POLITECNICO DI TORINO

Master's Degree in Computer engineering

Master's Degree Thesis

Designing a scalable network overlay for Kubernetes multi-cluster topologies

Supervisors

Prof. Fulvio RISSO

Dott. Alex PALESANDRO

Candidate

Davide FALCONE

Academic year 2020-2021

Summary

Nowadays, more and more organizations leverage Kubernetes as a way to deploy containerized applications. In addition, they tend to own different clusters in order to increase their geographical reachability or avoid the vendor lock-in, in case of public clouds. Given that, an efficient management of multi-cluster should be crucial in companies. This thesis analyzes the limitations existing on an already started project that enables the creation of multi-cluster environments when it deals with deployments spanning on more than two clusters. Indeed, the current solution experiences different issues when deploying such applications: the first deals with the network architecture of the solution that does not provide a way of communication between 'Spoke' clusters while the second is about how Pods are advertised in such clusters. The thesis proposes a new design based on an Hub and Spoke topology coupled with a new logic in the control-plane to advertise Pods and describe how it has been implemented. Despite the solution can be further improved with a Peer-to-Peer topology (analyzed as well), it is a good starting point to improve multi-cluster management and make it available in production environments.

Table of Contents

Chapter 1

Introduction

There are no doubts on the fact the internet is the main actor of the current digital age: many of the services we rely on every day are hosted in data centers far away and the remarkable steps forward done by network technologies allow us to seamlessly enjoy them. The COVID-19 pandemic has exasperate our dependency to be connected, as most of the activities originally carried out in person have to be designed by scratch in order to be done at home.

Studies on virtualization, operating systems and networking conducted during the last decades, permitted a new paradigm to become crucial in this context, its name is *Cloud Computing*. It enables computing resources to be delivered on-demand through the internet (i.e. 'the cloud') promoting elasticity, availability and separation of concerns. Therefore, a company could either rely on a public cloud, offered by platforms like *Amazon Web Services*, *Microsoft Azure* and *Google Cloud Platform* and on a private cloud, enhancing security and privacy.

As cloud computing gained more and more importance, the current tendency is to engineer applications to be cloud native: they are no more monolithic systems as they used to be, but instead they are a set of loosely coupled microservices interacting together. A technology that broke through the cloud market is Kubernetes, an open-source project proposed by Google to automate the deployment, the scaling and the management of containerized applications over a cluster, which is a set of computing nodes.

1.1 The need of multi clusters

Although cloud solutions have given companies the opportunity to remarkably improve the quality of their delivered services, some of them have foreseen cloud limitations and have started focusing on how to overcome them or how feasible workarounds would look like.

1.1.1 Network latency

Suppose a company has to deploy an application to Japanese users but it owns a data center located in United States: in average, an user requests would travel nine thousands kilometers to reach the server and other nine thousands kilometers for coming back (the time of request computation in the data center is clearly negligible). This example is intended to present that network latency is a major problem in cloud computing, especially if an organization manages a business based on real-time solutions, such as autonomous driving, where also nanoseconds can make the difference. In such contexts, it would be much better to move the computation near to the end user: the tendency to avoid long round trip time by shifting the place where requests are served not so far from where they are generated is called *Edge Computing* and is gaining more and more importance, as edge devices or edge servers with a built-in CPU are experiencing a notably proliferation.

1.1.2 Reliability and availability

Although nowadays major cloud providers can provide extremely high percentages of availability of their services, it is well known that also minutes of down-time can cause thousands of dollars of revenue loss for organizations [1]. In order to prevent similar issues, companies should rely on data centers located in different geographical areas: in this way, if one set of servers experience a problem, this will not result in a period of unavailability of the services offered by the company. Moreover, with such architectures, applications deployed on the cloud can scale better and therefore give their user a better experience.

1.1.3 Vendor lock-in

Netflix and similar media content delivery platforms are able to give personalized recommendations to their users: your watch history is analyzed to suggest content you may also like. However, after users have enjoyed their platform for a while and presumably consumed the most attractive content, they may want to switch to another provider. This transition would force them to train from scratch the new platform so that it can deliver fitting suggestions, and this can be troublesome for some end users. This is a simple example of a phenomenon called *Vendor lock-in*: is a mechanism which makes customers dependent on a specific product or service [2] and can take place in different markets, Cloud Computing is not excluded. Indeed, providers can make changes on their product offerings in a way they no longer meet the customer needs, the quality of these products may decrease or prices imposed by vendors can drastically rise up. Nevertheless, organizations are not willing to pay the cost of cloud migration, as it can be severe. Building a multi cloud environment, either public or hybrid, is doubtless a straightforward way to tackle the root problem.

1.1.4 Resource management

Cloud architectures are designed starting from a series of user requirements, such as availability (e.g. RPO[1]/RTO[2]), performance (e.g. number of transactions per unit of time) and so on. Consequently, they are built by considering also the upper limit on the amount of users to be served at the same time. This is obviously the worst case and hopefully the cloud won't experience such a traffic most of the time, leading to a resources underuse during normal traffic load periods. The possibility to make these resources available for external jobs would be surely appreciated by companies, as they could use them for a different service or share them to third parties, drafting kind of contracts.

[1]Recovery Point Objective (RPO) is the maximum targeted period in which data (transactions) might be lost from an IT service due to a major incident.[3]

[2]Recovery Time Objective (RTO) is the duration of time within which a business process must be restored after a disaster.[3]

1.2 Liqo

Leveraging a multi cluster environment is becoming a crucial exigency for a lot of companies. Moreover, Kubernetes is the de-facto standard for containerized environments, since half of organizations running containers use Kubernetes [4]. Given that, Liqo, an open source project started at the Polytechnic University of Turin, is definely a solution to consider for improving multi cluster management. Liqo enables "dynamic and decentralized resource sharing across Kubernetes clusters" [5] according to the official documentation, and is available both for private and most popular public clouds, which can be valuable to avoid vendor lock-in and improve service geographical availability. Liqo works in a completely transparent way for applications, indeed they do not need any modification. The project is compliant with different network providers[3] and provides also a mechanism of 'clusters automatic discovery', facilitating the peering procedure if clusters are on the same LAN.

1.3 The goal of the thesis

Liqo permits pods to be offloaded to 'remote clusters', i.e. clusters to which the 'home cluster' has an active peering session with. Thanks to a resource replication mechanism, the cluster administrator has the impression that pods are running on the home cluster, but they actually live on the remote one. However, Liqo currently supports only application deployments across two clusters, limiting the great potential of a multi cloud architecture. In brief, if Pods are offloaded on different remote clusters, it is impossible for them to communicate one to another; this limitation of the Liqo's network module is in contrast with the microservice nature of modern applications whose components need to have an efficient way of interact. The work described in the following chapters aims at overcome this issue by introducing the support for applications spanning on more than two clusters, enabling a whole new set of scenarios to become reality. A company that owns different clouds, would use Liqo to improve the scalability and reliability of the services it offers. Large sized applications could be deployed on a collection of desktop computers or edge devices with low computing power,

[3]Network providers are external modules in charge of managing Kubernetes networking, in particular Pod-to-Pod and Node-to-Pod communication; they do not deal with Pod-to-Service communication. One of their responsibilities is to implement the CNI (Container Network Interface) that is the interface between containers and the network provider.

exploiting the inherent microservice architecture. This document will give the background knowledge necessary to understand the context and comprehend the problem to face and will explore the adopted solution. The final part will be reserved to tests and benchmarks on the obtained result.

Chapter 2

Kubernetes

2.1 From virtualization to container orchestration

Our journey begins in the 60's: The Beatles publish their first album, NASA sends the first man on the moon and IBM dominates the market of mainframes. These machines (in particular the *System/360* and 370 families) were capable of unprecedented levels of computing power and therefore were employed in various situations: businesses, universities and laboratories are some examples [6]. It was unfeasible for organizations to own a mainframe for each user, given the cost of buying and maintaining those computers. Therefore, the introduction of **virtualization** gave companies the opportunity to share the same machine among different users, each of them having a personalized and isolated working environment, called *virtual machine* (VM in brief). After a period of undisputed popularity, interest in virtualization dropped due to the wide diffusion of personal computers, way cheaper than mainframes.

The remarkable increase of computing power offerings and the failure of popular operating systems to provide performance predictability and configuration isolation for applications led to another period of huge adoption of virtualization. In 2006, Amazon released *Elastic Compute Cloud* (aka EC2), a web platform to deliver computing power towards VMs: this is the beginning of cloud computing.

Although virtualization has been widely adopted in that period, some people realized that some use cases would not need strong isolation as the one provided by

virtualization. Moreover, some were not willing to wait long booting time or keep operating systems on virtual machines always up to date. This gave rise to the birth of a new kind of virtualization, called **lightweight virtualization**, which is suitable when there is no need of different OSes and hardware profiles or the overhead of virtual machines is not acceptable.

Taking advantage of new Linux kernel features (in particular *cgroups* [1] and *namespaces* [2]), **Docker** was released in 2013. It is an open-source project that automates the deployment of applications in packages called *containers*, which are sort of lightweight virtual machines. Soon it became valuable to have an efficient management of containers and **Kubernetes** proved to be the best platform to achieve this result.

2.2 Overview

Kubernetes is an open source project proposed by Google in 2014 for automating the deployment, the scaling and the management of containerized applications. It is highly influenced by *Borg*, a system used by Google to schedule jobs on different computing hosts, indeed some engineers that developed Borg were also involved in the Kubernetes project.

Logically, Kubernetes is located on the top of a **cluster**, a set of machines composed by at least a master node and a worker node. Typically, computing jobs are scheduled on worked nodes, while the computing power of the master node is entirely dedicated to the control plane of Kubernetes, but this behavior can be modified. The system implements a control loop approach, as it continuously observes the current **state** of the cluster and properly reacts if it differs from the desired one. The state is entirely represented through a series of **resources**. Each of them is provided of a *spec*, which is a description of the desired state for that object and a *status*, that represents the current state. The control loop approach mentioned previously is enforced thanks to *controllers*, Kubernetes components that respond to changes in resources status, meanwhile the resource spec is often provided by the user.

[1] cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage of a collection of processes [7]

[2] namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources[8]

2.3 Concepts

2.3.1 Resources

The cluster administrator can check or manipulate the cluster state thanks to the API server, which is the front-end of the Kubernetes control plane. This component exposes an HTTP API consumed by end users for administration purposes or by other cluster components, allowing them to communicate with one other. The interface allows the typical CRUD[3] actions to be performed on a set of resources. An user who wants to create a resource, will provide such information in form of a *yaml*[4] file. An example is proposed below:

Listing 2.1: Kubernetes resource.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-first-pod
  labels:
    role: myrole
spec:
# Expected status of the resource
status:
# Current status of the resource
```

The type of the resource is defined univocally by the *apiVersion* field, that indicates the version of the Kubernetes API used to create this object and the *kind* field. The *metadata* section is used to help identifying the object instance and consists of the *name* field (one can use also *generateName* field to make the API server to choose the name starting from a string prefix) and optionally a namespace (it will be analyzed later on) as well as one or more *labels*, used to identify attributes of the object that are relevant to the user; resources can be queried according to one of their labels by taking advantage of *selectors*.

[3]Create, Read, Update, and Delete (CRUD) are the four basic operations of persistent storage.

[4]YAML is a human-readable data-serialization language. Its name is a recursive acronym: YAML Ain't Markup Language.

2.3.2 Pod

The Pod is the basic execution unit that you can create and manage in Kubernetes. A Pod is a group of one or more containers sharing storage and network resources, indeed a container can provide data to another and both of them can communicate through *localhost*. This shared context is possible because Pods have been engineered by taking full advantage of Linux namespaces and cgroups.

Pods provide a way to schedule highly coupled containers on the same node, avoiding the scheduler to start them on different servers, indeed containers belonging to a common Pod are automatically scheduled by the control plane on the same node. This feature results to be effective when the main application container needs some external services to be active and ready to be consumed. Pods allows external services to be embedded in additional containers, typically called *sidecar*. However, Kubernetes itself does not know anything about sidecars, they are a pattern to solve some use cases.

The situation is different for *init* containers, indeed the Pod resource itself allows the creation of those containers with a specialized field. They are run before the app containers are started and therefore are utilized to prepare the environment for them: cloning a Git repository or generate configuration files are some examples of usage.

In practice, is not likely that one creates a Pod on a Kubernetes cluster, except for testing purposes. They are usually created by means of different resources, such as *ReplicaSet* and *Deployments*.

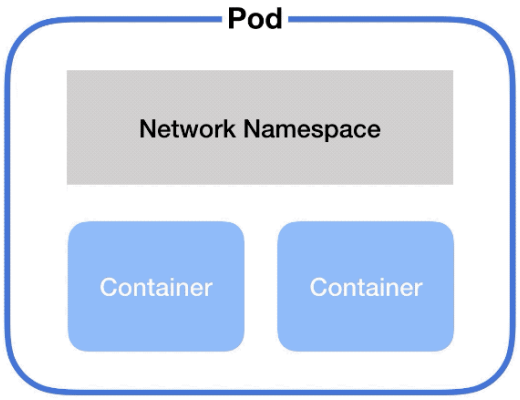

Figure 2.1: A Kubernetes Pod.

2.3.3 Job

When the user needs some Pods to necessarily complete their execution, it can leverage another Kubernetes abstraction. Jobs create one or more Pods and guarantee that a specified number of them correctly completes. A simple use case is a Pod that makes some kind of computation on each item of a queue and then completes. Jobs can also be waited for their completion.

2.3.4 ReplicaSet

The Cattle pattern [5] is implemented in Kubernetes thanks to the ReplicaSet object. It is used to maintain a stable set of Pods running at any time, so that if a Pod gets sick, it is suddenly replaced by a clone. Listing 2.2 shows that the spec section includes a replicas field (i.e. the number of Pods to deploy) as well as a template and a selector subsections. The former is a specification on how the resulting Pods have to be built; the latter enables ReplicaSet to own not only Pods specified in the template section, but also Pods whose label match the selector.

2.3.5 Deployment

Suppose a web application is deployed on a Kubernetes cluster; it is legitimate to assume the application is made by (at least) a back end (e.g. mysql) together with a front end (e.g. nginx). In order to enforce scalability and resilience, the cluster administrator has deployed both Pods through a couple of ReplicaSets, specifying the Docker images for containers in the template section. Unfortunately, a security issue is found on the version of mysql implemented in the image, therefore it is necessary to update it as soon as possible by creating a new ReplicaSet for the back end. The manual management of this kind of update can be cumbersome, especially if the organization owning the application wants to avoid downtime as much as possible.

In this situation, a Deployment can definely help: it provides controlled updates

[5] The Cattle pattern differs from the traditional Pets pattern in which servers running applications are unique and indispensable: when one server gets sick, it is nursed back to health. It is called Pets because typically names are used to identify server, as they were family pets. In the Cattle pattern instead, servers are almost identical, therefore when one gets sick, it can be easily replaced by another one, simplifying scaling and recovery.

Listing 2.2: A ReplicaSet yaml specification.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

for Pods and ReplicaSets. It owns almost the same fields of a ReplicaSet resource, indeed creating a Deployment automatically results in a ReplicaSet creation; the former will be entirely managed by the latter. Coming back to the previous example, if Deployments were used instead of ReplicaSets, an update of mysql would be as simple as issuing a command in a terminal:

```
kubectl set image deployment/mysql-deployment msql=msql:1.16.1
```

kubectl is a command line tool that embeds a Kubernetes API client and therefore allows the user to run commands against a cluster; mysql-deployment is the name of the imaginary deployment and msql:1.16.1 is the updated image name, hopefully without security weaknesses. Right after the command will be served by the control plane, Pods belonging to the old ReplicaSet will start terminating themselves, while a new ReplicaSet will spawn updated Pods. The whole process is automatically carried out in a controller way to avoid downtime or performance issues.

2.3.6 Horizontal Pod Autoscaler

The Black Friday is an informal term used to indicate the Friday following Thanksgiving day in US. Most people look forward to it, as many stores open very early and offer high sales on a wide set of products: this is the right time to upgrade your smartphone or buy the shoes you have been craving. It is legitimate to think that

whichever e-commerce website would experience a load peak during this period and nevertheless cluster administrators are expected to guarantee no down times or performance issues. Manually adapting a Deployment with a different number of replicas can definely help, but a careful monitoring of the load situation is mandatory to avoid resource under-use and maximize the user experience. The Horizontal Pod Autoscaler (HPA) has been designed to solve this exact problem, as it provides replica autoscaling. It is implemented via a resource and a controller; the former specifies the expected resource (e.g. CPU) utilization as well as the minimum and maximum number of replicas that the scaler is allowed to deploy; the latter periodically compares the target value with the current utilization and possibly it adjust the number of replicas. Like other Kubernetes resources, HPA is supported by kubectl: it is possible to create an autoscaler, listing all the autoscalers in the cluster or deleting an autoscaler. In addition, kubectl provides the autoscale command for creating a HorizontalPodAutoscaler object. For instance, executing:

```
kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80
```

will create an autoscaler for the ReplicaSet (that is what rs stands for) named foo, with target CPU utilization set to 80% and the number of replicas between 2 and 5 [9].

2.3.7 DaemonSet

DaemonSet are special Pods that are guaranteed to be in execution on each node of the cluster; this results in two main consequences:

- When a new node is added to the cluster, Kubernetes automatically schedules a new Pod on that node.
- When a node is removed from the cluster, the pod on that node is garbage collected.

In line with the Cattle model, people working with Kubernetes do not care about where Pods are scheduled, as nodes are all equal. However, some special use cases require a single Pod to be present in each node; these are some usage scenarios:

- Running a cluster storage daemon on each node.
- Running a logs collection daemon on each node.

- Running a node monitoring daemon on each node.

2.3.8 Service

Deployments and ReplicaSets guarantee a stable set of Pods is always running on the cluster; however Pods can be scaled up or down or they can be re-scheduled on a different node for many reasons. Moreover, in Kubernetes each Pod has its own IP address, hence if a Pod is deleted due to a scheduling decision, its IP address is no longer valid and it can even be used for other Pods. This leads to the problem of making Pods able to communicate.

Services have been proposed to address this problem, as they provide a way to expose a set of Pods. This mechanism is orthogonal with respect to the execution of Pods, indeed exposed ones, called *Endpoints*, are chosen through a label selector. Services do even more: contacting a Service results in a load-balancing among its Endpoints so that traffic is equally distributed among them.

kube-proxy is the Kubernetes component that makes Services work. It takes advantage of the Linux kernel's netfilter module, which is used to provide the user a way to specify firewall and NAT rules. Most of the times, kube-proxy is configured in iptables mode[6], this means that for each Service, Kubernetes chooses an IP address (typically called "ClusterIP") and kube-proxy adds DNAT[7] rules that capture the traffic to the ClusterIP and redirect it to the back end Endpoints. A *targetPort* field can be noticed by the yaml specification; this is the port on which exposed Pods are listening for incoming connections. The *port* field instead is related to the Service itself: connections to the Service ClusterIP on port 80 will be forwarded to Endpoints on port 9376 by kube-proxy.

Kubernetes supports 2 ways of finding Services: environment variables and DNS.

- Environment variables: when a Pod is run, Kubernetes adds a set of environment variables that facilitates the Service discovery. Consequently, if a Pod depends on a Service, the latter needs to be created before the former. The usage of the DNS method, allows the cluster admin to not worry about this aspect.

[6]kube-proxy can be configured to work in user space mode, iptables mode or IPVS mode.

[7]Destination NAT (DNAT) is a kind of NAT in which the destination IP address of a packet is changed so that traffic is redirected toward the new IP.

- DNS: typically, after the installation of a Kubernetes cluster, a cluster-aware DNS server should be deployed as well. It watches the Kubernetes API for new Services and creates a set of DNS records for each one.

Listing 2.3: A Service yaml specification.

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

A Service ClusterIP is valid only within the Kubernetes cluster; whichever client outside the cluster contacting an existing ClusterIP won't receive any response. This is the default mode of operation for Services, called ClusterIP. In case a Service has to serve requests coming from outside the cluster, a *NodePort* or *LoadBalancer* Services should be used. A NodePort type exposes the Service on each node at a static port (the NodePort); while a LoadBalancer type exposes the Service externally using a cloud provider's load balancer.

2.3.9 Ingress

Suppose an organization wants to publish a set of web pages but it owns only one public IP address. Typically, the company would have built a reverse proxy[8] configured to dispatch the traffic to internal servers, according to the URL. In Kubernetes such behavior is provided by means of Ingresses and Ingress controllers: they are used to expose HTTP/HTTPS routes form outside the cluster to Services within the cluster. The controller acts as a reverse proxy, and incoming traffic is load balanced between Services according to policies defined in Ingress resources. Unfortunately, Kubernetes does not include an Ingress controller, therefore the cluster admin has to deploy one on its own, such as ingress-nginx[10], depicted in

[8]Reverse proxy is a type of proxy that load balances client requests to one or more servers according to different policies. It can also keep cached copies of static content in order to further reduce the load on internal servers.

Figure 2.2. The following is an example of an Ingress resource that forwards traffic coming to the controller with URL ending in /testpath to the Service named test.

Listing 2.4: An Ingress yaml specification.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

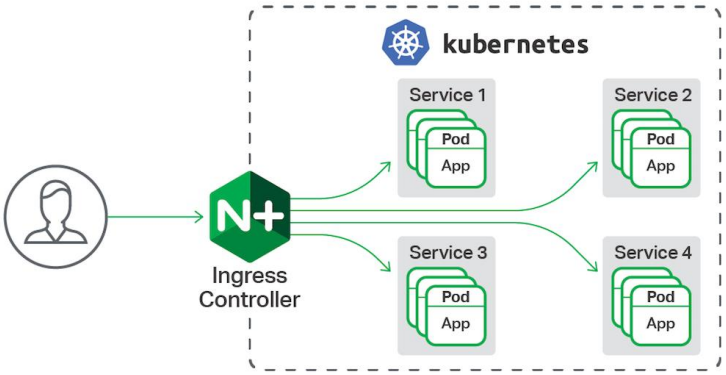

Figure 2.2: Ingress controller.

2.3.10 Namespace

Namespaces enable the creation of multiple logical clusters by using the same physical cluster and are intended to be used in environments with many users, spread across multiple teams or projects.

Kubernetes resources are divided in namespaces. Each resource can belong to a single namespace and resource names within the same namespace must be unique.

2.4 Components

After an insight on the abstract concepts that Kubernetes provides, it is time to look at the internal components that make all the system work. Figure 2.4 gives an overview of the Kubernetes architecture and suggests that it can be logically divided in 2 parts: the master components and the nodes components. The master components implements the control plane, the brain of Kubernetes and even if set up scripts typically run its components on the same machine, in production environments the tendency is to run the control plane across multiple computers. Node components, as the name suggests, run on every computing node and are in charge of managing Pods while following the directives of the control plane.

2.4.1 Master components

API server

The API server exposes the Kubernetes API, allowing end users and cluster components to inspect and alter the cluster state. The main implementation of the API server is kube-apiserver.

etcd

etcd is a distributed key-value store that provides Kubernetes a way to persist data; stored information is both related to running application and to the management of the cluster.

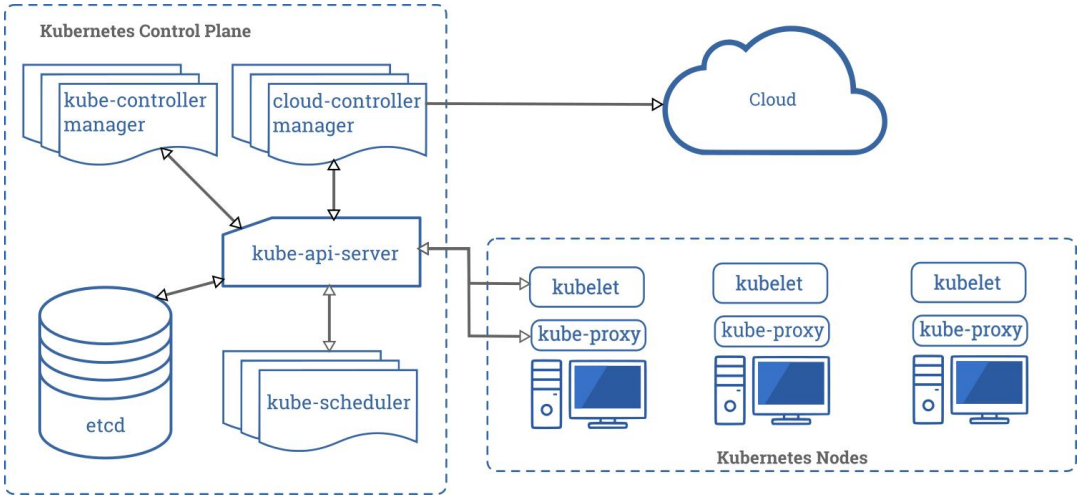

Figure 2.3: Kubernetes architecture.

kube-scheduler

As operating systems contain a scheduler component to dispatch processes on different CPU cores or plan their preemption, Kubernetes kube-scheduler makes decision on which node a Pod must be run. It takes in consideration the availability of node resources, data locality as well as node affinity Pod specifications.

kube-controller-manager

This component of the control plane manages the Kubernetes controllers. As mentioned in Section 2.2, controllers are control loops that aim at moving the current state of the cluster to its expected state. Kubernetes comes with a set of built-in controllers, that run in kube-controller-manager. Users can develop their own controllers and they can be run in Pods or externally to the cluster.

cloud-controller-manager

The cloud controller manager is the glue between the cluster and the provider, in case the cluster is deployed on a public cloud; indeed there is no need of such a controller for on-premise clouds. It links the cluster to the cloud provider's API, enabling vendors to release features at a different pace compared to Kubernetes.

2.4.2 Node components

Container runtime

The container runtime is the software consumed by Kubernetes to run containers. Examples are Docker, containerd and CRI-O.

kubelet

The kubelet is the component present in each node that ensures containers are running in Pods. If the API server is asked to create a new Pod, therefore the kubelet interacts with the container runtime of the node to run the containers indicated on the Pod specification.

kube-proxy

As discussed in 2.3, Pod to Service communication on Kubernetes is managed by kube-proxy. It is a proxy that redirects traffic toward Services to the appropriate Endpoints.

2.5 Related projects

As Kubernetes gain more and more importance, it becomes decisive for developers to interact efficiently with the platform and for the project to fit different use cases. Due to this necessities, a lot of Kubernetes' related projects have been kicked off in the last period and two of them are particularly relevant for this work: the Virtual Kubelet and Kubebuilder.

2.5.1 Virtual Kubelet

Virtual Kubelet is an open source kubelet implementation [11]. As kubelet is the primary agent that runs on a Kubernetes node, the virtual kubelet allows the creation of a virtual node and this is exactly the Liqo methodology to represent a cluster with an active peering session.

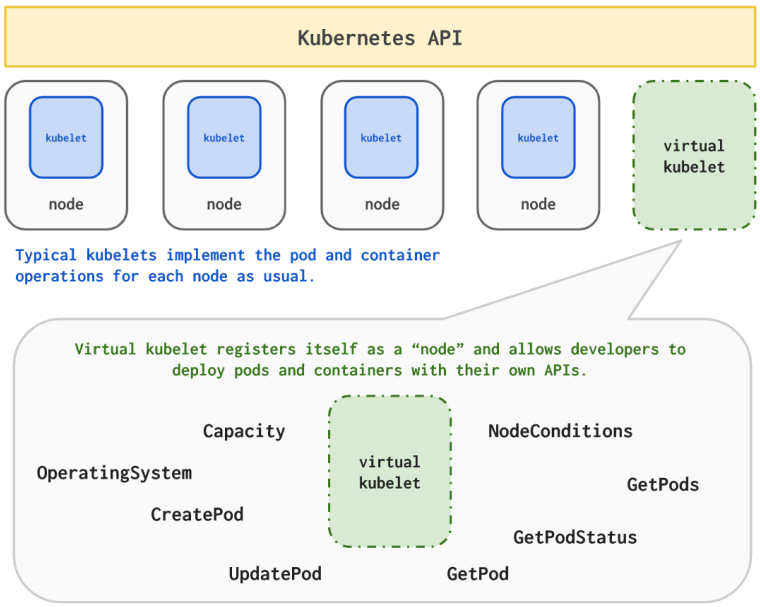

Figure 2.4: Virtual Kubelet.

2.5.2 Kubebuilder

Kubernetes permits and encourages developers to extend existing API through *Custom Resources*. After the installation, a custom resource can be managed via kubectl as happens for normal Kubernetes resources, like Pods. The deployment of a custom resource allows the programmer to store and retrieve structured data; a custom *controller* instead allows the creation of a *declarative API*: a way to declare a state for a resource and maintain that state. The practice of combining custom resources and custom controller is called **Operator pattern**.

Kubebuilder is an open source SDK[9] for rapidly building and publishing Kubernetes API in Go, being probably the most popular alternative to do so, thanks to its complete documentation and its simple abstractions for implementing API.

[9]A software development kit (SDK) is a collection of software development tools.

Chapter 3

Liqo

3.1 Overview

As discussed in Section 1.1, having an efficient multi cluster management can be valuable for several reasons and Liqo surely represent a valid candidate for achieving this result. This platform allows to run tasks on clusters with which the home cluster has an active peering session with. Peering sessions are obviously temporary and reversible: after a successful peering, it is always possible for a cluster to go back to the original state and stop the resource sharing. According to the Liqo terminology[12], the peering session results in the creation of a *Big Node*, a virtual node that represent and summarizes resources present in the remote cluster. Conversely, the home cluster becomes a *Big Cluster*, as it logically represent a cluster whose computing resources are sparse among several physical clusters.

3.2 Concepts

Discovery

The goal of the discovery phase is to detect other Liqo-enabled clusters (*Foreign clusters*) in order to gather all the information necessary for the peering phase. The cluster administrator can rely on three different methods for discovering new foreign clusters:

- **Manual configuration**: in Liqo, remote clusters are represented by ForeignCluster resources. The cluster administrator can forge manually a resource of such a type by identifying IP and port of the authentication service exposed by the remote cluster.

- **DNS discovery**: the documentation suggest that this mechanism is useful when dealing with multiple clusters, that are dynamically spawned and decommissioned. There are two steps to must be carried out for using DNS discovery: the administrator of cluster A need to register a set of records in its DNS server to let the cluster A be visible in its domain and the administrator of cluster B need to forge a SearchDomain resource that make Liqo perform periodical queries on the specified domains, looking for other clusters. In this case the ForeignCluster resource is automatically generated.

- **LAN automatic discovery**: Liqo can automatically discover foreign clusters on the same L2 broadcast domain of the home cluster. This mechanism is particularly suitable when the clusters owns a single node. As happens for DNS discovery, the ForeignCluster resource is automatically forger without any manual intervention. This mechanism takes advantage of the mDNS protocol.

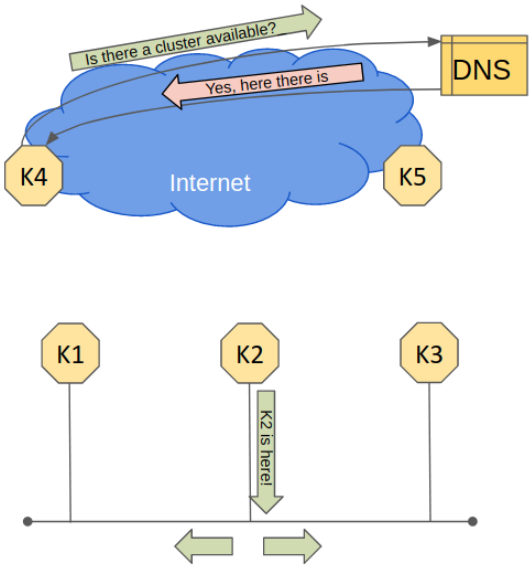

Figure 3.1: Liqo DNS discovery (on the top) and LAN automatic discovery (on the bottom).

Peering

There is no resource sharing before having completed a peering with a ForeignCluster. The peering process aims at establishing a connection with the foreign cluster, using data collected in the discovery phase. After a peering, clusters advertise their resource availability one to another; in this way clusters can deploy Big nodes representing peered clusters and what they want to share. It is important to notice that Liqo has a peer-to-peer architecture, therefore it does not conceive the idea of a master or server cluster.

There are three different types of peerings:

- **Incoming peering**: the foreign cluster can schedule Pods on the home cluster.
- **Outgoing peering**: the home cluster can schedule Pods on the foreign cluster.
- **Bidirectional peering**: both Incoming and Outgoing peering are active.

As Figure 3.2 shows, a cluster for which a ForeignCluster resource exists (i.e. it is discovered), is not necessarily a peered cluster. Thus, peerings can be manually activated and deactivated at any time, according with the administrator preferences or the cluster resource availability.

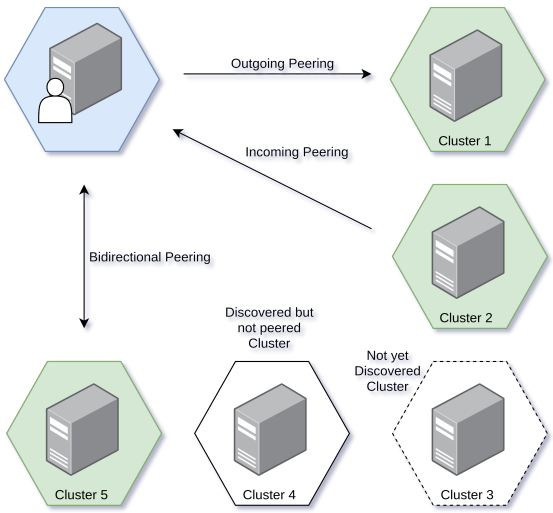

Figure 3.2: Liqo peering phases.

Networking

Among the data that clusters exchange during the peering phase, there are also what Liqo calls *NetworkConfigs*. The NetworkConfig is a Kubernetes resource created by a cluster that contains relevant information about the its networking configuration. It is by means of this resource that clusters know about the other's PodCIDR[1], for example. Moreover, the Liqo networking module is able to manage overlapping PodCIDR networks.

The Liqo control plane contains also a Gateway Pod, that acts as a VPN tunnel terminator: traffic between peered clusters passes through the clusters' Gateway Pod.

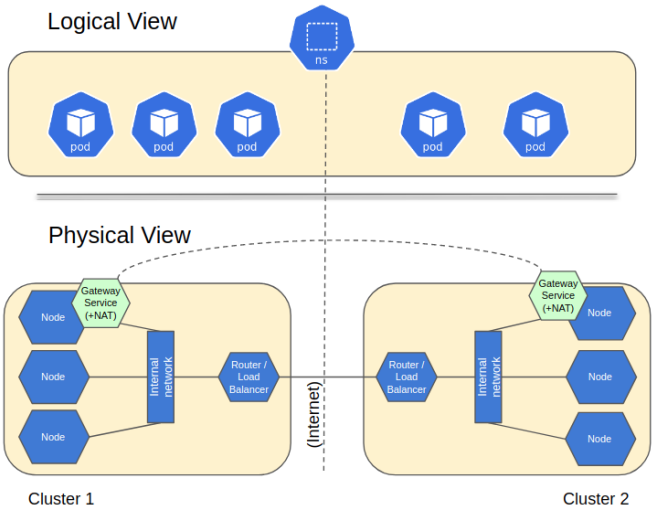

Figure 3.3: Liqo networking.

Offloading

After a peering with a foreign cluster, the home cluster can offload Pods and taking advantage of the newly available resources. Once a Big node is deployed, the Kubernetes control plane will take care of the rest, as it will see the Big node as a normal cluster node. If the kube-scheduler chooses the Big node as the best place to schedule a Pod, Liqo will create a new Pod in the foreign cluster. Again, the

[1]The network used for assigning IP addresses to Pods in a Kubernetes cluster.

Kubernetes control plane will select the appropriate node in the remote cluster to run the Pod in.

Even if a Pod is actually run in a remote cluster, Liqo creates and keeps up to date a local copy of the Pod, giving the impression that the Pod lives in the home cluster and facilitating its management by the user. A deletion of the local copy (that Liqo calls *Shadow Pod*) will result in the deletion of the corresponding remote Pod. On the contrary, the deletion of the remote copy, will result in the creation of a new remote copy, as Pods' life cycle is controlled by remote ReplicaSets.

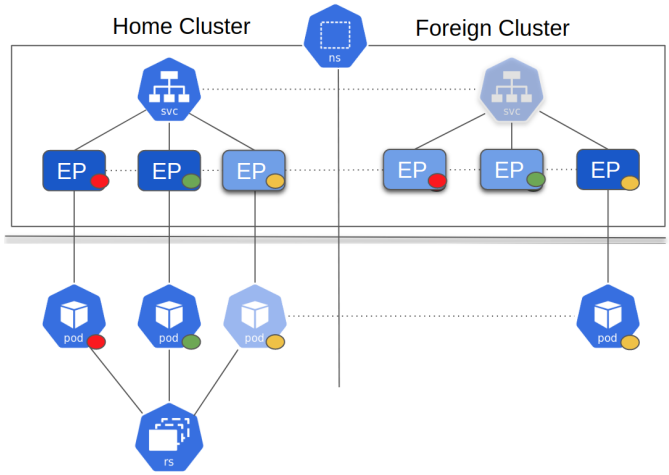

Figure 3.4: Pod shadowing.

3.3 Components

Liqo is mainly written in Go, as Kubernetes has been developed in the same programming language. It can be installed with Helm[2] and the cluster administrator need to pass some configuration data during installation, according to the peering he is willing to establish. The installation creates a series of Pods (they are actually Deployments for resiliency reasons) that compose the Liqo control plane.

Auth component

This component authenticates incoming peering request; thus is exposed via a NodePort or LoadBalancer service or even through an Ingress, as it has to be accessible by remote clusters.

CRD replicator component

As its name suggests, the CRD replicator replicates relevant local custom resources on peered clusters. For achieving this task, the API server of the foreign cluster must be accessible by the home cluster.

Discovery component

The Discovery component looks for new Liqo-enabled clusters. Therefore it embeds a mDNS Server coupled with a DNS and mDNS clients.

Gateway Component

As mentioned previously, the Gateway component is a VPN tunnel terminator and traffic between peered cluster passes through this component. It also manages firewalling and NAT rules: for example, it permits communication between clusters with overlapping PodCIDR by inserting appropriate NAT rules. The Gateway component is implemented with a Pod sharing the host's network namespace.

[2]Helm is an open source package manager for Kubernetes.

Network manager Component

This component manages the network configuration of a peering, for example storing the PodCIDR of a foreign cluster or assigning a new one if conflicts are found.

Route component

Traffic generated in the home cluster and directed to foreign clusters need to be forwarded to the Gateway component. In order to do so, routing tables of cluster nodes have to be modified: the Route component is a DaemonSet that manages routing tables of each node of the cluster. This can be achieved by sharing the network namespace with the worker node, similarly to the Gateway Pod.

Virtual Kubelet

This is a custom version of the Virtual Kubelet project. Whenever the home cluster establish a peering, a new virtual kubelet instance is spawned for the remote cluster. It is used to schedule tasks on the remote peer, as Kubernetes control plane has the impression it is a real cluster node and also to reflect core Kubernetes resources among clusters: a Service created on the home cluster will be reflected on the remote cluster, for example.

Chapter 4

Multi cluster deployments: Design

4.1 Liqo Service reflection

As services are the way Kubernetes provide to expose Pods, they are one of the most important abstraction of the system and they are highly used in applications. In order to be compliant to such a wide adoption of services, Liqo is expected to support them, and so it is. The virtual kubelet permits to make remote copies of core Kubernetes resources, such as Services. Thus, to create a Service on foreign cluster A, it is necessary to create it on a Liqo-enabled namespace[1]of the home cluster and the virtual kubelet related to cluster A will take care of the rest.

Service Endpoints can live either on the home cluster and on the foreign one: during the reflection, IP addresses of exposed Pods are translated according to where they are and where they are going to be reflected, as shown in Figure 4.6. This example assumes that during the peering procedure, cluster A has made a mapping between the PodCIDR of cluster B (10.244.0.0/24) and an available network (in this case 10.0.0.0/24), as the two clusters have the same PodCIDR. Thus, the virtual kubelet asked to reflect S1 on cluster A performs a 1:1 mapping of the Endpoint IP (running in B) to the remapped network. Conversely, if the Service created in B have exposed a Pod living in A, the IP would have been

[1]Liqo-enabled namespaces are special namespaces for which the resources offloading is enabled.

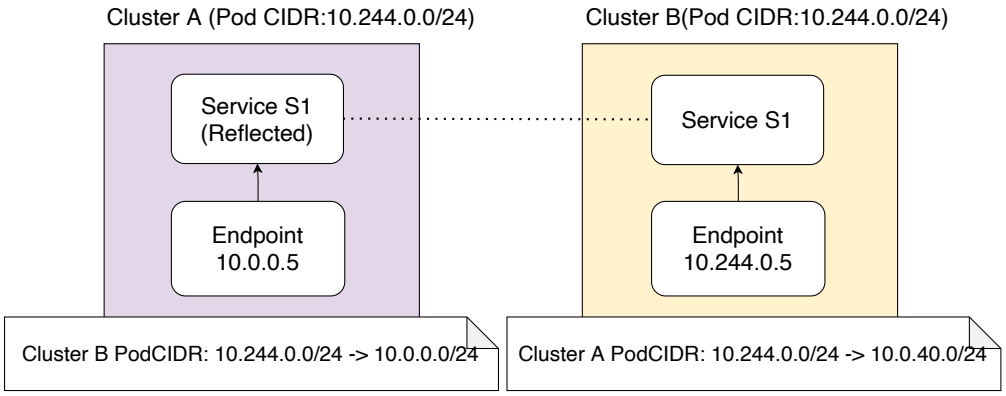

Figure 4.1: Endpoint reflection.

mapped in 10.0.40.0/24. Once the Service and the relative Endpoints are reflected from the home cluster to the foreign one, they become available for remote Pods: kube-proxy inserts NAT rules to catch traffic toward the Service and redirect it to the Endpoints, as would happen for a normal Service.

4.2 Liqo Pod-to-Pod connectivity

Thanks to the work of kube-proxy, a Pod-to-Service communication problem is translated in a Pod-to-Pod communication problem. Thus, the next step is to analyze how Pods in different clusters talk to each other.

Most of the work is carried out by the Gateway. Indeed, in order to make Pods communicate this complex component performs:

- Single NAT: if one of the two clusters has remapped the PodCIDR of the other.
- Double NAT: if both clusters have remapped the each other's PodCIDR.

Figure 4.2 shows an example of Single NAT: the translation happens on both clusters and the type (DNAT or SNAT) depends on the packet direction. Assuming b is the mask length of networks, it can be noticed that new IP addresses are forged concatenating the first b bits of new network CIDRs and the last 32-b bits from the old IP addresses: it is a NAT with the host part of the IP that remains unchanged. In case of Double NAT, both DNAT and SNAT are performed on both clusters.

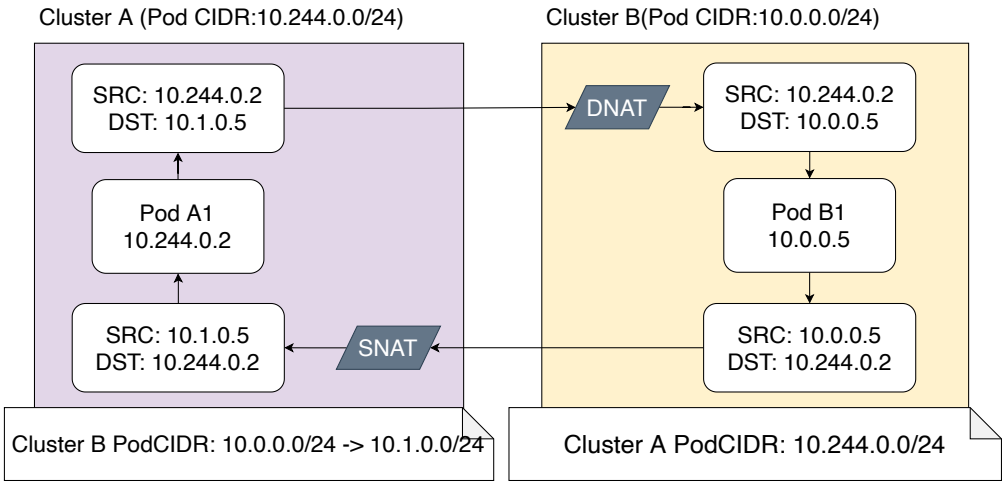

Figure 4.2: Pod-to-Pod communication with Single NAT.

After being NAT-ted, traffic is forwarded to the remote cluster; currently Liqo achieves this task by using WireGuard [13], a modern VPN that is secure and easy to configure. WireGuard uses the same network interface to communicate with any of its Endpoints, which permits to scale better when the number of peerings becomes larger.

4.3 Problem definition

Applications deployed on a Liqo-enabled namespace work well and smoothly, providing that they are spanned across the home cluster and only a foreign cluster. Problems arise when Pods are scheduled on different Big nodes, making an application live on more than two clusters. Consider the situation depicted in Figure 4.3, in which an active peering session is configured between Clusters A and B, and the latter peered also with Cluster C. Service A exposes Pod C1 and can be contacted without any problems from Pods living both on B and C. But what if a Pod of Cluster A, such as A1, would like to make a request to the same service?

Endpoint IP addresses are translated as described in Section 4.1, thus the IP address of exposed Endpoint C1 10.1.0.5 is mapped on 10.0.0.5 as Cluster B (which carries out the reflection) knows its Pod network has not been remapped by Cluster A and is still 10.0.0.0/24. After kube-proxy has configured NAT rules on A, any packet toward Service A will be forwarded to 10.0.0.5. Since a Liqo peering exists between A and B, traffic generated A's Pods and directed to network 10.0.0.0/24

is sent to Cluster B. At this point, 2 circumstances can happen:

- If there are no Pods on B with IP address equal to 10.0.0.5, the packet will be dropped.
- If there is a Pod on B with IP address equal to 10.0.0.5 (in Figure 4.3 is B1), the packet will be sent to that Pod.

In any case, the packet will not reach its expected destination, Pod C1. The first cause of the issue is the Service reflection mechanism: it can deal with Pods living on the home cluster and on the cluster in which the reflection will happen, but it is not capable to deal with Pods living on other clusters because the Virtual Kubelet, in charge of performing the reflection, treats at the same way local Pods and Pods living on a third cluster. Indeed, in the proposed example Cluster B tries to reflect on Cluster A an Endpoint living on Cluster C. Even if a peering session is active between Cluster A and C, applications are not guaranteed to work on this topology. A Liqo cluster maintains information only about its own peerings and is completely unaware of how peerings have been established between other clusters or how they have remapped each other Pod networks, therefore the virtual kubelet would not be capable of translate IP addresses.

Secondly, it is missing a channel of communication between clusters that do not have a direct peering but do have a peering with a common cluster. This results in the fact that even if Endpoints would have been reflected correctly, the application still did not work. Referring to the example in Figure 4.3, Cluster A knows how to reach Pods running on B, but does not know how to reach those running on C, as

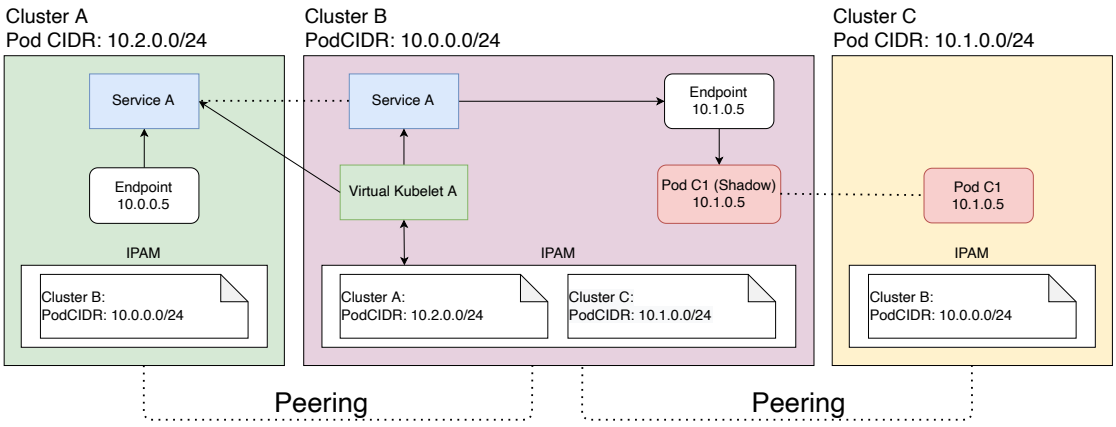

Figure 4.3: Application spanned on 3 Liqo clusters.

A and C do not have an active peering. Thus, if B would have reflected C1 in the right way, there were still no possibilities for A1 to send a packet to C1, as A was not capable of treat packets with that destination.

4.4 Solution overview

The solution proposed in the present work takes advantage of a new network that is used by clusters when reflecting Pods that are not running either on the home cluster and on the cluster the reflection is taking place, but living on a third cluster. As it is used when reflecting external Pods and, more in general, **external resources** (e.g. processes running on a worker node), it will be referenced from now on as *ExternalCIDR*. As Figure 4.4 shows, now clusters own both a network for their own Pods, the PodCIDR, and another one for external resources that are going to be reflected on some remote cluster, the ExternalCIDR. Information on both networks are exchanged among clusters during the peering mechanism and possibly the ExternalCIDR is remapped to another network if conflicts are found with used networks, as it already happens for the PodCIDR. In addition, clusters are configured to send to a remote cluster both the traffic directed to the foreign cluster's PodCIDR and ExternalCIDR. During the reflection of Service A1, Cluster B notices that the Pod it would like to reflect is an external Pod and therefore it maps the Endpoint IP address to another one, with the latter belonging to the ExternalCIDR and allocated exclusively for that Endpoint. It must be noticed that a random mapping is performed in this situation, differently with respect to

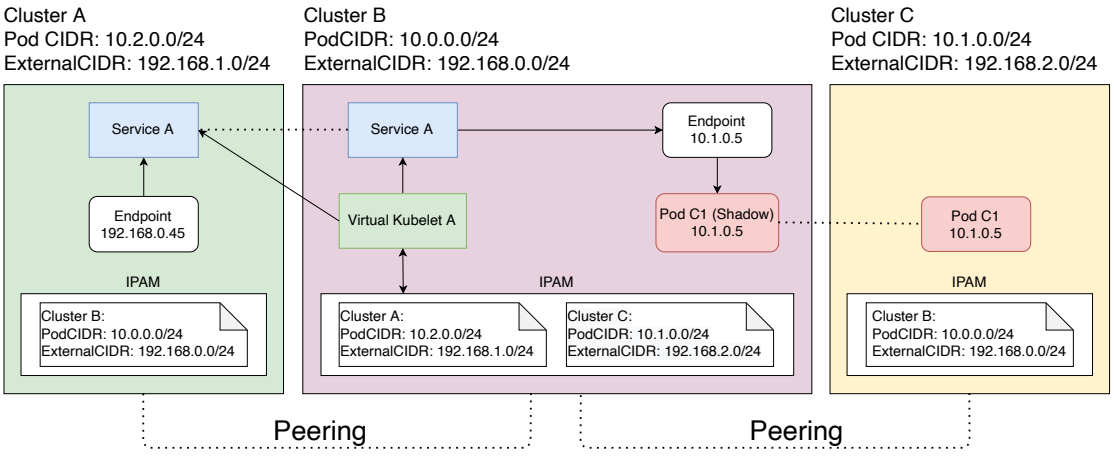

Figure 4.4: ExternalCIDR solution overview.

the traditional reflection, characterized by a 1:1 mapping of IP addresses. Cluster A has been configured in such a way that the packet generated by A1 and directed to 192.168.0.45 is forwarded to Cluster B. On its side, B is aware it performed the reflection of Pod C1 using the ExternalCIDR address 192.168.0.45 and therefore it redirects the received packet to that Pod. From now on, there is nothing new: since B and C have a peering session, B is perfectly capable of reaching Pod C1 which will receive the packet originally generated by A1. The opposite path works more or less in the same way.

Traffic directed to an ExternalCIDR IP have to go through the cluster that have reflected the Service and finally reach its real destination: it is an **star** topology. Since the proposed solution adopts this kind of network topology, it has to deal with the typical advantages and disadvantages of a start network. With respect to a **mesh** topology, in which clusters could communicate directly through a sort of network peering, this solution works by using only a peering for each cluster, resulting in network allocation savings. On the other hand, as all the traffic have to pass through the hub cluster, the solution could not scale if the number of clusters involved in the application deployment increases.

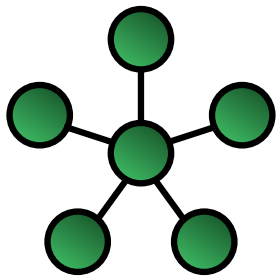

Figure 4.5: Star topology.

4.5 Solution design

The ExternalCIDR solution could be implemented by following 4 different steps:

1. Making Liqo clusters choose a network to be used as their ExternalCIDR and let them peer by exchanging not only each other's PodCIDR but also the ExternalCIDR.

2. Updating the reflection mechanism making it capable of use the new network for mapping IP addresses when reflecting external Pods.

3. Redirect traffic toward own ExternalCIDR to the appropriate external Pod.

4. Configure clusters to send traffic toward other's ExternalCIDR to the right foreign cluster.

Steps 1 and 4 are just a matter of replicate what already happens for PodCIDR also for ExternalCIDR. Conversely, steps 2 and 3 require new logic to be designed and implemented. In particular, step 2 proposes modifications to the virtual kubelet, which is the component in charge of performing the reflection. The virtual kubelet cannot complete the work only on its own since it cannot manage allocation of IP addresses which is something important to avoid that a single ExternalCIDR IP address is used to map two different Endpoint IP addresses and to efficiently manage the address space of the network itself, freeing an IP if it is no longer used. This task should logically be carried out by another component, which is embedded in the network manager: the IPAM module. Indeed, IPAM keeps track of used networks in the cluster and it is also capable of allocating and freeing IPs belonging to these networks. It's quite obvious that a mean of communication between these components will be necessary to complete the reflection, as the IPAM should provide available ExternalCIDR addresses to the virtual kubelet when the latter is asked to reflect a Service, and consequently an Endpoint, into a foreign cluster. Finally, the third step focuses on the Gateway module, as all the traffic reaching the home cluster go through this component and so the one directed to

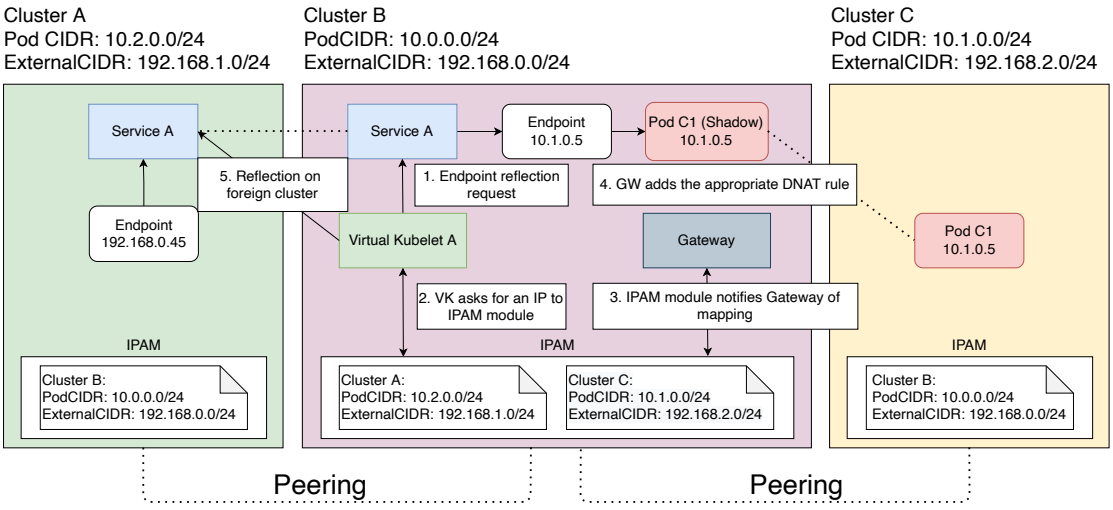

Figure 4.6: Endpoint reflection with ExternalCIDR.

the home cluster's ExternalCIDR does. However, these packets are not expected to be sent to a home Pod but to an external one; this means the Gateway has to catch them and redirect them to the appropriate Pod, by means of a DNAT operation. But how can the Gateway be aware of the IP address mappings carried out by the IPAM? These two components need to interact so that when the IPAM carries out a mapping between addresses, the Gateway is informed and can insert the appropriate NAT rules. Figure 4.6 depicts operations executed in steps 2 and 3: it can be easily noticed that they are quite dependent one to another.

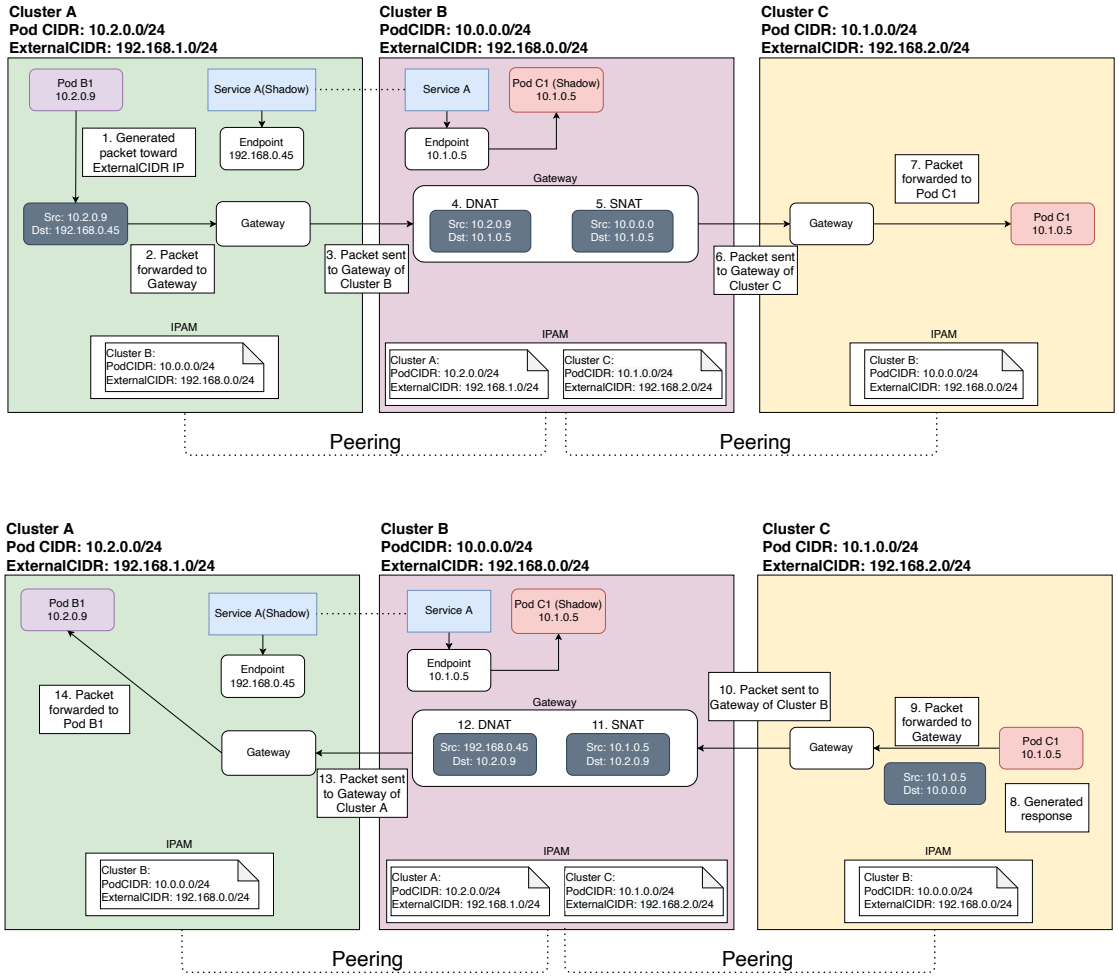

Figure 4.7: Flow of a packet directed to an ExternalCIDR IP.

Chapter 5

Multi cluster deployments: Implementation

5.1 ExternalCIDR configuration

Right after the installation of Liqo, networks used for Pods and Service ClusterIPs are announced by the Network Manager to the IPAM module, that in turn reserves them (i.e. mark them as used so that further network allocation requests do not return networks equal to those that are used or that are overlapping with them). The IPAM module adopts a custom resource, called *IpamStorage*, to persist its configuration: the PodCIDR and the ServiceCIDR, for example, are stored right there after their allocation. Only the Spec section of the resource is used because there is the only exigency to store information. In this way, if the Network Manager (that embeds the IPAM module) is rescheduled due to a failure or a decision of the Kubernetes control plane, networks are no more allocated, since the IPAM finds them in IpamStorage.
What has been done for the ExternalCIDR network is quite similar to the presented procedure, with the exceptions that is the IPAM itself that chooses the network for external resources among those that are available and returns it to the Network Manager. Indeed the latter needs this piece of information, along with the PodCIDR, to exchange network configuration with foreign clusters during the peering procedure.

Listing 5.1: Relevant cluster networks are stored in the IpamStorage resource.

```
// IpamSpec defines the desired state of Ipam.
type IpamSpec struct {
    /* ... */
    // Cluster PodCIDR
    PodCIDR string `json:"podCIDR"`
    // Cluster ServiceCIDR
    ServiceCIDR string `json:"serviceCIDR"`
    // Cluster ExternalCIDR
    ExternalCIDR string `json:"externalCIDR"`
    /* ... */
}
```

5.2 Peering

Peered clusters need to exchange network information in order to make their Pods talk to each other. The way Network Managers of clusters achieve this task is very clever. First of all, a different component comes to the rescue: the CRD Replicator. Its main goal is to make copies of local custom resources on foreign clusters and to maintain them always updated. When clusters are allowed to peer to each other, they produce a custom resource containing relevant network information, called *NetworkConfig*. A cluster creates different NetworkConfigs for each foreign cluster it did have discovered. The snippet 5.2 shows that this custom resource contains both a Spec section and a Status section. The Spec section is populated by the local cluster before it is replicated into the remote one and contains local cluster network information, such as PodCIDR, ExternalCIDR and VPN configuration parameters as long as the identity of the cluster the NetworkConfig is directed to. The identity of the cluster sending the NetworkConfig is instead included within the annotations. The remote cluster will inspect the received resource and will check if the indicated networks are available. At this point, two different circumstances can happen:

1. A network (PodCIDR and/or ExternalCIDR) is available, therefore the remote cluster reserves it for the home one.
2. A network (PodCIDR and/or ExternalCIDR) is not available because it is reserved or used for another cluster, therefore the remote cluster picks an available network and makes a 1:1 association between the two. In this case, traffic between clusters will be NAT-ted.

The Status section is managed by the remote cluster that uses it to notify the

Listing 5.2: The NetworkConfig custom resource.

```
// NetworkConfigSpec defines the desired state of NetworkConfig.
type NetworkConfigSpec struct {
    // The ID of the remote cluster that will receive this CRD.
    ClusterID string `json:"clusterID"`
    // Network used in the local cluster for the pod IPs.
    PodCIDR string `json:"podCIDR"`
    // Network used for local service Endpoints.
    ExternalCIDR string `json:"externalCIDR"`
    // Public IP of the node where the VPN tunnel is created.
    EndpointIP string `json:"EndpointIP"`
    // Vpn technology used to interconnect two clusters.
    BackendType string `json:"backendType"`
    // Connection parameters
    BackendConfig map[string]string `json:"backend_config"`
}

// NetworkConfigStatus defines the observed state of NetworkConfig.
type NetworkConfigStatus struct {
    // Indicates if this network config has been processed by the remote cluster.
    // +kubebuilder:default=false
    Processed bool `json:"processed"`
    // The new subnet used to NAT the podCidr of the remote cluster. The original PodCidr may have been mapped to this
    // network by the remote cluster.
    PodCIDRNAT string `json:"podCIDRNAT,omitempty"`
    // The new subnet used to NAT the externalCIDR of the remote cluster. The original ExternalCIDR may have been mapped
    // to this network by the remote cluster.
    ExternalCIDRNAT string `json:"externalCIDRNAT,omitempty"`
}

type NetworkConfig struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec   NetworkConfigSpec   `json:"spec,omitempty"`
    Status NetworkConfigStatus `json:"status,omitempty"`
}
```

local one about if and how networks have been remapped. In particular, fields PodCIDRNAT and ExternalCIDRNAT can have the following values:

- The string literal "None" if the network was available and was correctly reserved by the remote cluster.

- The CIDR of the network used to remap the network in case of address space conflicts.

After the local and the remote NetworkConfigs have been processed (i.e. their Status section is filled), the Network Manager proceeds with the creation of a new custom resource which represents the network interconnection of clusters; its name is *TunnelEndpoint* and can be thought as a merger between the two NetworkConfigs. This new resource is consumed by the Liqo Route component, in charge of forwarding traffic toward remote clusters coming from worker nodes to the Gateway, and the Gateway component itself, that has to route that traffic to the foreign cluster via the VPN tunnel, applying NAT rules if necessary. Figure 5.1 describes the steps that occur between the receiving of a remote NetworkConfig and the creation of the TunnelEndpoint resource; for brevity, only the ExternalCIDR is shown, but the same process happens at the same time also for the PodCIDR network. The IPAM maintains a map[1] that associates to a remote cluster ID a

Listing 5.3: IPAM stores peering network information.

```
// Subnets type contains relevant networks related to a remote cluster.
type Subnets struct {
    // Network used in the remote cluster for local Pods. Default is "None": this means remote cluster uses local cluster PodCIDR.
    LocalNATPodCIDR string `json:"localNATPodCIDR"`
    // Network used for Pods in the remote cluster.
    RemotePodCIDR string `json:"remotePodCIDR"`
    // Network used in remote cluster for local service Endpoints. Default is "None": this means remote cluster uses local cluster ExternalCIDR.
    LocalNATExternalCIDR string `json:"localNATExternalCIDR"`
    // Network used in local cluster for remote service Endpoints.
    RemoteExternalCIDR string `json:"remoteExternalCIDR"`
}
// IpamSpec defines the desired state of Ipam.
type IpamSpec struct {
    /* ... */
    // Map used to keep track of networks assigned to clusters. Key is the remote cluster ID, value is a the set of networks used by the remote cluster.
    ClusterSubnets map[string]Subnets `json:"clusterSubnets"`
    /* ... */
}
```

[1]Map is a built-in Go data type that implements a hash table; it maps keys to values.

struct containing four fields. Those fields are used to keep track of networks used in the home cluster for Pods and external resources of the foreign one (respectively RemotePodCIDR and RemoteExternalCIDR) and networks used in the remote cluster for local Pods and external resources that the local cluster can expose (respectively LocalNATPodCIDR and LocalNATExternalCIDR).

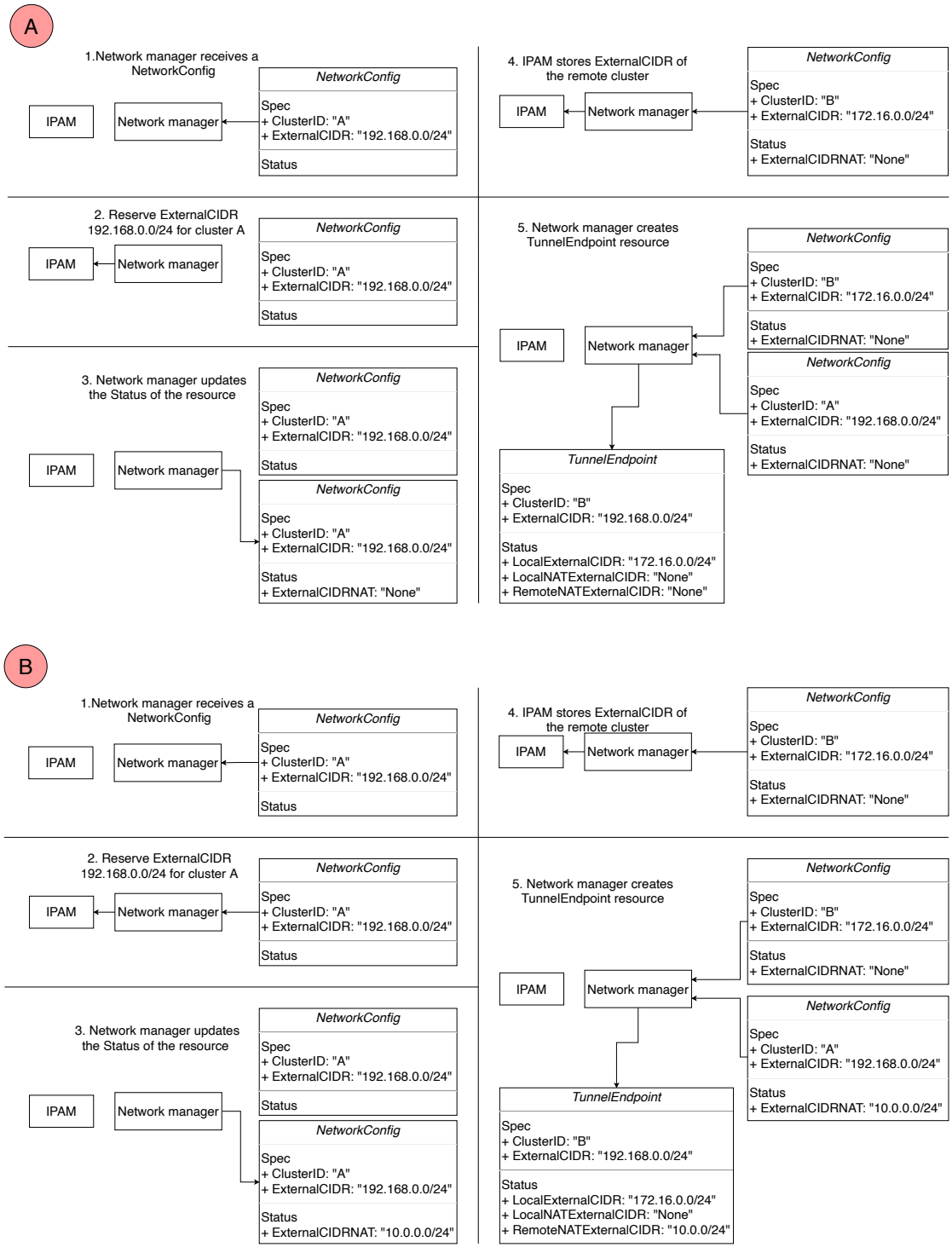

Figure 5.1: ExternalCIDR exchange between clusters when the network is available (A) and when it is not (B).

5.3 Reflection

Now it is time to focus on how Service Endpoints are reflected on foreign clusters. It is worth to remark that the Endpoints that are going to be reflected on a peered cluster x can live either on the local cluster y or on a third cluster z. The two cases have to be managed differently: the first one consists on the reflection of a local Endpoint; it uses the network information exchanged during the peering phase to translate IP addresses and it is already supported by Liqo (therefore, it will be later referenced with the name *traditional reflection*); the second one instead is the reflection of an external Pod and cannot be carried out similarly; the local cluster is not aware of a potential peering between the cluster where the Pod lives and the cluster on which the reflection has to take place. Even if such a peering existed and the local cluster was aware of it, the local cluster would not known how clusters can communicate one to another, as networks can have been remapped. As mentioned in the last Chapter, in order to face this issue the new ExternalCIDR network has to be used for Endpoints living on a third cluster. Figure 5.2 depicts what happen in both circumstances: the Virtual Kubelet relative to foreign cluster A does not deal with IP addresses and networks, so it needs the intervention of the IPAM module, which announces the IP to reflect according to the couple (Endpoint IP, Reflection destination Cluster).

In order to make the Virtual Kubelet and the IPAM module talk to each other, the Network Manager has to be exposed by a new Liqo Service. Then, different

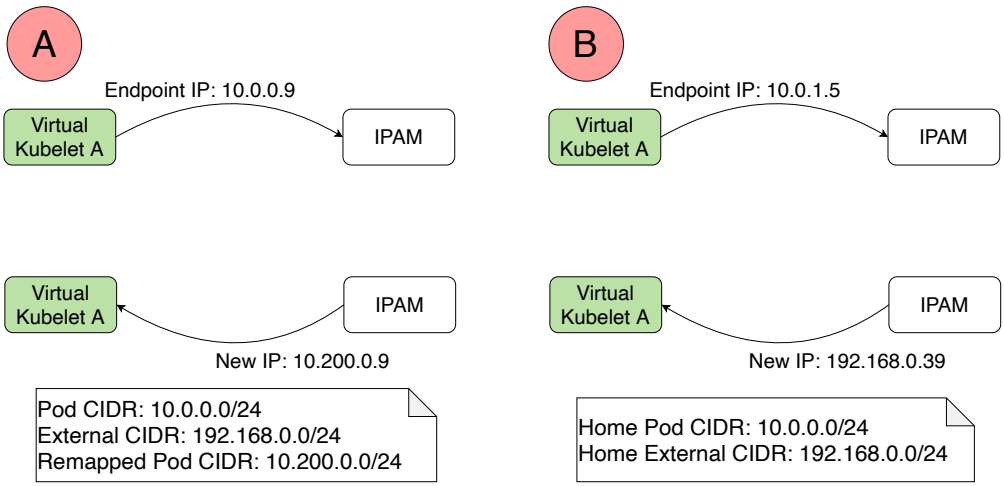

Figure 5.2: Endpoint traditional reflection (A) and Endpoint reflection with ExternalCIDR (B).

Listing 5.4: gRPCs exposed by IPAM module using Protocol Buffer language

```
/* ipam_gRPC.proto */
service ipam {
    rpc MapEndpointIP (MapRequest) returns (MapResponse);
    rpc UnmapEndpointIP (UnmapRequest) returns (UnmapResponse);
}

message MapRequest {
    string clusterID = 1;
    string ip = 2;
}

message MapResponse {
    string ip = 1;
}

message UnmapRequest {
    string clusterID = 1;
    string ip = 2;
}

message UnmapResponse {}
```

solutions could be adopted: it has been decided to let the IPAM expose a couple of gRPCs[2][14]. An alternative could be also to use a RESTful API. The prototypes must be declared in a file using an interface definition language, called Protocol Buffer. The file, whose extension is *.proto*, have to be compiled with a Protocol Buffer compiler that will produce the Go source code of the client and the server of the gRPCs.

Thus, the IPAM exposes two procedures;

1. MapEndpointIP: invoked by the Virtual Kubelet when starting the reflection of a Service and used to activate a mapping of an Endpoint IP address. This function also returns the new IP of the Endpoint.
2. UnmapEndpointIP: invoked by the Virtual Kubelet to signal the IPAM that the reflection is no longer active, and therefore to free resources if necessary.

Those calls are used in both situations resumed in Figure 5.2; is the IPAM itself

[2]A Remote Procedure Call (RPC) is a process of activate a procedure or a function in a different computer with respect to the one in which the caller process lives.

that know how to response properly to the request. Both requests, indeed, include the ID of the cluster where the reflection is going to happen, as long as the Endpoint original IP address. As regards responses instead, the first one contains the new Endpoint IP, the second one is empty. After having upgraded the Virtual Kubelet data structures with the gRPC client produced by the Protocol Buffer compiler, it is sufficient to use that client to make a MapEndpointIP request on every Service Endpoint add/update in a Liqo-enabled namespace and an UnmapEndpointIP request on every deletion of such Endpoints.

On the IPAM side, it is sufficient to implement the Go interface of the server produced by the Protocol Buffer compiler. Before looking at the pseudo-code, the reader should recall that:

- The IPAM has to distinguish between the cases in which the Endpoint is run on the local cluster or not. To do so, it is necessary to look at its IP address: if it belongs to the local PodCIDR, the reflection will be carried out in the traditional way; otherwise the ExternalCIDR network will be used.
- In both cases, the local PodCIDR and ExternalCIDR could have been remapped by the cluster on which the reflection has to take place and therefore a mapping between addresses, previously described in Section 4.1, can became necessary.
- When an address is allocated from the ExternalCIDR network and it is associated with an external Endpoint IP for the first time, further requests related to the same Endpoint IP should use and return the already allocated address. Thus, becomes necessary a way of keeping track of these address associations. For this reason, the IpamStorage resource has been upgraded with a new field, called *EndpointMappings*.

The pseudo-code in Algorithm 1 uses function *MapIPToNetwork* to translate addresses with the policy described in Section 4.1 and function *AllocateIP* to get an available IP address from the network specified as parameter. Curious readers can found the pseudo-code of *MapIPToNetwork* in Appendix. The first if statement determines the type of reflection that is going to take place: in case the condition is met, IPAM will proceed with a traditional reflection: mapping the address on *localRemappedPodCIDR* results in a no-operation or in the mapping on the network used in the remote cluster to identify the local PodCIDR. Otherwise, a reflection with ExternalCIDR will be carried out. At this point, it is ensured that no ExternalCIDR addresses have been already allocated for the Endpoint IP received as argument. A new address is allocated from those available in ExternalCIDR network in this case. Finally, the IPAM keeps track of the fact the Endpoint address

Listing 5.5: IpamStorage new fields for Endpoint IP addresses mapping.

```
// ipamStorage_types.go
/* ... */
// ClusterMapping is an empty struct.
type ClusterMapping struct{}

// EndpointMapping describes a relation between an enpoint IP and an
    IP belonging to ExternalCIDR.
type EndpointMapping struct {
    // IP belonging to cluster ExtenalCIDR assigned to this Endpoint.
    IP string `json:"ip"`
    // Set of clusters to which this Endpoint has been reflected.
    Only the key, which is the ClusterID, is useful.
    ClusterMappings map[string]ClusterMapping `json:"clusterMappings"
    `
}

// IpamSpec defines the desired state of Ipam.
type IpamSpec struct {
    /* ... */
    // Endpoint IP mappings. Key is the IP address of the local
    Endpoint, value is the IP of the remote Endpoint, so it belongs to
     an ExternalCIDR
    EndpointMappings map[string]EndpointMapping `json:"
    EndpointMappings"`
    /* ... */
}
/* ... */
```

is currently reflected with an ExternalCIDR IP in the foreign cluster specified in the procedure arguments. This results in a clever management of ExternalCIDR addresses: when an IP is no longer used in any reflection, IPAM can free it making it available for future reflections. The address returned to the Virtual Kubelet, *newIP*, is *externalIP* mapped on *localRemappedExternalCIDR*: this guarantees a correct translation if the home cluster's ExternalCIDR have been remapped by the foreign cluster. Algorithm 2 is the implementation of UnmapEndpointIP in pseudo-code. The unstructured termination at the very beginning of the snippet is triggered in two circumstances:

1. *endpointIP* belongs to the local PodCIDR. This means MapEndpointIP has carried out a traditional reflection, therefore no actions have to be done.

2. The Endpoint with address *endpointIP* is not a local Pod and an ExternalCIDR reflection has been executed by MapEndpointIP. However, the ExternalCIDR

address relative to *endpointIP* has been already freed. Also in this case, there is nothing to do.

Function *FreeIP*, as its name suggests, is used to represent the freeing of an IP in a network.

Algorithm 1 MapEndpointIP logic

```
1: function MapEndpointIP(clusterID, endpointIP)
2:     if EndpointIP belongs to localPodCIDR then
3:         newIP := MapIPToNetwork(endpointIP, localRemappedPodCIDR)
4:         return newIP
5:     end if
6:     if an ExternalCIDR IP has not been allocated for endpointIP yet then
7:         externalIP := AllocateIP(localExternalCIDR)
8:         store association between externalIP and endpointIP
9:     end if
10:    add clusterID to the list of clusters in which endpointIP has been reflected
11:    newIP := MapIPToNetwork(externalIP, localRemappedExternalCIDR)
12:    return newIP
13: end function
```

Algorithm 2 UnmapEndpointIP logic

```
1: function UnmapEndpointIP(clusterID, endpointIP)
2:     if an ExternalCIDR IP has not been allocated for endpointIP then
3:         return
4:     end if
5:     remove clusterID from the list of clusters in which endpointIP is reflected
6:     if the list of clusters in which endpointIP has been reflected is empty then
7:         retrieve externalIP relative to endpointIP
8:         FreeIP(localExternalCIDR, externalIP)
9:     end if
10: end function
```

5.4 Traffic redirection (IPAM side)

Making the translation of IP addresses on the control plane is not enough: some of these translations have to generate side effects on the data plane in order to make everything work. This is not the case of traditional reflections, indeed they involve Endpoints with addresses of the cluster's PodCIDR that are already reachable by remote Pods, as a Liqo peering exists among the two clusters. In other words, just the fact that a peering has been established among the clusters, guarantees that remote Pods can contact the reflected Service without any extra operation on the data plane. This happens because a Liqo peering enables inter-cluster Pod-to-Pod connectivity, and reflecting an Endpoint with IP belonging to a cluster's PodCIDR means no more than make Pods on the other clusters aware that for reaching that Service they must contact that Pod, and they are already capable to do so.

The situation is quite different when talking about a reflection with the ExternalCIDR. The reader should mind that this type of reflection is carried out for Endpoints with an address that does not belong to the cluster's PodCIDR but belongs to a different network (e.g.: the PodCIDR of a third cluster, the NodeCIDR[3], etc.). After the IPAM module of cluster x performs this kind of reflection, x will sooner or later receive packets directed to the address used for map the original Endpoint IP and x must be properly prepared. In conclusion, the reader should consider that these situations have to be treated differently: the traditional reflection requires no extra work, while the reflection with ExternalCIDR requires the Gateway to be configured to redirect traffic toward this network to its real recipient. In particular, a Destination NAT operation is what the Gateway should carry out.

The Gateway is notified about new IP mappings by means of a new custom resource updated by the IPAM module after a new ExternalCIDR reflection, called *NatMapping*. There is a NatMapping resource for each peered cluster and it contains addresses association that the home cluster have to transform in DNAT rules. Even if the new resource results useful in the context of study, it can also be adopted in other circumstances, whenever there is the need of dynamically redirect some traffic elsewhere. At this point, Algorithm 1 have to be modified in order to make the IPAM update the new resource whenever is necessary. In this way the Gateway, once upgraded to reconcile the resource, will be notified about the new mapping between addresses. The reader should notice that the association stored in NatMapping is between *endpointIP* and *newIP* and *externalIP* is not

[3]The NodeCIDR is the network addresses of worker nodes belong to.

Listing 5.6: NatMapping resource contains the DNAT rules for packets coming from a specified cluster.

```
// Mappings is the type describing a set mappings for a remote
    cluster. Key is the old IP, value is the new NAT-ted IP.
type Mappings map[string]string

// NatMappingSpec defines the desired state of NatMapping.
type NatMappingSpec struct {
    // ClusterID is the cluster this resource refers to.
    ClusterID string `json:"clusterID"`
    // ClusterMappings is the set of NAT mappings currently active.
    ClusterMappings Mappings `json:"clusterMappings"`
}
```

involved. The reason is that, exactly like the PodCIDR, the ExternalCIDR of the home cluster could have been remapped by the foreign cluster into a new address space, therefore *externalIP* may have no meaning in the other cluster. Since *newIP*, that have been mapped to *localRemappedExternalCIDR*, has been returned to the Virtual Kubelet, it will be used by remote Pods to reach the Endpoint. Similarly, Algorithm 2 has to be modified to take advantage of the new resource.

Algorithm 3 MapEndpointIP logic

```
1: function MapEndpointIP(clusterID, endpointIP)
2:     if endpointIP belongs to localPodCIDR then
3:         newIP := MapIPToNetwork(endpointIP, localRemappedPodCIDR)
4:         return newIP
5:     end if
6:     if an ExternalCIDR IP has not been allocated for endpointIP yet then
7:         externalIP := AllocateIP(localExternalCIDR)
8:         store association between externalIP and endpointIP
9:     end if
10:    add clusterID to the list of clusters in which endpointIP has been reflected
11:    newIP := MapIPToNetwork(externalIP, localRemappedExternalCIDR)
12:    store association between endpointIP and newIP in NatMapping resource
       relative to cluster clusterID
13:    return newIP
14: end function
```

Algorithm 4 UnmapEndpointIP logic

```
1: function UnmapEndpointIP(clusterID, endpointIP)
2:     if an ExternalCIDR IP has not been allocated for endpointIP then
3:         return
4:     end if
5:     remove clusterID from the list of clusters in which endpointIP is reflected
6:     remove association containing endpointIP in NatMapping resource of cluster
       clusterID
7:     if the list of clusters in which endpointIP has been reflected is empty then
8:         retrieve externalIP relative to endpointIP
9:         FreeIP(localExternalCIDR, externalIP)
10:    end if
11: end function
```

5.5 Traffic redirection (Gateway side)

The Gateway has been upgraded in such a way it can react to changes in NatMapping resource, thus a new controller has been integrated in the component. The new NatMappingController reconciles the resource and interacts with the NAT driver module to insert the appropriate set of DNAT rules. The NAT driver, under the hoods, uses the netfilter Linux module to ensure rules are present on a node. Assume the following NatMapping resource exists in the cluster:

```
apiVersion: net.liqo.io/v1alpha1
  kind: NatMapping
  metadata:
    name: natmapping-1
  spec:
    clusterID: cluster-1
    clusterMappings:
        10.0.0.5: 192.168.0.45
```

Once reconciled by the NatMappingController, the following rule will be inserted within the NAT rules of the Gateway network namespace:

```
iptables -t nat -A LIQO-PRRT-MAP-CLS-cluster-1 -d 192.168.0.45 -j
   DNAT --to-destination 10.0.0.5
```

where *LIQO-PRRT-MAP-CLS-cluster-1* is the iptables chain relative to cluster *cluster-1* containing prerouting rules extracted by its NatMapping resource.

5.6 ExternalCIDR traffic routing

The last step is to make Liqo clusters send to a peered cluster x not only the traffic toward the PodCIDR of x (or the network used to remap it) but also the traffic directed to cluster x's ExternalCIDR. This requires two steps:

1. Configure nodes to forward the traffic toward the ExternalCIDR of a remote cluster to the worker node where the Gateway lives.

2. Configure Gateway to forward the traffic toward the ExternalCIDR of a remote cluster to the appropriate VPN tunnel.

5.6.1 Routing between worked nodes

Currently, Liqo forward inter-cluster traffic toward the Gateway node by means of an overlay network[4] established between worker nodes. Therefore, nodes that are not running the Gateway have been configured so that they route that traffic to the interface belonging to the overlay. At that point, the data plane of the overlay network will take care of the packet.

5.6.2 Routing between clusters

On the Gateway side, it was necessary to add a route used to forward traffic to the local VPN Endpoint and configure the VPN (currently WireGuard) to send those packets to the tunnel Endpoint relative to the remote cluster.

[4]An overlay network is a computer network that is layered on top of another network.

Chapter 6

Experimental evaluation

6.1 Functional tests

The solution has been widely tested with different CNIs and has proven to be fully working. Indeed, End to End tests have been carried out both manually and programmatically. Google's microservices-demo [15] was used to deploy workloads on different (virtual) nodes. It is a microservice application that deploys an e-commerce website where users can browse items, add them to the cart, and purchase them and is composed of several Deployments and Services.

Manual tests were carried out by browsing the whole website, trying to complete some orders. The automated test instead was a simple check of the HTTP reply status code received by the website.

6.2 Perfomance tests

Performance tests have been developed in order to evaluate the goodness of the solution in terms of scalability. Therefore, it has been decided to collect the following direct and indirect measures:

1. Response Time: Time required by the IPAM module to serve n address translation requests made by m clients in parallel.
2. Average Response Time: Average time required by the IPAM module to serve

a single request. This measure will be extracted by the first one.

3. Latency: Time required by the IPAM and the Gateway to serve completely (IPAM response to Virtual Kubelet and Gateway insertion of DNAT rule) n requests made by m clients in parallel.

4. Average Latency: Average time interval between an IPAM address translation response and the insertion of the relative DNAT rule by the Gateway module. This measure will be extracted by the third one.

The first couple of measures focus only on the IPAM module and can give an insight of how it behaves when experiencing an heavy load in terms of parallel address translation requests. Latency measures instead can help understanding how much time the system takes to become able to handle traffic toward an ExternalCIDR IP after a translation request. Indeed, when the Virtual Kubelet receives a response to an address translation request (time $t1$), it is about to reflect the Endpoint in the foreign cluster and make it available to remote Pods. From the reflection completion on, traffic toward the ExternalCIDR IP can be generated and forwarded to the home cluster. In addition, the insertion time $t2$ of the DNAT rule is equivalent to the time the system becomes capable of redirecting traffic toward the ExternalCIDR IP to its real destination. This means that during the interval $t = t2 - t1$ the system is not inherently coherent, as the home cluster can potentially receive some traffic and it does not know where it must be forwarded.

6.2.1 Test environment

For the sake of simplicity, these tests were not performed on real clusters and Kind[16] was used instead. Kind is a tool that enables the user to get a Kubernetes cluster using Docker containers under the hood. Thus, tests were carried out on a Kind cluster installed on a Virtual Machine equipped with an Intel Xeon (Cascadelake) CPU @ 2195 MHz and 14 GB of RAM.

6.2.2 Micro benchmark

Micro benchmark B1 has been created to collect the response time (and consequently the average response time) and aims at testing only the IPAM module, therefore this component has been extracted by the Liqo Network Manager and has been inserted in a normal process. Although the IPAM is now a process on its own, it

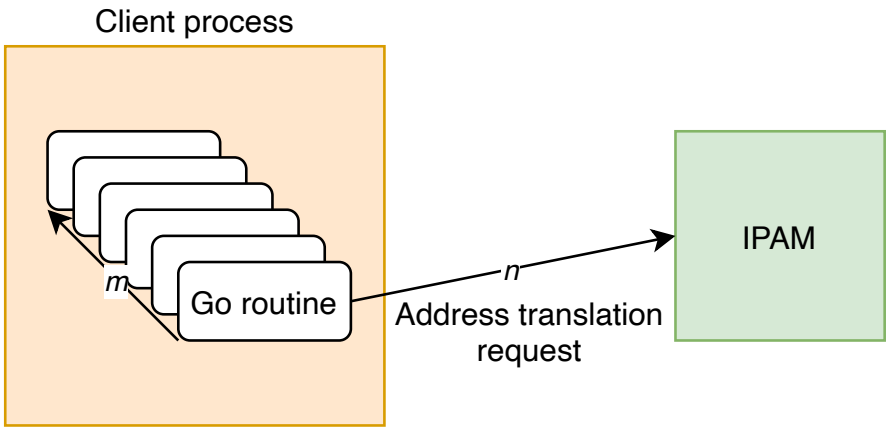

Figure 6.1: Micro benchmark setup.

still needs an API server to work. This is the reason why the Kind cluster has been deployed anyway. B1 is composed of a couple of processes:

- The IPAM, that after the initial setup logic, launches a Go routine that listens on a certain port waiting for gRPCs.
- A client process that starts m Go routines that in turn generate n address translation requests to the IPAM. The parameters n and m are received as command line arguments. It is worth to notice that a Go routine generated by the helper process is a stub representing a Virtual Kubelet.

Results

B1 has been executed with the following values for n and m:

- $m = 5$.
- n varying in [10, 50, 100, 150, 200].

Consequently, the number of total requests r relative to the i-th n can be computed as:

$$r = 5n[i]$$

The average processing time a for each request has been computed as:

$$a = \frac{t}{r}$$

where t is the total time of processing.

Collected response times are shown in Figure 6.2: the IPAM module takes 260.68 seconds (almost 4 minutes) to serve 500 requests and 675.42 seconds (almost 11 minutes) to serve one thousand requests. It can be noticed that the solution scales linearly when the number of requests grows.

Figure 6.3 instead summarizes the average response times and suggests that the system composed by the IPAM and the Gateway takes almost the same time to process a single request despite it experiences loads that differs for different orders of magnitude.

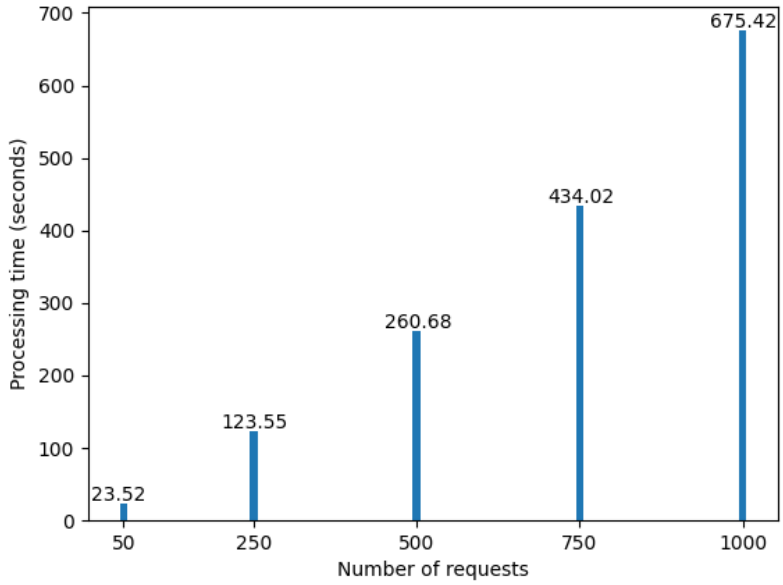

Figure 6.2: IPAM processing time of address translation requests for different number of requests.

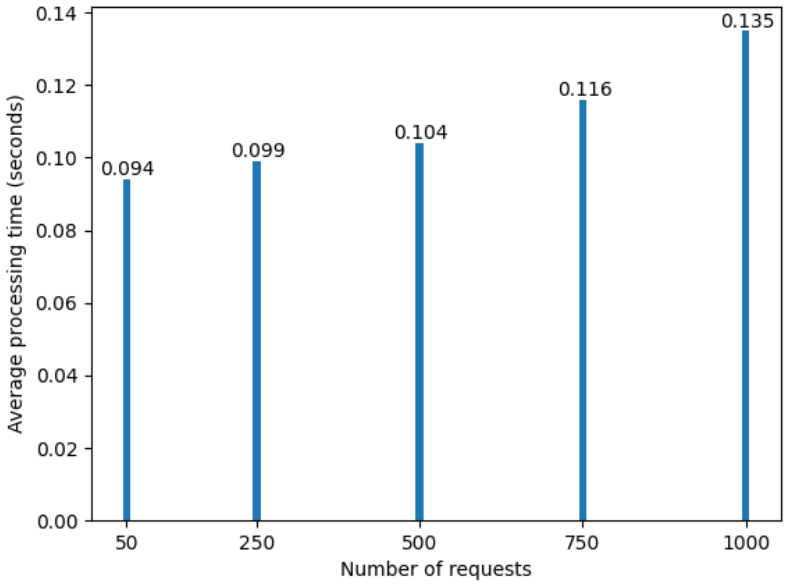

Figure 6.3: IPAM average processing time per request for different number of total requests.

6.2.3 Macro benchmark

Macro benchmark B2 has been engineered to collect latencies and stress the system composed by the IPAM and the Gateway. With respect to B1, in this case the IPAM lives in the Network Manager Pod. A client Pod, managed by a Kubernetes Job, has the responsibility of creating m Go routines. Similarly to what happened in B1, each routine generates n address translation requests.

Results

Values for m and n have been unchanged from B1 and a and r have been computed as described in the previous Section. Figure 6.5 and Figure 6.6 depict the obtained results: also in this case the system scales. However, when the number of requests exceeds 500 it can be noticed that the average processing time grows reasonably. This tiny overhead is probably caused by the insertion of NAT rules that requires the Gateway to interact with the netfilter kernel module and consequently to carry out multiples context switches.

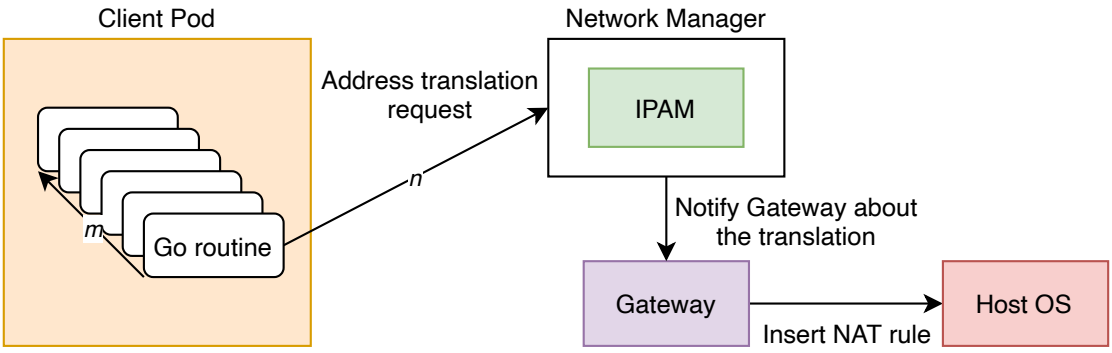

Figure 6.4: Macro benchmark setup.

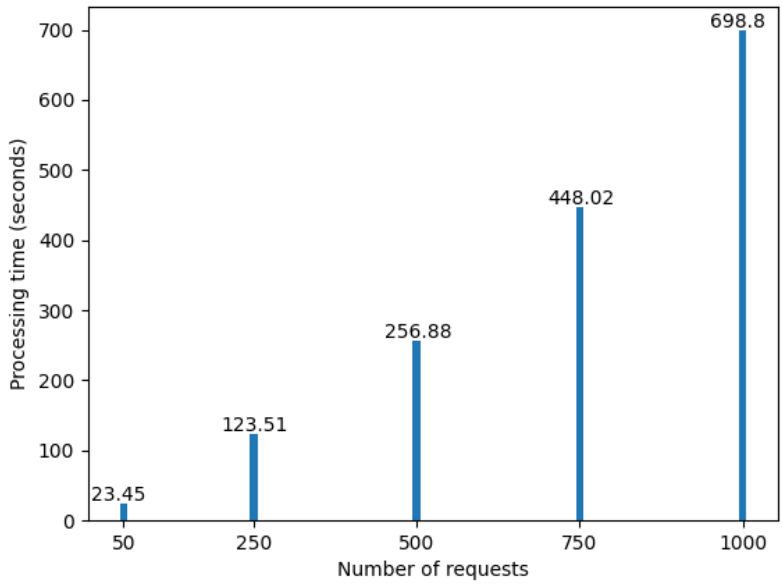

Figure 6.5: System processing time of address translation requests for different number of requests.

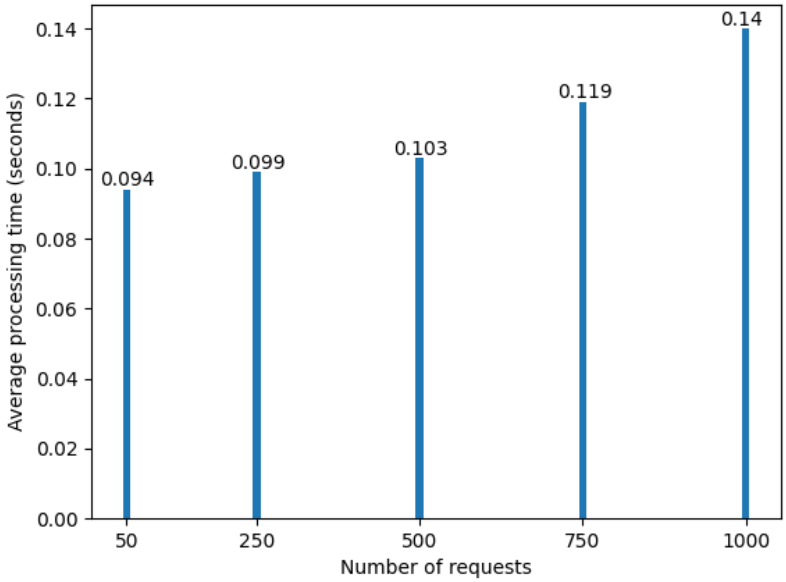

Figure 6.6: System average processing time per request for different number of total requests.

Chapter 7

Conclusions and future works

The Kubernetes and cloud native operations report 2021 by Canonical [17] gives an insight of how strong is the current tendency of organizations to own different Kubernetes clusters in production environments. As mentioned in Section 1.1, the reasons that push companies to deploy different clusters are multiple, therefore is reasonable to think that this trend is expected to grow further.

The work described in this thesis improved Liqo, an existing framework to build multi cluster topologies, enabling cluster administrators to exploit all the potential of such environments. When deploying an application spanning on more than two clusters, Liqo reflected all Kubernetes Endpoints in the same way regardless where those Endpoints actually were run (locally or remotely). Furthermore, it was missing a way of communication between the cluster the Endpoint lived in and the cluster the reflection was going to happen. A new control plane logic during the reflection of Kubernetes Endpoints fixed the problem by using the ExternalCIDR network when reflecting remote Endpoints.

Although the ExternalCIDR solution reached promising results during the experimental evaluation, as suggested in Section 4.4 it leverages a star topology, where traffic always goes through the central cluster before reaching its final destination. Even if the star topology is characterized by some advantages, it is easy to guess that if the number of involved cluster grows, the load on the central cluster becomes irreparably huge, slowing down the entire multi cluster environment.

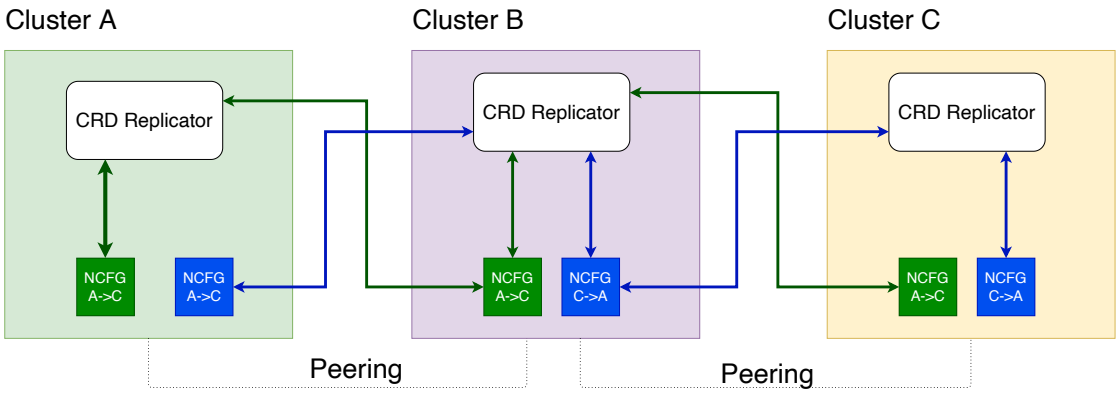

Figure 7.1: Induced peering configuration.

In this respect, the logical evolution of this work consists on switching to a mesh topology, in which clusters can communicate directly. Considering a simple multi cluster environment in which a cluster B has a peering with A and another one with C, the problem of making A and C in communication can be solved by enabling their NetworkConfigs flowing through B as depicted in Figure 7.1. In this way, A and C could create TunnelEndpoint resources and establish a special peering, in which only networking is enabled: it is called *induced peering*. After A and C are connected one to another, it remains the problem of reflecting Endpoint with valid IP addresses. However, B has its own copies of the NetworkConfigs exchanged during the induced peering configuration, therefore it is aware of how clusters remapped each other's PodCIDR. Thus, solving the second problem is just a matter of upgrading the Virtual Kubelet making it able to reflect Endpoints according to the information in those NetworkConfigs.

Appendix A

Network Manager

Listing A.1: NetworkConfig resource.

```
package v1alpha1

import (
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/apimachinery/pkg/runtime/schema"

    crdclient "github.com/liqotech/liqo/pkg/crdClient"
)

// NetworkConfigSpec defines the desired state of NetworkConfig.
type NetworkConfigSpec struct {
    // The ID of the remote cluster that will receive this CRD.
    ClusterID string `json:"clusterID"`
    // Network used in the local cluster for the pod IPs.
    PodCIDR string `json:"podCIDR"`
    // Network used for local service endpoints.
    ExternalCIDR string `json:"externalCIDR"`
    // Public IP of the node where the VPN tunnel is created.
    EndpointIP string `json:"endpointIP"`
    // Vpn technology used to interconnect two clusters.
    BackendType string `json:"backendType"`
    // Connection parameters
    BackendConfig map[string]string `json:"backend_config"`
}

// NetworkConfigStatus defines the observed state of NetworkConfig.
type NetworkConfigStatus struct {
    // Indicates if this network config has been processed by the
   remote cluster.
```

```
29      // +kubebuilder:default=false
30      Processed bool `json:"processed"`
31      // The new subnet used to NAT the podCidr of the remote cluster.
        The original PodCidr may have been mapped to this
32      // network by the remote cluster.
33      PodCIDRNAT string `json:"podCIDRNAT,omitempty"`
34      // The new subnet used to NAT the externalCIDR of the remote
        cluster. The original ExternalCIDR may have been mapped
35      // to this network by the remote cluster.
36      ExternalCIDRNAT string `json:"externalCIDRNAT,omitempty"`
37  }
38
39  type NetworkConfig struct {
40      metav1.TypeMeta   `json:",inline"`
41      metav1.ObjectMeta `json:"metadata,omitempty"`
42
43      Spec   NetworkConfigSpec   `json:"spec,omitempty"`
44      Status NetworkConfigStatus `json:"status,omitempty"`
45
46
47  // NetworkConfigList contains a list of NetworkConfig.
48  type NetworkConfigList struct {
49      metav1.TypeMeta `json:",inline"`
50      metav1.ListMeta `json:"metadata,omitempty"`
51      Items           []NetworkConfig `json:"items"`
52  }
53
54  func init() {
55      SchemeBuilder.Register(&NetworkConfig{}, &NetworkConfigList{})
56
57      crdclient.AddToRegistry("networkconfigs", &NetworkConfig{}, &
        NetworkConfigList{}, nil, schema.GroupResource{
58          Group:    TunnelEndpointGroupResource.Group,
59          Resource: "networkconfigs",
60      })
61  }
```

Listing A.2: TunnelEndpoint resource.

```
1   package v1alpha1
2
3   import (
4       metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
5   )
6
7   // TunnelEndpointSpec defines the desired state of TunnelEndpoint.
8   type TunnelEndpointSpec struct {
9       // Important: Run "make" to regenerate code after modifying this
        file
```

```
10      // The ID of the remote cluster that will receive this CRD.
11      ClusterID string `json:"clusterID"`
12      // PodCIDR of remote cluster.
13      PodCIDR string `json:"podCIDR"`
14      // ExternalCIDR of remote cluster.
15      ExternalCIDR string `json:"externalCIDR"`
16      // Public IP of the node where the VPN tunnel is created.
17      EndpointIP string `json:"endpointIP"`
18      // Vpn technology used to interconnect two clusters.
19      BackendType string `json:"backendType"`
20      // Connection parameters.
21      BackendConfig map[string]string `json:"backend_config"`
22  }
23
24  // TunnelEndpointStatus defines the observed state of TunnelEndpoint.
25  type TunnelEndpointStatus struct {
26      Phase string `json:"phase,omitempty"`
27      // PodCIDR of local cluster.
28      LocalPodCIDR string `json:"localPodCIDR,omitempty"`
29      // Network used in the remote cluster to map the local PodCIDR,
        in case of conflicts(in the remote cluster).
30      // Default is "None".
31      LocalNATPodCIDR string `json:"localNATPodCIDR,omitempty"`
32      // Network used in the local cluster to map the remote cluster
        PodCIDR, in case of conflicts with Spec.PodCIDR.
33      // Default is "None".
34      RemoteNATPodCIDR string `json:"remoteNATPodCIDR,omitempty"`
35      // ExternalCIDR of local cluster.
36      LocalExternalCIDR string `json:"localExternalCIDR,omitempty"`
37      // Network used in the remote cluster to map the local
        ExternalCIDR, in case of conflicts(in the remote cluster).
38      // Default is "None".
39      LocalNATExternalCIDR string `json:"localNATExternalCIDR,omitempty
        "`
40      // Network used in the local cluster to map the remote cluster
        ExternalCIDR, in case of conflicts with
41      // Spec.ExternalCIDR.
42      // Default is "None".
43      RemoteNATExternalCIDR string      `json:"remoteNATExternalCIDR,
        omitempty"`
44      RemoteEndpointIP      string      `json:"remoteTunnelPublicIP,
        omitempty"`
45      LocalEndpointIP       string      `json:"localTunnelPublicIP,
        omitempty"`
46      TunnelIFaceIndex      int         `json:"tunnelIFaceIndex,
        omitempty"`
47      TunnelIFaceName       string      `json:"tunnelIFaceName,omitempty
        "`
```

```
48      VethIFaceIndex          int         `json:"vethIFaceIndex,omitempty"
        `
49      VethIFaceName           string      `json:"vethIFaceName,omitempty"`
50      GatewayIP               string      `json:"gatewayIP,omitempty"`
51      Connection              Connection  `json:"connection,omitempty"`
52  }
53
54  // Connection holds the configuration and status of a vpn tunnel
        connecting to remote cluster.
55  type Connection struct {
56      Status              ConnectionStatus   `json:"status,omitempty"`
57      StatusMessage       string             `json:"statusMessage,
        omitempty"`
58      PeerConfiguration map[string]string `json:"peerConfiguration,
        omitempty"`
59  }
60
61  // ConnectionStatus type that describes the status of vpn connection
        with a remote cluster.
62  type ConnectionStatus string
63
64  const (
65      // Connected used when the connection is up and running.
66      Connected ConnectionStatus = "connected"
67      // Connecting used as temporary status while waiting for the vpn
        tunnel to come up.
68      Connecting ConnectionStatus = "connecting"
69      // ConnectionError used to se the status in case of errors.
70      ConnectionError ConnectionStatus = "error"
71  )
72
73  type TunnelEndpoint struct {
74      metav1.TypeMeta   `json:",inline"`
75      metav1.ObjectMeta `json:"metadata,omitempty"`
76
77      Spec   TunnelEndpointSpec   `json:"spec,omitempty"`
78      Status TunnelEndpointStatus `json:"status,omitempty"`
79  }
80
81  // TunnelEndpointList contains a list of TunnelEndpoint.
82  type TunnelEndpointList struct {
83      metav1.TypeMeta `json:",inline"`
84      metav1.ListMeta `json:"metadata,omitempty"`
85      Items           []TunnelEndpoint `json:"items"`
86  }
87
88  func init() {
89      SchemeBuilder.Register(&TunnelEndpoint{}, &TunnelEndpointList{})
90  }
```

Appendix B

IPAM

Listing B.1: IpamStorage resource.

```
package v1alpha1

import (
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// Subnets type contains relevant networks related to a remote cluster.
type Subnets struct {
    // Network used in the remote cluster for local Pods. Default is "None": this means remote cluster uses local cluster PodCIDR.
    LocalNATPodCIDR string `json:"localNATPodCIDR"`
    // Network used for Pods in the remote cluster.
    RemotePodCIDR string `json:"remotePodCIDR"`
    // Network used in remote cluster for local service endpoints. Default is "None": this means remote cluster uses local cluster ExternalCIDR.
    LocalNATExternalCIDR string `json:"localNATExternalCIDR"`
    // Network used in local cluster for remote service endpoints.
    RemoteExternalCIDR string `json:"remoteExternalCIDR"`
}

// ClusterMapping is an empty struct.
type ClusterMapping struct{}

// ConfiguredCluster is an empty struct used as value for NatMappingsConfigured.
type ConfiguredCluster struct{}

```

```
25 // EndpointMapping describes a relation between an enpoint IP and an
       IP belonging to ExternalCIDR.
26 type EndpointMapping struct {
27     // IP belonging to cluster ExtenalCIDR assigned to this endpoint.
28     IP string `json:"ip"`
29     // Set of clusters to which this endpoint has been reflected.
       Only the key, which is the ClusterID, is useful.
30     ClusterMappings map[string]ClusterMapping `json:"clusterMappings"
       `
31 }
32
33 // IpamSpec defines the desired state of Ipam.
34 type IpamSpec struct {
35     // Map consumed by go-ipam module. Key is prefic cidr, value is a
        Prefix.
36     Prefixes map[string][]byte `json:"prefixes"`
37     // Network pools.
38     Pools []string `json:"pools"`
39     // Map used to keep track of networks assigned to clusters. Key
       is the remote cluster ID, value is a the set of
40     // networks used by the remote cluster.
41     ClusterSubnets map[string]Subnets `json:"clusterSubnets"`
42     // Cluster ExternalCIDR
43     ExternalCIDR string `json:"externalCIDR"`
44     // Endpoint IP mappings. Key is the IP address of the local
       endpoint, value is the IP of the remote endpoint, so it belongs to
        an ExternalCIDR
45     EndpointMappings map[string]EndpointMapping `json:"
       endpointMappings"`
46     // NatMappingsConfigured is a map that contains all the remote
       clusters
47     // for which NatMappings have been already configured.
48     // Key is a cluster ID, value is an empty struct.
49     NatMappingsConfigured map[string]ConfiguredCluster `json:"
       natMappingsConfigured"`
50     // Cluster PodCIDR
51     PodCIDR string `json:"podCIDR"`
52     // ServiceCIDR
53     ServiceCIDR string `json:"serviceCIDR"`
54 }
55
56 // IpamStorage is the Schema for the ipams API.
57 type IpamStorage struct {
58     metav1.TypeMeta   `json:",inline"`
59     metav1.ObjectMeta `json:"metadata,omitempty"`
60
61     Spec IpamSpec `json:"spec,omitempty"`
62 }
63
```

```
64 // +kubebuilder:object:root=true
65
66 // IpamStorageList contains a list of Ipam.
67 type IpamStorageList struct {
68     metav1.TypeMeta `json:",inline"`
69     metav1.ListMeta `json:"metadata,omitempty"`
70     Items           []IpamStorage `json:"items"`
71 }
72
73 func init() {
74     SchemeBuilder.Register(&IpamStorage{}, &IpamStorageList{})
75 }
```

Algorithm 5 MapIPToNetwork logic

```
1: function MapIPToNetwork(oldIP, network)
2:     if network == 'None' then
3:         return oldIP
4:     end if
5:     maskLength := GetMask(network)
6:     forge newIP concatenating the first maskLength bits of network and the
       last 32-maskLength bits of oldIP
7:     return newIP
8: end function
```

Bibliography

[1] *Calculating the cost of downtime*. URL: https://www.atlassian.com/incident-management/kpis/cost-of-downtime (cit. on p. 2).

[2] *5 Vendor Lock-In Strategies for Your Online Business*. URL: https://straal.com/5-vendor-lock-in-strategies-for-your-online-business/ (cit. on p. 3).

[3] *Disaster recovery - Wikipedia*. URL: https://en.wikipedia.org/wiki/Disaster_recovery#Recovery_Point_Objective (cit. on p. 3).

[4] *11 Facts About Real-World container use*. URL: https://www.datadoghq.com/container-report/ (cit. on p. 4).

[5] *Liqo GitHub repository*. URL: https://github.com/liqotech/liqo (cit. on p. 4).

[6] *IBM's success story*. URL: https://startuptalky.com/ibm-success-story/#IBMIBM-History (cit. on p. 6).

[7] *cgroups - Wikipedia*. URL: https://en.wikipedia.org/wiki/Cgroups (cit. on p. 7).

[8] *namespaces - Wikipedia*. URL: https://en.wikipedia.org/wiki/Linux_namespaces (cit. on p. 7).

[9] *Horizontal Pod Autoscaler | Kubernetes*. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (cit. on p. 12).

[10] *Welcome - NGINX Ingress Controller*. URL: https://kubernetes.github.io/ingress-nginx/deploy/ (cit. on p. 14).

[11] *Virtual Kubelet - GitHub*. URL: https://github.com/virtual-kubelet/virtual-kubelet (cit. on p. 19).

[12] *Liqo documentation*. URL: https://doc.liqo.io/ (cit. on p. 21).

[13] *WireGuard homepage*. URL: https://www.wireguard.com/ (cit. on p. 30).

[14] *gRPC*. URL: https://grpc.io (cit. on p. 44).

[15] *microservice-demo*. URL: https://github.com/GoogleCloudPlatform/microservices-demo (cit. on p. 53).

[16] *Kind homepage*. URL: https://kind.sigs.k8s.io (cit. on p. 54).

[17] *Kubernetes and cloud native operations report 2021*. URL: https://juju.is/cloud-native-kubernetes-usage-report-2021?utm_source=blog#size-is-important-how-many-machines-and-clusters-are-people-running (cit. on p. 60).