-based network experimentation.

7.1.1 ICMP Round-Trip Time

Under the zero-latency configuration, direct provider-to-provider tunneling already delivers substantial latency improvements over the default consumer-mediated path. By eliminating the extra forwarding hop through the consumer cluster, the direct tunnel shortens the data path and reduces processing overhead at each relay.

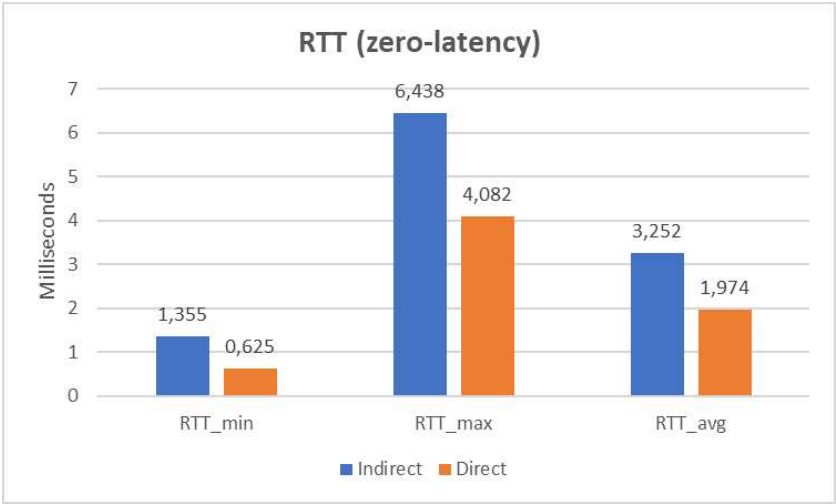

Figure 7.1: Comparison of 100-probe RTT under zero-latency conditions (indirect vs direct).

Figure 7.1 highlights three key benefits of the direct mode:

- **Lower base latency:** Removing the intermediary cuts the minimum RTT by more than 50 %, accelerating the initial packet exchange and improving responsiveness for control-plane traffic.
- **Reduced average latency:** The mean RTT falls by nearly 40 %, which translates directly into faster round-trip interactions for TCP handshakes, DNS lookups, and short-lived HTTP requests.
- **Tighter latency distribution:** Jitter (max–min RTT) shrinks by roughly one-third, yielding a more predictable timing envelope that benefits real-time and streaming applications.

These latency gains under ideal conditions underline the efficiency of a provider-to-provider overlay, setting a performance baseline that is further amplified when network delays or losses are introduced.

7.1.2 HTTP GET Latency

Under zero-latency conditions, the elimination of the intermediary hop yields lower application-layer delays.

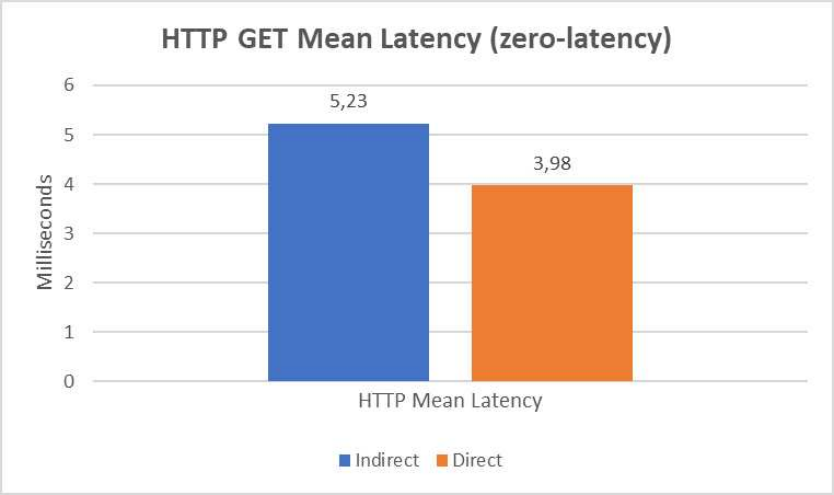

Figure 7.2: Mean HTTP GET latency under zero-latency conditions, comparing indirect (5.23 ms) and direct (3.98 ms) routing.

Figure 7.2 shows:

- **Mean latency reduction:** direct mode decreases average GET time by 24 %, from 5.23 ms to 3.98 ms.

- **Reduced variability:** the direct tunnel exhibits a more compact range of response times, indicating fewer high-latency samples.
- **Enhanced consistency:** lower and more uniform request durations support improved performance for time-sensitive applications.

7.1.3 TCP Throughput

The comparison of sustained TCP transfer rates under zero-latency conditions reveals a clear advantage for direct provider-to-provider tunnels.

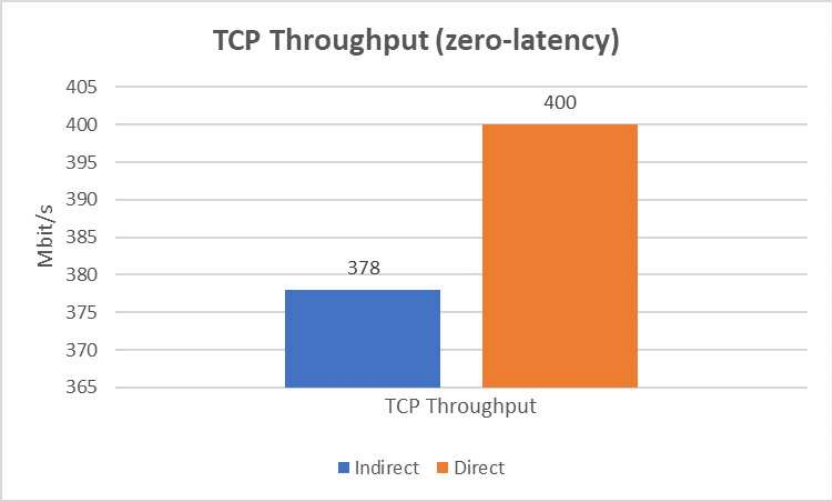

Figure 7.3: Average TCP throughput under zero-latency conditions: **Indirect mode:** 378 Mbit s^{-1}; **Direct mode:** 400 Mbit s^{-1} (6 % improvement).

As shown in Figure 7.3, direct routing increases average throughput from 378 Mbit s^{-1} to 400 Mbit s^{-1}, yielding an improvement of approximately 6 %. This uplift is achieved by removing the intermediary forwarding hop in the consumer cluster, thereby eliminating per-packet processing delays and local queue buildup. The resulting path reduction enables TCP to attain a higher steady-state sending rate, improving bulk transfer efficiency and optimizing utilization of the available network capacity.

7.2 Results under Emulated WAN Conditions

Under emulated WAN conditions (with 30 ms delay, 5 ms variance, and 0.01 % packet loss per gateway), the direct provider-to-provider tunnel consistently outperforms the indirect mode across all measured metrics. RTT minimum, average, and

maximum values are roughly halved, demonstrating consistently lower end-to-end delays and moderately reduced jitter.HTTP GET latency is similarly reduced by nearly 50 %, enabling more efficient and predictable request handling. Despite increased delay and loss, TCP throughput remains substantially higher and more stable, highlighting the robustness of direct tunneling under adverse conditions.

7.2.1 ICMP Round-Trip Time under WAN-like Delay

Figure 7.4 presents the distribution of 100 ICMP probes in both routing modes. In indirect mode, the minimum, average and maximum RTTs are 113.005 ms, 124.719 ms and 139.888 ms , respectively. Direct mode lowers these values to 54.876 ms, 63.098 ms and 71.984 ms.

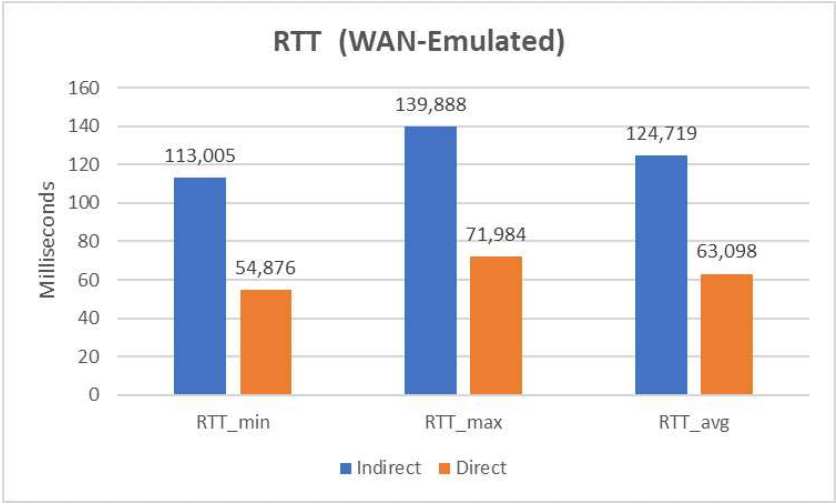

Figure 7.4: Comparison of 100-probe RTT under emulated WAN conditions (indirect vs direct).

The direct tunnel halves the base latency (−51 %) and reduces the peak RTT by over 67 ms (−48 %). Notably, the slowest direct-mode probe (71.984 ms) completes faster than the fastest indirect-mode probe (113.005 ms), removing the intermediary hop's impact on worst-case latency and delivering a consistently lower and tighter latency envelope.

7.2.2 HTTP GET Latency under WAN-like Delay

Application-level request times exhibit even greater divergence when per-hop delay and loss are introduced. The inherent efficiency of a direct provider-to-provider tunnel minimizes round-trip handshake overhead and reduces the probability of retransmission at the HTTP layer.

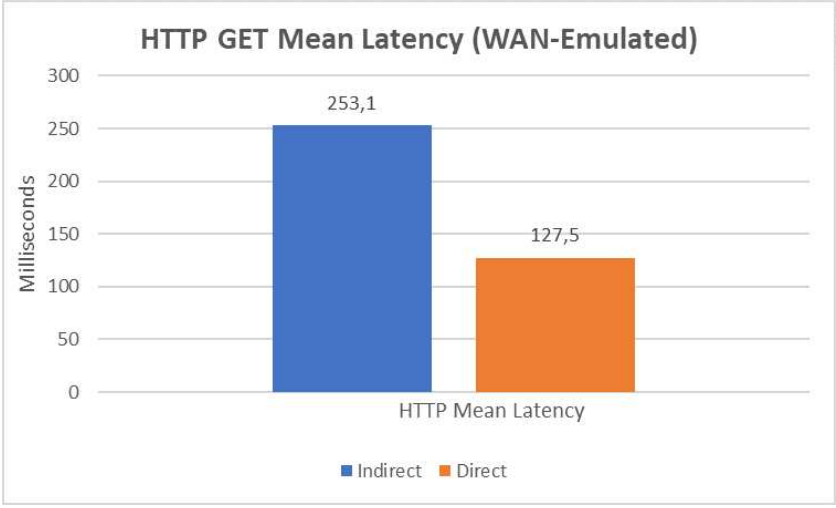

Figure 7.5: Mean HTTP GET latency under emulated WAN conditions (indirect vs direct).

Under the indirect routing path, the average HTTP GET completes in approximately 252 ms. By contrast, direct tunneling achieves a mean of 127 ms a 50 % reduction. This halving of latency not only accelerates page load and API responses but also constricts the tail of the latency distribution, yielding a more deterministic user-perceived performance. Such predictability is critical for time-sensitive applications and high-frequency request patterns.

7.2.3 TCP Throughput under Emulated WAN Conditions

As expected, TCP performance degrades markedly in WAN-like conditions due to added per-hop delay and loss. The indirect mode achieves approximately 50 Mbit s^{-1}, whereas the direct tunnel maintains 243 Mbit s^{-1}, demonstrating substantially improved utilization of the available path.

Under the indirect routing path, TCP throughput falls below 50 Mbit s^{-1}, reflecting repeated retransmissions and restricted congestion-avoidance growth. In contrast, the direct tunnel preserves a high, stable rate of 243 Mbit s^{-1}—an approximate 380 % increase—indicating more efficient loss recovery and sustained window expansion despite added per-packet latency and minor loss. Such throughput resilience is critical for bulk data transfers over multi-region overlays.

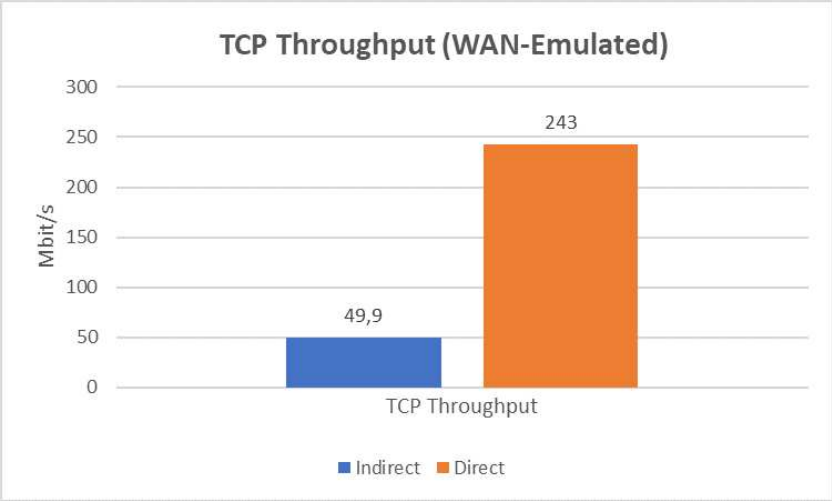

Figure 7.6: Average TCP throughput under emulated WAN conditions (indirect vs direct).

7.3 Tunnel Provisioning Time

Provisioning duration for a direct provider-to-provider tunnel was evaluated under two workflows: manual configuration and controller-driven automation. Manual setup requires explicit delivery of each provider's kubeconfig and endpoint details. In contrast, the automated approach leverages the `ForeignClusterConnection` controller: only the identifiers of the two foreign clusters are specified, the necessary kubeconfig credentials are fetched programmatically, and the central control plane is informed of the new direct tunnel.

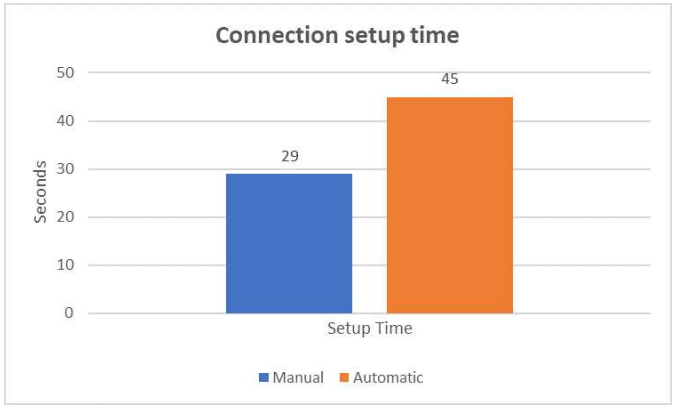

Figure 7.7: Comparison of direct-tunnel provisioning times: manual setup (29 s) vs. controller-driven setup (45 s).

Automated provisioning exhibits a 55 % increase in elapsed time (from 29 s to 45 s), equivalent to a 16 s overhead. This modest penalty is offset by the elimination of manual credential management, the enforcement of declarative and repeatable deployment, and the central cluster's real-time awareness of the direct connectivity—capabilities not achievable with purely manual configuration.

7.4 Evaluation Summary

Direct provider-to-provider tunnels deliver clear operational and performance advantages:

- **Elimination of the intermediate hop:** traffic follows a single WireGuard path without relaying through the consumer cluster, reducing queuing and simplifying routing.
- **Consistent latency improvements:** tighter RTT distributions ensure predictable response times in both ideal and WAN-like scenarios.
- **Better application responsiveness:** HTTP request latency drops significantly under all conditions, benefiting user-facing services.
- **Superior bulk-transfer performance:** more stable TCP windows and reduced retransmissions enable much higher throughput in adverse networks.
- **Streamlined, automated provisioning:** controller-based setup eliminates manual credential exchange, offering repeatability and central oversight despite a slight increase in setup time.

These results demonstrate that controller-managed direct tunnels are an effective approach for multi-site Kubernetes deployments requiring reliable performance and simplified operations. Future work will extend this analysis to dynamic network configurations, larger topologies, and encryption overhead.

Chapter 8

Conclusions and Future Work

This thesis examined a key limitation in current Liqo-based multi-cluster deployments: the lack of direct, optimized communication paths between provider clusters in peer-to-peer topologies. By default, Liqo routes all inter-cluster traffic through the consumer cluster, introducing higher latency, increased egress traffic, and a potential single point of failure.

To address these challenges, a new approach was designed and implemented based on a new Kubernetes resource: the `ForeignClusterConnection` custom resource definition (CRD). This resource enables declarative management of provider-to-provider tunnels, with full lifecycle automation handled by a dedicated Kubernetes Operator. The solution integrates with Liqo's architecture and leverages its dynamic discovery and remapping mechanisms, enabling clusters to advertise direct-tunnel CIDRs while maintaining consistency within the federated control model.

The controller, built using the Kubebuilder framework, demonstrates the feasibility and scalability of this approach. It eliminates the need for manual network configuration, supports automated status reporting through CRD status fields, and provides a foundation for centrally managed, policy-driven tunnel optimization in the future.

Experimental evaluation confirms significant benefits in latency reduction and resilience under WAN-like conditions. By removing dependence on the consumer cluster for inter-provider traffic, the solution reduces unnecessary network hops and improves robustness against consumer cluster failures.

8.1 Future Work

Several promising directions can extend this work:

- **EndpointSlice Rewriting:** Develop a dedicated controller to automatically rewrite `EndpointSlices` in provider clusters, advertising direct-tunnel CIDRs when a valid `ForeignClusterConnection` exists. This would enable optimal routing without manual annotations or auxiliary configurations.
- **Policy-Driven Routing Logic:** Integrate SLA-aware policies—such as latency thresholds, bandwidth quotas, or availability metrics—into the controller to support intelligent routing decisions when multiple paths are available. Policies could dynamically activate or deactivate tunnels based on real-time performance data.
- **Support for Mesh Topologies:** Extend the current design, focused on pairwise links, to support full mesh topologies. Automatic management of transit routes (e.g., Rome → Milan → Berlin) could further improve performance in geographically distributed deployments.
- **Security and Trust Expansion:** Enhance the existing model, which relies on Liqo's built-in authentication and encryption, by integrating with external certificate authorities or service meshes to enable advanced trust models, observability, and auditability.
- **GitOps and UI Integration:** Leverage the declarative nature of the solution to integrate it into GitOps workflows. Future enhancements could include dashboards or visual tools for tunnel monitoring and management, providing metrics, health status, and topology visualization.
- **Compatibility with Non-Liqo Clusters:** Investigate adaptation for non-Liqo Kubernetes clusters through an adapter or network gateway, broadening applicability to hybrid and heterogeneous environments.

Overall, this work lays the foundation for a more efficient, resilient, and extensible model of multi-cluster communication in Kubernetes environments. By moving from a consumer-centric routing paradigm to a declarative, peer-aware tunnel orchestration model, it enhances both the scalability and reliability of Liqo-based infrastructures.

Bibliography

[1] Kubernetes Authors. *Kubernetes Documentation*. Accessed May 2025. 2025. URL: https://kubernetes.io/docs/home/ (cit. on p. 3).

[2] Kubernetes Authors. *Kubernetes Concepts: Cluster Architecture*. Accessed May 2025. 2025. URL: https://kubernetes.io/docs/concepts/architecture/ (cit. on p. 4).

[3] Kubernetes SIG API Machinery. *Kubebuilder Book*. Accessed May 2025. Kubernetes Project, 2025. URL: https://book.kubebuilder.io/ (cit. on p. 12).

[4] Liqo Authors. *What is Liqo?* Accessed May 2025. 2025. URL: https://docs.liqo.io/en/stable/ (cit. on p. 13).

[5] Liqo Authors. *Peering*. Accessed May 2025. 2025. URL: https://docs.liqo.io/en/stable/features/peering.html (cit. on p. 14).

[6] Liqo Authors. *Network Fabric*. Accessed May 2025. 2025. URL: https://docs.liqo.io/en/stable/features/network-fabric.html (cit. on p. 16).

[7] Jason A. Donenfeld. *WireGuard: fast, modern, secure VPN tunnel*. Accessed May 2025. 2022. URL: https://www.wireguard.com/ (cit. on p. 17).

[8] Internet Engineering Task Force. *Generic Network Virtualization Encapsulation (Geneve), RFC 8926*. Accessed May 2025. 2020. URL: https://datatracker.ietf.org/doc/html/rfc8926 (cit. on p. 17).

[9] Liqo Authors. *Offloading*. Accessed May 2025. 2025. URL: https://docs.liqo.io/en/stable/features/offloading.html (cit. on p. 20).

[10] Santo Calderone. *foreign_cluster_connector*. Accessed June 2025. 2025. URL: https://github.com/scal110/foreign_cluster_connector.git (cit. on p. 31).

[11] Santo Calderone. *CLI-liqo-shortcut*. Accessed June 2025. 2025. URL: https://github.com/scal110/cli-liqo-shortcut.git (cit. on p. 55).