POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Scalable network overlay to support direct communications within Kubernetes multi-cluster topologies

Supervisors

Prof. Fulvio RISSO

Dott. Marco IORIO

Candidate

Luca FRANCESCATO

Academic year 2021-2022

Summary

The growth of the Internet knows no rest, and brings with it the necessity to deliver services at unprecedented orders of magnitude. The cloud has become the foundation upon which services can be delivered and managed at scale, along with engineering applications to be cloud-native. In this context, Kubernetes has been the driving force behind the successful management, deployment and delivery of large-scale services. With Kubernetes, computing nodes are brought together into clusters, where organizations have full visibility over the overall resources at their disposal, for an effective utilization of the deployed infrastructure. Moreover, companies may need to control multiple clusters to ensure their presence in multiple regions, increase availability, scale better in terms of maintenance, and may rely on different cloud providers for cost management or vendor lock-in avoidance.

To take full advantage of a multi-cluster infrastructure, clusters may be connected together to create a single yet flexible environment where the deployment of heavy workloads can leverage the aggregate resources that clusters provide together. Liqo, an open-source project started at Politecnico di Torino, embraces this idea and makes it a reality. Liqo makes it possible to deploy an application consisting of several micro-services to different clusters. To preserve the overall functioning of the application, the different application's components have to communicate with one another, from one cluster to the other. This was achieved by creating a hub-and-spoke topology, where traffic passed through a central hub cluster.

This thesis presents the design and implementation of a solution whereby workloads deployed in a multi-cluster scenario can directly exchange traffic. With this approach, clusters connected to a central cluster are able to discover one another and exchange the network parameters required for direct communications, using the central cluster as a relay. Once this initial setup completes, traffic starts flowing directly, without traversing the central cluster, which is freed from networking overhead and overloads the previous topology implied, reducing the latency in communications and making them immune to temporary network outages on the central cluster. This results in a mesh, peer-to-peer topology and was made possible by evolving the networking and service reflection logic at the core of Liqo.

Table of Contents

List of Figures

Chapter 1

Introduction

Recent years have witnessed the rise of cloud-native solutions to handle the massive amount of requests that big companies deal with on a daily basis to deliver their services to millions of users. When developing software, this approach wins over the traditional monolithic software architecture: the idea is to develop applications as composed of many small highly-cohesive and loosely-coupled parts, whose life-cycle can be easily and independently managed to accomplish the goal of the overall application. The delivery of such small components meets an unprecedented ease of management when associated with the containerization techniques and the capabilities offered by container orchestrators that have spread across the software industry.

Among those orchestrating systems, Kubernetes has earned a primary role in the cloud scene. Thanks to its features, ease of use and powerful declarative API, it enables users to effectively handle the high dynamism that characterizes the modern working loads. Its value resides in these tools that are at the disposal of developers, as well as in its open-source nature. All these elements contributed to its spread, but even more importantly to the growth of the cloud community as a whole. Indeed, cloud-native principles are no longer a prerogative of big companies, but are becoming more and more used by small and medium sized firms.

As a result, clusters are being used in many aspects of the software industry, and with this comes the necessity to interconnect them to fully exploit their capabilities.

1.1 Introducing Liqo

In this scenario, Liqo [1] takes its place and aims to contribute to enhance the cloud-native experience even more and open the gates to new possibilities. Liqo is an open-source project that enables dynamic and seamless Kubernetes multi-cluster topologies, supporting heterogeneous on-premise, cloud and edge infrastructures.

It establishes itself over the Kubernetes API, and expands it to make different clusters combine into a multi-cluster network of computing nodes.

Liqo provides automatic peer-to-peer establishment of relationships to share and consume resources and services between independent and heterogeneous clusters. Once the peering is established, a cluster can seamlessly offload workloads to its remote peer, without requiring any modifications to Kubernetes or the applications. Liqo makes multi-cluster native and transparent: remote clusters are simply seen as nodes that add up to the other available nodes of the local cluster, thus being compliant with the standard Kubernetes approaches and tools. To make remote pods be able to communicate, Liqo provides a network fabric that enables multi-cluster pod-to-pod connectivity.

1.2 Goal of the thesis

This thesis stems from the analysis of how Liqo currently implements the management multi-cluster deployments. In particular, when a cluster creates a peering session with multiple remote clusters and then offloads to them its own workloads, pods that are scheduled to those remote clusters might need to communicate. In this case, their communications cannot flow directly from one remote cluster to the other, but need to be forwarded to the central cluster from which the workloads were offloaded, and after traversing it, they can land on the remote cluster. The goal of this thesis is to propose a design of a network overlay that is automatically configured and deployed across those remote clusters, providing their pods the ability to communicate directly without relying on the central cluster and therefore cutting down the overhead introduced by this additional intermediate step, and even reducing the latency of pods communications.

The result is a peer-to-peer, full mesh network of clusters that enables direct pod-to-pod traffic flows. To achieve it, several updates were required on different core elements within Liqo, with particular focus on the logic that exchanges the network parameters and sets up the network connections between clusters, the proper reflection of services and endpoints, and the correct translation of IP addresses from one address range to the other to avoid address conflicts.

1.3 Structure of the work

This thesis will unfold with the following structure:

- **Chapter 2** provides an overview on Kubernetes, the technology at the root of Liqo for automatic deployment, scaling, and management of containerized applications.

- **Chapter 3** provides an overview on Liqo, including a detailed description of its core concepts and main components.
- **Chapter 4** provides a thorough analysis on the different multi-cluster topologies handled by Liqo and how they affect the remote offloading of workloads.
- **Chapter 5** provides a conceptual description of the changes this work required to materialize the goal of this thesis.
- **Chapter 6** provides an in-depth study of the implementation details that together constitute the proposed solution.
- **Chapter 7** provides a review of the proposed solution in terms of functional and performance tests.
- **Chapter 8** concludes the presented work, summarizing the achieved results and taking a quick glance at some possible improvements that future works may provide.

Chapter 2

Kubernetes

This chapter introduces the Kubernetes technology, giving an overview of the evolution of the way applications are deployed and managed, from the early days of virtualization up to the latest methodologies of the present day.

2.1 What is Kubernetes

Kubernetes [2] is a platform for managing containerized workloads and services. It is based on a declarative configuration, meaning that developers declare intents and Kubernetes responds to these intents by applying the built-in logic that is shipped with.

Kubernetes is Greek for "helmsman" or "pilot". The name is often found in its short form, "K8s", which results from counting the eight letters between "K" and "s".

The Kubernetes project [3] has been developed by Google, and it was open-sourced in 2014. It combines over 15 years of Google's experience running production workloads at scale and it brings forward the best solutions and practices of the cloud computing world.

Kubernetes is already the de-facto standard in the DevOps community, and nowadays it undergoes a massive utilization in production environments across the entire world. In order to understand why Kubernetes has gained so much traction and has got a paramount role in solving cloud-native problems, we can start by analyzing what technologies traced the path to its ascension.

2.2 Evolution of workloads management

Traditional deployment era In the traditional deployment era, organizations ran applications on physical servers. There was no way to define application

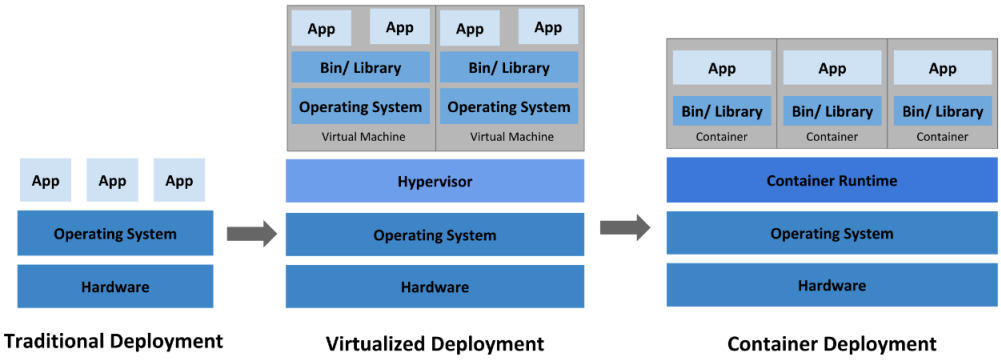

Figure 2.1: Evolution of applications deployments

constraints to limit resource usage, and some applications would end up taking most of the resources available, making the remaining applications starve. This led system managers to deploy one server per application, increasing costs and maintenance work. At this point, the community rediscovered the abandoned concept of virtualization.

Virtualized deployment era In the virtualized deployment era, developers could run multiple Virtual Machines (VMs) on a single physical server, and ensure applications would not interfere with one another, by running one VM per application. Virtualization allows defining resource-usage constraints for each VM, and makes software running on one VM isolated from the rest of the system and other VMs, leading to a much more stable and secure environment, as applications cannot interfere with one another, nor freely access private application data. Moreover, it allows better scalability as application instances can be scaled up or down easily by spawning or deleting VMs as needed. Each VM includes a whole operating system and can be tweaked to include the properly versioned dependencies as requested by the running application: this creates sealed compartments that are easy to manage and maintain, as well as to debug. Overall, less physical servers are deployed, costs are lower and companies can get the most out of their available servers, preventing them from being underused.

Containerized deployment era The next step in the evolution of workloads deployment came with the rise of containerization. Containers work similarly to VMs, but with less strict isolation properties so that different applications can share the same Operating System. For this, they are considered lightweight. Just like VMs, containers have their own filesystem, share of CPU, memory, process space, and more. Containers are decoupled from the underlying infrastructure: this makes them portable across clouds and OS distributions. What makes them so

popular is the set of extra benefits they provide, such as:

- The agile application creation and deployment, given the ease of creation of container images compared to VM images.
- Continuous development, integration and deployment, thanks to the reliable and frequent container image build and deployments.
- Application health checks and observability.
- Cloud and OS distribution portability.
- Application-centric management, raising the abstraction level in order to simply focus on running the application.
- Resource utilization that yields high efficiency and density.

In parallel to the sheer technological advancements, an improvement on the workload management methods has been observed: from handling VMs as single entities, we moved to a "cattle" model where VMs were handled in a more general way (although their management would still be quite coupled to their lives), to move further and reach a decoupled approach, that is the one used by Kubernetes: a declarative way that expresses general intentions that are taken by the system and applied to all of the interested resources, without having to deal with the single instances, resulting in a more detached view where resources are seen as commodities that can be created, destroyed, and replaced as needed.

2.3 Kubernetes concepts

2.3.1 Kubernetes objects

Kubernetes objects are persistent entities in a cluster. Kubernetes uses these entities to represent the state of a cluster. Specifically, they describe:

- What containerized applications are running and on which nodes.
- The resources available to those applications.
- The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance.

A Kubernetes object is a "record of intent": once an object is created with its specifications, the Kubernetes system will constantly work to ensure that object exists and its specifications are met. By creating an object, Kubernetes shapes the

cluster to make it compliant to the expressed workload: this is the cluster's desired state.

To work with Kubernetes objects—whether to create, modify, or delete them—a user needs to interact with the Kubernetes API. For example, it is possible to use the kubectl command-line interface, and the CLI will make the necessary API calls corresponding to the entered commands.

Almost every Kubernetes object includes two nested fields that govern the object's configuration: the object's "spec" and "status" fields. The former specifies the desired state and is required upon the object creation, while the latter tracks the current state of the object. The Kubernetes control plane continually manages every object's actual state to make it match the specified desired state.

When creating an object, a user most often provides the information in the form of a .yaml [1] file, as the one presented below for the Pod resource:

Listing 2.1: Kubernetes resource

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: example-pod
5 spec:
6   # user's desired state for this object
```

The kind of object to be created is defined with the *kind* field, while the *apiVerson* defines which version of the Kubernetes API is used to create the object. The *metadata* helps uniquely identify the object, including a name string. Finally, the *spec* field is the user's desired state for the object.

The following is a non-exhaustive list of the built-in Kubernetes objects that can be found inside an operating cluster.

Namespace

A namespace is a mechanism that provides a scope for names, enabling isolation of groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects, such as Deployments and Services, and not for cluster-wide objects, such as StorageClasses, Nodes and PersistentVolumes.

[1] YAML is a human-readable and easy to understand data serialization language that is often used for writing configuration files and is oriented to data. YAML stands for *YAML ain't markup language* (a recursive acronym).

Pod

The smallest deployable unit of computing that can be created and managed in Kubernetes. A Pod (as in a pod of whales or a pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod represents a logical host, because the containerized applications it contains are generally tightly coupled and work in harmony to provide a workload-specific service.

ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of Pod replicas running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

Deployment

A Deployment provides declarative updates for Pods and ReplicaSets. The Deployment contains the desired state, and the Deployment controller changes the actual state to the desired state at a controlled rate.

DaemonSet

A DaemonSet ensures that all or some Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are therefore added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created. A common use case for a DaemonSet is to have a running logs collection daemon on every node.

Service

A Service is an abstract way to expose an application running on a set of Pods as a network service. Since Pods can be destroyed and recreated at any time to match the cluster's desired state, their IP address can change. Services are used to abstract the IP tracking process by giving a single DNS name for a set of Pods.

A Service is associated through a set of Pods by specifying a selector that has to match the labels of such Pods. To make the Service aware of which container port to forward the incoming requests, it is required to specify the targetPort: the reason is that a Pod can host more than one container, each with a different port within the same Pod. In case more than one container needs to be accessible, more targetPort fields will be configured in the Service definition.

A single Service can match more than one Pod, in which event it will load-balance between them.

Services are used to access the final containers from within the cluster, but even more importantly from the outside. By specifying the serviceType, it is possible to have different behaviors in terms of Pod exposure:

- **ClusterIP Service** (default), only accessible from resources inside the cluster, it requires an Ingress in order to be reachable externally. It has an IP address that is unique within the cluster, and a DNS lookup of its name would result in its IP address.
- **Headless Service**, a special kind of ClusterIP Service that has no IP address assigned. The use case is when a Pod requests to communicate with a specific Pod, not a randomly chosen Pod in the set of Pods behind the Service. In this case, a DNS lookup of its name results in the IP address of the Pod it exposes.
- **NodePort Service**, accessible externally from a static port on each node in the cluster. In this case, external traffic can access directly the Service by specifying the Node IP and the static port. In the definition of the Service, the "port" attribute has to be specified, because a ClusterIP service will be automatically created and be routed traffic from the NodePort Service.
- **LoadBalancer Service**, accessible externally but only via the cloud provider's load balancer. Each cloud provider has its own native load balancer implementation that routes the traffic. What happens is that NodePort and ClusterIP Services are automatically created by K8s and the load balancer routes the traffic to them. This time, however, the port open on the nodes will not be directly accessible externally but only through the load balancer itself. So the entry point becomes the cloud provider's load balancer.

Endpoints and EndpointSlice

An Endpoints object represents the connection between Pods and Services, with the objec. It is used by K8s to keep track of Pod IP addresses that are the endpoints of the related Service. EndpointSlices provide a simple way to track network endpoints within a Kubernetes cluster, by grouping network endpoints together. EndpointSlices are an improvement over the original Endpoints API.

By creating a Service, K8s automatically creates an Endpoints object named after the service. They are used by K8s to keep track of Pod IP addresses and ports the Service has to route traffic to. Endpoints are an additional layer of data that binds Pods and Services.

Ingress

An Ingress manages external access to the services in a cluster, typically HTTP.

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

ConfigMap

A ConfigMap is an object used to store non-confidential data in key-value pairs. Pods consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume, that is a directory containing data, accessible to the containers in a pod. Data in ConfigMaps is set separately from application code: this allows decoupling environment-specific configurations from container images, so that applications can be easily portable.

ConfigMaps do not provide secrecy. Confidential data should be stored in Secrets.

Secret

A Secret is an object designed to contain a small amount of sensitive data, such as password, tokens, or keys. Such information might otherwise be put in a Pod specification or in a container image. Using a Secret means not including confidential data in the application code.

Considering their nature, Secrets are similar to ConfigMaps, but are specifically intended to hold confidential data.

2.3.2 Controllers

Kubernetes controllers stem from robotics and automation, where a control loop is a non-terminating process that regulates the state of an observed system. A Kubernetes controller is a control loop that watches the state of the deployed cluster, then makes or requests changes where needed. Each controller tries to move the current cluster state to the desired state.

A controller tracks at least one type of Kubernetes object. The controllers for that resource are responsible for making the current state come closer to the desired state expressed in the object's "spec" field.

Kubernetes comes with a set of built-in controllers that run inside the kube-controller-manager component.

2.3.3 Extending the Kubernetes API

Custom Resources

Custom resources (shortened as CRs) are extensions of the Kubernetes API. Once a custom resource is installed, users can create and access its objects using kubectl, just as they do for built-in resources like Pods.

On their own, custom resources let you store and retrieve structured data. When you combine a custom resource with a custom controller, custom resources provide a true declarative API.

The Kubernetes declarative API enforces a separation of responsibilities. You declare the desired state of your resource. The Kubernetes controller keeps the current state of Kubernetes objects in sync with your declared desired state. This is in contrast to an imperative API, where you instruct a server what to do.

You can deploy and update a custom controller on a running cluster, independently of the cluster's lifecycle. Custom controllers can work with any kind of resource, but they are especially effective when combined with custom resources.

Custom Resource Definitions

The CustomResourceDefinition API resource (shortened as CRD) allows defining custom resources. The Kubernetes API is programmed to serve and handle the storage of custom resources.

CRDs free developers from writing their own API server to handle the custom resource.

The Operator pattern

The Operator pattern combines custom resources and custom controllers. You can use custom controllers to encode domain knowledge for specific applications into an extension of the Kubernetes API.

Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. Operators follow Kubernetes principles, notably the control loop as described in Subsection 2.3.2.

2.4 Kubernetes components

Deploying Kubernetes means creating a Kubernetes cluster. A Kubernetes cluster is a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. The worker nodes host the application workloads. Among them, one or more are also designated to host the Kubernetes control plane. The control plane is the very core of the Kubernetes orchestration

logic, and constitutes the container orchestration layer that exposes the API and interfaces to define, deploy, and manage the lifecycle of containers. Usually, the control plane runs on multiple computers and a cluster runs multiple nodes, providing fault-tolerance and high availability.

2.4.1 Control plane components

The control plane components make global decisions about the cluster management, scheduling and general status, and respond to cluster events, such as scaling up an application when certain resource-usage thresholds are exceeded.

Control plane components can be run on any node in the cluster. For a matter of simplicity and ease of setup, those components are generally started up on the same node, which is entirely dedicated to them and won't run user-defined application workloads.

kube-apiserver

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane. The main implementation of a Kubernetes API server is kube-apiserver. It is designed to scale horizontally: one can run several instances of kube-apiserver and balance traffic between those instances.

etcd

This is a consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data.

kube-scheduler

This component watches for newly created Pods with no assigned node, and selects a node for them to run on. When making a scheduling decision, it considers the individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

kube-controller-manager

This component runs the main Kubernetes control logic. It embeds several controllers, but it is compiled as a single process. Each controller is a control loop that watches the shared state of the cluster through the API server and makes changes attempting to move the current state towards the desired state.

Some of its controllers are:

- **Node controller**, responsible for noticing and responding when nodes go down.
- **Job controller**, watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
- **Endpoints controller**, populates the Endpoints object (that is, joins Services and Pods).
- **Service Account and Token controllers**, create default accounts and API access tokens for new namespaces.

cloud-controller-manager

This component embeds cloud-specific control logic. It lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster. Its controllers are specific to the selected cloud provider. Therefore, if Kubernetes is ran on-premise or in a learning environment, like inside a PC, the cluster does not have a cloud controller manager.

2.4.2 Node components

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

kubelet

This component runs on each node in the cluster and makes sure that containers are running in a Pod. It takes a set of PodSpecs (provided through various mechanisms) and ensures that the containers described in those specifications are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

kube-proxy

This component is a network proxy that runs on each node in the cluster, implementing part of the Kubernetes Service concept. It maintains network rules on nodes. These network rules allow network communications to Pods from network sessions inside or outside of the cluster. Moreover, it uses the operating system packet filtering layer if there is one and it's available, otherwise forwards the traffic itself.

Container runtime

This is the software that is responsible for running containers. Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

2.4.3 Addons

Addons use Kubernetes resources to implement and provide cluster-level features.

DNS

All Kubernetes clusters should have a cluster DNS, which is a DNS server that serves DNS records for Kubernetes services and adds up to the other DNS servers that might be present in the environment. Containers started by Kubernetes automatically include this DNS server in their DNS searches.

In particular, Kubernetes DNS schedules a DNS Pod and Service on the cluster, and configures the kubelets to tell individual containers to use the DNS Service's IP to resolve DNS names. Services and Pods of a Kubernetes cluster are assigned a DNS record.. A DNS query may return different results based on the namespace of the pod making it. DNS queries that don't specify a namespace are limited to the pod's namespace. To access services in other namespaces, one needs to specify the namespace in the DNS query.

2.5 RBAC

Kubernetes defines several APIs for managing access permissions over resources. The Role-Based Access Control (shortened as RBAC) is a method of regulating access to compute or network resources based on the roles of individual users.

The RBAC API declares four kinds of Kubernetes objects: Role object, defines permissions within a particular namespace. ClusterRole object, same as Role, but defines cluster-wide permissions (all namespaces), because this resource is non-namespaced. RoleBinding, grants permissions within a specific namespace to a user or set of users. ClusterRoleBinding, grants cluster-wide permissions to a user or a set of users.

- **Role object**, defines permissions within a particular namespace.
- **ClusterRole object**, defines cluster-wide permissions (all namespaces), because this resource is non-namespaced.
- **RoleBinding object**, grants permissions within a specific namespace to a user or set of users.

- **ClusterRoleBinding object**, grants cluster-wide permissions to a user or a set of users.

2.6 Virtual Kubelet

Virtual Kubelet [4] is an open source Kubernetes kubelet implementation that masquerades as a kubelet for the purposes of connecting Kubernetes to other APIs. Virtual Kubelet features a pluggable architecture and direct use of Kubernetes primitives, making it much easier to build on.

From the official documentation, Virtual Kubelet is focused on providing a library that developers can consume in their projects to build a custom Kubernetes node agent. This project features an interface developers can implement that defines the actions of a typical kubelet (such as creating, deleting and updating pods, retrieving container logs and metrics, getting pod, pods and pod status).

Thus, Virtual Kubelet allows the creation of a special node within the cluster supported by customized APIs. This is what Liqo requires: a special node that represents an entire remote cluster and therefore needs some custom APIs to deploy pods and containers.

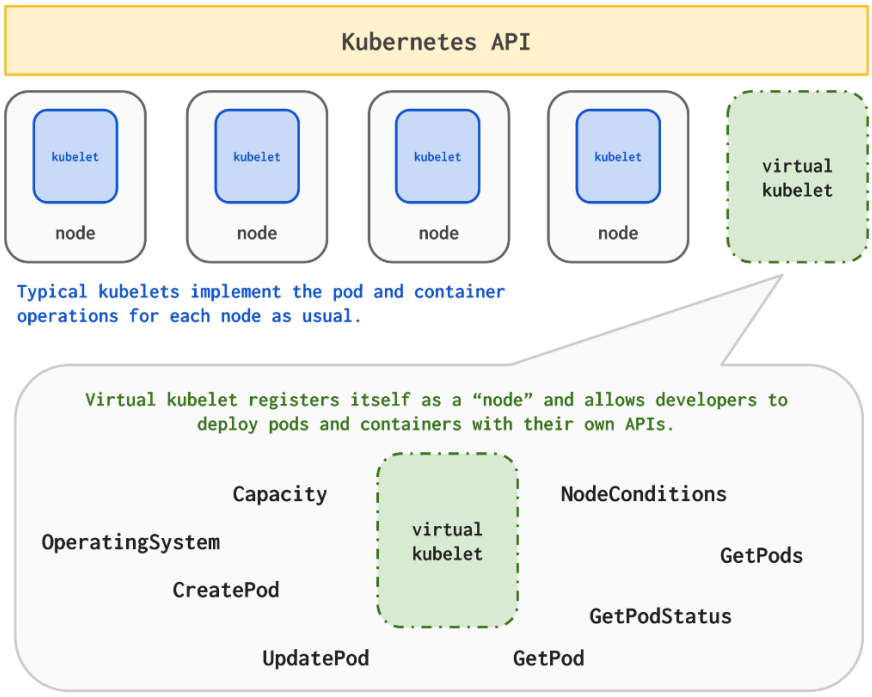

Figure 2.2: Virtual Kubelet

2.7 Kubebuilder

Kubebuilder is a framework for building Kubernetes APIs using custom resource definitions (CRDs).

Similar to web development frameworks such as Ruby on Rails and SpringBoot, Kubebuilder increases velocity and reduces the complexity managed by developers for rapidly building and publishing Kubernetes APIs in Go. It builds on top of the canonical techniques used to build the core Kubernetes APIs to provide simple abstractions that reduce boilerplate and toil.

Chapter 3

Liqo

This chapter introduces the conceptual foundations at the base of Liqo [5], as well as the core elements that make up its architecture.

3.1 Liqo: an overview

The Kubernetes technology is widely employed to handle cloud tasks. Clusters are designed to provide more resources—in terms of sheer computing power, available memory, storage capacity—than the ones normally required, to handle temporary peaks of load. This means that this excess of capabilities could be used by other clusters that undergo a period of higher load. Liqo [1] aims to unleash this potential power by connecting clusters together and have them work synergically to pursue their goals.

To accomplish this task, clusters establish a peering session that results in a larger virtual cluster that hosts the sum of the resources exposed by each cluster involved in the peering process.

The benefit of using Liqo is that it takes the core concepts that are well-known in the Kubernetes environment and exploits them to achieve more possibilities. Indeed, a cluster sees its peers simply as (virtual) nodes that add up to its (physical) ones, and schedules tasks to its nodes regardless of their actual nature.

The next sections will describe more in depth the presented concepts, starting with a core element and how to establish it: the Liqo peering.

3.2 The Liqo peering

Once two (or more) Kubernetes clusters are available to host workloads, they can become part of a multi-cluster topology by activating a peering session between them. This is where the Liqo experience starts off. A Liqo peering takes separate

entities and joins them into a wider environment that is capable of handling larger workloads. As a result, each involved cluster becomes aware of the existence of other remote peers, modeled by the ForeignCluster Custom Resource (CR). This process entails the exchange network parameters and other cluster information, so as to create a secure VPN that pods will leverage to communicate with one another as part of a large distributed cross-cluster application.

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

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

When an outgoing peering is active, it is of paramount importance to control what could be offloaded and what should not. This is done by leveraging some native Kubernetes concepts, namely Namespaces and label selectors, and some logic provided by Liqo to select which namespaces to offload, which pods within such namespaces to offload, and even which remote peers as the target of this offloading mechanism. The possibilities are endless.

The basic requirements to start a peering session is to have access to the remote Kubernetes API Server. This allows clusters to exchange information and create resources remotely, with the result of having a VPN that remote pods use to communicate as if they were all in the same Kubernetes cluster.

3.3 The Liqo reflection

Once a peering is established, the workload offloading is enabled by leveraging the virtual node abstraction and the namespace extension.

A virtual node represents a remote cluster and all of its shared resources (e.g. CPU and memory). This allows for a transparent extension of the local cluster's resources, as the virtual node added to the cluster is seamlessly taken into account by the vanilla Kubernetes scheduler when selecting the best place for executing workloads.

In addition to that, Liqo enables the extension of Kubernetes namespaces across the cluster boundaries. Once a namespace is selected for offloading, Liqo automatically creates twin namespaces in the selected subset of remote peers. These remote twin namespaces will host the remotely offloaded pods, as well as other resources living in the local namespace that has been extended remotely, such as

those related to service exposition (Ingress, Service and Endpoints resources), or storing configuration data (ConfigMaps and Secrets), to name a few.

3.4 Liqo Custom Resources

The following subsections present some of the Custom Resources used by Liqo to provide the peering and reflection features.

3.4.1 The NetworkConfig CR

This CR represents a set of network parameters (mainly IP addresses) by means of which clusters know how a remote peer has remapped the local PodCIDR, as well as the remote peer's PodCIDR. The "spec" part includes data related to the local cluster, while its "status" part reports the changes to the specifications. The idea is that a cluster creates this CR and sends it to the remote cluster it is going to establish a peering with. The remote cluster processes this CR and annotates in the "status" part everything it had to change in terms of IP address ranges to avoid any conflicts. These updates are reported back to the owning cluster.

Concurrently, the same happens in the opposite direction, so the remote cluster generates a NetworkConfig, writes its "spec" part and sends it to the local cluster, which annotates any changes in the "status" part to make the remote cluster aware of any modifications to the original specifications.

Once both the CRs are processed, a Liqo control loop reconciles them to create the TunnelEndpoint CR.

3.4.2 The TunnelEndpoint CR

This CR contains the relevant network configuration to establish a VPN tunnel with the remote cluster. This is used to make pods reach out to other remote pods as if they were in the same network.

3.4.3 The ForeignCluster CR

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

3.4.4 The ShadowPod CR

When a Pod is scheduled onto a virtual node, a Pod is created in the remote cluster for the actual workload execution. In the local cluster, a new object paired with the remote Pod is created: this is the ShadowPod. This resource, combined with the appropriate logic, represents the remote Pod in the local cluster and controls it.

3.5 Liqo Components

3.5.1 The CRD Replicator component

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

The CRD Replicator architecture is quite complex, but essentially it is implemented through a so-called reflector, which is a data structure containing the required objects and data to detect changes in local and remote namespaces (using local and remote informers), as well as to perform the traditional CRUD[1] operations in those namespaces (using local and remote clients). In particular, when an object, such as a NetworkConfig, is created in a namespace enabled for reflection and with the proper metadata labels set up, the local reflector (that is the one belonging to the cluster that created the object) follows these steps:

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

[1] Create, read, update, and delete (CRUD) are the four basic operations of persistent storage.

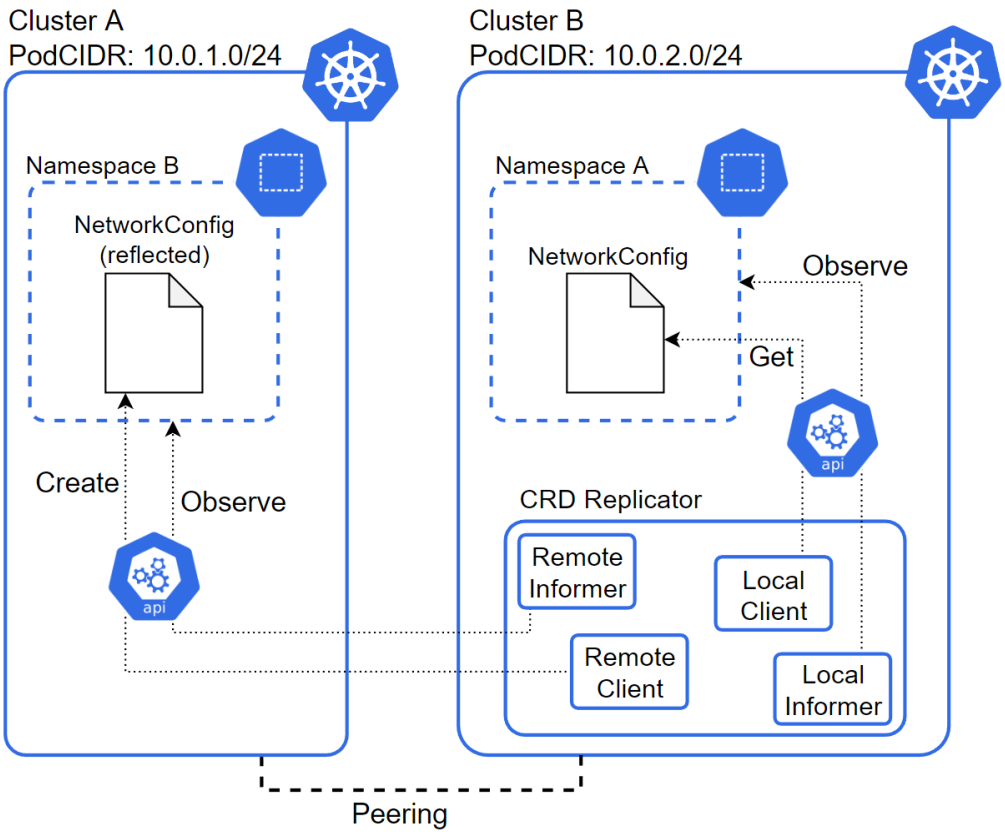

Figure 3.1: CRD Replicator

3.5.2 The Virtual Kubelet component

This component is a custom version of the Virtual Kubelet project. Whenever a peering session is established with a remote cluster, a dedicated instance of this component is created. Once created, it is used to offload pods to remote clusters, seen by the Kubernetes control plane as normal cluster nodes onto which to schedule a normal task. In addition to that, it is used to reflect core Kubernetes resources, such as Services and Endpoints: once deployed in a Liqo-enabled namespace, that is a namespace extended remotely, they will always be reflected to the selected remote peers.

3.5.3 The IPAM component

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

Chapter 4

Workloads in a multi-cluster environment

This chapter analyzes how offloading workloads to multiple peered clusters impacts the performance of the overall cluster architecture. Then, it proceeds by describing the current adopted strategy to handle complex cluster topologies, in order to lay down the foundation upon which it is possible to understand how the proposed work evolves from the current solution and to make a well-informed comparison between the two.

4.1 Overview

Thanks to the Liqo peering technology, clusters can be offloaded workloads from other clusters, participating in a synergetic effort to execute complex applications. This works well between two peered clusters, where one cluster either offloads some pods to the other end of the peering, while having other pods executing locally, or delegates the execution of all the pods that make up the workload to its peer. Both cases constitute the simplest usage of the capabilities that Liqo can provide. An example is shown in Figure 4.1.

More advanced uses of the Liqo technology may lead to a situation where one cluster has multiple peers to which it offloads parts of an application workload, while such peers have no peerings established between them. This can be represented as a tree where peers are the leaves and the first cluster is their common parent node. Figure 4.2 exemplifies this concept.

As a consequence, a pod executing in one leaf cluster might need to communicate to another pod that has been deployed to another leaf cluster. In this case, the generated traffic would be able to reach the other leaf via the parent cluster the two leaves have in common. This makes the parent cluster a central hub where all

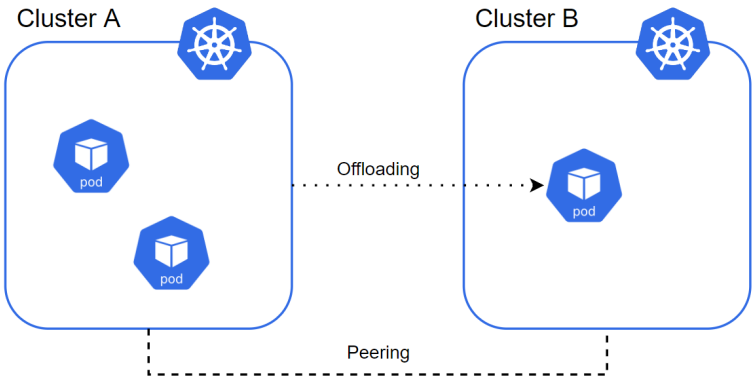

Figure 4.1: Two-cluster deployment

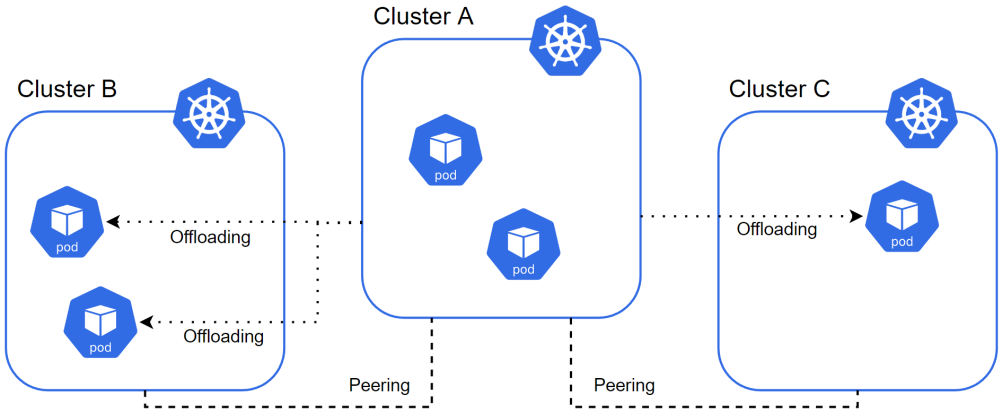

Figure 4.2: Three-cluster deployment

the communications between peers necessarily have to go through before hitting the destination, due to how the peering was set up, as visualized in Figure 4.3.

4.1.1 Problem assessment

The scenario presented above translates to a single point of failure in the central cluster, adds up overhead to the network communications, and increases the pressure on the central cluster, which now has to manage its own communications as well as those that simply need to pass through before hitting the other end, as they originate from a leaf cluster and are headed toward another leaf cluster.

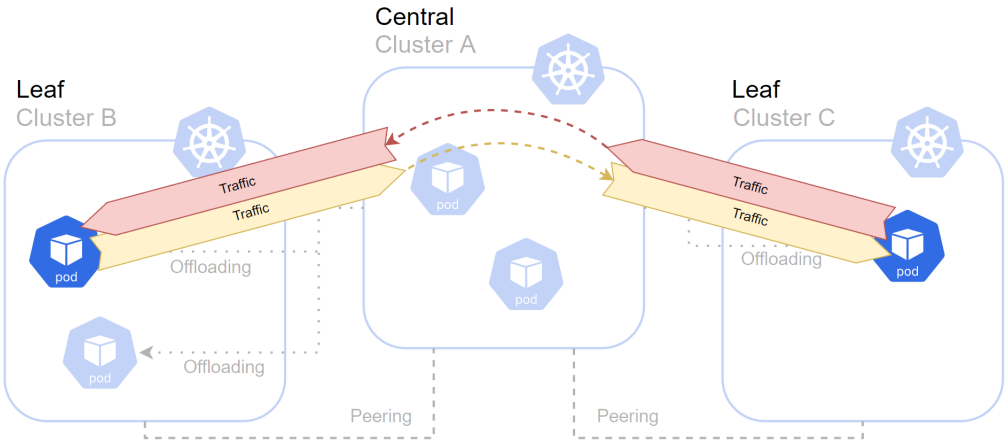

Figure 4.3: Traffic is routed through the central cluster

4.1.2 Possible solutions

First things first, to maintain an engineering spirit throughout the exposition of this paper, it should be mentioned that, for small applications, the overhead and increased pressure on the central cluster might even be acceptable in favor of a simpler setup and management. Of course, this depends on assessments regarding the nature of the cluster architecture and what workloads it is going to host.

Having characterized the problem and established what issues might arise when handling cross-cluster workloads, different solutions can be proposed. Each solution is a different approach to addressing the same problem, where the results are similar but the implementations are widely different and influence the general outcome. It should be also noted that these approaches are not mutually-exclusive, and could be deployed together to reach the common goal of a less overloaded central cluster.

Workflows affinity

One possibility is to consider business logic affinity and consequently deploy closely related pods in the same cluster to significantly reduce the amount of cross-cluster traffic that needs to traverse the central cluster. This approach focuses on the application domain and requires a deep understanding of the interactions of the application components. The effort put into this analysis could be quite large, as it deals with how the core business logic works, and, depending on the complexity of the application, might not translate into an actual improvement in the amount of cross-cluster traffic that traverses the central cluster.

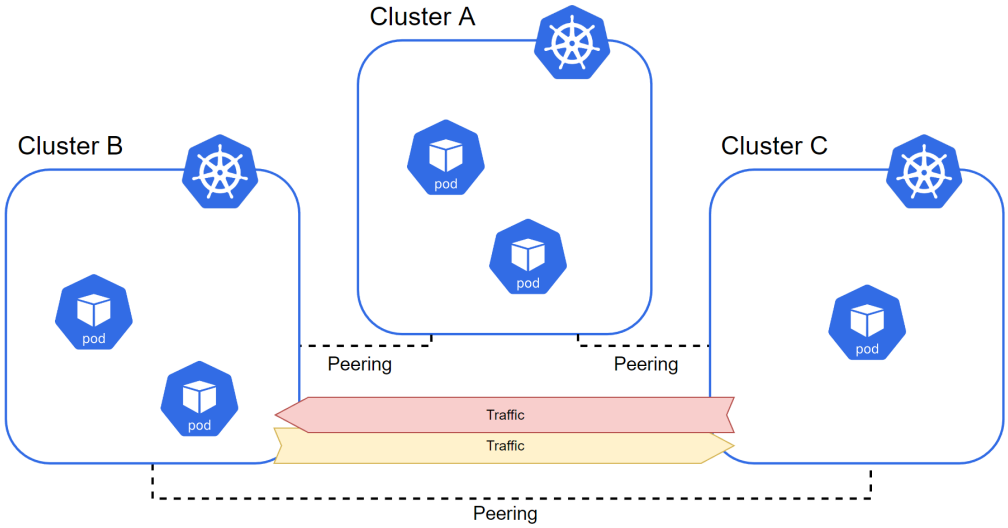

Figure 4.4: Establishing another full peering between the two leaf clusters; traffic can be routed directly to the destination.

Adding full peerings

Another possibility is to establish a direct peering between leaf clusters, thus making the initial tree topology a cyclic graph, as shown in Figure 4.4. This increases the complexity of the overall architecture, as well as its management and maintenance. Moreover, it places stricter requirements in terms of peering establishment that not always can be satisfied, e.g. a cluster administrator is not willing to establish a peering with a third entity due to permission constraints.

Implementing induced peerings

A different approach is to build a lightweight peering mesh between leaf clusters so as to make traffic travel directly to the destination without passing through the central cluster. As later sections will point out, this solution can be automated and does not require manual setup: this is the reason such peerings are called "induced". The advantages are a reduced overhead, latency, and less pressure on the central cluster, at the cost of a slightly increased complexity within the Liqo components. From a user perspective, this is completely transparent as it does not require to manually operate on the cluster, nor to change application workloads.

This solution appears to be a desirable candidate to solve the above mentioned problem, mostly because it provides transparency to application developers, and frees cluster administrators from reviewing their cluster's peering policies.

Before getting into the details of the concepts behind the induced peering and its implementation aspects, it is worth taking a step back and looking at the current

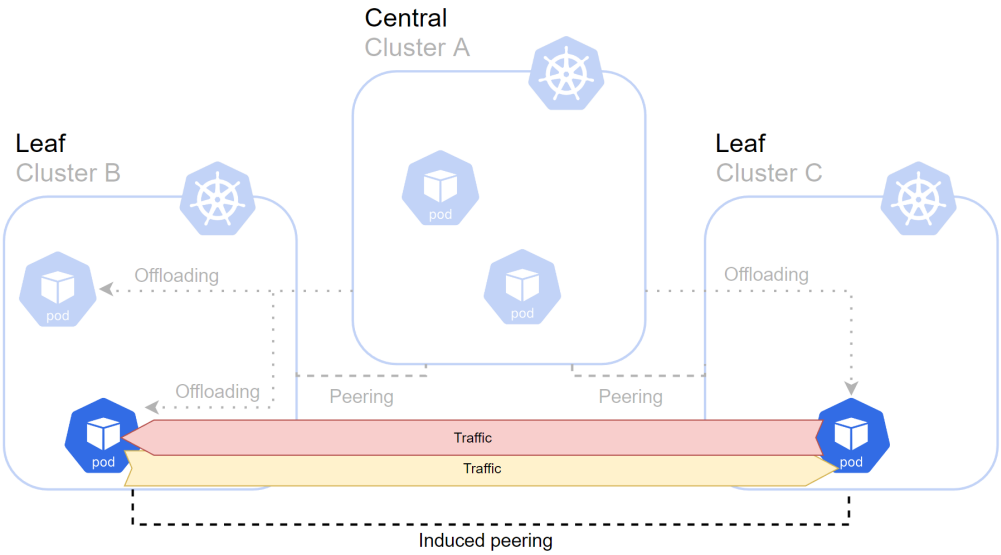

Figure 4.5: Induced peering between the two leaf clusters; traffic is routed directly to the destination.

state of the things so as to make it possible to compare the current solution and the one proposed in the present work. Therefore, the next section will outline how Liqo works under the hood when handling an offloading request, in particular by describing the key concept of resource reflection.

4.2 Liqo resource reflection

Liqo makes it possible to offload workloads to remote clusters. When considering the set of pods that compose an application, the offloading might be full (i.e. all the pods are executed remotely), or just partial (i.e. some pods are executed locally and some remotely). In both cases, Liqo has the responsibility to keep the overall application in a functioning state, letting pods (of the same application or even other pods) across different clusters communicate successfully in order to fulfill their tasks. To this end, the Liqo control plane makes use of two important components, namely the Virtual Kubelet and the IPAM service, to reflect the necessary resources to the remote clusters where pods have been offloaded and to keep track of them in the local cluster.

The following sections will focus on the reflection of Services, Endpoints and Pods, and will assume that the peering process between the considered clusters has already completed, for which reason another core Liqo component, that is the CRD Replicator, will not be mentioned. Indeed, this component is required to exchange the networking data to establish the peerings, thanks to which the actual

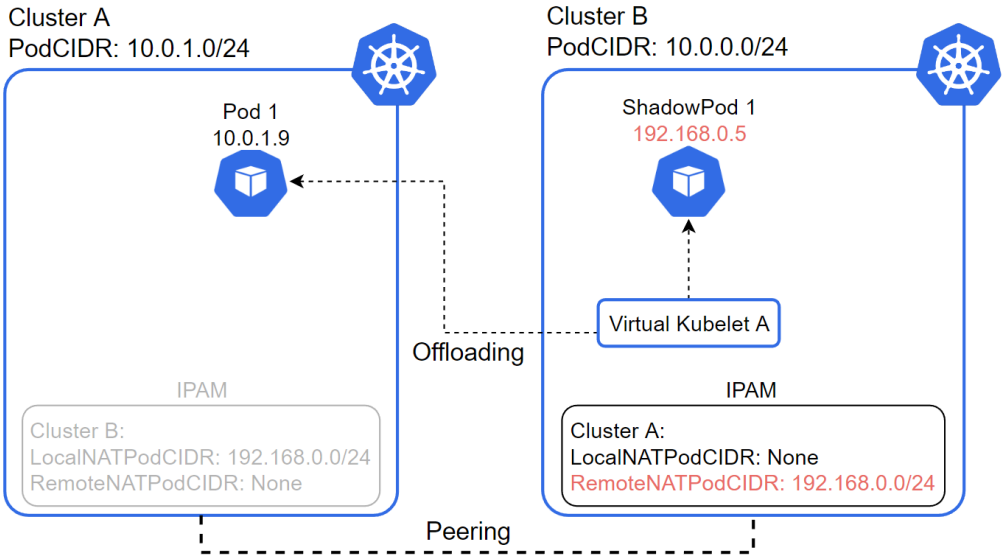

Figure 4.6: Reflection in a two-cluster deployment

offloading phase can happen. Once this initial setup is completed, the offloading and reflection operations carried out by the Virtual Kubelet with the aid of the IPAM component can start.

4.2.1 Two-cluster workloads

Starting with a simple two-cluster architecture, we can analyze how the Liqo reflection logic handles multi-cluster workloads effectively. Note that the following examples assume that pods and services live in Liqo-enabled namespaces, which are those that have been extended remotely for offloading and resource reflection purposes.

Figure 4.6 shows an example of a Pod offloading to a remote cluster. Cluster B has already established a peering session with cluster A, and has got all the data about possible remappings between the two clusters in the IPAM component. Specifically, it has not been remapped by cluster A, while the last has been remapped to another PodCIDR (192.168.0.0/24). Cluster B's scheduler decides to offload Pod 1 to A: the Virtual Kubelet that refers to A offloads Pod 1, which gets assigned a free IP address within the cluster A's PodCIDR. Moreover, before creating ShadowPod 1, it contacts the local IPAM and establishes that the IP address to assign to it is 192.168.0.5 (remapped). At this point, Pod 1 is actually executed in cluster A.

Such an example lays down the basis for more complex deployments, where service and endpoints resources enter the scene and thus make the offloading

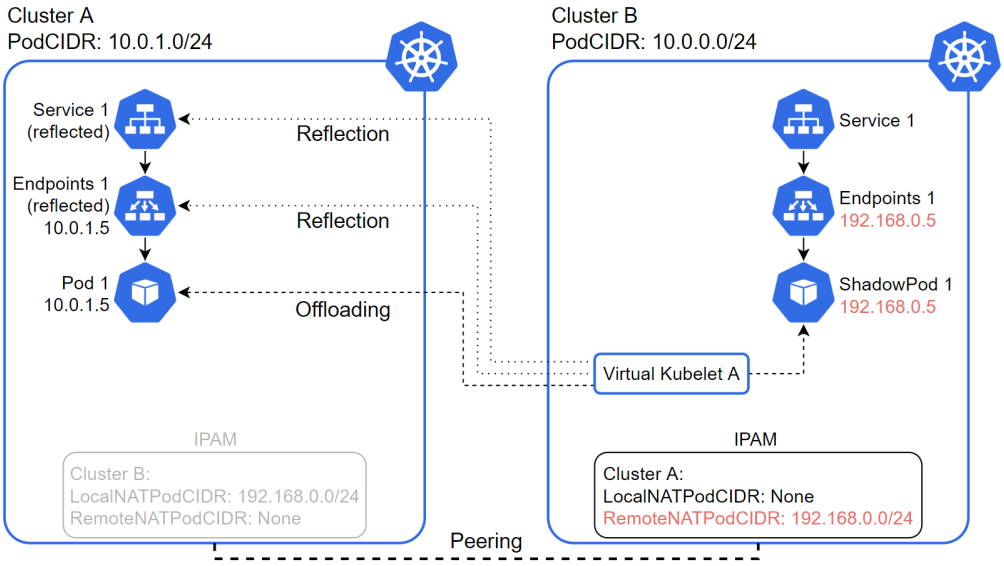

Figure 4.7: Advanced reflection in a two-cluster deployment

mechanism a very powerful tool to distribute workloads across clusters and make pods accessible remotely. Figure 4.7 shows an example of this possibility.

Just like the previous example, Pod 1 is offloaded to cluster A, with its counterpart ShadowPod 1 in cluster B. The difference is that the (shadow) pod is exposed through Service 1 and its related Endpoints 1 deployed in cluster B, and reflected to cluster A to expose the offloaded (actually running) pod in cluster A as well. Because the pod is running in cluster A, it gets its IP address from the cluster A's PodCIDR (same for the reflected Endpoints 1). Moreover, the Virtual Kubelet learns from the local IPAM that the IP address to assign to ShadowPod 1 and Endpoints 1 is 192.168.0.5 (remapped, see *RemoteNATPodCIDR*). By having the service and the endpoints resources in both clusters, any pod living in A or B is able to communicate with Pod 1 via its local service.

Please note that because the pod is executed in cluster A, its IP address belongs to A's PodCIDR (same for the reflected Endpoints 1), while the shadow pod in cluster B has an IP address out of B's PodCIDR: it should be within A's PodCIDR, but due to the remapping it is within 192.168.0.0/24.

Alternative configuration

For the sake of completeness, Figure 4.8 shows another possible configuration with two clusters. The real difference here is that the pod is not offloaded to a remote cluster. Nonetheless, the related service and endpoints resources are reflected to the remote cluster by the Virtual Kubelet, which learns from the local

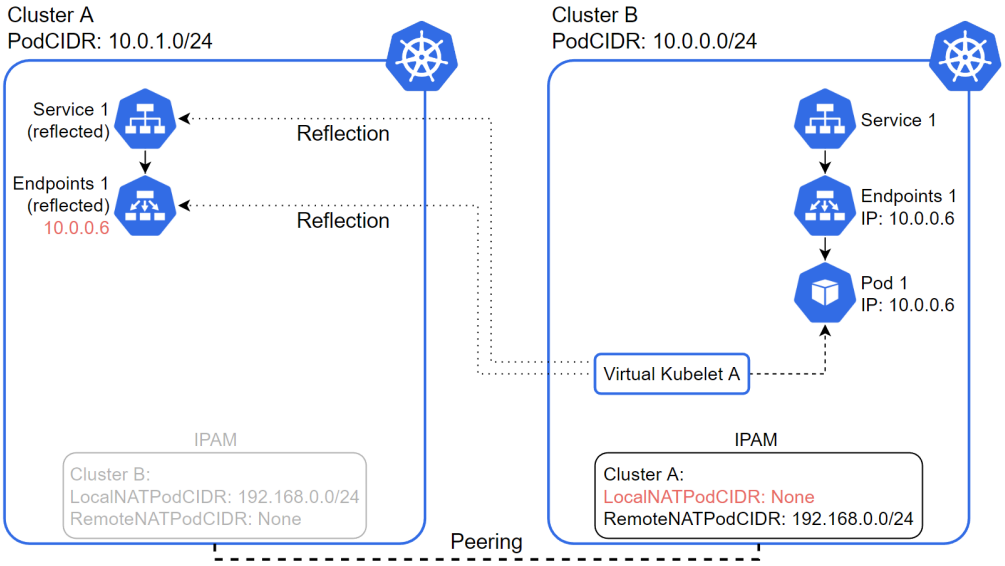

Figure 4.8: Alternative reflection in a two-cluster deployment

IPAM that the IP address to assign to Endpoints 1 is 10.0.0.6 (not remapped, see *LocalNATPodCIDR*). Still, any pod in cluster A can communicate to Pod 1 via the reflected Service 1.

This time, note that the reflected Endpoints 1 gets an IP address in B's PodCIDR (A has not remapped B's PodCIDR to a different address range).

4.2.2 Three-cluster workloads

Now that the basics of reflection are covered, even more advanced layouts can be proposed, such as a three-cluster architecture with a central cluster that has established a peering session with two remote clusters, seen as the leaves of this tree-shaped topology. Once more, by making the overall setup more complex, new challenges arise that Liqo has to deal with in order to stick to its main goal of keeping applications functional when deployed in a multi-cluster environment.

In particular, the challenge to overcome is to make remote pods, running in the leaf clusters, exchange traffic even though their clusters have not established any peering session. Currently, Liqo leverages the addition of a new network, called ExternalCIDR, specifically designed for this purpose and deployed in each cluster. As in the case of the PodCIDR networks, also ExternalCIDR ones can be remapped to other ranges to avoid address conflicts.

However, the adopted strategy has some scalability issues that will become visible with the following example. What is presented now is a cornerstone around which the present work unfolds: it will propose a natural advancement in terms

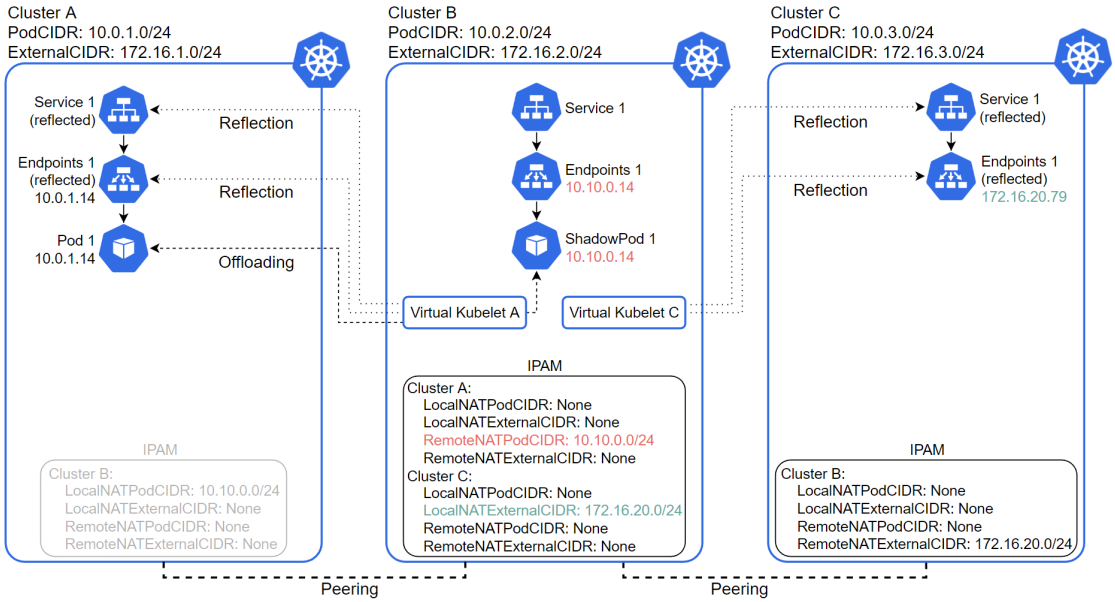

Figure 4.9: Advanced reflection in a three-cluster deployment

of scalability, while retaining the ease of use that is characteristic of the current implementation.

Figure 4.9 illustrates the aforementioned use case. Pod 1 is offloaded to cluster A, as a result of the decision made by cluster B's scheduler. Just like before, ShadowPod 1 is assigned an IP address that complies with the RemoteNATPodCIDR entry (for cluster A) in the central cluster IPAM, namely 10.10.0.14 (remapped). The pod is exposed through Service 1 and Endpoints 1, which are present in the central cluster and are reflected to both the leaf clusters. Speaking of the reflected Endpoints 1 in cluster A, everything stays the same in terms of IP address allocation. Quite different is the matter of assigning an IP address to the reflected Endpoints 1 in cluster C. The definition of the right IP address made by the Virtual Kubelet dedicated to cluster C depends on the newly introduced ExternalCIDR belonging to cluster B. This new address range became necessary as the normal cluster B PodCIDR is already reserved by C to host endpoints directly reflected from B, but the endpoints resource that is being reflected this time actually refers to a pod that runs on a third cluster (A), for which reason the name "external" has been chosen. The underlying principle is to always avoid IP address conflicts, and the ExternalCIDR ensures this does not happen. Moreover, as indicated in the RemoteNATExternalCIDR entry, cluster C has remapped the original 172.16.2.0/24 address range to 172.16.20.0/24.

At this point, any pod running in cluster C and willing to reach the remote Pod 1 would simply need to contact the reflected Service 1, and the communication flow

will traverse the central cluster before landing to Pod 1. Under the hood, some Liqo components are deputed to handle all the remappings and understand what destination the traffic is headed to.

From a user perspective, this is completely transparent, which means that:

- Cluster managers are not required to manually operate on the cluster to make this work.
- Application developers are not required to change their applications, which is one of Liqo's strengths.

As anticipated, it is clear that now the central cluster is burdened with the additional task of handling the traffic flows that are exchanged between pods deployed in the two leaf clusters. This poses a scalability problem that, with large workloads, results in much more work to be done by the central cluster, increased latency and networking overhead. Even worse, when the central cluster becomes unavailable, such passthrough flows come to a halt.

The next chapter will start from these weaknesses and build a set of concepts that will represent an evolution of pod-to-pod communication between clusters that share no peering.

Chapter 5

Evolution of multi-cluster workload deployments: concepts

The primary concept that will be discussed in this chapter is a new type of peering that will be created between clusters that share a common peer, meaning that the configuration under examination is the one that comprises at least three clusters in a tree-shaped topology. This addition is backed by a set of new Custom Resource Definitions (CRDs) and some updates to how the CRD Replicator works.

5.1 The induced peering

So far, when offloading pods from a central cluster to multiple (at least two) peers that have no direct peering between them, the pod-to-pod traffic could only happen by sending it to the central cluster, whose task was to forward it to another remote destination. In this way, an application can continue working as communications are made possible between its pods spread over many clusters. It is important to notice that those leaf clusters are not bound to any ongoing peering session, therefore are not required to know or be aware of each other.

A concrete improvement would consist of being able to avoid this passthrough traffic and instead make it flow directly between the relevant pods, while still preserving an important point, that is not to require them to establish an additional peering session, which would affect the ease of set up and would entail access to the remote API servers from each party, something that might not be feasible, or simply not desirable.

By considering this communication necessity carefully, the only true demand

is to have a functioning networking phase between the two remote clusters. No offloading is requested: if this was the case, a normal peering, as the ones presented so far, would have sufficed.

For this purpose, a new kind of peering has been designed: the *induced peering*. The word "induced" recalls something that is just a consequence of an already existent full-featured peering. As anticipated, this peering only provides the networking elements that allow for direct pod-to-pod communication. Peers linked by an induced peering are called *induced peers*.

With the induced peering, pod-to-pod communications between leaf clusters flow directly from source to destination with no additional networking overhead introduced by an intermediary in the form of a central cluster, which is not burdened with the generated traffic that is not within its competencies. As a consequence, pods offloaded to leaf clusters can continue to communicate even in case the central cluster becomes temporarily unreachable, as their routing policies are configured to bypass the central cluster. Of course, during this outage, the reflected endpoints that directly point to the remote pods running in an induced peer cannot be updated to the changes that potentially occur on the IP address of those pods, as this is controlled by the central cluster. Therefore, if pods change their IP address for any reasons, the overall deployed application that such pods compose together will experience a denial of service.

The next sections will detail the basic building blocks and the processes for establishing an induced peering.

5.2 APIs: Neighborhood and ForeignCluster

Now that clusters have a new type of peering at their disposal, the IPAM component must be adapted to accommodate more information, specifically the one related to potential induced peers. In order to fill this data, the current APIs have been updated: the new Neighborhood API, and the updated ForeignCluster API.

5.2.1 The Neighborhood CR

A Neighborhood object contains all the clusters that have established a full peering session with the cluster that originates this resource—as the name suggests, they are its neighbors—, except for the cluster this piece of information is intended for.

In the example shown in Figure 5.1, the central cluster creates two Neighborhood resources and sends them to its direct peers. Thanks to this mechanism, each peer gains knowledge about the existence of a remote cluster that otherwise would have remained unknown. For convenience, the destination cluster does not appear in the list of the central cluster neighbors, as it of course already knows about itself.

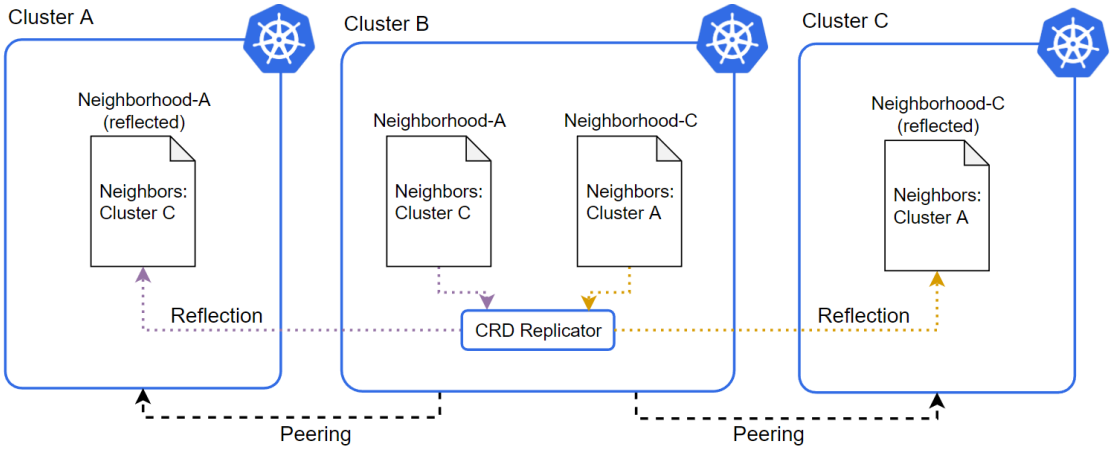

Figure 5.1: Neighborhoods exchange

Leaf clusters also send a Neighborhood object to the central cluster with an empty list of neighbors.

The component dedicated to the remote reflection of the Neighborhoods is the CRD Replicator, whose access to the remote API Servers allows to create remotely such resources in a dedicated namespace.

5.2.2 The (induced) ForeignCluster CR

At this point, each receiver of the Neighborhood resource has a controller that reconciles the neighbors data and creates an induced ForeignCluster object. This is a streamlined ForeignCluster resource that annotates the remote cluster as an induced peer, but lacks fields that are normally necessary in a full-fledged peering session. This process is visualized in Figure 5.2.

5.3 Networking setup

5.3.1 Exchange of NetworkConfigs

Once the induced ForeignCluster has been created, another Liqo custom controller reconciles it to produce a NetworkConfig just like the ones created in a normal peering. Such a resource thus contains the cluster's local network information in the "spec" field, and will report, within the "status" field, the changes made by the target cluster (the induced peer). However, this time such an object has to travel across the central cluster before arriving at the destination. Therefore, it is called a *passthrough NetworkConfig*.

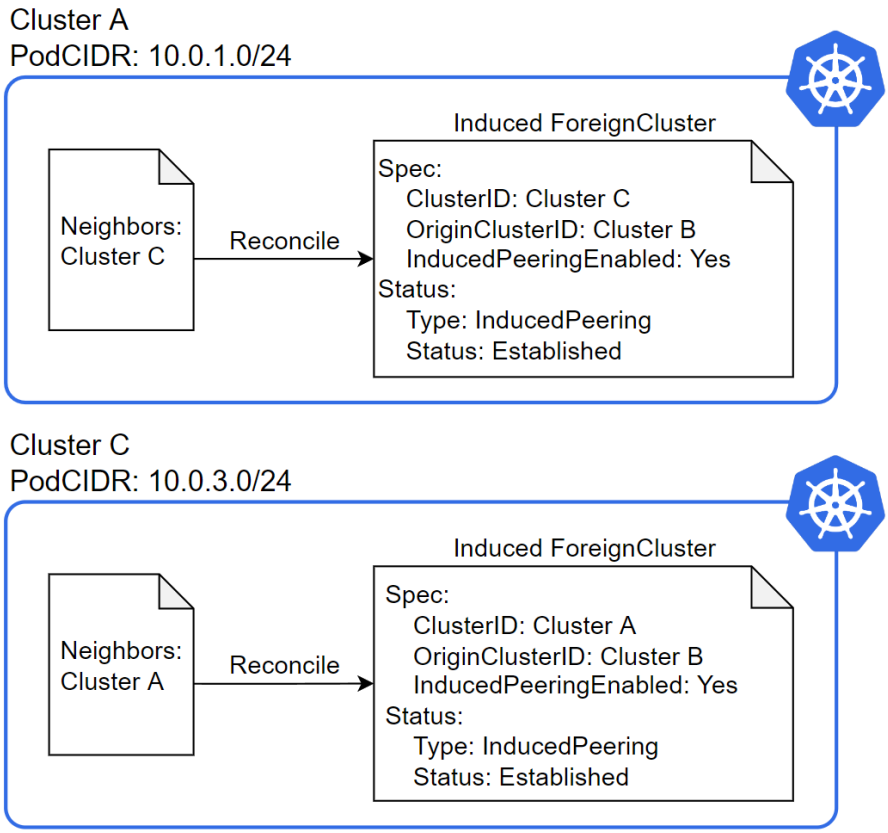

Figure 5.2: Induced ForeignClusters creation

For the sake of clarity, the purpose of these objects is to ultimately create the VPN endpoints that will enable the direct communications between pods.

In Figure 5.3, cluster C receives A's NetworkConfig and learns what PodCIDR is used by A. In this case, by assuming that cluster C has already reserved the address range 10.0.1.0/24 for other purposes, it remaps it to 10.81.0.0/24 and writes that information in the *PodCIDRNAT* entry under the status field. This information returns back to A, so that cluster A can correctly configure its local IPAM. Moreover, since the destination NAT is done on the destination cluster, cluster A needs to know about this remapping to correctly NAT any destination IP addresses that belong that remapped range and map them back to the local PodCIDR 10.0.1.0/24 (this is done by a dedicated Liqo component).

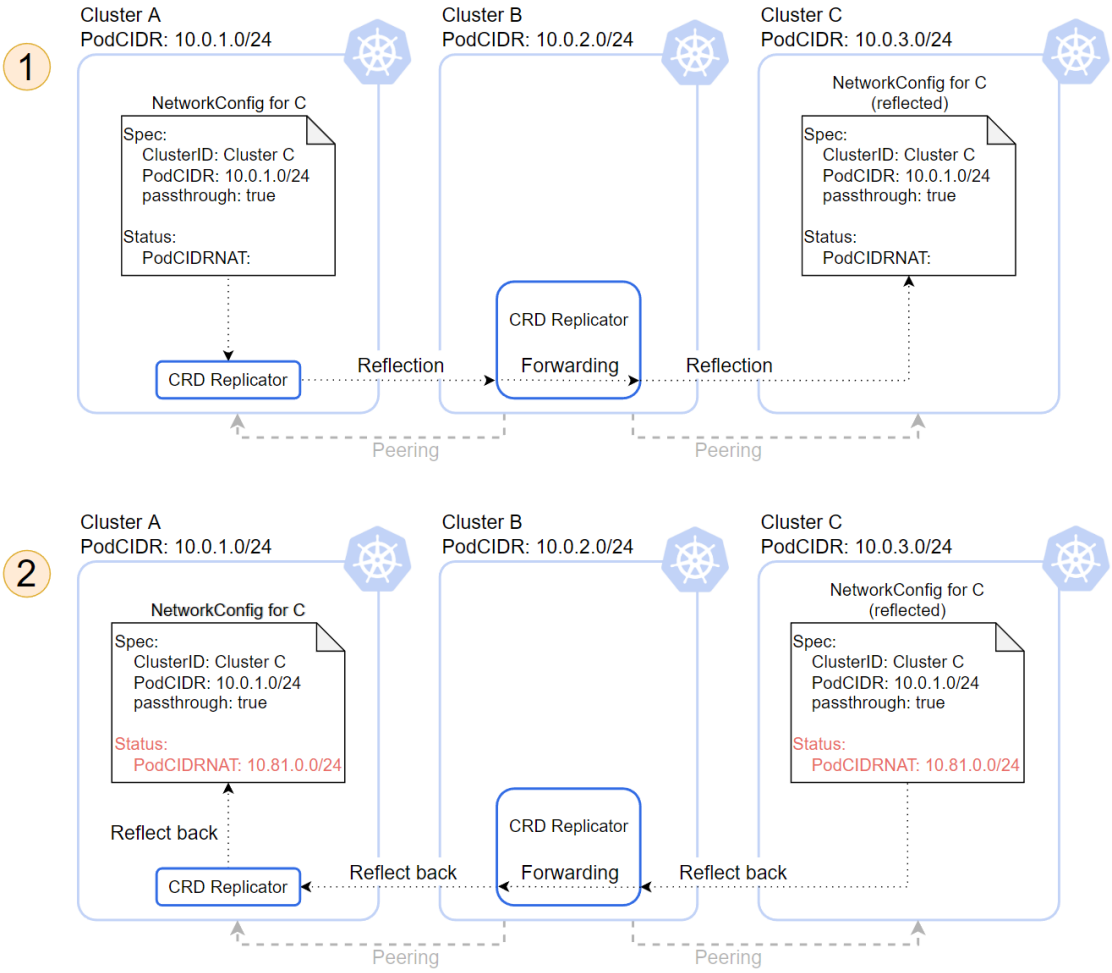

Figure 5.3: Exchange of passthrough NetworkConfigs: A to C

On the other hand, C creates its own NetworkConfig, the central cluster forwards it to A, which learns that C's PodCIDR is 10.0.3.0/24. If this address range is not already reserved, as it is in this example, cluster A will set the *PodCIDRNAT* entry, under the status field, to None, otherwise it will pick a free address range and use it to remap C's PodCIDR. In both cases, this information gets reflected back to C. This is depicted in Figure 5.4.

A note regarding the forwarding of passthrough NetworkConfigs

The reason for which NetworkConfigs have to follow this forwarding procedure operated by the central cluster is because leaf clusters have no access to the induced peer's API server. Therefore, they simply leverage the already established normal peering session with the central cluster, which then is capable of setting this kind of

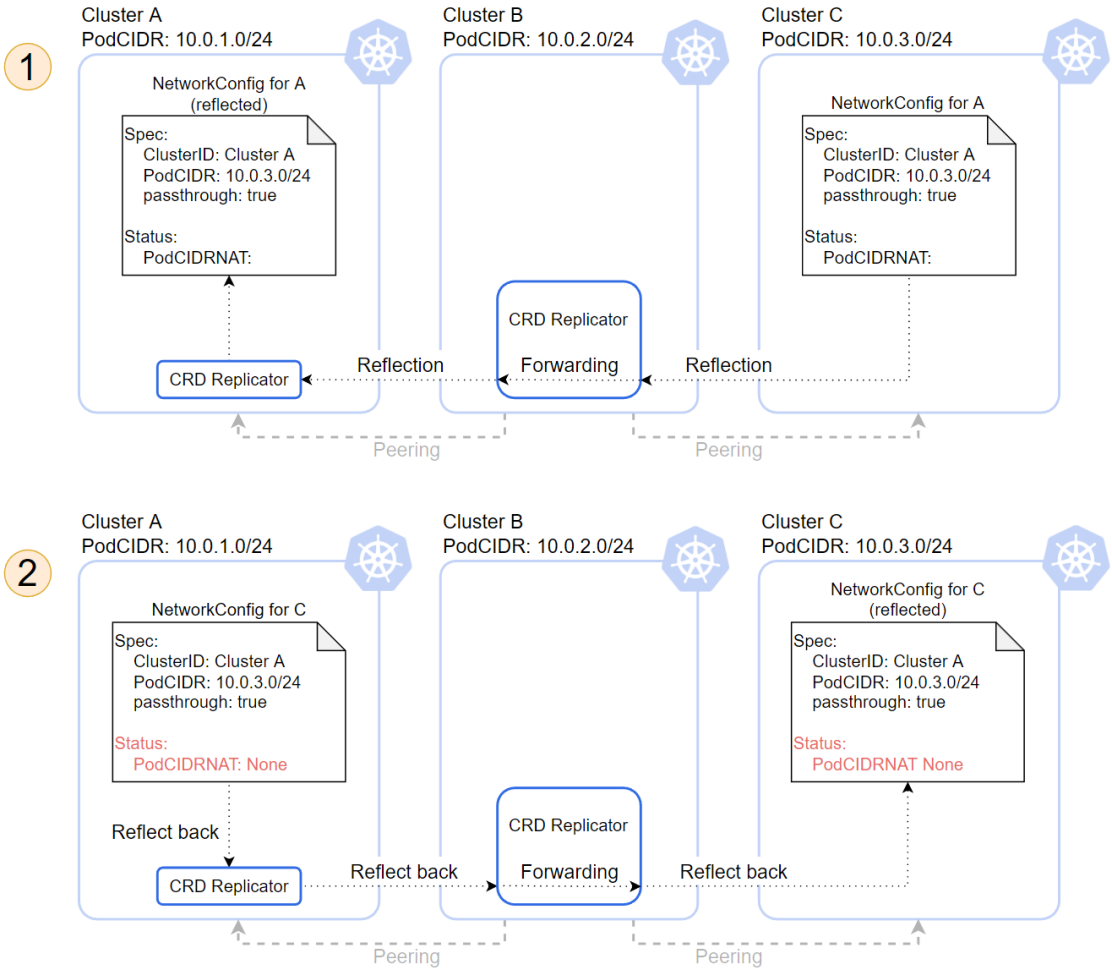

Figure 5.4: Exchange of passthrough NetworkConfigs: C to A

passthrough NetworkConfigs apart from the normal ones, and then forwards it to the other remote cluster, again leveraging the already established normal peering session.

5.3.2 TunnelEndpoints creation

At the end of this exchange process, the two induced peers have the same two copies of the passthrough NetworkConfigs. Again, by reusing an already existing Liqo custom controller, each cluster merges the two NetworkConfigs (one is local, one is the reflected one) and produces a TunnelEndpoint object, as Figures 5.5 and 5.6 present.

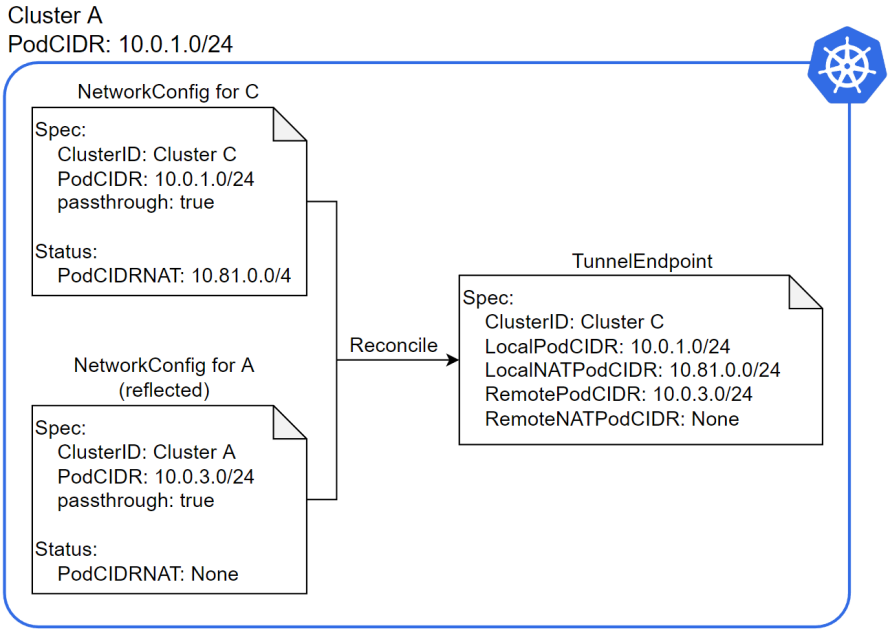

Figure 5.5: Cluster A: TunnelEndpoint creation

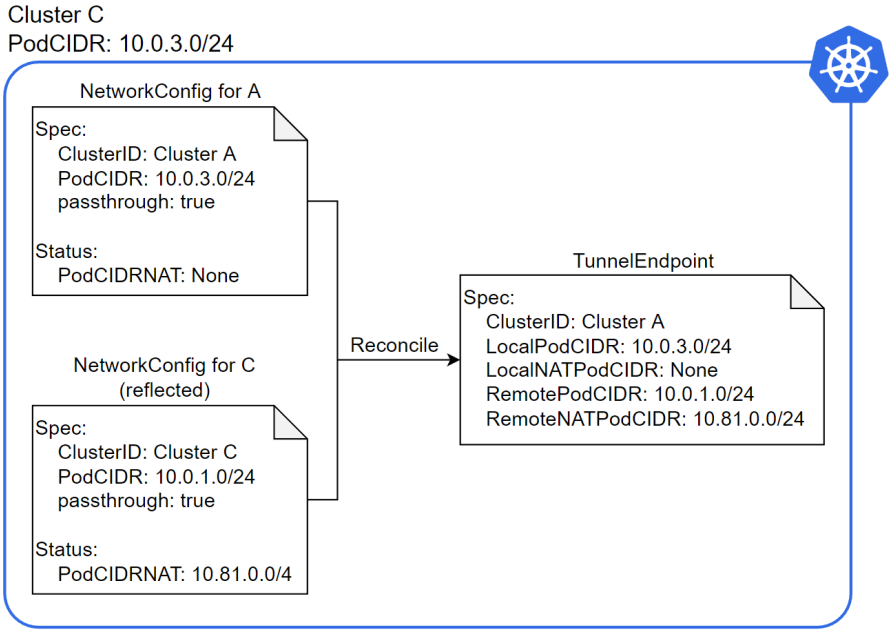

Figure 5.6: Cluster C: TunnelEndpoint creation

The two resulting resources contain all the data required to establish a VPN tunnel toward the other induced peer.

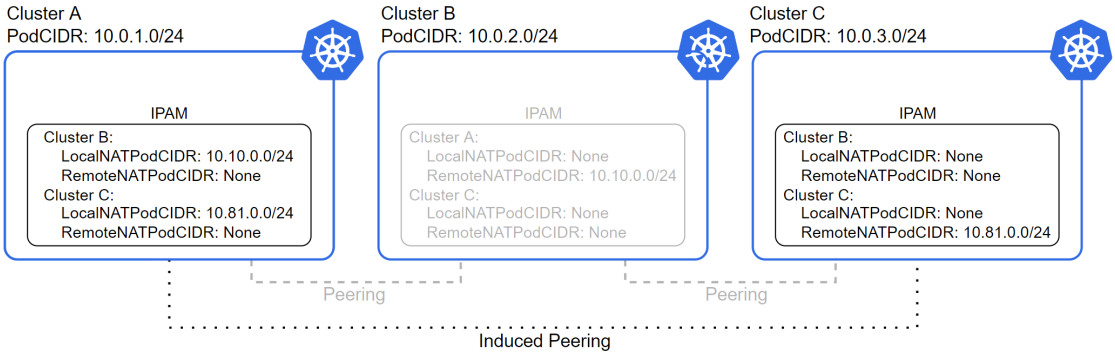

Figure 5.7: IPAM status once the networking setup has completed

5.3.3 IPAM data

To accommodate the additional networking information related to induced peers, the IPAM storage is used as it was for normal peers, since the kind of data to store is the exactly the same.

Figure 5.7 shows what IPAM stores in each cluster after the networking setup is complete. The focus is on the two induced peers.

Note the remapping that exists between the two induced peers. In particular, cluster C has remapped A's PodCIDR to 10.81.0.0/24. This will constitute a key piece of information, as later subsections will highlight.

5.3.4 Changes to the CRD Replicator

The CRD Replicator has a key role in the process of network parameters exchange between the induced peers. It has been updated to ensure each party receives the NetworkConfig from the counterpart via the central cluster.

The forwarding operated by the central cluster is aided by the data that is carried in the labels within the NetworkConfig resource. More specifically, this data reports the peculiar nature of this object, as well as the identity of the actual recipient. The CRD Replicator component has been updated to adapt to this scenario and treat such a resource differently from the normal NetworkConfigs.

The greatest changes in its behavior can be observed in the central cluster, since it has the job to set passthrough NetworkConfigs apart from traditional ones, and forward them to the destination instead of consuming them locally. However, to keep the implementation consistent with the current code base, the core reflection principles discussed in Subsection 3.5.1 have been preserved. The idea is to leverage what has been already established when the central cluster activated a direct peering with clusters A and C, including namespaces, reflectors, and data structures. In other words, the direct peerings have paved the way to the

establishment of the induced peering.

From the point of view of a passthrough NetworkConfig that is created in cluster A and has to reach cluster C, this translates into the following:

- A already has access to B's API server, and has already configured the namespace to reflect toward B.
- B already has access to C's API server, and has already configured the namespaces to receive from A and reflect toward C.
- C has already configured the namespace to receive from B.

Figure 5.8 visualizes this process. It shows that the central cluster has got a new CRD Replicator that is dedicated to the aforementioned forwarding process. This process consists of a normal reflection activity, this time though being between two namespaces of the same cluster. As the next Chapter will highlight, the central cluster physically has one CRD Replicator that splits into multiple reflection modules, each with its own tasks, therefore Figure 5.8 is just a logical view of the central cluster reflection process.

The status gets written by the destination cluster, in this case C, and the same reflection modules that have been used to reflect forward the resource from A to B to C have the capability to detect this change and reflect it back from C to B to A.

The same applies in the opposite direction, with the due variations.

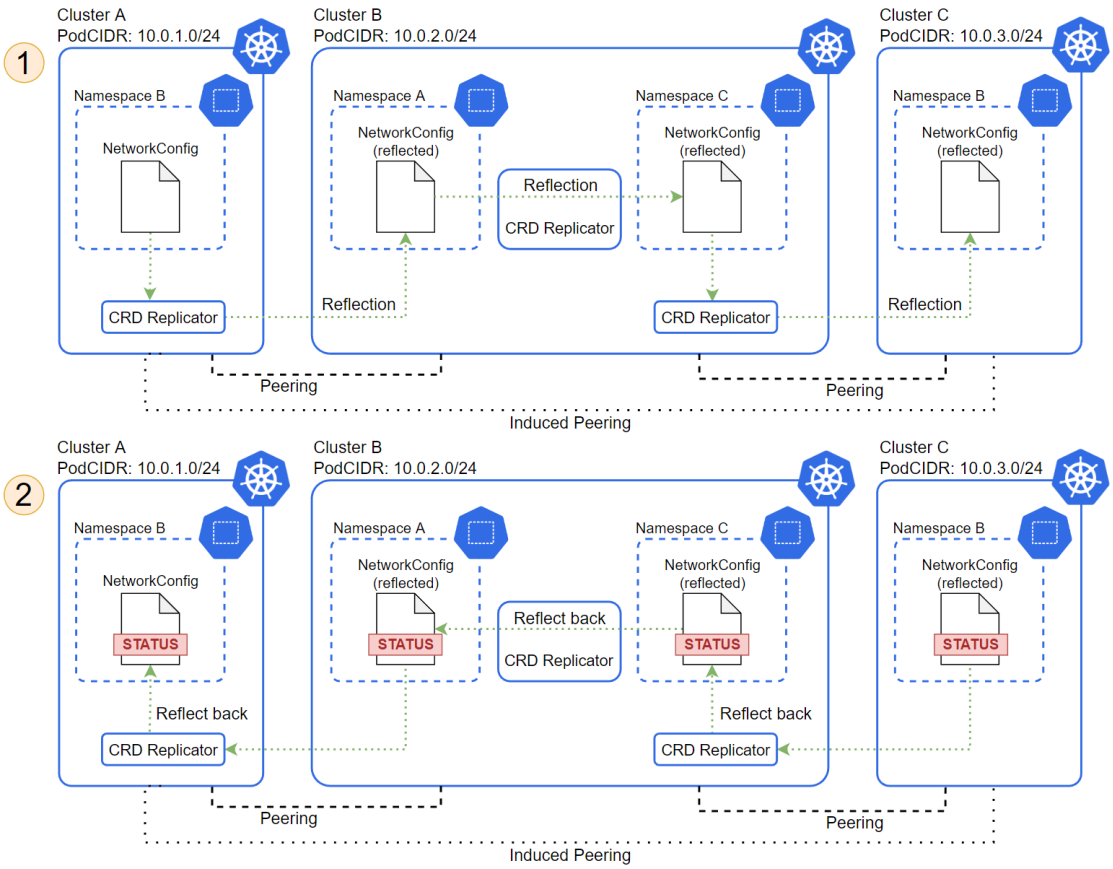

Figure 5.8: CRD Replicators and Namespaces for the reflection of a passthrough NetworkConfig

5.4 Resource reflection

5.4.1 Endpoints reflection

Once the networking between the two induced peers is configured, the induced peering is set up. All that remains is to use it to make remote pods directly communicate with one another. To achieve this, endpoints resources must be reflected and their IP address must point to the actual running pods. This is shown in Figure 5.9.

Pod 1, exposed through Service 1, is offloaded to cluster A. Thus, Service 1 and Endpoints 1 are reflected to that same cluster, as well as to cluster C. A corresponding ShadowPod 1 is running in cluster B. Any pod running in cluster C willing to reach out to Pod 1 would first pass through the (local) reflected Service 1. The destination IP address of Pod 1 would then be obtained through the reflected

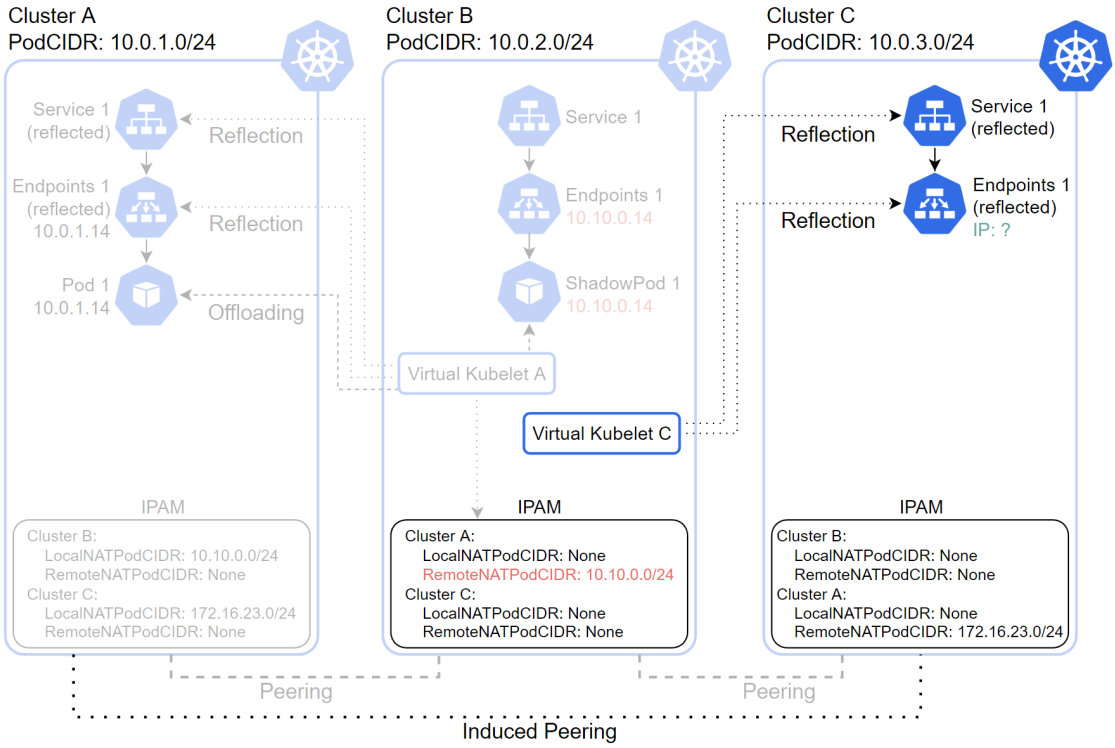

Figure 5.9: Endpoints reflection: unknown IP address

Endpoints 1 that is connected to the service. Thus, the reflection of the endpoints plays a key role.

However, the decision the Virtual Kubelet C has to take about what IP address to write in that reflected Endpoints object is not trivial, and was solved with the introduction of the ExternalCIDR address range, with all the limitations that led to the induced peering, as explained earlier in Section 4.2.2.

At first glance, it appears that the IP address to write to that Endpoints resource is the one owned by the running Pod 1 in cluster A. A closer analysis reveals that cluster C might have remapped A's PodCIDR to another range. This key information—that is the potential remapping operated by C toward A—is something the Virtual Kubelet C (running in cluster B) must be aware of when reflecting the Endpoints object to cluster C. However, it is nowhere to be found in the local (cluster B) IPAM, but can be found in cluster A's IPAM, under the *LocalNATPodCIDR* entry dedicated to cluster C that has been configured in a previous step.

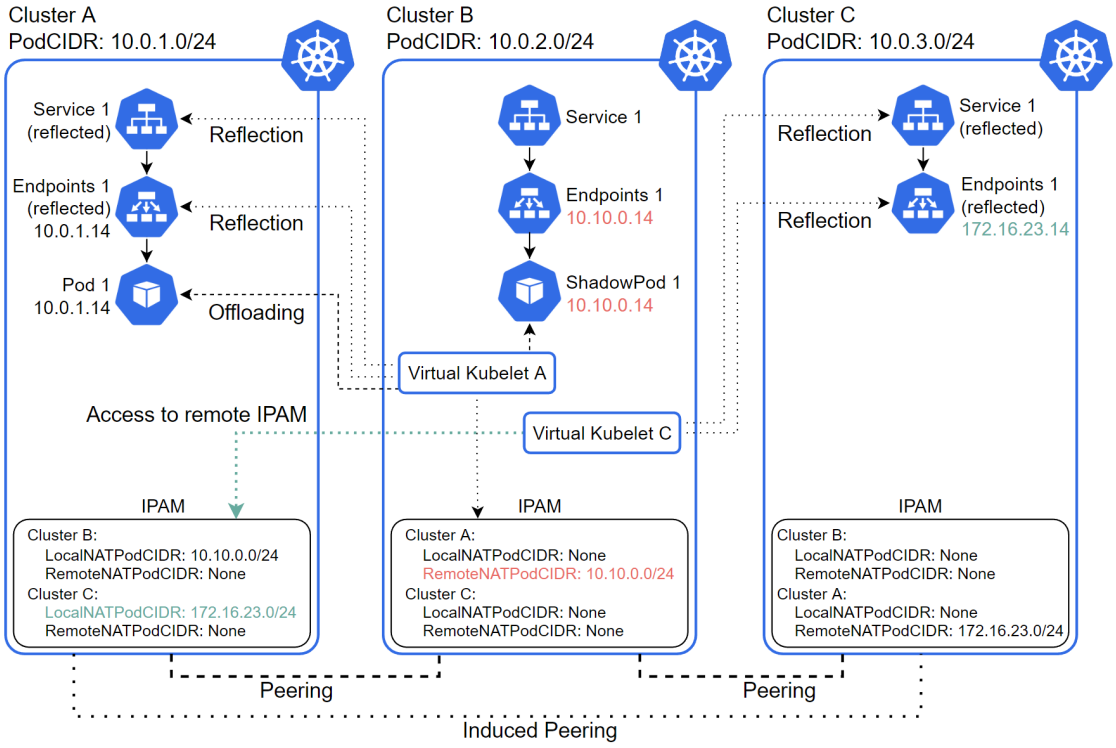

Figure 5.10: Endpoints reflection: access to remote IPAM

5.4.2 Remote IPAM access

As just mentioned, the Virtual Kubelet C requires access to the remote IPAM service before being able to reflect the endpoints. To make this possible, the "liqo" namespace of cluster A, where A's IPAM service resides, needs to be offloaded toward central cluster B. This operation follows the standard namespace extension that Liqo provides to enable the workload offloading, as presented in Section 3.3. At this point, the Virtual Kubelet C, running in cluster B, can learn what address range to use to write the IP address in the reflected endpoints, by simply following the reflected IPAM service. The IPAM in cluster A will then use the *LocalNATPodCIDR* value for cluster C and return an IP address of 172.16.23.14, which combines the mapped network range with the host part of the original IP address. Figure 5.10 displays this behavior.

Chapter 6

Evolution of multi-cluster workload deployments: implementation

This chapter illustrates how the presented concepts unfold into the actual implementation, providing a closer look at the different components adopted for the solution.

This will allow sharing a deeper level of detail on the APIs, as well as on the changes done to the CRD Replicator and the Virtual Kubelet to support the induced peering.

6.1 APIs implementation

6.1.1 The Neighborhood CR

Listing 6.1 shows that the Neighborhood resource comprises a Spec and a Status section. The former contains the identity of the cluster that sent this resource remotely to another cluster, and specifically the unique cluster ID, as well as the Neighbors map, which contains the IDs and names of all the clusters that have peered with the local cluster (except the receiver of this resource) that creates and sends this resource. The Status is left empty, as it won't be updated by the receiver, differently from what happens with NetworkConfigs, for example.

The NeighborhoodList struct lists Neighborhood resources.

Listing 6.1: The Neighborhood custom resource

```
type Neighbor struct {
    ClusterName string `json:"clusterName"`
}

// NeighborhoodSpec defines the desired state of Neighborhood
type NeighborhoodSpec struct {
    // ClusterID is the ID of the sender of this resource
    ClusterID string `json:"clusterID"`
    // Neighbors contains the clusters that have peered with the
    local cluster
    Neighbors map[string]Neighbor `json:"neighbors"`
}

// NeighborhoodStatus defines the observed state of Neighborhood
type NeighborhoodStatus struct {}

// Neighborhood is the Schema for the neighborhoods API
type Neighborhood struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec   NeighborhoodSpec   `json:"spec,omitempty"`
    Status NeighborhoodStatus `json:"status,omitempty"`
}

type NeighborhoodList struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ListMeta `json:"metadata,omitempty"`
    Items           []Neighborhood `json:"items"`
}
```

6.1.2 The ForeignCluster CR and the induced peering

As already mentioned, the ForeignCluster API is used also for the induced peering case. It has been extended to keep track of the Spec and Status of this new type of peering session. Therefore, the Spec now has a FullPeering struct and an InducedPeering struct. The former contains data about the full peering session, which, for an induced ForeignCluster, means just some general information between the two clusters, and the latter contains data about the induced pereing session, i.e. whether the induced peering is enabled and the cluster identity (ID and name) of the cluster that created and sent the Neighborhood resource.

On the other hand, the Status field kept its original structure, while being able to store the new induced peering type.

Listing 6.2: The induced ForeignCluster custom resource

```
// ClusterIdentity contains the information about a remote cluster (ID and Name)
type ClusterIdentity struct {
    // Foreign Cluster ID
    ClusterID string `json:"clusterID"`
    // Foreign Cluster Name
    ClusterName string `json:"clusterName"`
}

type FullPeering struct { /* full peering fields */}

type InducedPeering struct {
    // InducedPeeringEnabled indicates whether the induced peering is active
    InducedPeeringEnabled PeeringEnabledType `json:"inducedPeeringEnabled"`
    // Cluster Identity of the sender of the Neighborhood
    OriginClusterIdentity ClusterIdentity `json:"originClusterIdentity,omitempty"`
}

// ForeignClusterSpec defines the desired state of ForeignCluster.
type ForeignClusterSpec struct {
    // Foreign Cluster Identity
    ClusterIdentity ClusterIdentity `json:"clusterIdentity,omitempty"`
    // FullPeering defines the configuration for a full peering
    FullPeering FullPeering `json:"fullPeering,omitempty"`
    // InducedPeering defines the configuration for an induced peering
    InducedPeering InducedPeering `json:"inducedPeering,omitempty"`
}

// ForeignClusterStatus defines the observed state of ForeignCluster
type ForeignClusterStatus struct {
    // fields containing the peering status, namely type (e.g. "InducedPeering") and condition (e.g. "Established")
}

// ForeignCluster is the Schema for the foreignclusters API
type ForeignCluster struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`
    Spec   ForeignClusterSpec   `json:"spec,omitempty"`
    Status ForeignClusterStatus `json:"status,omitempty"`
}
```

For example, a three-cluster architecture with cluster B as the central one and clusters A and C as the leaves, this resource, when sent to cluster A by cluster B, would look like the following:

Listing 6.3: Example of ForeignCluster resource

```
apiVersion: discovery.liqo.io/v1alpha1
kind: ForeignCluster
metadata:
  name: cluster-c-induced
spec:
  clusterIdentity:
    clusterID: c
    clusterName: cluster-c
  fullPeering:
    incomingPeeringEnabled: "No"
    networkingEnabled: "Yes"
    outgoingPeeringEnabled: "No"
  inducedPeering:
    inducedPeeringEnabled: "Yes"
    originClusterIdentity:
      clusterID: b
      clusterName: cluster-b
status:
  peeringConditions:
  - status: Established
    type: InducedPeering
```

6.2 Changes to the CRD Replicator

The CRD Replicator plays a key role in the correct and working definition of an induced peering. It has been updated so that NetworkConfigs can traverse the central cluster, land on a remote cluster that soon will be an induced peer and let it know about the necessary network parameters used to establish a VPN tunnel that will make pods communicate with one another. Although its architecture has been rethought to support the induced peering scenario, its distinctive reflection logic has been preserved and extended for this use case.

Before introducing the changes done to the CRD Replicator, it is worth giving a deeper insight over the workings of the CRD Replicator, and in particular on its main building block: the reflector.

6.2.1 The reflector

At the core of the CRD Replicator is the reflector, a piece of software that is responsible for creating a copy of a local resource to a remote namespace. This

logical component is set up with all the data it requires to access its local API server, the remote peer's API server and to configure two informer objects that are instructed to watch for any addition, update, and deletion events occurring to specific resources in a local and a remote namespace. Every time they detect such an event, they put it in a working queue. This queue of events is consumed item by item, and for each event a different operation can be expected.

For example, in case a NetworkConfig has been just created and added to a namespace, the local informer, which watches for additions of NetworkConfigs in that namespace, places that event in the queue. Then, that event is popped out of the queue and processed, and in particular the external reflector uses the client to the remote peer's API server to replicate that NetworkConfig to a remote namespace. Now, the remote informer keeps track of that NetworkConfig in the remote namespace. It occurs that the remote peer updates the NetworkConfig status, and therefore an update event is placed in the queue thanks to the remote informer. This event is consumed from the queue by triggering a corresponding update on the status of the original copy of the resource, which now reports the same data that was written in the remote reflected copy. This is the mechanism that is used to replicate resources remotely and reflect back their status once it has been updated by the remote peer. The remote informer is also capable of detecting deletions of resources in the remote namespace, by adding a deletion event in the working queue, which leads the reflector to recreate the resource remotely, using the client to the remote API server. In case the deletion event affects the original local copy of the resource, the client to the remote API server will destroy the remote copy.

6.2.2 External and internal reflectors

Before the introduction of the induced peering, the CRD Replicator contained one reflector per remote peer. Now, in addition to that, it contains another reflector, which is responsible for the forwarding of all the passthrough NetworkConfigs. This reflector has been called an internal reflector, while the normal ones have been called external reflectors. As the name suggests, the distinction is based on the scope of the reflection mechanism. Even though their name is different, they are built the same: they are simply reflectors as just described. What actually changes their behavior is the data on which they operate. To achieve this, the CRD Replicator internals have been redesigned with the goal of having a shared data structure and behavior. For sure, this helped to keep things consistent and easier to implement, without introducing brand new logic that would complicate the reflection mechanisms.

As stated above, external and internal reflectors are basically the same. What changes their behavior is the data on which they operate. This data is contained

in a dedicated data structure, as shown in the Listing 6.4.

Listing 6.4: ResourceToReflect

```
type ResourceToReflect struct {
    gvr          schema.GroupVersionResource

    sourceNamespace string
    targetNamespace string

    sourceClusterID string
    targetClusterID string
    localClusterID  string

    listerForSource cache.GenericNamespaceLister
    listerForTarget cache.GenericNamespaceLister

    /* other fields */
}
```

Each ResourceToReflect object is an instance of a resource that has to be reflected remotely. It contains the required information to operate such a reflection:

- The type of the resource.
- The source namespace, which is the local namespace from which to replicate the resource.
- The target namespace, which is the namespace the resource will be replicated to. In case of an external reflector, it is a remote namespace, while in case of an internal reflector, it is another local namespace within the same local cluster.
- The source cluster ID, which is the local cluster ID.
- The target cluster ID, which is the cluster ID of the target namespace. In case of an external reflector, it is the remote cluster ID, while in case of an internal reflector, it is the local cluster ID.
- A reference to the local cluster ID.
- The source lister, which is used to retrieve local instances of the resource to reflect in the source namespace.
- The target lister, which is used to retrieve local or remote instances of the resource to reflect in the target namespace.

The Listing below shows the implementation of the reflector data structure.

Listing 6.5: Reflector

```
type Reflector struct {
    // TenantNamespaces is a map of clusterIDs and tenant namespaces.
    TenantNamespaces map[string]string

    clientForTarget dynamic.Interface

    resources map[schema.GroupVersionResource][]*resourceToReflect
    workqueue workqueue.RateLimitingInterface
    isLocalToLocal bool

    /* other fields */
}
```

Each Reflector instance contains the data to handle the reflection:

- A map of cluster IDs and namespaces, used by the reflector to keep track of the remote cluster IDs and their namespaces.
- A reference to the client of a target cluster. In case of an external reflector, it is the client towards a remote API server, while in case of an internal reflector, it is the client towards the local API server.
- A list of ResourceToReflect instances, therefore knowing how to deal with each of the resources to reflect.
- A working queue to store all the addition, update, and deletion events of the tracked resources.
- An indication on its nature, telling whether it is an internal reflector or not.

6.2.3 Induced peering scenario

When the central cluster receives the passthrough NetworkConfig from one of its peers, it has to move it forward to the destination cluster because it is not intended for itself. By having established a peering with the sender of the passthrough NetworkConfig and another peering with the final receiver of that same resource, the central cluster already has the proper reflectors that will permit forwarding that object. The only thing missing is the logic that copies the NetworkConfig from one reflector to the other. This actually translates into moving that resource from one local namespace, where the resource has been copied to, to another local namespace, from which the resource will be copied and then written to a remote namespace belonging to the final receiver.

The CRD Replicator therefore has been updated to include the internal reflector, used only to reflect NetworkConfig resources between local namespaces that are bound to the normal reflectors created for the full peering sessions.

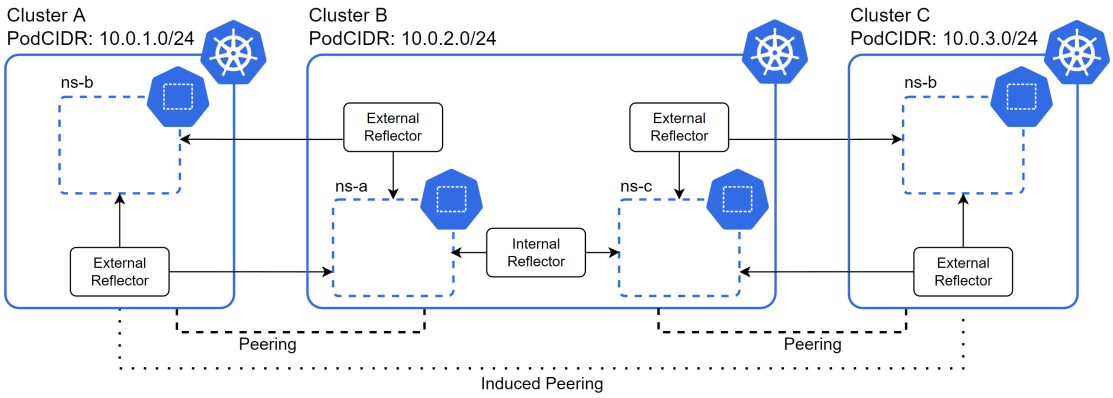

Figure 6.1: Reflectors

The NetworkConfigs are the only resources that need to undergo this special treatment, as they carry precious network information that induced peers have to exchange in order to successfully set up a VPN connection.

As depicted in the Figure above, each cluster has a number of external reflectors that reflects the number of active peering sessions: the central cluster, due to having two peerings, has two external reflectors. As explained earlier, each external reflector keeps track of a local and a remote namespace, by watching for additions, updates, and deletions of specific resources in those namespaces. Those events are collected in a working queue that holds the details of the events and allows the external reflectors to operate accordingly: in case the event is the addition of a NetworkConfig to the local namespace, the external reflector has to replicate that resource remotely to the remote namespace, leveraging the client to the remote peer's API server that holds within.

Cluster B has also an internal reflector, which carries out a similar job to the external reflectors: it watches for additions, updates, and deletions on the two local namespaces, and replicates NetworkConfigs from one namespace to the other. This is done to forward a passthrough NetworkConfig coming from A and headed towards C. The internal reflector leverages the same namespaces already configured for the external reflection logic.

6.2.4 The resulting NetworkConfigs

When considering the usual three-cluster architecture with cluster B as the central cluster, having two bidirectional (outgoing and incoming) peerings with cluster A and C, and those two cluster having an induced peering established, several NetworkConfigs are processed and treated to set up all the proper VPN communications. The following Figure shows the resulting number of NetworkConfigs once their exchange between the three clusters is completed.

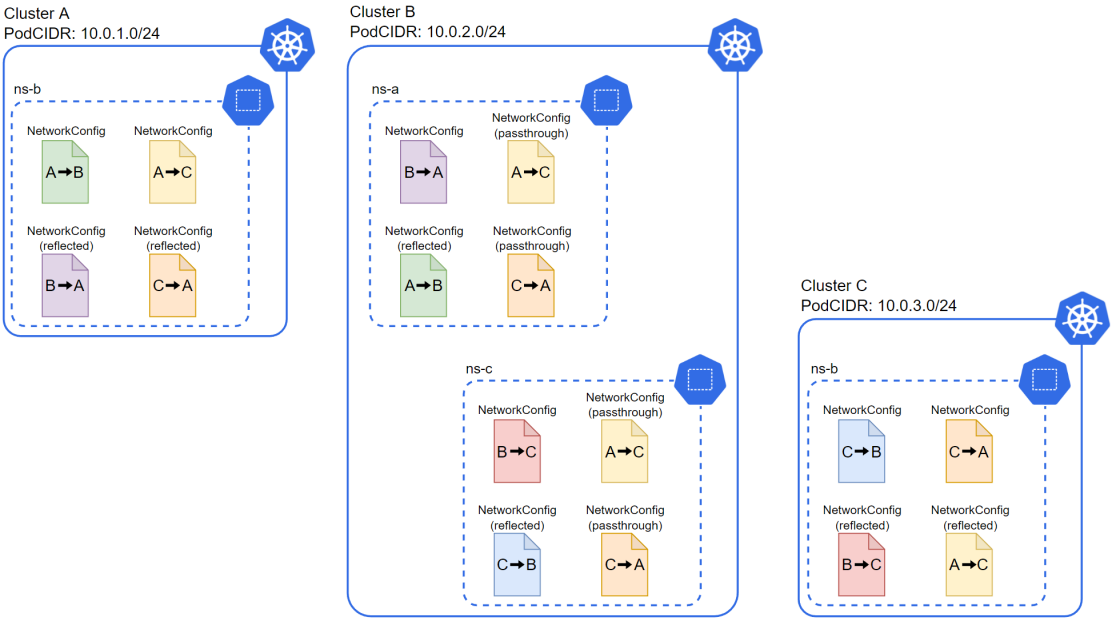

Figure 6.2: The resulting NetworkConfigs

The total amount of NetworkConfigs shown in Figure 6.2 are divided as follows:

- Cluster A has four NetworkConfigs in namespace ns-b, namely one pair of local and remote copies reflected to and by B (peering with B), one pair of local and remote copies reflected to and by C via B (induced peering with C).
- Cluster C has four NetworkConfigs in namespace ns-b, namely one pair of local and remote copies reflected to and by B (peering with B), one pair of local and remote copies reflected to and by A via B (induced peering with A).
- Cluster B has eight NetworkConfigs, divided into two namespaces. In namespace ns-a, these are one pair of local and remote copies reflected to and by A (peering with A), one pair of remote copies reflected by A to C and by C to A (passthrough NetworkConfigs). In namespace ns-c, these are one pair of local and remote copies reflected to and by C (peering with C), one pair of remote copies reflected by A to C and by C to A (passthrough NetworkConfigs).

Handling NetworkConfigs in the central cluster

The example above showed the resulting NetworkConfigs in the context of a three-cluster architecture. When considering a growing number of leaf clusters, all peered with a common central cluster and therefore participating in the full mesh of induced peerings, the number of NetworkConfigs handled by the central cluster becomes

larger and larger. In particular, the number N of passthrough NetworkConfigs stored and handled by the central cluster grows as the square of the total number n of involved clusters (including the central one), following the equation:

$$N = n(n-1)$$

This amount is added to the standard, non-passthrough NetworkConfigs that the central cluster exchanges with its peers as part of the full-fledged, normal peering sessions. This can pose a problem in terms of scalability for the central cluster, due to the number of NetworkConfigs and the resources used to handle them.

6.3 Changes to the Virtual Kubelet

The Virtual Kubelet component has been updated to enable the correct reflection of Endpoints toward a remote cluster when the relevant Pod is deployed to a different remote cluster that has an induced peering with the first cluster. The reason is that the central cluster's Virtual Kubelet needs to know whether one induced peer has remapped the PodCIDR of the other induced peer, therefore it requires access to the remote IPAM before assigning the correct IP address to the reflected Endpoints resource.

To ensure the central cluster is able to connect to the remote IPAM, this component needs to be exposed via a service and such a service needs to be reflected on the central cluster, so that the last knows about it and can reach out to it. This reflection is done by simply offloading the "liqo" namespace toward the central cluster, thus leveraging a feature that Liqo already provides. The reason behind this is because that namespace contains all the pods that enable the various Liqo-related features, including the IPAM service. Once the IPAM service is reflected, the central cluster only needs to understand whether to use the local IPAM or the remote one when translating an IP address for the purpose of reflecting a Service and the relevant Endpoints resource. Of course, this is not limited to one peer of the central cluster: any peers can offload its own "liqo" namespace toward the central cluster, and the central cluster will thus receive the reflected IPAMs, each one in its own namespace.

Before introducing the rest, it is worth noticing that the actual resource type that carries the IP address of the Pod and that is connected to the Service resource for the Pod exposition is the EndpointSlice resource, even though the discussion so far always mentioned the Endpoints resource as the concept that represents the tracking of Pod IP addresses. The EndpointSlice is a different and improved implementation of the Endpoints resource, but both do the same job.

The following listing presents the core changes done to the EndpointSlice reflection logic within the Virtual Kubelet component. Some parts have been omitted for convenience and to keep the discussion clear.

Listing 6.6: Virtual Kubelet

```
func MapEndpointIPs(/* params */) /* returns */ {
    /* ... */
    // Get the cluster ID of the cluster the address belongs to
    response, err := ipamclient.GetClusterIdentity(ctx, &ipam.ClusterIdentityRequest{Ip: original})
    clusterID := response.GetClusterID()
    clusterName := response.GetClusterName()

    useLocalIPAM := true
    if clusterID != "" {
        // Remote cluster ID found, check whether cluster mapping exists
        ipamClient = remoteIpamClients[clusterID]
        if ipamClient == nil {
            identity := discoveryv1alpha1.ClusterIdentity{
                ClusterID:   clusterID,
                ClusterName: clusterName,
            }
            ipamClient = initRemoteIpamClient(ctx, &identity)
            remoteIpamClients[clusterID] = ipamClient
        }

        if ipamClient != nil {
            clusterMappingResponse, err := ipamClient.DoesClusterMappingExist(ctx, &ipam.ClusterMappingRequest{ClusterID: forge.RemoteClusterID})
            if clusterMappingResponse.GetDoesExist() {
                useLocalIPAM = false
            }
        }
    }

    if useLocalIPAM {
        // Remote cluster ID not found, use local IPAM
        ipamClient = ipamclient
    } // else remote cluster ID found and cluster mapping found, use remote IPAM

    mapResponse, err := ipamClient.MapEndpointIP(ctx, &ipam.MapRequest{ClusterID: forge.RemoteClusterID, Ip: original, IsInduced: !useLocalIPAM})
    /* ... */
}
```

The logic shown above can be better understood by referring to an example consisting of a central cluster B, and two leaf clusters A and C. Among A and C an induced peering has been already established. The central cluster offloads a pod

to cluster A. This Pod is exposed through the relevant Service and EndpointSlice resources, which need to be reflected toward cluster C so that other pods can directly communicate with the first pod. To properly set the IP address of the EndpointSlice resource that is going to be reflected to cluster C, the Virtual Kubelet that runs in cluster B and that refers to cluster A accesses the cluster A's remote IPAM and learns of a possible remapping done by C on the A's PodCIDR. This information is found under the *LocalNATPodCIDR* entry. This example assumes that C has remapped A's PodCIDR from 10.1.0.0/24 to 10.3.3.0/24.

Given this, the presented code refers to that specific Virtual Kubelet. Starting from an IP address of 10.1.0.18, this code determines the cluster identity (i.e. cluster ID and name) of the cluster the address belongs to, which is A. This is done using the local IPAM of cluster B. Then, since the address belongs to a remote cluster, it determines whether to use the local or the remote IPAM for the translation of the IP address to the address space 10.3.3.0/24. This is done by querying the A's remote IPAM in order to know whether it stores the information saying that A has been remapped by C from the original address range 10.1.0.0./24 to a different one. This is the case, since A has been remapped to 10.3.3.0/24. The last line of code is the actual translation, which takes the original IP 10.1.0.18 and returns 10.3.3.18.

This code interleaves with some important functions introduced in the IPAM component to make all this work happen correctly. The following section will give some details about the changes done on the IPAM service.

6.4 Changes to the IPAM component

The IPAM component stores all the required data about remappings operated by the local cluster toward remote ones, and vice versa. Among this information, it also contains the remappings between induced peers, something that is fundamental for the correct EndpointSlice resource reflections.

In addition to that, some logic has been introduced to support the induced peering case when translating IP addresses from one address range to the other. Listing 6.7 reports the most important updates to the IPAM component. Some parts have been simplified for convenience.

Listing 6.7: IPAM

```
1 func findClusterID(ip string, subnets map[string]netv1alpha1.Subnets)
      (clusterID, clusterName string) {
2     for clusterID = range subnets {
3         doesBelong, err := ipBelongsToNetwork(ip, subnets[clusterID].
      RemotePodCIDR)
4         if err == nil && doesBelong {
5             return clusterID, subnets[clusterID].ClusterName
```

```
6           }
7       }
8       return "", ""
9   }
10
11  func (liqoIPAM *IPAM) getClusterIdentity(ip string) (clusterID,
        clusterName string, err error) {
12      parsedIP := net.ParseIP(ip)
13      if parsedIP == nil { /* error management */ }
14
15      liqoIPAM.mutex.Lock()
16      defer liqoIPAM.mutex.Unlock()
17
18      // Get all subnets
19      subnets := liqoIPAM.ipamStorage.getClusterSubnets()
20
21      // Find cluster ID of the cluster the ip address belongs to
22      // In case clusterID == "", return it as is and don't return an
        error
23      clusterID, clusterName = findClusterID(ip, subnets)
24
25      return
26  }
27
28  func (liqoIPAM *IPAM) doesClusterMappingExist(clusterID string) (
        doesExist bool) {
29      subnets := liqoIPAM.ipamStorage.getClusterSubnets()
30      _, doesExist = subnets[clusterID]
31      return
32  }
```

Chapter 7

Evaluation

This chapter shows a set of functional and performance tests that have been conducted to demonstrate the behavior of the induced peering solution compared to the normal peering one, as well as other measurements to confirm the induced peering was working as expected.

7.1 Functional tests

Functional tests have been conducted manually by using Google's microservices demo application [6]. This application, called Online Boutique, is a cloud-native microservices demo application consisting of a 11-tier microservices application. The application is a web-based e-commerce service where users can browse items, add them to the cart, and purchase them.

Google uses this application to demonstrate use of technologies like Kubernetes/GKE[1] and gRPC[2]. This application works on any Kubernetes cluster, as well as Google Kubernetes Engine.

The Online Boutique demo application has been deployed in a three-cluster scenario, and functional tests have been carried out to verify its correct functioning, as well as that the offloaded Pods could communicate directly. In particular, this test was focused on the reflection of endpoints to the leaf clusters that were connected through an induced peering, and on the traffic measurements between them and the central cluster.

---

[1]Google Kubernetes Engine (GKE) provides a managed environment for deploying, managing, and scaling containerized applications using Google infrastructure.[7]

[2]gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment.[8]

The deployment of the application started on the central cluster, and thanks to the proper node affinity configurations, some pods were scheduled to one leaf cluster, other pods were scheduled to the other cluster, and the pod related to the application frontend was scheduled on the central cluster. For example, the cart service pod was running on the leaf cluster A, and its related Endpoints and Service resources were reflected to leaf cluster C: the Endpoints resource contained an IP address that logically pointed to the remote pod executing on cluster A. Similarly, the database pod was running in cluster C, and its related Endpoints and Service resources were reflected to cluster A, with the correct IP address configurations that logically pointed to the remote pod in cluster C. Thanks to this setup, pods were able to communicate directly without having their traffic exchanges pass through the central cluster. This has been verified by measuring the traffic on the central cluster: the results showed that the traffic was properly being exchanged directly between the two leaf clusters A and C.

7.2 Performance tests

For the performance benchmarking of the induced peering, the focus was on the measurement of the latencies between leaf clusters with respect to the adoption of the induced peering solution compared to the usage of the normal peering solution, as well as on the measurement of the time required to establish the peer-to-peer, full mesh network of clusters using the induced peering solution, with respect to the number of involved clusters that had a full peering session with a designated central cluster.

7.2.1 Latency measurements

For the purpose of measuring the latencies of a three-cluster topology, where one cluster works as a central cluster, a testbed was hosted by a Kubernetes cluster composed of six worker nodes, totally encompassing 332 virtual cores and 2 TB of RAM. The testbed leveraged the Liqo Benchmarks GitHub repository [9], which contains a set of tools to streamline the benchmarking of Liqo. In particular, the Liqo K3s[3] cattle tool has been used to deploy the three-cluster setup: this is an Helm chart that streamlines the creation of a given number of single-node K3s clusters on top of a pre-existing Kubernetes cluster.

The testbed was created to simulate a scenario whereby an organization controls a cluster in the West US region and activates two peering sessions with two other

[3]K3s is a highly available, certified Kubernetes distribution designed for production workloads in unattended, resource-constrained, remote locations or inside IoT appliances.[10]

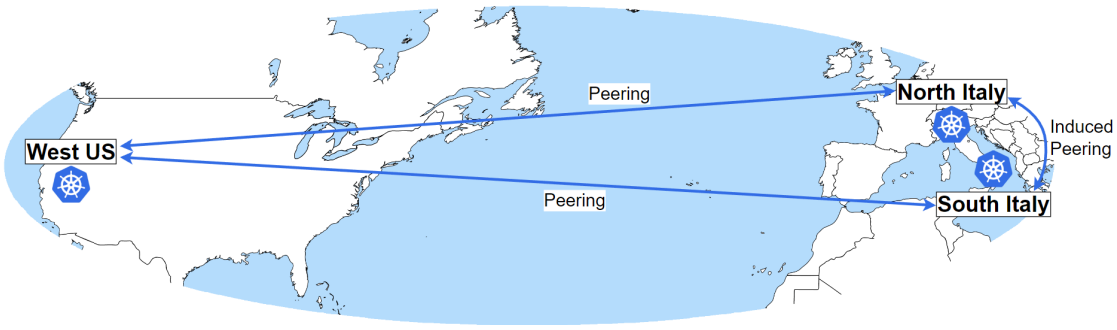

Figure 7.1: Three-cluster scenario: clusters spanning over multiple continents

clusters located in Europe, in particular in the regions of North and South Italy. Between those two clusters no active peering is established, but once the West US cluster sets up the two peering sessions (and becomes the central cluster of this topology), an induced peering is established between the two clusters in Italy. The company then deploys pods so that they get scheduled in the remote clusters of North and South Italy. Such pods are then able to communicate directly by exploiting the induced peering, with massive advantages in terms of latency. As presented in the following diagrams, the solution is compared to a traditional peering where pods' communications need to pass through the central cluster before hitting the other leaf cluster, resulting in high latencies even though the clusters are quite close to each other: the formed path is very inefficient, as it starts and ends in Italy, but passes through the USA.

To reflect the actual latencies of such an intercontinental communication, latencies have been added to this benchmark, so that the round trip time between the central cluster and the leaf clusters has been set to around 180 ms, as it would be between West US and Italy, while the latency between the two leaf clusters has been set to around 20 ms, as it would be between North and South Italy.

The following charts show the latency measurements and demonstrate the value provided by the induced peering in the effort of cutting down the delay of communications between remote pods belonging to the Italian regions.

Figure 7.2, reported for completeness, simply reveals that no differences are found in the latency between West US and the two Italian clusters, with or without the induced peering, as expected. The real value is shown in Figures 7.3 and 7.4, where the latencies between the two Italian clusters are drastically reduced to about 24 ms, compared to the 365 ms that a normal peering would produce as the sum of the two 180 ms latencies that characterize the path between Italy and West US, thanks to the usage of the induced peering network setup: indeed, their latencies are the ones normally observed between two close regions like the ones of North and South Italy.

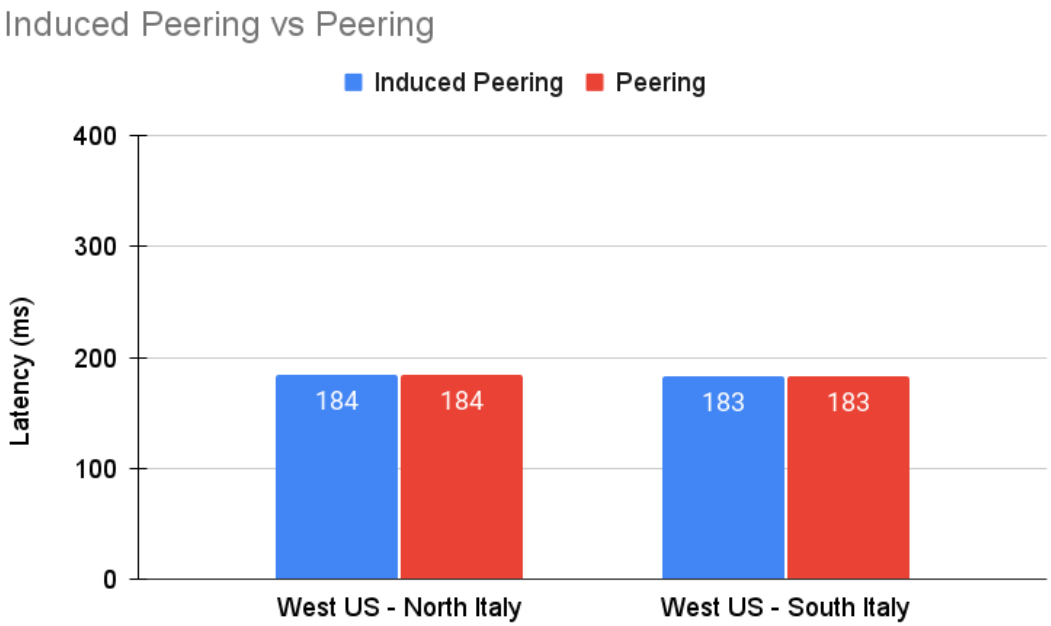

Figure 7.2: Latency between West US cluster and the other ones

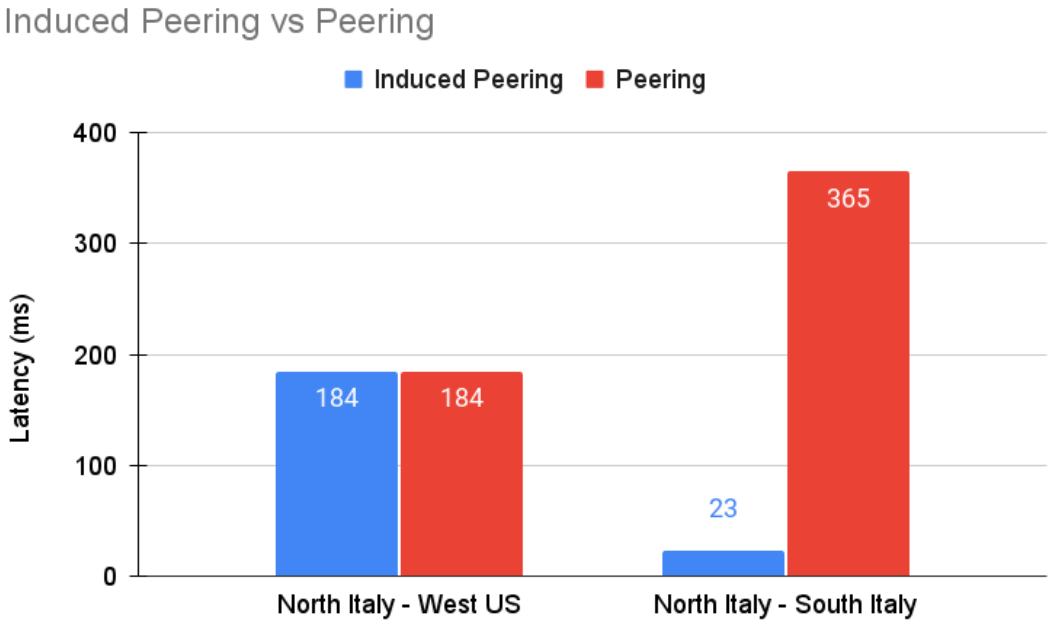

Figure 7.3: Latency between North Italy cluster and the other ones

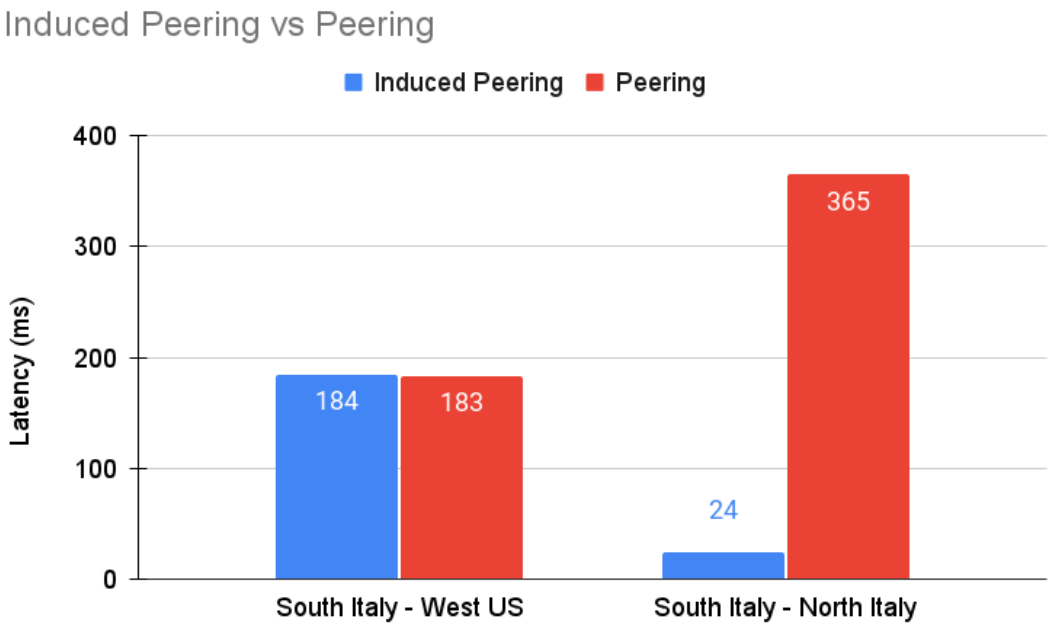

Figure 7.4: Latency between South Italy cluster and the other ones

The comparison highlights the great benefits of the induced peering in such a scenario, with an improved latency that is fifteen times better. Of course, the more the central cluster is closer to the leaf clusters, the less evident is the latency improvement. However, in general terms, the latency has a solid benefit from the induced peering, as it cuts off an intermediate step in the communication exchange between remote pods.

7.2.2 Time measurements

A set of measures has been conducted to analyze the behavior of the establishment of a full mesh of induced peers. The testbed was hosted by a Kubernetes cluster composed of six worker nodes, totally encompassing 332 virtual cores and 2 TB of RAM, running the Liqo K3s cattle tool from the Liqo Benchmarks GitHub repository, just like the one already mentioned. A growing number of clusters has been used in these measures, starting from the simplest three-cluster topology. In each experiment, one of the clusters has been designated as the central one, while all the remaining ones were the leaves of the tree-shaped topology. More precisely, by considering the connections deployed between every induced peer, so as to form a full mesh, the final topology becomes a graph. However, by only considering the full-fledged peering sessions that are established between the central cluster and

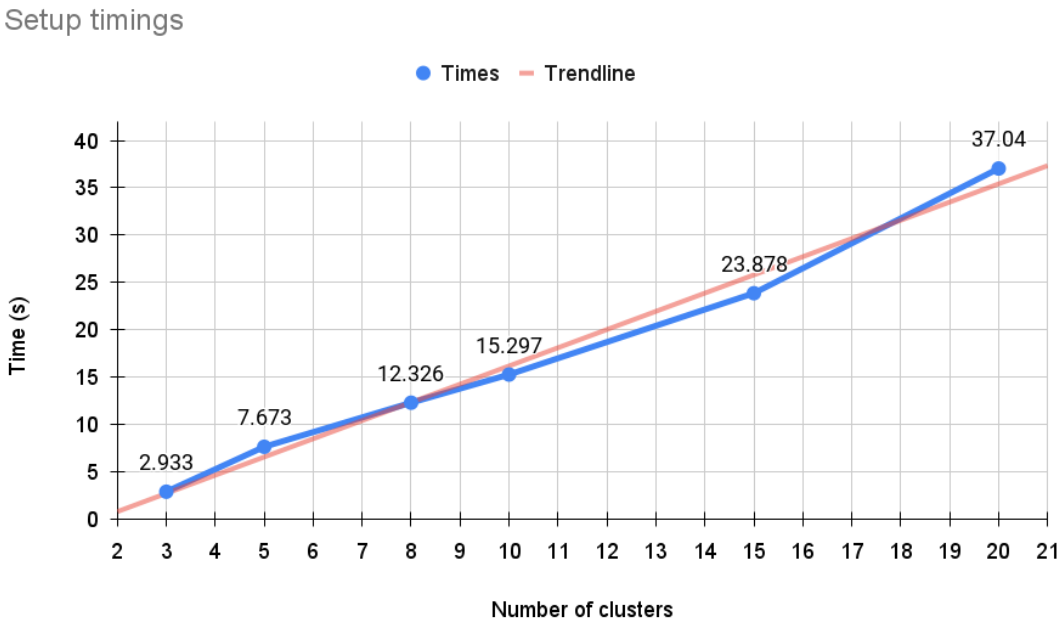

Figure 7.5: Time to setup a full mesh of induced peerings

each of the leaf clusters, the topology can still be seen as a tree of nodes, where there is a common parent node which is the central cluster.

The starting and final points in time among which the measures were taken are, respectively, the first moment in which the central cluster creates the first Neighborhood resource (which is sent over to the other clusters by means of the CRD Replicator), and the last moment in which one of the leaf clusters completes the creation of the last TunnelEndpoint resource that stores the VPN tunnel data.

Figure 7.5 reports the results of the time measurements. It highlights a linear growth with the number of clusters that together constitute the full mesh of induced peers, including the central cluster. Therefore, the solution scales linearly with the total number of deployed clusters, even though the number of passthrough NetworkConfigs handled by the central cluster and the total number of TunnelEndpoints that represent the full mesh of interconnections in the whole multi-cluster architecture both grow as the square of the total number of clusters.

It is worth noting that this result was obtained by activating all the normal peering sessions with the central cluster at the same time. Instead, in case the peering sessions are established progressively in several separate moments, the time required to extend the full mesh of induced peers at each progressive step decreases, as the number of resources to handle is lower. Moreover, this operation constitutes an initialization process, whose time span affects the initial moments

of the multi-cluster setup and no longer impacts the performance of the clusters when the peerings are fully operational, that is when the actual offloading phase starts and workloads can be scheduled remotely. However, to help with the timings and reduce them, it is possible to increase the concurrency of the reconcilers and of the CRD Replicator's workers, at the cost of increased consumption of cluster resources.

Chapter 8

Conclusions

This work constitutes a proof of concept that shows how it is possible to establish direct communications between offloaded pods without requiring a central cluster to forward the traffic, therefore avoiding the scalability issues that emerge on such a centralized setup. By means of induced peering and with the aid of a common central cluster, clusters can automatically discover remote peers and thus create the network connections that allow for lower latency pod-to-pod communications and less overhead on the central cluster.

As a proof of concept, it may require additional tweaks to smooth out some issues with the induced peering, such as the large number of NetworkConfig resources that are handled by the central cluster once it activates a large number of peering sessions with its remote peers. Other improvements may be directed towards the decrease of the time required to set up the full mesh of induced clusters, when their number increases, by means of increasing the concurrency on the controllers' reconcilers, which handle the various resources and CRs, as well as the CRD replicator's concurrent workers of the central cluster, which reflect an amount of NetworkConfigs that increases with the increase of the number of established peering sessions.

This work created a valid solution performance-wise, without losing the inherent ease of use of Liqo. From a user perspective, this solution can translate to an improved experience of the overall application. For a developer, it is completely transparent and does not require them to operate any change to their applications. From the standpoint of a cluster administrator, the induced peering represents an addition to their toolset they can use to carry out fine-tuning adjustments to their multi-cluster architecture.

Bibliography

[1] *Liqo GitHub repository*. URL: https://github.com/liqotech/liqo (cit. on pp. 1, 17).

[2] *Kubernetes documentation*. URL: https://kubernetes.io/docs/home/ (cit. on p. 4).

[3] *Kubernetes GitHub repository*. URL: https://github.com/kubernetes/kubernetes (cit. on p. 4).

[4] *Virtual Kubelet GitHub repository*. URL: https://github.com/virtual-kubelet/virtual-kubelet (cit. on p. 15).

[5] *Liqo documentation*. URL: https://docs.liqo.io/ (cit. on p. 17).

[6] *Online Boutique GitHub repository*. URL: https://github.com/GoogleCloudPlatform/microservices-demo (cit. on p. 57).

[7] *GKE documentation*. URL: https://cloud.google.com/kubernetes-engine/docs/ (cit. on p. 57).

[8] *gRPC*. URL: https://grpc.io/ (cit. on p. 57).

[9] *Liqo Benchmarks GitHub repository*. URL: https://github.com/liqotech/liqo-benchmarks (cit. on p. 58).

[10] *K3s: Lightweight Kubernetes*. URL: https://k3s.io/ (cit. on p. 58).