POLITECNICO DI TORINO

Master's Degree in Computer engineering

Master's Degree Thesis

Seamless Network Connectivity across Different Kubernetes Clusters

Supervisors

Prof. Fulvio RISSO

Dott. Alex PALESANDRO

Candidate

Aldo LACUKU

Academic year 2019-2020

Summary

In the last two decades the cloud has gained a lot of importance, indeed the current trend is to engineer the new web applications to be cloud native, thus to be split up in loosely-coupled micro-services, each one containerized and deployed as a part of a bigger application. The use of containers allows to cut oneself off the hosting physical hardware and operating system, letting to focus on the main purposes of a web application: to be widespread and high-available. The cloud allows to achieve this goal, by gathering the infrastructure control under the cloud provider tenants and implementing the IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) paradigms: the computational, networking and storage resources are provided on demand to the cloud provider's customers as if they were services. A technology that broke through the cloud market is Kubernetes, a project kicked off by Google in 2014 that allows to automate deployment, scaling, and management of containerized applications. Beside the cloud, in recent years the edge computing has gained a lot of importance: it is a distributed computing paradigm that brings the computational and storage resources close to the final user, in order to improve the QoS standards in terms of latency and bandwidth.

The goal of the project in which this thesis is involved is to create a federation of Kubernetes clusters that cooperate at the network edge: many different tenants are connected together to cooperate in creating a federation of clusters with computational, storage and networking resources shared between them. In this scenario every tenant can make its own resource cluster available to the federation by sharing or leasing them out in a federated environment.

This solution needs a network model able to install a seamless network connectivity across different Kubernetes clusters. This work proposes a solution with minimal dependencies, which could be installed in a running cluster without bringing changes to the configuration of the local environment. Permitting micro-services to consume other services independently from the location where they are deployed.

Table of Contents

1 Introduction 1
1.1 Goal of the thesis 2

2 Kubernetes 3
2.1 Kubernetes: a bit of history 3
2.2 Applications deployment evolution 4
2.3 Container orchestrators 5
2.4 Kubernetes architecture 6
2.4.1 Control plane components 7
2.4.2 Node components 9
2.5 Kubernetes objects 10
2.5.1 Label & Selector 11
2.5.2 Namespace 11
2.5.3 Pod 11
2.5.4 ReplicaSet 12
2.5.5 Deployment 12
2.5.6 DaemonSet 13
2.5.7 Service 14
2.6 Kubernetes network architecture 15
2.6.1 Container communication within same pod 16
2.6.2 Pod communication within the same node 16
2.6.3 Pod communication on different nodes 17
2.6.4 CNI (Container Network Interface) 17
2.6.5 Pod to service networking 18
2.7 Virtual-Kubelet 18
2.8 Kubebuilder 19

3 Multi-Cluster networking: state of the art 21
3.1 Submariner 21
3.1.1 Broker 21
3.1.2 Gateway engine 22

3.1.3 Route agent . . . 23
3.2 Cilium cluster mesh . . . 23
3.2.1 Control plane . . . 24
3.2.2 Pod IP routing . . . 25

4 Multi-Cluster networking: design 27
4.1 The problem . . . 27
4.1.1 Compatibility with running clusters . . . 29
4.2 Architecture . . . 30

5 Multi-Cluster networking: implementation 35
5.1 Kubernetes programming interface . . . 36
5.2 CRDReplicator . . . 37
5.2.1 Labels and Selectors . . . 37
5.2.2 Watching Resources . . . 39
5.2.3 Watching Local Resources . . . 39
5.3 Watching Remote Resources . . . 42
5.3.1 Network Parameters Exchange . . . 43
5.4 TunnelEndpointCreator . . . 45
5.4.1 TunnelEndpoint API . . . 45
5.4.2 IPAM . . . 47
5.4.3 Network API Management . . . 50
5.5 Tunnel-Operator . . . 52
5.5.1 GRE Tunneling Protocol . . . 53
5.6 Route-Operator . . . 54
5.6.1 VxLAN overlay network . . . 55
5.6.2 Routes . . . 57
5.6.3 IPtables rules . . . 59

6 Experimental evaluation 62
6.1 Functional Tests . . . 62
6.1.1 Test environment . . . 62
6.1.2 Tests . . . 63
6.1.3 Functional tests results . . . 65
6.2 Performance and scalability tests . . . 65
6.2.1 Tests . . . 66
6.2.2 Performance test results . . . 67
6.3 Test limitations . . . 69

7 Conclusions and future work 70

Bibliography 71

Chapter 1

Introduction

In the last several years, ICT world has seen an incredible innovation with the introduction of virtualization first, then with containerization and finally with orchestrators. In this last field, one of the main actors is Kubernetes, an open source system for managing containerized applications in a clustered environment. The spread of Kubernetes is rapidly increasing; in cloud providers such as Google Cloud Platform and Microsoft Azure it is the most popular choice [1] and many companies and organizations have started to set up their own clusters in order to migrate their applications on it. With the advent of 5G and edge computing also telecommunications companies are moving towards a Kubernetes solution [2].

The growing demand for edge computing resources, particularly due to increasing popularity of Internet of Things (IoT), and distributed machine/deep learning applications poses a significant challenge. In a similar scenario, if we could share resources between clusters this would open many use cases:

- different users with their small clusters (for example Minikube [3]) can partially or totally offload their applications to others;

- different companies could interconnect and get payed for hosting others' applications;

- in an IoT scenario, edge nodes (which typically have limited resources) can send requests to more powerful ones;

- in an edge computing scenario, an application can be scheduled on the best cluster in order to reduce latency.

1.1 Goal of the thesis

Kubernetes does not support natively this sharing of resources among clusters: a concept of "Federation" has been defined and is being developed, but the project is relatively new and not very mature (it is still alpha [4]).

This work, carried out by the Computer Networks Group at Politecnico di Torino proposes a network plug-in able to extend the network connection of a Kubernetes cluster to services hosted by remote clusters. It is a control plane that configures Kubernetes clusters to have a seamless network connectivity between them.

This thesis is structured as follows:

- **Chapter 2** provides an extensive presentation of Kubernetes, its architecture and concepts;
- **Chapter 3** describes the state of the art of multi-cluster networking;
- **Chapter 4** formalizes the global design of the network plug-in and its logical parts;
- **Chapter 5** provides an implementation of the network plug-in by means of Kubernetes operators;
- **Chapter 6** analyzes the compatibility of the implementation with the existing container network interfaces;
- **Chapter 7** is the thesis conclusion and enunciation of future work directions.

Chapter 2

Kubernetes

In this chapter we analyse Kubernetes architecture, showing also its history and evolution through time, in order to lay the foundations for all the work which will be exposed later on. Kubernetes (often shortened as K8s) is a huge framework and a deep examination of it would require much more time and discussion, hence we only provide here a description of its main concepts and components. Further details can be found in the official documentation [5].

The chapter continues with an introduction to other technologies and tools used to develop the solution, in particular **Virtual-Kubelet** [6], a project which allows to create virtual nodes with a particular behaviour, and **Kubebuilder** [7], a tool to build custom resources.

2.1 Kubernetes: a bit of history

Around 2004, Google created the **Borg** [8] system, a small project with less than 5 people initially working on it. The project was developed as a collaboration with a new version of Google's search engine. Borg was a large-scale internal cluster management system, which "ran hundreds of thousands of jobs, from many thousands of different applications, across many clusters, each with up to tens of thousands of machines" [8].

In 2013 Google announced **Omega** [9], a flexible and scalable scheduler for large compute clusters. Omega provided a "parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, in order to achieve both implementation extensibility and performance scalability".

In the middle of 2014, Google presented **Kubernetes** as on open-source version of Borg. Kubernetes was created by Joe Beda, Brendan Burns, and Craig McLuckie, and other engineers at Google. Its development and design were heavily influenced by Borg and many of its initial contributors previously used to work on it. The

original Borg project was written in C++, whereas for Kubernetes the Go language was chosen.

In 2015 Kubernetes v1.0 was released. Along with the release, Google set up a partnership with the Linux Foundation to form the **Cloud Native Computing Foundation** (CNCF) [10]. Since then, Kubernetes has significantly grown, achieving the CNCF graduated status and being adopted by nearly every big company. Nowadays it has become the de-facto standard for container orchestration [11, 12].

2.2 Applications deployment evolution

Kubernetes is a portable, extensible, open-source platform for running and coordinating containerized applications across a cluster of machines. It is designed to completely manage the life cycle of applications and services using methods that provide consistency, scalability, and high availability.

What does "containerized applications" means? In the last decades, the deployment of applications has seen significant changes, which are illustrated in figure 2.1.

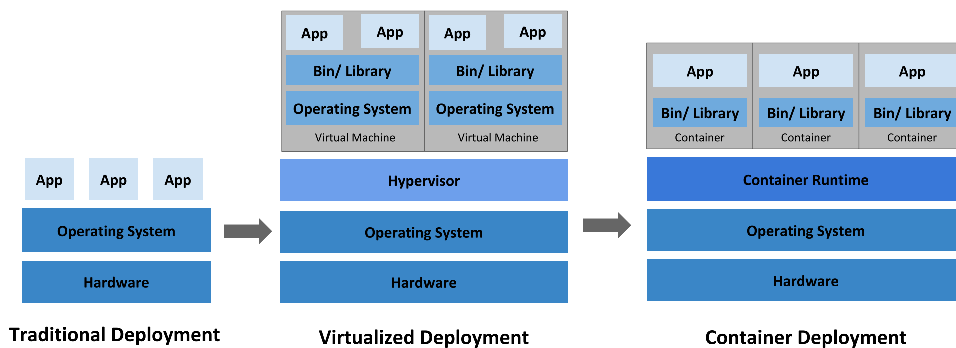

Figure 2.1: Evolution in applications deployment.

Traditionally, organizations used to run their applications on physical servers. One of the problems of this approach was that resource boundaries between applications could not be applied in a physical server, leading to resource allocation issues. For example, if multiple applications run on a physical server, one of them could take up most of the resources, and as a result, the other applications would starve. A possibility to solve this problem would be to run each application on a different physical server, but clearly it is not feasible: the solution could not scale, would lead to resources under-utilization and would be very expensive for organizations to maintain many physical servers.

The first real solution has been **virtualization**. Virtualization allows to run

multiple Virtual Machines on a single physical server. It grants isolation of the applications between VMs providing a high level of security, as the information of one application cannot be freely accessed by another application. Virtualization enables better utilization of resources in a physical server, improves scalability, because an application can be added or updated very easily, reduces hardware costs, and much more. With virtualization it is possible to group together a set of physical resources and expose it as a cluster of disposable virtual machines. Isolation certainly brings many advantages, but it requires a quite 'heavy' overhead: each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

A second solution which has been proposed recently is **containerization**. Containers are similar to VMs, but they share the operating system with the host machine, relaxing isolation properties. Therefore, containers are considered a lightweight form of virtualization. Similarly to a VM, a container has its own filesystem, CPU, memory, process space etc. One of the key features of containers is that they are portable: as they are decoupled from the underlying infrastructure, they are totally portable across clouds and OS distributions. This property is particularly relevant nowadays with cloud computing: a container can be easily moved across different machines. Moreover, being "lightweight", containers are much faster than virtual machines: they can be booted, started, run and stopped with little effort and in a short time.

2.3 Container orchestrators

When hundreds or thousands of containers are created, the need of a way to manage them becomes essential; container orchestrators serve this purpose. A container orchestrator is a system designed to easily manage complex containerization deployments across multiple machines from one central location. As depicted in figure 2.2, Kubernetes is by far the most used container orchestrator. We provide a description of such system in the following.

Kubernetes provides many services, including:

- **Service discovery and load balancing** A container can be exposed using the DNS name or using its own IP address. If traffic to a container is high, a load balancer able to distribute the network traffic is provided.
- **Storage orchestration** A storage system can be automatically mounted, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** The desired state for the deployed containers can be described, and the actual state can be changed to the desired state at a controlled rate. For example, it is possible to automate the

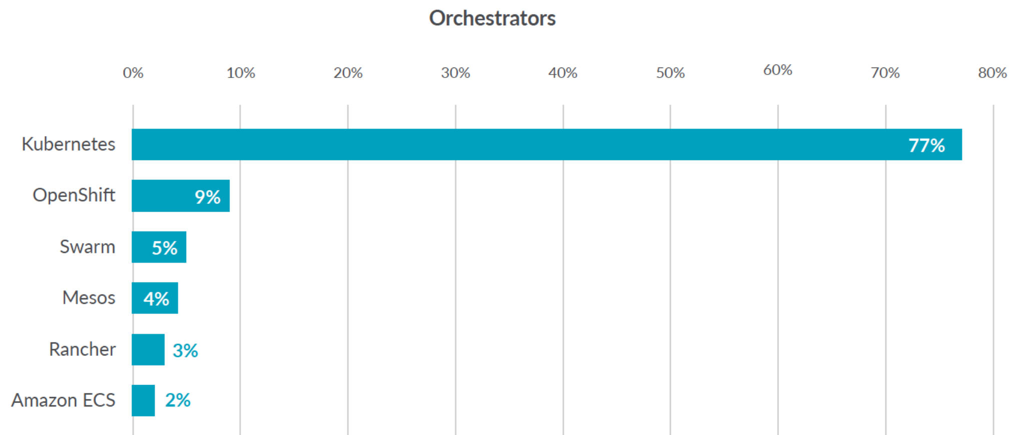

Figure 2.2: Container orchestrators use [13].

creation of new containers of a deployment, remove existing containers and adopt all their resources to the new container.

- **Automatic bin packing** Kubernetes is provided with a cluster of nodes that can be used to run containerized tasks. It is possible to set how much CPU and memory (RAM) each container needs, and automatically the containers are sized to fit in the nodes to make the best use of the resources.
- **Secret and configuration management** It is possible to store and manage sensitive information in Kubernetes, such as passwords, OAuth tokens, and SSH keys. It is possible to deploy and update secrets and application configuration without rebuilding the container images, and without exposing secrets in the stack configuration.

2.4 Kubernetes architecture

When Kubernetes is deployed, a cluster is created. A Kubernetes cluster consists of a set of machines, called **nodes**, that run containerized applications. At least one of the nodes hosts the control plane and is called **master**. Its role is to manage the cluster and expose an interface to the user. The **worker** node(s) host the **pods** that are the components of the application. The master manages the worker nodes and the pods in the cluster. In production environments, the control plane usually runs across multiple machines and a cluster runs on multiple nodes, providing fault-tolerance and high availability.

Figure 2.3 shows the diagram of a Kubernetes cluster with all the components linked together.

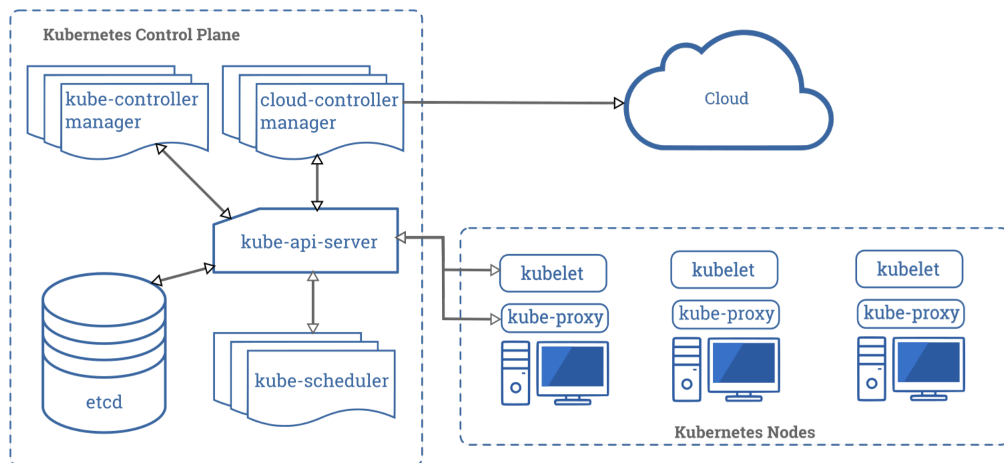

Figure 2.3: Kubernetes architecture

2.4.1 Control plane components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod). Although they can be run on any machine in the cluster, for simplicity, they are typically executed all together on the same machine, which does not run user containers.

API server

The API server is the component of the Kubernetes control plane that exposes the Kubernetes REST API, and constitites the front end for the Kubernetes control plane. Its function is to intercept REST request, validate and process them. The main implementation of a Kubernetes API server is `kube-apiserver`. It is designed to scale horizontally, which means it scales by deploying more instances. Moreover, it can be easily redounded to run several instances of it and balance traffic among them.

etcd

`etcd` is a distributed, consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. It is based on the Raft consensus algorithm [**raft_algorithm**], which allows different machines to work as a coherent group and survive to the breakdown of one of its members. `etcd` can be stacked in the master node or external, installed on dedicated host. Only the API server can communicate with it.

Scheduler

The scheduler is the control plane component responsible of assigning the pods to the nodes. The one provided by Kubernetes is called `kube-scheduler`, but it can be customized by adding new schedulers and indicating in the pods to use them. `kube-scheduler` watches for newly created pods not assigned to a node yet, and selects one for them to run on. To make its decisions, it considers singular and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

kube-controller-manager

Component that runs controller processes. It continuously compares the desired state of the cluster (given by the objects specifications) with the current one (read from `etcd`). Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process. These controllers include:

- Node Controller: responsible for noticing and reacting when nodes go down.
- Replication Controller: in charge of maintaining the correct number of pods for every replica object in the system.
- Endpoints Controller: populates the Endpoint objects (which links Services and Pods).
- Service Account & Token Controllers: create default accounts and API access tokens for new namespaces.

cloud-controller-manager

This component runs controllers that interact with the underlying cloud providers. The `cloud-controller-manager` binary is a beta feature introduced in Kubernetes 1.6. It only runs cloud-provider-specific controller loops. You can disable these controller loops in the `kube-controller-manager`.

`cloud-controller-manager` allows the cloud vendor's code and the Kubernetes code to evolve independently of each other. In prior releases, the core Kubernetes code was dependent upon cloud-provider-specific code for functionality. In future releases, code specific to cloud vendors should be maintained by the cloud vendor themselves, and linked to `cloud-controller-manager` while running Kubernetes. Some examples of controllers with cloud provider dependencies are:

- Node Controller: checks the cloud provider to update or delete Kubernetes nodes using cloud APIs.

- Route Controller: responsible for setting up network routes in the cloud infrastructure.
- Service Controller: for creating, updating and deleting cloud provider load balancers.
- Volume Controller: creates, attaches, and mounts volumes, interacting with the cloud provider to orchestrate them.

2.4.2 Node components

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

Container Runtime

The `container runtime` is the software that is responsible for running containers. Kubernetes supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

kubelet

An agent that runs on each node in the cluster, making sure that containers are running in a pod. The `kubelet` receives from the API server the specifications of the Pods and interacts with the `container runtime` to run them, monitoring their state and assuring that the containers are running and healthy. The connection with the `container runtime` is established through the Container Runtime Interface and is based on gRPC.

kube-proxy

`kube-proxy` is a network agent that runs on each node in your cluster, implementing part of the Kubernetes Service concept. It maintains network rules on nodes, which allow network communication to your Pods from inside or outside of the cluster. If the operating system is providing a packet filtering layer, `kube-proxy` uses it, otherwise it forwards the traffic itself.

Addons

Features and functionalities not yet available natively in Kubernetes, but implemented by third parties pods. Some examples are DNS, dashboard (a web gui), monitoring and logging.

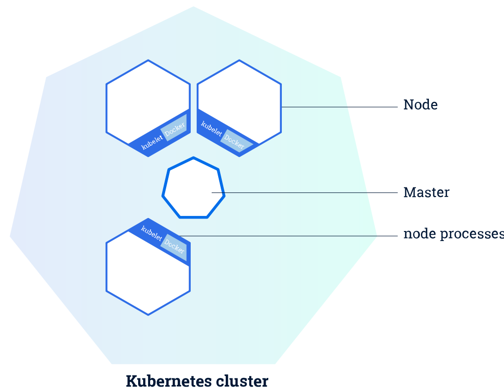

Figure 2.4: Kubernetes master and worker nodes [5].

2.5 Kubernetes objects

Kubernetes defines several types of objects, which constitutes its building blocks. Usually, a K8s resource object contains the following fields [**k8s_api_doc**]:

- `apiVersion`: the versioned schema of this representation of the object;
- `kind`: a string value representing the REST resource this object represents;
- `ObjectMeta`: metadata about the object, such as its name, annotations, labels etc.;
- `ResourceSpec`: defined by the user, it describes the desired state of the object;
- `ResourceStatus`: filled in by the server, it reports the current state of the resource.

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

- **Create**: create the resource in the storage backend; once a resource is created, the system applies the desired state.
- **Read**: comes with 3 variants
  - **Get**: retrieve a specific resource object by name;
  - **List**: retrieve all resource objects of a specific type within a namespace, and the results can be restricted to resources matching a selector query;
  - **Watch**: stream results for an object(s) as it is updated.

- **Update**: comes with 2 forms
  - **Replace**: replace the existing spec with the provided one;
  - **Patch**: apply a change to a specific field.
- **Delete**: delete a resource; depending on the specific resource, child objects may or may not be garbage collected by the server.

In the following we illustrate the main objects needed in the next chapters.

2.5.1 Label & Selector

Labels are key-value pairs attached to a K8s object and used to organize and mark a subset of objects. Selectors are the grouping primitives which allow to select a set of objects with the same label.

2.5.2 Namespace

Namespaces are virtual partitions of the cluster. By default, Kubernetes creates 4 Namespaces:

- **kube-system**: it contains objects created by K8s system, mainly control-plane agents;
- **default**: it contains objects and resources created by users and it is the one used by default;
- **kube-public**: readable by everyone (even not authenticated users), it is used for special purposes like exposing cluster public information;
- **kube-node-lease**: it maintains objects for heartbeat data from nodes.

It is a good practice to split the cluster into many Namespaces in order to better virtualize the cluster.

2.5.3 Pod

Pods are the basic processing units in Kubernetes. A pod is a logic collection of one or more containers which share the same network and storage, and are scheduled together on the same pod. Pods are ephemeral and have no auto-repair capacities: for this reason they are usually managed by a controller which handles replication, fault-tolerance, self-healing etc.

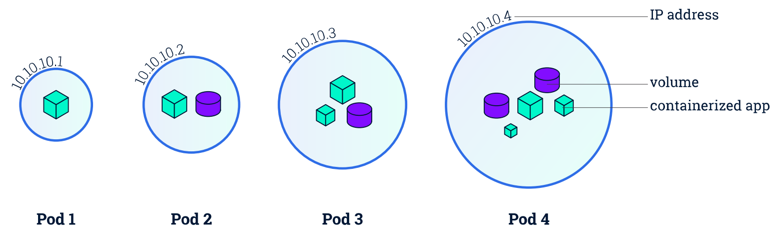

Figure 2.5: Kubernetes pods [5].

2.5.4 ReplicaSet

ReplicaSets control a set of pods allowing to scale the number of pods currently in execution. If a pod in the set is deleted, the ReplicaSet notices that the current number of replicas (read from the `Status`) is different from the desired one (specified in the `Spec`) and creates a new pod. Usually ReplicaSets are not used directly: a higher-level concept is provided by Kubernetes, called **Deployment**.

2.5.5 Deployment

Deployments manage the creation, update and deletion of pods. A Deployment automatically creates a ReplicaSet, which then creates the desired number of pods. For this reason an application is typically executed within a Deployment and not in a single pod. The listing 2.1 is an example of deployment.

Listing 2.1: Basic example of Kubernetes Deployment [5].

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

```
18      - name: nginx
19        image: nginx:1.7.9
20        ports:
21        - containerPort: 80
```

The code above allows to create a Deployment with name `nginx-deployment` and a label `app`, with value `nginx`. It creates three replicated pods and, as defined in the `selector` field, manages all the pods labelled as `app:nginx`. The template field shows the information of the created pods: they are labelled `app:nginx` and launch one container which runs the nginx DockerHub image at version 1.7.9 on port 80.

2.5.6 DaemonSet

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created [5]. Some typical uses of a DaemonSet are:

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

Listing 2.2: Basic example of Kubernetes daemonset [5].

```
1  apiVersion: apps/v1
2  kind: DaemonSet
3  metadata:
4    name: fluentd-elasticsearch
5    namespace: kube-system
6    labels:
7      k8s-app: fluentd-logging
8  spec:
9    selector:
10     matchLabels:
11       name: fluentd-elasticsearch
12   template:
13     metadata:
14       labels:
15         name: fluentd-elasticsearch
16     spec:
17       tolerations:
18       - key: node-role.kubernetes.io/master
19         effect: NoSchedule
20       containers:
```

```
21        - name: fluentd-elasticsearch
22          image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
```

2.5.7 Service

A Service is an abstract way to expose an application running on a set of Pods as a network service. It can have different access scopes depending on its ServiceType:

- **ClusterIP**: Service accessible only from within the cluster, it is the default type;
- **NodePort**: exposes the Service on a static port of each Node's IP; the NodePort Service can be accessed, from outside the cluster, by contacting `<NodeIP>:<NodePort>`;
- **LoadBalancer**: exposes the Service externally using a cloud provider's load balancer;
- **ExternalName**: maps the Service to an external one so that local apps can access it.

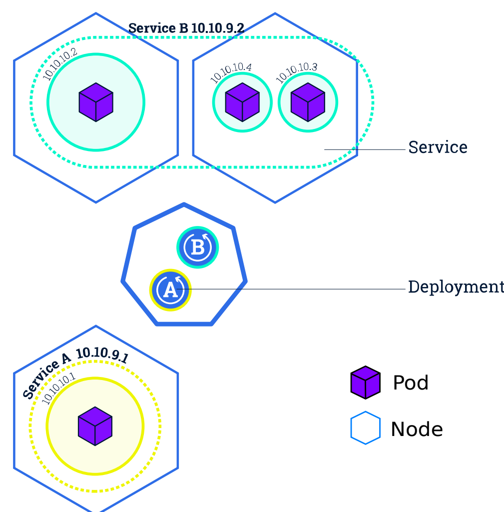

Figure 2.6: Kubernetes Services [5].

The following Service is named `my-service` and redirects requests coming from TCP port 80 to port 9376 of any Pod with the `app=MyApp` label.

Listing 2.3: Basic example of Kubernetes Service [5].

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

2.6 Kubernetes network architecture

Kubernetes defines a network model that helps provide simplicity and consistency across a range of networking environments and network implementations. The Kubernetes network model provides the foundation for understanding how containers, pods, and services within Kubernetes communicate with each other [14]. The Kubernetes network model specifies:

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

As a result, pods can be treated much like VMs or hosts (they all have unique IP addresses), and the containers within pods very much like processes running within a VM or host (they run in the same network namespace and share an IP address). This model makes it easier for applications to be migrated from VMs and hosts to pods managed by Kubernetes. In addition, because isolation is defined using network policies rather than the structure of the network, the network remains

simple to understand. This style of network is sometimes referred to as a “flat network”.

2.6.1 Container communication within same pod

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

2.6.2 Pod communication within the same node

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

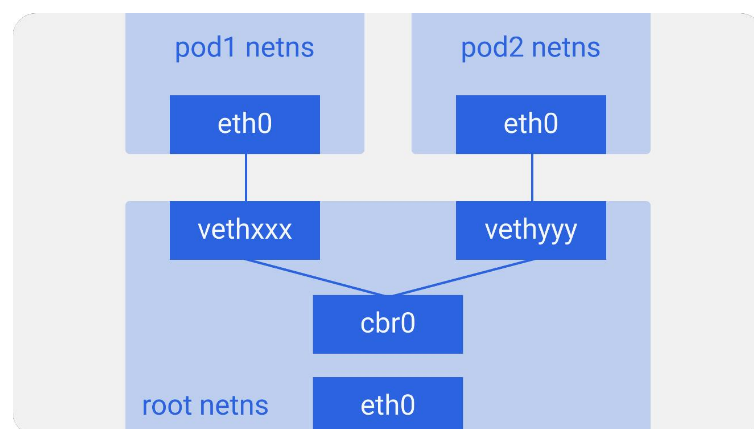

Figure 2.7: Pod to pod communication within same node.

2.6.3 Pod communication on different nodes

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

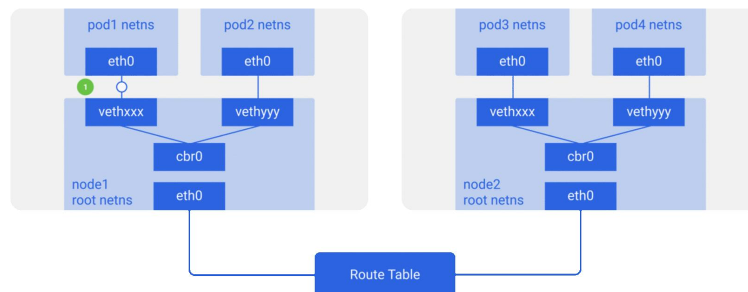

Figure 2.8: Pod to pod communication across different nodes.

2.6.4 CNI (Container Network Interface)

CNI (Container Network Interface) is a Cloud Native Computing Foundation project consisting of a specification and libraries for writing plugins to configure network interfaces in Linux containers. CNI concerns itself only with network connectivity of containers and removing allocated resources when the container is deleted. Kubernetes uses the CNI specifications and plug-ins to orchestrate networking. Also, it can address other container's IP addresses without using the Network Address Translation (NAT). Every time a Pod is initialized or removed, the default CNI plug- in is called with the default configuration, which this CNI plug-in creates a pseudo interface, attaches it to the underlay network, sets IP Address, routes, and maps it to the Pod namespace. It should be passed –network-plugin = cni to the Kubelete when launching it for using the CNI plugin. If the environment is not using the default configuration directory (`/etc/cni.net.d`), the CNI plugin passes the correct configuration directory as a value to `-cni-conf-dir`.

Moreover, the Kubelet looks for the CNI plugin binary at `/opt/cni/bin`, but it can be specified an alternative location with `-cni-bin-dir`.

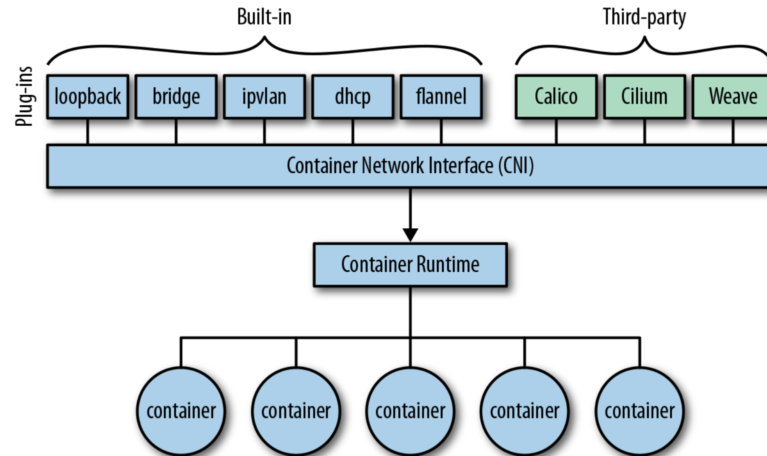

Figure 2.9: Container network interface [15]

2.6.5 Pod to service networking

Pod IP addresses are not durable and will appear and disappear in response to scaling up or down, application crashes, or node reboots. Each of these events can make the Pod IP address change without warning. Services were built into Kubernetes to address this problem. The Kubernetes service manages the state of Pods, allowing us to track a set of the pod IP address that dynamically changes over time. Services act as an abstraction over Pods and assign a single virtual IP address to a group of Pod IP addresses. Any traffic addressed to the virtual IP of the service will be routed to the set of Pods that are associated with the virtual IP. This allows the set of Pods associated with a service to change at any time clients only need to know the service's virtual IP, which does not change [16].

2.7 Virtual-Kubelet

Two Kubernetes-based tools which have been used during the development of this project are Virtual-Kubelet and Kubebuilder. Virtual Kubelet is an open source Kubernetes kubelet implementation that masquerades a cluster as a kubelet for the purposes of connecting Kubernetes to other APIs [6]. Virtual Kubelet is a Cloud Native Computing Foundation sandbox project.

The project offers a provider interface that developers need to implement in order to use it. The official documentation [6] says that "providers must provide

the following functionality to be considered a supported integration with Virtual Kubelet:

1. Provides the back-end plumbing necessary to support the lifecycle management of pods, containers and supporting resources in the context of Kubernetes.
2. Conforms to the current API provided by Virtual Kubelet.
3. Does not have access to the Kubernetes API Server and has a well-defined callback mechanism for getting data like secrets or configmaps".

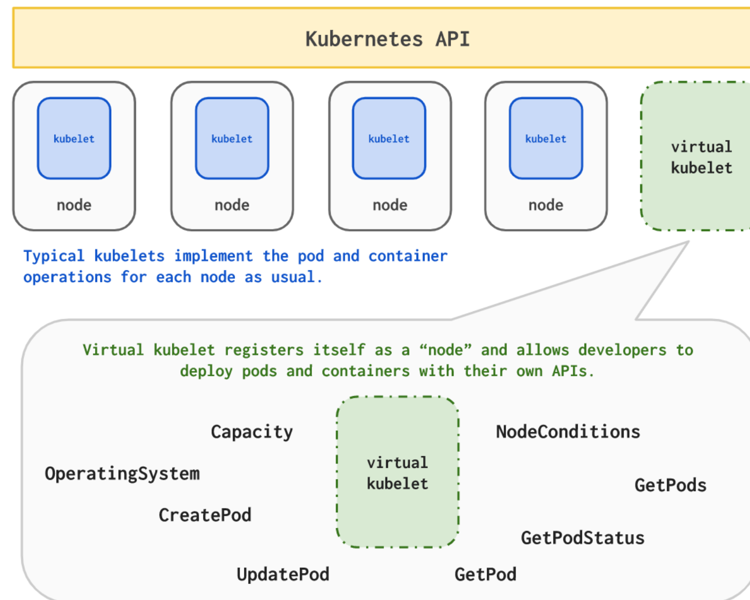

Figure 2.10: Virtual-Kubelet concept [6].

2.8 Kubebuilder

Kubebuilder is a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs) [7].

CustomResourceDefinition is an API resource offered by Kubernetes which allows to define Custom Resources (CRs) with a name and schema specified by the user. When a new CustomResourceDefinition is created, the Kubernetes API server creates a new RESTful resource path; the CRD can be either namespaced or cluster-scoped. The name of a CRD object must be a valid DNS subdomain name.

A **Custom Resource** is an endpoint in the Kubernetes API that is not available in a default Kubernetes installation and which frees users from writing their own

API server to handle them [5]. On their own, custom resources simply let you store and retrieve structured data. In order to have a more powerful management, you also need to provide a custom controller which executes a control loop over the custom resource it watches: this behaviour is called Operator pattern [17].

Kubebuilder helps a developer in defining his Custom Resource, taking automatically basic decisions and writing a lot of boilerplate code. These are the main actions operated by Kubebuilder [7]:

1. Create a new project directory.
2. Create one or more resource APIs as CRDs and then add fields to the resources.
3. Implement reconcile loops in controllers and watch additional resources.
4. Test by running against a cluster (self-installs CRDs and starts controllers automatically).
5. Update bootstrapped integration tests to test new fields and business logic.
6. Build and publish a container from the provided Dockerfile.

Chapter 3

Multi-Cluster networking: state of the art

Emerging capabilities of networks have enabled cloud to successfully provide on-demand services which can unilaterally provision computing capabilities such as servers, network, OS and storage. The network has possibly the highest impact on a cloud deployment's success because users need to access applications and data residing in the cloud from remote locations. For this reason, cloud computing requires a holistic approach to designing networks. A cloud network infrastructure must support the dynamic volatility of network traffic and its inherent characteristics, which includes not only performance, throughput, and latency, but also security, and availability.

3.1 Submariner

Submariner consists of several main components that work in conjunction to securely connect workloads across multiple Kubernetes clusters.

3.1.1 Broker

Submariner uses a central Broker component to facilitate the exchange of metadata information between Gateway Engines deployed in participating clusters. The Broker is basically a set of Custom Resource Definitions (CRDs) backed by the Kubernetes datastore. The Broker also defines a ServiceAccount and RBAC components to enable other Submariner components to securely access the Broker's API. There are no Pods or Services deployed with the Broker [18].

Submariner defines two CRDs that are exchanged via the Broker: `Endpoint` and `Cluster`. The `Endpoint` CRD contains the information about the active Gateway

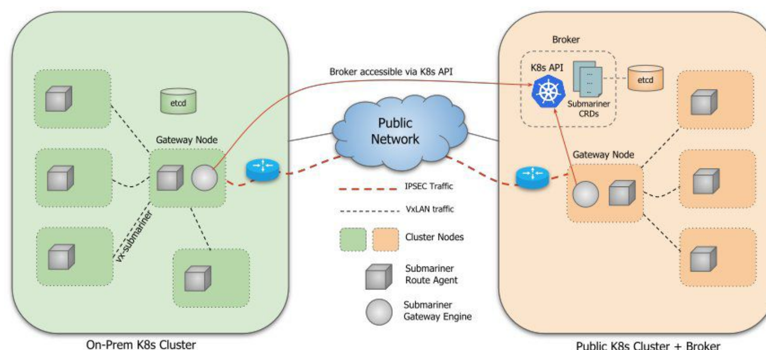

Figure 3.1: Basic architecture of Submariner [18]

Engine in a cluster, such as its IP, needed for clusters to connect to one another. The `Cluster` CRD contains static information about the originating cluster, such as its Service and Pod CIDRs.

The Broker is a singleton component that is deployed on a cluster whose Kubernetes API must be accessible by all of the participating clusters. If there is a mix of on-premises and public clusters, the Broker can be deployed on a public cluster. The Broker cluster may be one of the participating clusters or a standalone cluster without the other Submariner components deployed. The Gateway Engine components deployed in each participating cluster are configured with the information to securely connect to the Broker cluster's API.

3.1.2 Gateway engine

The Gateway Engine component is deployed in each participating cluster and is responsible for establishing secure tunnels to other clusters [18]. It has a pluggable architecture for the cable engine component that maintains the tunnels. The following implementations are available:

1. IPsec implementation using strongSwan (via the goStrongswanVici library); this is currently the default;
2. IPsec implementation using Libreswan;
3. implementation for WireGuard.

Instances of the Gateway Engine run on specifically designated nodes in a cluster of which there may be more than one for fault tolerance. Submariner supports active/passive High Availability for the Gateway Engine component, which means that there is only one active Gateway Engine instance at a time in a cluster. They perform a leader election process to determine the active instance and the others await in standby mode ready to take over should the active instance fail.

The active Gateway Engine communicates with the central Broker to advertise its `Endpoint` and `Cluster` resources to the other clusters connected to the Broker, also ensuring that it is the sole Endpoint for its cluster.

3.1.3 Route agent

The Route Agent component runs on every worker node in each participating cluster. It is responsible for setting up **VXLAN** tunnels and routing the cross cluster traffic from the node to the cluster's active Gateway Engine which subsequently sends the traffic to the destination cluster [18].

When running on the same node as the active Gateway Engine, Route Agent creates a **VXLAN** VTEP interface to which Route Agent instances running on the other worker nodes in the local cluster connect by establishing a **VXLAN** tunnel with the VTEP of the active Gateway Engine node. The **MTU** of the **VXLAN**tunnel is configured based on the **MTU** of the default interface on the host minus the**VXLAN** overhead.

Route Agents use Endpoint resources synced from other clusters to configure routes and to program the necessary IP table rules to enable full cross-cluster connectivity.

When the active Gateway Engine fails and a new Gateway Engine takes over, Route Agents will automatically update the route tables on each node to point to the new active Gateway Engine node.

3.2 Cilium cluster mesh

Cilium is open source software for providing and transparently securing network connectivity and load-balancing between application workloads such as application containers or processes. Cilium operates at Layer 3/4 to provide traditional networking and security services as well as Layer 7 to protect and secure use of modern application protocols such as HTTP, gRPC and Kafka. Cilium is integrated into common orchestration frameworks such as Kubernetes and Mesos [19].

A new Linux kernel technology called eBPF is at the foundation of Cilium. It supports dynamic insertion of eBPF bytecode into the Linux kernel at various integration points such as: network IO, application sockets, and trace-points to implement security, networking and visibility logic. eBPF is highly efficient and flexible.

Cluster Mesh, Cilium's multi-cluster implementation provides:

1. Pod IP routing across multiple Kubernetes clusters at native performance via tunneling or direct-routing without requiring any gateways or proxies;

2. Transparent service discovery with standard Kubernetes services and coredns/kube-dns;

3. Network policy enforcement spanning multiple clusters. Policies can be specified as Kubernetes NetworkPolicy resource or the extended CiliumNetworkPolicy CRD;

4. Transparent encryption for all communication between nodes in the local cluster as well as across cluster boundaries.

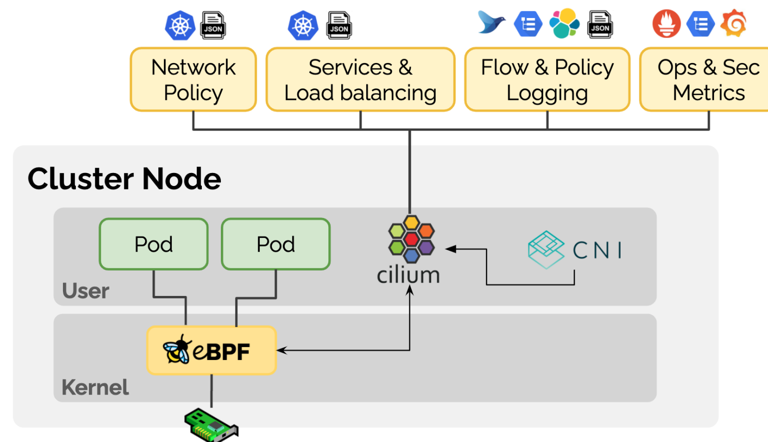

Figure 3.2: Basic architecture of Cilium [19]

3.2.1 Control plane

In order for the solution to work some requirements have to be met by the existing clusters where it is deployed:

- All Kubernetes worker nodes must be assigned a unique IP address and all worker nodes must have IP connectivity between each other;
- All clusters must be assigned unique PodCIDR ranges;
- Cilium must be configured to use etcd as the kvstore;
- The network between clusters must allow the inter-cluster communication.

The control plane is based on etcd and kept as minimalistic as possible. Each Kubernetes cluster maintains its own etcd cluster which contains the state of that cluster. State from multiple clusters is never mixed in etcd itself. The etcd is exposed via a set of etcd proxies. Cilium agents running in other clusters connect

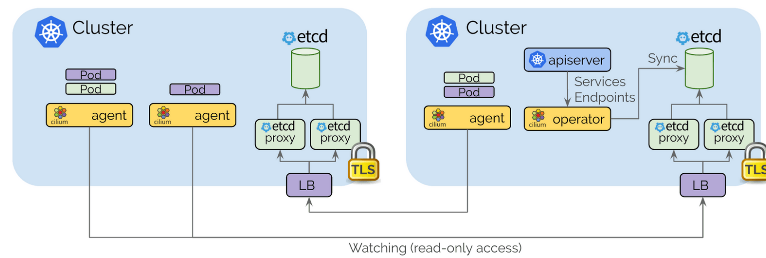

Figure 3.3: Control Plane of Cilium [20]

to the etcd proxies to watch for changes and replicate the multi-cluster relevant state into their own cluster. Use of etcd proxies ensures scalability of etcd watchers. Access is protected with TLS certificates. The access from one cluster into another is always read-only. This ensures that the failure domain remains unchanged, i.e. failures in one cluster never propagate into other clusters. The configuration is done via a simple Kubernetes secrets resource that contains the addressing information of the remote etcd proxies along with the cluster name and the certificates required to access the etcd proxies.

3.2.2 Pod IP routing

The pod IP routing is the foundation of the multi-cluster ability. It allows pods across clusters to reach each other via their pod IPs. Cilium can operate in several modes to perform pod IP routing. All of them are capable to perform multi-cluster pod IP routing.

Tunneling mode

Tunneling mode encapsulates all network packets emitted by pods in a so-called encapsulation header. The encapsulation header can consist of a **VXLAN** or **Geneve** frame. This encapsulation frame is then transmitted via a standard **UDP** packet header. The concept is similar to a **VPN** tunnel.

- The pod IPs are never visible on the underlying network. The network only sees the IP addresses of the worker nodes. This can simplify installation and firewall rules.
- The additional network headers required will reduce the theoretical maximum throughput of the network. The exact cost will depend on the configured **MTU** and will be more noticeable when using a traditional **MTU** of 1500 compared to the use of jumbo frames at **MTU** 9000.

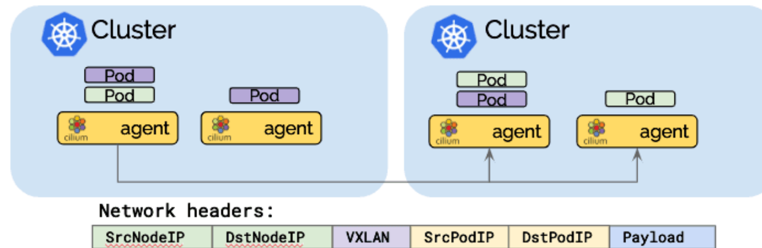

Figure 3.4: Cilium tunneling mode [20]

- In order to not cause excessive CPU, the entire networking stack including the underlying hardware has to support checksum and segmentation offload to calculate the checksum and perform the segmentation in hardware just as it is done for "regular" network packets.

Direct-routing mode

In the direct routing mode, all network packets are routed directly to the network. This requires the network to be capable of routing pod IPs.When a point is reached where the network no longer understands pod IPs, network packet addresses need to be masqueraded.

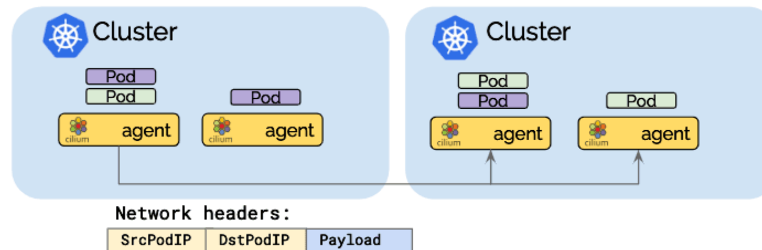

Figure 3.5: Cilium direct-routing mode [20]

Chapter 4

Multi-Cluster networking: design

Some organizations have multiple Kubernetes clusters. Usually the number of clusters is quiet static but the number of nodes in a cluster and the number of pods in a service may change frequently according to load and growth. The reasons to have multiple clusters include:

- strict security policies requiring isolation of one class of work from another;
- location, placing services in specific locations to address availability, latency, and locality needs;
- test clusters to canary new Kubernetes releases or other cluster software.

When the resources in one clusters are not enough, usually new hardware is made available and installed in the existing clusters. It is not an acceptable solution when the heavy load peaks are transitory causing the new hardware not to be used most of the time.

Here comes **Liqo**, an open source project started at **Politecnico of Turin** that allows Kubernetes to seamlessly and securely share **resources** and **services**, so you can run your tasks on any other cluster available nearby.

4.1 The problem

When a cluster offloads some jobs on another cluster there comes some dependencies that have to be satisfied. Considering a **micro-services** application such the one depicted in figure 4.1:

- Micro-service A consumes micro-service C;

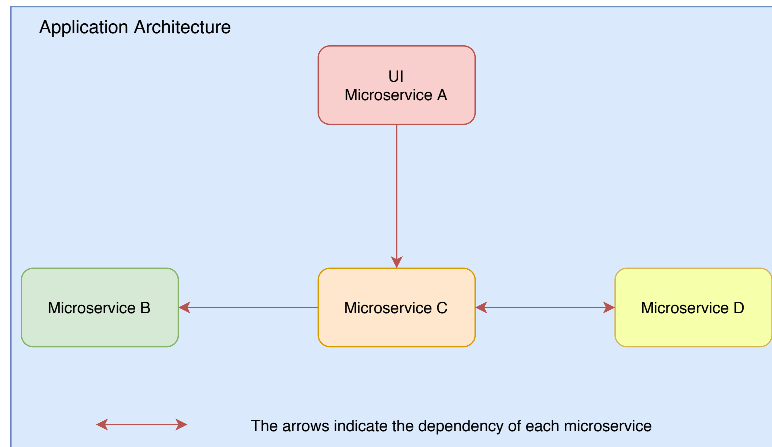

Figure 4.1: Micro-services dependencies

- Micro-service C consumes micro-service B and D;
- Micros-service D consumes micro-service C.

Picturing this application in multi-cluster environment it could be something like figure 4.2 The two **micro-services** C and D can communicate with each other

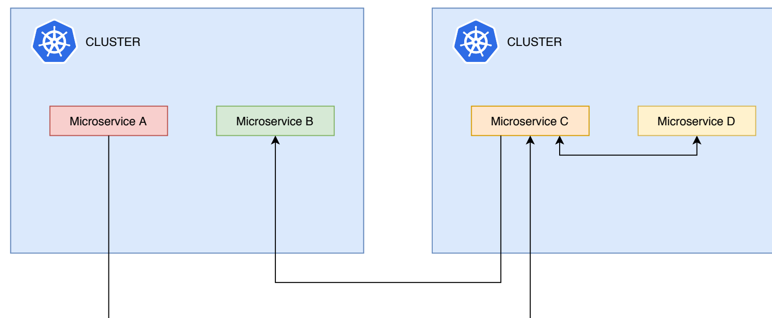

Figure 4.2: Micro-services deployed in two clusters

since they are in the same cluster, but C can not consume B and the same is for A depending on C.

By default the Kubernetes ecosystem does not offer solutions for inter-cluster networking. Each cluster ends up by having its own network meaning that overlay networks, pods, services are routable and reachable only from inside the cluster itself. The aim of this work is to design and implement a network plug-in for Kubernetes in order to inter-connect multiple clusters. Once two clusters discovers each other and exchanges the network parameters the plug-in configures the required resources to allow the `pod-to-pod` and `node-to-pod` communication across the clusters.

Another scenario figure 4.2 could be that a cluster called Cluster1 has peering sessions with clusters:

- Cluster2;
- Cluster3;
- Cluster4.

An actor deploys its application in Cluster1 made up by four micro-services called:

- A: running in Cluster1
- B: running in Cluster4
- C: running in Cluster2
- D: running in cluster3

In order for the application to work, there should be also a network connection established between Cluster2 and Cluster3, and also between Cluster2 and Cluster4. But this clusters could not be in an active session peering and hence no network connection between them.

4.1.1 Compatibility with running clusters

The aim of the **Liqo** project is to create **dynamic**, **opportunistic** data centers including also commodity desktops computers, laptops, single boards, other than powerful server machines. The clusters participating to this "peering" model, without any central point of control, establish dynamic and automatic relationships among them.

Since this peering model takes per granted that the clusters participating in a "peering" session are not necessarily under the administration of the same entity than the network plug-in has to be flexible and run on top of existing network configurations. It supports the main CNI such as:

- **Calico**: an open source networking and network security solution for containers, virtual machines, and native host-based workloads;
- **Flannel**: a simple and easy way to configure a layer 3 network fabric designed for Kubernetes;
- **Canal**: an integration of **Calico** and **Flannel**

The network inter-connection between the peering clusters is established only if necessary, when resources and services are shared between them.

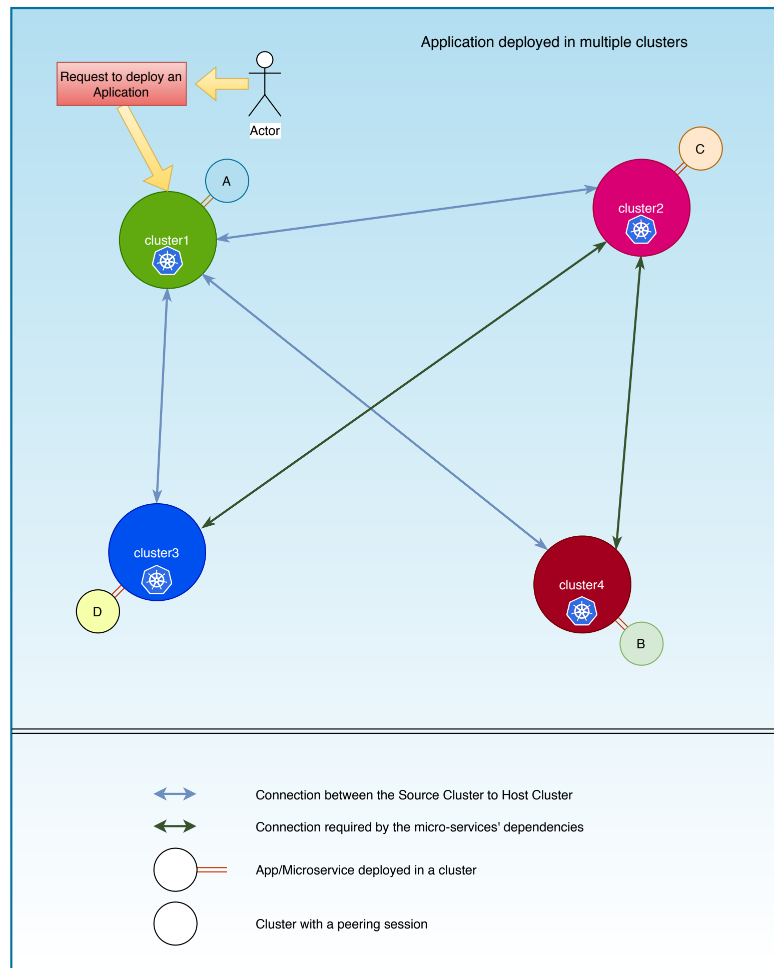

Figure 4.3: Micro-services deployed in multiple clusters

4.2 Architecture

Here we describe the general architecture of the network plug-in. The components that realize the plug-in functionalities are:

- **IPAM-Module**
- **Route-Agent**
- **Tunnel-Agent**

A general overview of this components is displayed in figure 4.4

Given two Kubernetes clusters here called Cluster1 and Cluster2 in order share resources among them all the `Pods` running in Cluster1 have to reach the `Pods`

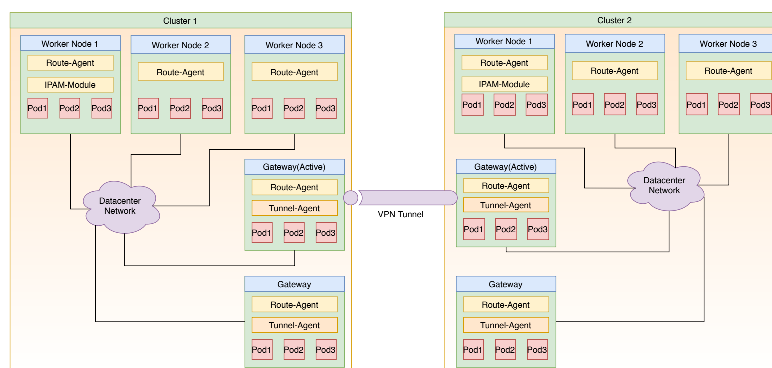

Figure 4.4: Network plug-in architecture

running in Cluster2 and vice-versa. But that is not enough, because `Pods` could be behind Kubernetes services such as a **NodePort** which implies that also the `Nodes` should be able to reach all the `Pods`, in this case the remote ones.

The first step is to have a point-to-point connection between the two clusters. That can be accomplished by establishing a **VPN** connection using two `Nodes` as endpoints, one residing in Cluster1 and the other in Cluster2. Such `Nodes` act as **gateways** for the resources that lives outside them. All the `Nodes` of a cluster needs to be configured how to route the traffic for a remote peering cluster to the **gateway node**.

IPAM-Module

Our solution supports clusters with overlapping networks, automatically resolving conflicts and assigning new network address spaces when necessary. The **IPAM-module** (IP address management) manages the assignment and use of IP address spaces for the peering clusters. The network address spaces that could be a source of conflicts are the **PodCIDR** and **ServiceCIDR**. The former is the IP pool used to assign to each pod an IP address and the later is used for the services created in Kubernetes. Usually this address spaces belong to the private IP address ranges, routable and reachable only from inside the data center. Other than the two mentioned networks there could be IP address spaces already in use inside the data center which could have conflicts with the **PodCIDR** and **ServiceCIDR** of the peering clusters.

The module needs to know in advance what are the IP address spaces used by the own cluster and the possible networks already used in the data center such as overlay networks. Having a new peering cluster that wants to establish a peering session, it has to make available his network configuration specifying his own network address spaces. The flow chart in figure 4.5 describes the main logic

behind this module.

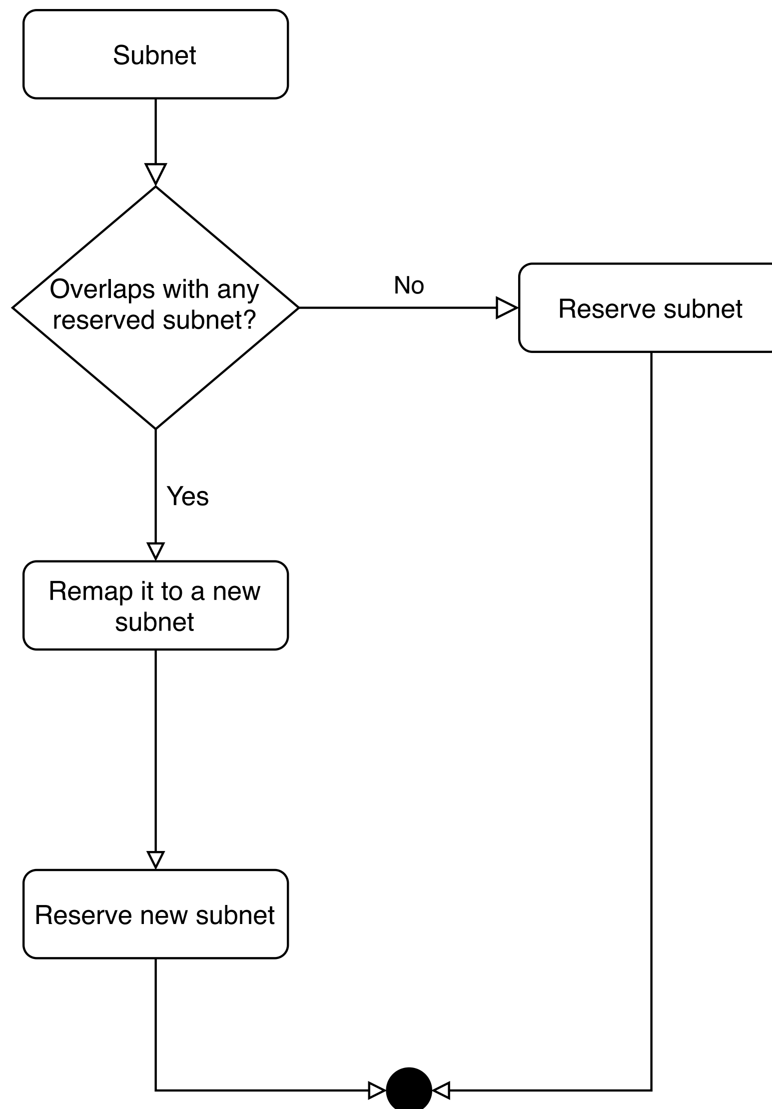

Figure 4.5: IPAM flow chart

Tunnel-Agent

The **Tunnel-Agent** is responsible to establish secure tunnels to the other peering clusters. It maintains the **VPN** tunnels and is in charge to configure them through a plug-able architecture supporting different technologies such as:

- IPSec

- OpenVPN
- Wireguard

The plug-able architecture is flexible and permits to support new **VPN** technologies just by implementing the interfaces required by the **Southbound driver**. It exports two interfaces:

- `InstallTunnel(configuration)`: it takes an opaque structure holding the configuration required based on the technology used;
- `RemoveTunnel(interfaceName)`: given the name of the network interface it takes care to bring down the **VPN** connection and clean up the resources.

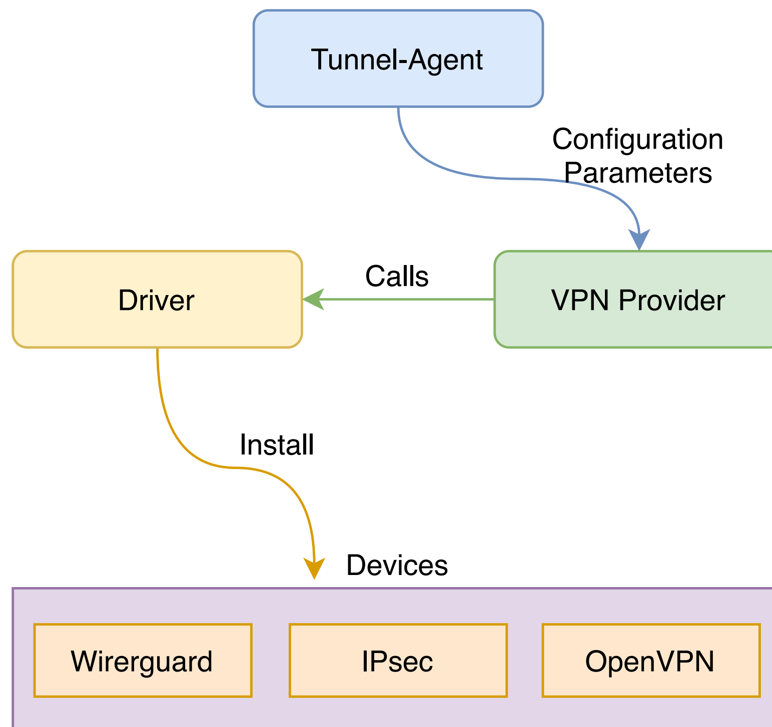

Figure 4.6: Southbound Driver Interface

The **Gateway node** is a single point of failure being it the only point of connection to the remote clusters. To overcome this limitation the **Tunnel-Agent** is deployed on multiple nodes, but only one instance is active managing the traffic for remote services. In case of failure of the active **Gateway** another node will change its status from `Idle` to `Active` configuring the **VPN** tunnels to the peering clusters.

Route-Agent

Pods running in `nodes` that are not the **Active Gateway** needs to know how to reach the **Gateway** in order to consume remote services. The **Route-Agent** is in charge of configuring the routing tables on each node to send all the traffic to the right `node` acting as the **Active Gateway**. A **VXLAN** overlay network is created. The traffic destined to a remote peering cluster is routed through the overlay network in order to reach the **Gateway**. The traffic for remote clusters is separated from **east-west traffic**. The **Route-Agent** handles only the outgoing traffic for a peering cluster such as depicted in figure 4.7. The packets are routed through the **VPN** tunnel and reach the remote cluster, where they are handled by the local CNI and routed to the correct node where the requested `POD` is running. Doing so we do not need to handle the traffic originated from a peering cluster, but it is treated the same as **east-west** and **north-south** traffic by the local CNI.

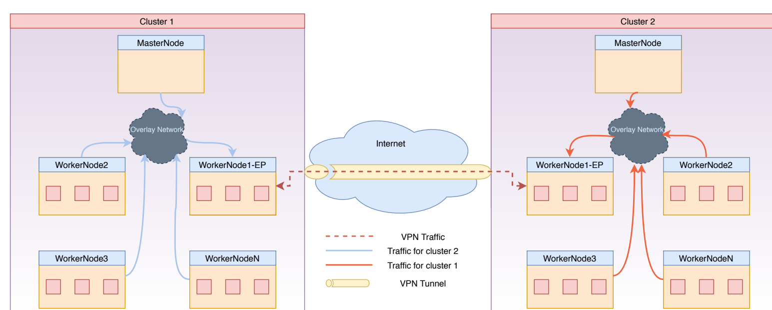

Figure 4.7: VXLAN overlay network

The **Route-Agent** also manages the `iptables-rules` in all the nodes in order to prevent **SNAT**(source natting) when the **PODS** communicate with IP addresses that are outside the **PodCIDR**. When the **Agent** is running in the **Gateway** it inserts the rules to perform:

- **Single NAT**: when one of the two clusters have remapped the **Pod CIDR** of the other in to a new network address space;
- **Double NAT**: when the two clusters have remapped the each others' **Pod CIDR** to new network address spaces.

The **NAT** rules are needed only when the network address spaces used by the clusters overlaps.

Chapter 5

Multi-Cluster networking: implementation

This chapter will present the implementation of the software solution discussed in the previous chapter. The developed code has been written in Go, which is the language in which Kubernetes and all related projects are written. The modules implemented in this work are the following:

- **CRDReplicator**: an operator that connects to a peering cluster's API server and replicates local `CRDs` on it. `Labels` and `Selectors` are used to design which `CRDs` to replicate and on which cluster.

- **TunnelEndpointCreator**: this operator is in charge to get the network parameters received from a peering cluster and create a resource instance of kind `tunnelendpoints.net.liqo.io`. More information on this component in the next sections. It implements the **IPAM-Module** presented in section 4.2

- **Tunnel-Operator**: the component runs on the designated **Gateway Node** and consumes a resource of type `tunnelendpoints.net.liqo.io` and brings up a point to point **VPN** connection with a peering cluster. It implements some of the ideas of the **tunnel-agent** discussed in section 4.2.

- **Route-Operator**: this operator is deployed as a `Daemonset` and runs on each `node` of the cluster. It ensures that the correct **routes** and **iptables rules** have been installed to make remote services reachable. All the information needed by the operator are found on the resource of kind `tunnelendpoints.net.liqo.io` that represents a specific peering cluster. This operator is the software implementation of the **route-agent** presented in section 4.2.

5.1 Kubernetes programming interface

The Kubernetes programming interface in Go mainly consists of the `k8s.io/client-go` library. `Client-go` is a typical web service client library that supports all API types that are officially part of Kubernetes. It eases the creation of a Kubernetes client object used to access resources in a Kubernetes cluster. Here we are going to describe some concepts used to implement our solution.

Dynamic client

The dynamic client in `k8s.io/client-go/dynamic` is totally agnostic to known resources. It does not use any Go types other than `unstructured.Unstructured`, which is a straightforward representation of `YAML/JSON` object in Go. Doing so the **CRDRepicator** supports third party **CRDs** and the **core resources** of Kubernetes.

Watches

Clients offer a method called `Watch`. It gives an event interface for all changes (adds, removes, and updates) to objects. The use of the `watch.interface` is discouraged in favor of informers.

Informers

Informers give a higher-level programming interface for the most common use case for `watches`: in-memory caching and fast, indexed lookup of objects by name or other properties in-memory. They can react to changes of objects nearly in real-time instead of requiring polling requests. Specifically, they:

- Get input from the **API server** as events.
- Offer a client-like interface called `Lister` to get and list objects from the in-memory cache.
- Register event handlers for `adds`, `removes`, and `updates`.
- Implement the in-memory cache using a store.

They also have advanced error behavior: when the long-running watch connection breaks down, they recover from it by trying another watch request, picking up the event stream without losing any events. If the outage is long, and the **API server** lost events because **etcd** purged them from its database before the new watch request was successful, the informer will relist all objects.

5.2 CRDReplicator

The exchange of information between two peering clusters is crucial in order to enable the sharing of resources and services. A way could be a domain-specific protocol to exchange network parameters and configuration. This could be cumbersome to maintain and operate:

- extension of the protocol when new parameters need to be exchanged;
- converting the messages to the `CRD api` consumed by the Kubernetes operators;

The **CRDReplicator** leverages **declarative API server** mechanisms of Kubernetes to exchange configurations and general messages through `CRDs`. The operator could handle every type of `CRD`, it only needs to know the **GVR** (Group, Version, Resource). The **GVR** is a tuple that uniquely identifies a type of resource. Replicating a new `CRD` is as simple as adding its **GVR** to the configuration of the operator. Now we will describe how the **CRDReplicator** manages the connection to a peering cluster and how does it decide which instances of a given resource need to be replicated and where.

5.2.1 Labels and Selectors

Each peering cluster that participates to a peering session has a unique **ClusterID**. The operator uses this parameter to determine the replication destination of a given resource. For each resource that we want to replicate to a specific peering cluster we have to add two `labels`:

- `liqo.io/replication=true`: the operator uses this `label` to know if a resource has to be replicated;
- `liqo.io/remodeID='clusterID of destination peering cluster'`: the `label` tells the operator which is the destination cluster and the right **API server** where to create the resource.

It is not enough just to replicate a local resource to a peering cluster we have to keep track of the resources replicated and to reflect changes that are made to them by the services consuming them. So the the replicated resources present the following labels:

- `liqo.io/originID='clusterID of the origin cluster`: the label holds the **ClusterID** of the cluster that replicated the resource. This is needed for the services that consumes the resource to know who sent that configuration.

- `liqo.io/replicated=true`: it says that the resource is a replicated one and it is ready to be consumed by the operators running in the local cluster.
- **liqo.io/replication=false**: needed for the the instances of the **CRDReplicator** running in the peering cluster where the resource has been replicated. It knows that it does need to process and replicate the resource because it is already a replicated one.

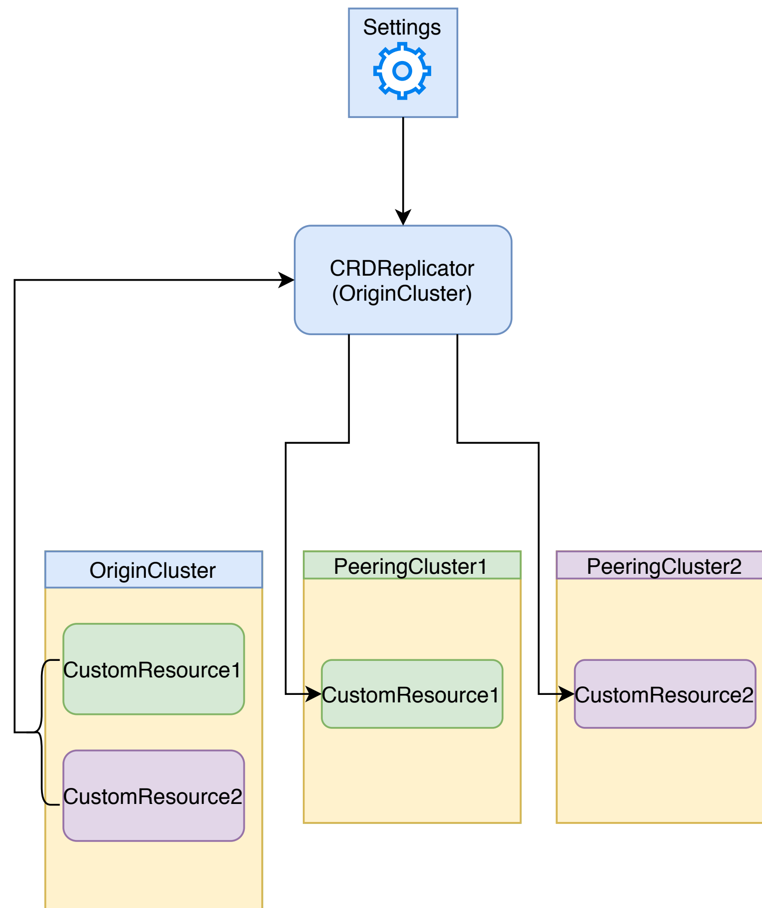

Figure 5.1: Replicating local resources to peering clusters

Referring to figure 5.1, the resource named `CustomResource1` in order to be replicated to the peering cluster with **ClusterID** 'PeeringCluster1' it should have the following `labels`:

- `liqo.io/replication=true`;
- `liqo.io/remoteID=PeeringCluster1`.

And the instance of the resource living in the peering cluster will present the labels:

- `liqo.io/originID=OriginCluster`;
- `liqo.io/replicated=true`;
- `liqo.io/replication=false`.

5.2.2 Watching Resources

Each resource to be replicated on remote peering clusters is given to the operator through a configuration **CRD**. The configuration is just a list of **GVRs**, an example si showed in listing 5.1

Listing 5.1: CRDReplicator configuration.

```
crdReplicatorConfig:
  resourcesToReplicate:
  - group: net.liqo.io
    version: v1alpha1
    resource: networkconfigs
  - group: net.liqo.io
    version: v1alpha1
    resource: tunnelendpoints
```

The operator creates a **dynamic client** to the **API servers** of the peering cluster and also a client to connect to the **API server** of the local cluster. A `shared informer factory` is instantiated for each **dynamic client**. The `shared factory` is then used to start the **informers** for the registered resources. For a better understanding in figure 5.2 is shown the relationship of the informers, factories, and clients.

5.2.3 Watching Local Resources

The operator needs to `watch` the instances for each registered resource denoted by its **GVR**. That is accomplished by using a **dynamic client** and **shared informers**. As shown in listing 5.2 the operator at startup time creates a **dynamic client** to interact with the local **API server**. After that a `dynamic shared informer factory` is instantiated, used to create the informers for each resource that we need to `watch`.

Listing 5.2: Creation of local informer shared factory.

```
dynClient := dynamic.NewForConfigOrDie(cfg)
dynFac := dynamicinformer.NewFilteredDynamicSharedInformerFactory(
    dynClient, crdReplicator.ResyncPeriod, metav1.NamespaceAll,
    crdReplicator.SetLabelsForLocalResources)
```

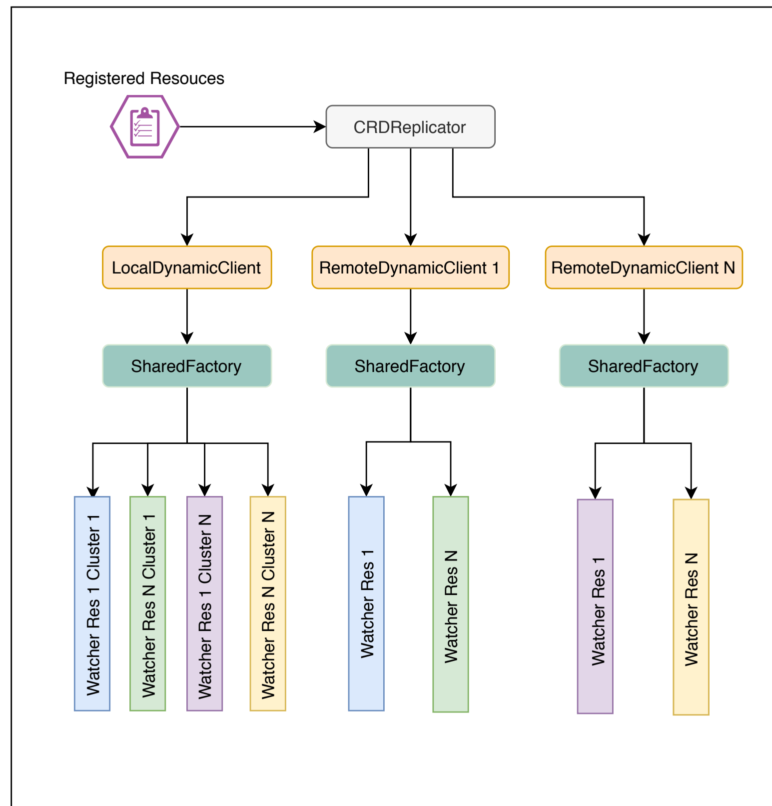

Figure 5.2: Informers, factories, and clients

A function shown in listing 5.3 is passed at creation time of the `shared factory` in order to filter only the resources that need to be replicated on a remote cluster.

Listing 5.3: Local selectors.

```
func SetLabelsForLocalResources(options *metav1.ListOptions) {
    //we want to watch only the resources that should be replicated
   on a remote cluster
    if options.LabelSelector == "" {
        newLabelSelector := []string{LocalLabelSelector, "=", "true"}
        options.LabelSelector = strings.Join(newLabelSelector, "")
    } else {
        newLabelSelector := []string{options.LabelSelector,
   LocalLabelSelector, "=", "true"}
        options.LabelSelector = strings.Join(newLabelSelector, "")
    }
}
```

While new peering clusters are discovered and a peering session is established the operator starts `local watchers` for the registered resources. For each remote

peering cluster and each registered resource to be replicated:

1. check if a `local watcher` is running and if not start a new one;

2. register event handlers for `adds`, `removes`, and `updates`;

3. save the `stop channel` passed to the `watcher`, which is used to track and stop, when needed, the running `watcher`.

Listing 5.4: Starting local watchers.

```
...
//for each remote cluster check if the local watchers are running for
    each registered resource
    for remCluster := range d.RemoteDynClients {
        watchers := d.LocalWatchers[remCluster]
        if watchers == nil {
            watchers = make(map[string]chan struct{})
        }
        for _, res := range d.RegisteredResources {
            //if there is not a running local watcher then start one
            if _, ok := watchers[res.String()]; !ok {
                stopCh := make(chan struct{})
                watchers[res.String()] = stopCh
                go d.Watcher(d.LocalDynSharedInformerFactory, res,
    cache.ResourceEventHandlerFuncs{
                    AddFunc:    d.AddFunc,
                    UpdateFunc: d.UpdateFunc,
                    DeleteFunc: d.DeleteFunc,
                }, stopCh)
                klog.Infof("%s -> starting local watcher for resource
    : %s", remCluster, res.String())
            }
        }
        d.LocalWatchers[remCluster] = watchers
    }
...
```

Add handler

The `d.AddFunc` showed in listing 5.4 is invoked each time a new resource is created. It checks if a `dynamic client` connected to the **API server** of the peering cluster, to which the resource has to be replicated, already exists. After that a the resource is created on the peering cluster if it does not exist.

Update handler

The `d.UpdateFunc` is called when modifications are performed on a local resource. It checks if the a copy of the modified resource exists on the peering cluster and if it is not present then creates a new one. Other wise it gets the remote instances and performs a diffs between the two instances. If there are differences than this changes are applied to the remote resource.

Delete handler

When a local resource is deleted it is also deleted on the peering clusters where it has been replicated. This is performed by the `d.DeleteFunc`.

5.3 Watching Remote Resources

Once a resource is replicated on a peering cluster, it is consumed hence modified. So the operator has to watch the replicated resources and reflect the changes to their **status** on the origin cluster. After that a **dynamic client** and a **shared informer factory** have been created the remote watchers are started as shown in listing 5.5

Listing 5.5: Starting remote watchers.

```
//for each remote cluster check if the remote watchers are running
    for each registered resource
    for remCluster, remDynFac := range d.
    RemoteDynSharedInformerFactory {
        watchers := d.RemoteWatchers[remCluster]
        if watchers == nil {
            watchers = make(map[string]chan struct{})
        }
        for _, res := range d.RegisteredResources {
            //if there is not then start one
            if _, ok := watchers[res.String()]; !ok {
                stopCh := make(chan struct{})
                watchers[res.String()] = stopCh
                go d.Watcher(remDynFac, res, cache.
    ResourceEventHandlerFuncs{
                    UpdateFunc: d.remoteModifiedWrapper,
                }, stopCh)
                klog.Infof("%s -> starting remote watcher for
    resource: %s", remCluster, res.String())
            }
        }
        d.RemoteWatchers[remCluster] = watchers
    }
```

Only the `update` handler is registered because we are interested only in updates of the **status** field of a replicated resource.

5.3.1 Network Parameters Exchange

Now we will see a real application of the **CRDReplicator**. It is used to exchange **network parameters** between two peering clusters. Before diving in the workflow we present the `networkconfigs.net.liqo.io` **API**.

NetworkConfigs

The **network parameters** are exchanged using a Custom Resource. Kubebuilder has been used to implement the operators and define the **CRD API**.

A `NetworkConfig` contains the four fields:

1. `metav1.TypedMeda` and `metav1.ObjectMeta`: the metadata object;

2. `Spec`: the desired state of the object;

3. `Status`: the observed state of the object.

Listing 5.6 shows how the custom resource is defined in golang.

Listing 5.6: NetworConfig API schema.

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

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

type NetworkConfigSpec struct {
    ClusterID string `json:"clusterID"`
    PodCIDR string `json:"podCIDR"`
    TunnelPublicIP string `json:"tunnelPublicIP"`
}

type NetworkConfigStatus struct {
    NATEnabled string `json:"natEnabled,omitempty"`
    PodCIDRNAT string `json:"podCIDRNAT,omitempty"`
}
```

`NetworkConfigSpec` holds the network parameters of the local cluster. The `ClusterID` is the **ID** of the peering cluster to whom we sent the parameters. The

`PodCIDR` is the address space of the Pods. And the `TunnelPublicIP` is the IP address of the node where the **VPN** tunnel will be created.

`NetworkConfigStatus` is reserved to be populated by the peering cluster. It has the `NATEnabled` field which says if the local cluster has been NATed by the remote one. The `PodCIDRNAT` holds the new address spaces used by the remote cluster to remap the original **PodCIDR** of the local cluster.

This way we have a protocol where the **sender** owns the `spec` field of a resource and is allowed to modify only this part. The **receiver** owns the `status` field and used it to give a feedback or communicate back the results of a request.

Workflow

Using the **NetworkConfig** custom resources and the CRDReplicator we are now able to exchange the parameters needed to establish a **VPN** connection.

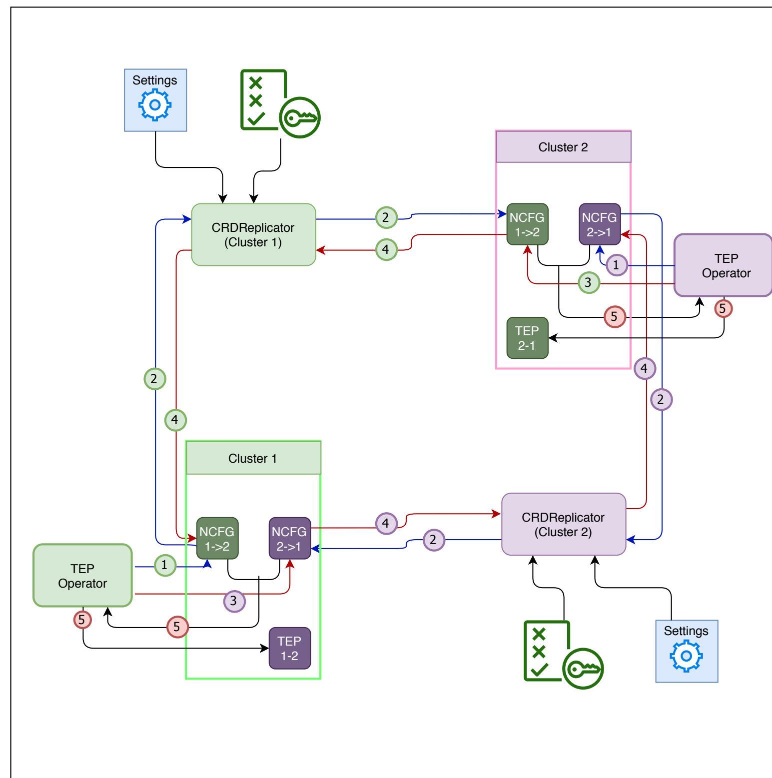

Figure 5.3: Exchanging network parameters

After that a two clusters start a peering session with each other they need to mutually exchange the **network parameters**. For the sake of clarity we will refer to figure 5.3 in order to explain all the steps involved in the process:

1. The network configuration of **Cluster1** is saved in a local custom resource (NCFG 1->2) containing the **ClusterID** of the remote cluster to whom is destined. Same step done in **Cluster2**;

2. NCFG is replicated to the remote cluster by the **CRDReplicator**, running in**Cluster1** who knows how to interact with the **API server** of **Cluster2**. Same step done in **Cluster2**;

3. In **Cluster2** the custom resource NCFG (1->2) is processed and its status updated with the **NAT** information if any. She same is done with NCFG (2->1) in **Cluster1**.

4. The **CRDReplicator** running in reflects this changes in the local resource NCFG(1->2). Same step done in **Cluster2**;

5. The **TunnelEndpointCreator** in **Cluster1** combining the `status` of NCFG (1->2) and the `spec` of NCFG (2->1) it has all the needed information to create the custom resource TEP (1-2) of type **tunnelendpoints.net.liqo.io**. More on the **TunnelEndpointCreator** and **tunnelendpoints** resource in the next section.

The blue arrows indicates that the connection handles only the `spec` field and the red arrows stands for the connection handling the `status` fields.

5.4 TunnelEndpointCreator

The **TunnelEndpointCreator** is an operator that manages the creation of the **networkconfigs** custom resources for the remote clusters. It also processes the **networkconfigs** received from the remote clusters remapping the remote **PodCIDRs** if they overlap with any of the address spaces used in the local cluster. And when the **networkconfigs** exchanged by the two clusters are ready the **TunnelEndpointCreator** creates a new resource of type **tunnelendpoints.net.liqo.io** which models the network interconnection between the two clusters.

5.4.1 TunnelEndpoint API

TunnelEndpoint custom resource is used to model the point-to-point connection among the clusters. It is the resource consumed by the **Route-Operator** and **Tunnel-Operator**. As shown in listing 5.7 it has the same `spec` values as the **networconfig API** (section 5.3.1).

Listing 5.7: TunnelEndpoint API schema.

```
// TunnelEndpoint is the Schema for the endpoints API
```

```
2  type TunnelEndpoint struct {
3      metav1.TypeMeta   `json:",inline"`
4      metav1.ObjectMeta `json:"metadata,omitempty"`
5
6      Spec   TunnelEndpointSpec   `json:"spec,omitempty"`
7      Status TunnelEndpointStatus `json:"status,omitempty"`
8  }
9
10 // TunnelEndpointSpec defines the desired state of TunnelEndpoint
11 type TunnelEndpointSpec struct {
12     ClusterID      string `json:"clusterID"`
13     PodCIDR        string `json:"podCIDR"`
14     TunnelPublicIP string `json:"tunnelPublicIP"`
15 }
16
17 // TunnelEndpointStatus defines the observed state of TunnelEndpoint
18 type TunnelEndpointStatus struct {
19     Phase                 string `json:"phase,omitempty"`
20     LocalRemappedPodCIDR  string `json:"localRemappedPodCIDR,
    omitempty"`
21     RemoteRemappedPodCIDR string `json:"remoteRemappedPodCIDR,
    omitempty"`
22     RemoteTunnelPublicIP  string `json:"remoteTunnelPublicIP,
    omitempty"`
23     LocalTunnelPublicIP   string `json:"localTunnelPublicIP,omitempty
    "`
24     TunnelIFaceIndex      int    `json:"tunnelIFaceIndex,omitempty"`
25     TunnelIFaceName       string `json:"tunnelIFaceName,omitempty"`
26 }
```

The **TunnelEndpointStatus** is made up by the following fields:

- **Phase**: when set the values is `Ready`. Used by the **Route-Operator** to know then the **tunnel network interface** has been installed and configured;
- **LocalRemappedPodCIDR**: the address space used by the peering cluster to remap the local **PodCIDR**;
- **RemoteRemappedPodCIDR**: the address space used by the local cluster to remap the **PodCIDR** of the peering cluster;
- **RemoteTunnelPublicIP**: the IP address of the node that has the role of **Gateway** on the peering cluster;
- **LocalTunnelPublicIP**: the IP address of the node that has the role of **Gateway** on the local cluster;
- **TunnelIFaceIndex**: the index value of the tunnel network interface which is a unique identifying number for a physical or logical network device;

- **TunnelIfaceName**: name of the tunnel network interface.

5.4.2 IPAM

The **TunnelEndpointCreator** implements a simple **IPAM** used to manage peering clusters that have overlapping **PodCIDR** address spaces. The **IPAM** is defined as a golang `interface`. The interface declaration is shown in listing 5.8.

Listing 5.8: IPAM interface.

```
type Ipam interface {
    Init() error
    GetNewSubnetPerCluster(network *net.IPNet, clusterID string) (*
   net.IPNet, error)
    RemoveReservedSubnet(clusterID string)
}
```

Now we will consider how this interface has been implemented in the operator. A golang `struct` of type `IpManager` has been declared with different nested fields:

- UsedSubnets: a map which contains all the currently used networks;
- FreeSubnets: a map containing all the subnets derived from the network `10.0.0.0/8`. The free networks has as **netmask** `255.255.255.0`;
- ConflictingSubnets: it contains all the networks that ovelaps with any of the address spaces present in the UsedSubnet map;
- SubnetPerCluster: for each peering cluster it contains the subnet assigned to it. It could be the original **PodCIDR** or a new value assigned by the **IPAM**.

Listing 5.9: IPAM Implementation.

```
type IpManager struct {
    UsedSubnets         map[string]*net.IPNet
    FreeSubnets         map[string]*net.IPNet
    ConflictingSubnets  map[string]*net.IPNet
    SubnetPerCluster    map[string]*net.IPNet
}
```

Init method

The `Init()` function divides the **CIDR block** 10.0.0.0/8 in 256 subnets and saves them in the `FreeSubnets` map. It initializes the **IPAM** at start-up time.

Listing 5.10: Init function.

```
func (ip IpManager) Init() error {
    CIDRBlock := "10.0.0.0/16"
    _, subnet, err := net.ParseCIDR("10.0.0.0/16")
    if err != nil {
        klog.Errorf("unable to parse the first subnet %s: %s",
    CIDRBlock, err)
        return err
    }
    //The first subnet /16 is added to the FreeSubnets
    ip.FreeSubnets[subnet.String()] = subnet
    //here we divide the CIDRBlock 10.0.0.0/8 in 256 /16 subnets
    for i := 0; i < 255; i++ {
        subnet, _ = cidr.NextSubnet(subnet, 16)
        ip.FreeSubnets[subnet.String()] = subnet
    }
    return nil
}
```

GetNewSubnetPerCluster method

`GetNewSubnetPerCluster()` function is called when a new peering cluster sends its **network parameters**. It gets as input the **ClusterID** and **PodCIDR** of the remote cluster. First it checks if it has already assigned an subnet to remote cluster. If so, it returns the allocated subnet, and a `nil` for the error. If the remote cluster is a new one, then it checks that its **PodCIDR** does not overlaps with any of the subnets saved in `UsedSubnets` map and remove from the free pool all the address spaces that overlaps with cluster's **PodCIDR**. The conflicting subnets are added to the `ConflictingSubnets` map. If there are no conflicts the original **PodCIDR** is returned and reserved. Otherwise it will get a new subnet from the free pool, return it and add the original network address space to the `ConflictingSubnets`

Listing 5.11: GetNewSubnetPerCluster function.

```
func (ip IpManager) GetNewSubnetPerCluster(network *net.IPNet,
    clusterID string) (*net.IPNet, error) {
    //first check if we already have assigned a subnet to the cluster
    if _, ok := ip.SubnetPerCluster[clusterID]; ok {
        return ip.SubnetPerCluster[clusterID], nil
    }
    //check if the given network has conflicts with any of the used
    subnets
    if flag := VerifyNoOverlap(ip.UsedSubnets, network); flag {
        //if there are conflicts then get a free subnet from the pool
     and return it
        //return also a "true" value for the bool
        if subnet, err := ip.getNextSubnet(); err != nil {
```

```
11              return nil, err
12          } else {
13              ip.reserveSubnet(subnet, clusterID)
14              klog.Infof("%s -> NAT enabled, remapping original subnet
    %s to new subnet %s", clusterID, network.String(), subnet.String()
    )
15              return subnet, nil
16          }
17      }
18      ip.reserveSubnet(network, clusterID)
19      klog.Infof("%s -> NAT not needed, using original subnet %s",
    clusterID, network.String())
20      return network, nil
21  }
```

RemoveReservedSubnet method

`RemoveReservedSubnet()` function is used to free the network resources allocated for a remote cluster. The input is the **ClusterID** of the remote cluster that is leaving a peering session. The function checks if a subnet has been assigned to the cluster by getting it from the `SubnetsPerCluster`. If no subnet is found for the given **ClusterID** then it does need to free any resource. In the other case the method removes the reserved network address space from `UsedSubnets` and **SubnetPerCluster** maps. Then if any of the address spaces in the `ConflictinSubnets` map does not have any more conflicts with the subnets in the `UsedSubnets` is moved to the `FreeSubnets` map.

Listing 5.12: RemoveReservedSubnet function.

```
1  func (ip IpManager) RemoveReservedSubnet(clusterID string) {
2      subnet, ok := ip.SubnetPerCluster[clusterID]
3      if !ok {
4          return
5      }
6      //remove the subnet from the used ones
7      delete(ip.UsedSubnets, subnet.String())
8      delete(ip.SubnetPerCluster, clusterID)
9      //check if there are subnets in the conflicting map that can be
    made available in to the free pool
10     for _, net := range ip.ConflictingSubnets {
11         if overlap := VerifyNoOverlap(ip.UsedSubnets, net); !overlap
    {
12             delete(ip.ConflictingSubnets, net.String())
13             ip.FreeSubnets[net.String()] = net
14         }
15     }
16 }
```

5.4.3 Network API Management

The operator manages the **networkconfig** and **tunnelendpoint** resources, the former is used to exchange the network parameters and the later to model the interconnection between two clusters. We are going to describe how this resources are processed:

1. creation of a local **networkconfig**;
2. processing of a received **networkconfig**;
3. creation of a **tunnelendpoint** based on the exchanged network parameters.

Local networkconfigs

Given a peering cluster the operator creates a **networkconfig**. It sets the **spec** fields with the local **ClusterID**, **PodCIDR** and **GatewayIP**. Then it waits for the remote cluster to set the **status** of the resource as shown in section 5.3.1

Listing 5.13: Hetworkconfig spec.

```
...

Spec: netv1alpha1.NetworkConfigSpec{
    ClusterID:       localClusterID,
    PodCIDR:         localPodCIDR,
    TunnelPublicIP:  localGatewayIP,
}

...
```

Remote networkconfigs

When a **networkconfig** is received by a remote cluster the operator processes it. It takes the **remote PodCIDR** and using the **IPAM** implementation seen in section 5.4.2 reserves the subnet. If the remote **PodCIDR** has not bee remapped then the **status** of the the **netwoconfig** resource is set to:

- `.Status.PodCIDRNAT = None`
- `.Status.NATEnabled = "false"`

in the other hand if present conflicts with used subnets the **status** is set to:

- `.Status.PodCIDRNAT = newSubnet`
- `.Status.NATEnabled = "true"`

Listing 5.14 shows the function in charge of processing a remote **networkconfig**.

Listing 5.14: ProcessRemoteNetConfig function.

```
func (r *TunnelEndpointCreator) processRemoteNetConfig(netConfig *netv1alpha1) error {
    ...
    newSubnet, err := r.IPManager.GetNewSubnetPerCluster(clusterSubnet, netConfig.Labels[crdReplicator.RemoteLabelSelector])
    if err != nil {
        klog.Errorf("an error occurred while getting a new subnet for resource %s: %s", netConfig.Name, err)
        return err
    }
    if newSubnet.String() != clusterSubnet.String() {
        if newSubnet.String() != netConfig.Status.PodCIDRNAT {
            //update netConfig status
            netConfig.Status.PodCIDRNAT = newSubnet.String()
            netConfig.Status.NATEnabled = "true"
            err := r.Status().Update(context.Background(), netConfig)
            if err != nil {
                klog.Errorf("an error occurred while updating the status of resource %s: %s", netConfig.Name, err)
                return err
            }
        }
        return nil
    }
    if netConfig.Status.PodCIDRNAT != defaultPodCIDRValue {
        //update netConfig status
        netConfig.Status.PodCIDRNAT = defaultPodCIDRValue
        netConfig.Status.NATEnabled = "false"
        err := r.Status().Update(context.Background(), netConfig)
        if err != nil {
            klog.Errorf("an error occurred while updating the status of resource %s: %s", netConfig.Name, err)
            return err
        }
        return nil
    }
    return nil
}
```

Tunnelendpoints creation

A **tunnelendpoint** resource is the result of two **networkconfigs** resources. The first is the one that carries the network parameters of a local cluster and the second

carries the information of a remote cluster. The life-cycle of a **tunnelendpoints** endpoints instance is directly related to the **netwoconfigs** instances exchanged by the two clusters that wants to establish a point-to-point network connection. It is created only when the network parameters have been exchanged, and removed when the local instance of **networkconfigs** is removed.

An auxiliary data structure expressed in golang as `type networkParam struct` is used to save all the information needed to create a **tunnelendpoint** instance. The fields are the same described in section 5.4.1

Listing 5.15: Networkparam data structure.

```
type networkParam struct {
    remoteClusterID   string
    remoteGatewayIP   string
    remotePodCIDR     string
    remoteNatPodCIDR  string
    localGatewayIP    string
    localNatPodCIDR   string
}
```

For each local **networkconfigs** instance the operator performs the following actions:

1. check if the resources has been processed by the remote cluster, if not re-queue the resource and try later;
2. get the remote **networkconfigs** instance, if it is not present it retries later;
3. check if the remote instance is ready, otherwise retries later starting from step 1;
4. fill the `networkParam` data structure;
5. check if already exists a **tunnelendpoints** instance. If already present updates the fields that are different that the one in the `networkparam` object. If not found, then create the resource.

5.5 Tunnel-Operator

The **tunnel-operator** implements some of the features described in section 4.2. It reconciles the **tunnelendpoints** resources and for each of them it creates point-to-point connection with the remote peering cluster.

5.5.1 GRE Tunneling Protocol

So far the operator supports only the `GRE` protocol as technology to establish `VPN` tunnel between two clusters. Generic function has been implemented to manage the network interfaces on the `Gateway Node` where the operator is deployed.

The configuration parameters to know in order to create the tunnel network interface are displayed in listing 5.16

Listing 5.16: GRE configuration.

```
type gretunAttributes struct {
    name   string
    local  net.IP
    remote net.IP
    ttl    uint8
}
```

The `local` field is the IP address of the local **Gateway Node** and the `remote` is the one of the remote `Gateway Node`. Both are found on the **tunnelendpoint** resource.

The `InstallGreTunnel()` function requires as argument an instance of **tunnelendpoints** and return the interface `index`, `name` or an `error` if something goes wrong. It creates a network interface of type `GRE` and configures it to be up and running.

Listing 5.17: InstallGreTunnel function.

```
func InstallGreTunnel(endpoint *netv1alpha1.TunnelEndpoint) (int,
    string, error) {
    name := tunnelNamePrefix
    local, err := GetLocalTunnelPublicIP()
    if err != nil {
        return 0, "", err
    }
    remote := net.ParseIP(endpoint.Spec.TunnelPublicIP)
    ttl := tunnelTtl
    attr := gretunAttributes{
        name:   name,
        local:  local,
        remote: remote,
        ttl:    uint8(ttl),
    }
    gretunnel, err := newGretunInterface(&attr)
    if err != nil {
        return 0, "", err
    }
    if err != nil {
        return 0, "", err
    }
```

```
22      if err = gretunnel.setUp(); err != nil {
23          return 0, "", err
24      }
25      return gretunnel.link.Index, gretunnel.link.Name, nil
26  }
```

Another function called `RemoveGreTunnel()` is used to remove the network interface when two peering clusters ends a peering session. As the previous methods this one also takes a **tunnelendpoints** instance as argument. It checks if a network interface has been installed for the remote cluster, and if so it manages to remove it.

Listing 5.18: RemoveGreTunnel function.

```
1   func RemoveGreTunnel(endpoint *netv1alpha1.TunnelEndpoint) error {
2       //check if the interface index is set
3       if endpoint.Status.TunnelIFaceIndex == 0 {
4           log.Info("no tunnel installed. Do nothing")
5           return nil
6       } else {
7           existingIface, err := GetIfaceByIndex(endpoint.Status.
        TunnelIFaceIndex)
8
9           if err != nil {
10              if err.Error() == "Link not found" {
11                  log.Error(err, "Interface not found")
12                  return nil
13              }
14              log.Error(err, "unable to retrieve tunnel interface")
15              return err
16          }
17          //Remove the existing gre interface
18          if err = netlink.LinkDel(existingIface); err != nil {
19              log.Error(err, "unable to delete the tunnel after the
        tunnelEndpoint CR has been removed")
20              return err
21          }
22      }
23      return nil
24  }
```

5.6 Route-Operator

The **Route-Operator** is the implementation of the **route-agent** presented in section 4.2. It is deployed as `daemonset` in order to run on every node of the Kubernetes cluster. The operator coordinates the setup of all the routes that allow

each local pod/node to communicate with the pods of the peering cluster, through the elected gateway node. It will ensure state and react on **tunnelendpoints** custorm resources changes, which means that it is able to add/remove routes as soon as a new cluster peers/de-peers with the local cluster.

5.6.1 VxLAN overlay network

Each instance of the **Route-Operator** creates a `VXLAN` network interfaces. Doing so an overlay network is created inside the cluster. Every node is part of it. At start-up time the `CreateVxLANInterface()` function is called by the operator. It performs the following steps:

1. retrieve the `Node` IP address where the operator is running;
2. automatically generate an IP address for the **VXLAN** interface;
3. set the configuration for the network interface. The configuration can be passed to the operator, if not a default values will be used;
4. create the **VXLAN** network interface;
5. get all the nodes running in the cluster, generates their IP address used in the overlay network and adds them to the `Forwarding database`.

Listing 5.19: CreateVxLANInterface() function.

```
func CreateVxLANInterface(clientset *kubernetes.Clientset,
   vxlanConfig VxlanNetConfig) error {
    podIPAddr, err := getPodIP()
    if err != nil {
        return err
    }
    token := strings.Split(vxlanConfig.Network, "/")
    vxlanNet := token[0]
    mtu, err := getDefaultIfaceMTU()
    if err != nil {
        return err
    }
    temp := strings.Split(podIPAddr.String(), ".")
    temp1 := strings.Split(vxlanNet, ".")
    vxlanIPString := temp1[0] + "." + temp1[1] + "." + temp1[2] + "."
    + temp[3]
    vxlanIP := net.ParseIP(vxlanIPString)

    vxlanMTU := mtu - vxlanOverhead
    vni, err := strconv.Atoi(vxlanConfig.Vni)
    if err != nil {
```

```
20          return fmt.Errorf("unable to convert vxlan vni \"%s\" from
    string to int: %v", vxlanConfig.Vni, err)
21      }
22      port, err := strconv.Atoi(vxlanConfig.Port)
23      if err != nil {
24          return fmt.Errorf("unable to convert vxlan port \"%s\" from
    string to int: %v", vxlanConfig.Port, err)
25      }
26      attr := &VxlanDeviceAttrs{
27          Vni:      uint32(vni),
28          Name:     vxlanConfig.DeviceName,
29          VtepPort: port,
30          VtepAddr: podIPAddr,
31          Mtu:      vxlanMTU,
32      }
33      vxlanDev, err := NewVXLANDevice(attr)
34      if err != nil {
35          return fmt.Errorf("failed to create vxlan interface on node
    with ip -> %s: %v", podIPAddr.String(), err)
36      }
37      err = vxlanDev.ConfigureIPAddress(vxlanIP, net.IPv4Mask(255, 255,
     255, 0))
38      if err != nil {
39          return fmt.Errorf("failed to configure ip in vxlan interface
    on node with ip -> %s: %v", podIPAddr.String(), err)
40      }
41      remoteVETPs, err := getRemoteVTEPS(clientset)
42      if err != nil {
43          return err
44      }
45      for _, vtep := range remoteVETPs {
46          macAddr, err := net.ParseMAC("00:00:00:00:00:00")
47          if err != nil {
48              return fmt.Errorf("unable to parse mac address. %v", err)
49          }
50          fdbEntry := Neighbor{
51              MAC: macAddr,
52              IP:  net.ParseIP(vtep),
53          }
54          err = vxlanDev.AddFDB(fdbEntry)
55          if err != nil {
56              return fmt.Errorf("an error occured while adding an fdb
    entry : %v", err)
57          }
58      }
59      return nil
60  }
```

5.6.2 Routes

When a `tunnelendpoints` resource is created and configured the **Route-Operator** instances running on the nodes are triggered and configures the routing tables of each node to enable the communication between the pods and nodes of the local cluster with remote pods running on the remote cluster. The configuration is different if the node is a **Gateway** or not.

Referring to figure 5.4 and to the cluster named **Cluster1** we see that an overlay network has been configured and all the nodes belong to it. The **WorkerNode1** is the **Gateway** node and its IP address for the `tunnel interface` is 192.168.100.1 and its **VXLAN** IP address is10.1.0.2/24. The **PodCIDR** of the peering cluster is 10.244.0.0/16. For the **Gateway** node there will be a routing rule to send all network traffic with destination the remote pods to the `tunnel interface` such this one: `10.244.0.0/16 via 192.168.100.2` On the other nodes the overlay network is used to send the network traffic to the **Gateway** node. In their routing tables will be a rule like this `10.244.0.0/16 via 10.1.0.2`.

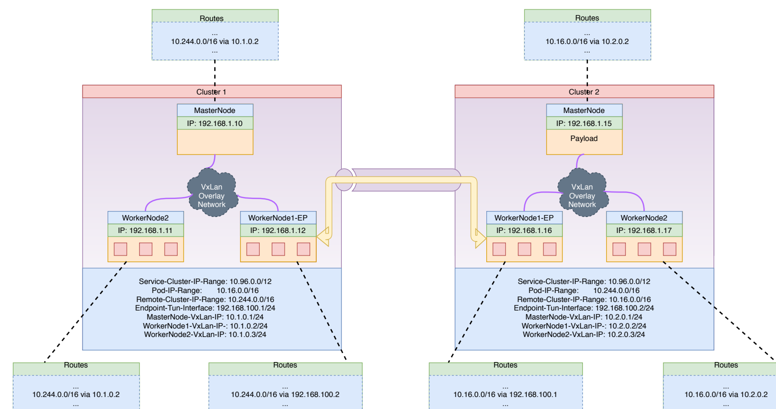

Figure 5.4: Route configuration

Dynamic convergence of the network

When one of the following parameters changes:

- **VXLAN** IP address of the **Gateway Node**;
- **PodCIDR** address space of the peering cluster;
- IP address of the remote **Gateway** node.

then the routing tables are dynamically updated by the **Route-Operator** instances running on each node of the cluster. As shown in listing 5.20 the operator checks

if the route already exists. If it is already present then it is compared with the current values in the **tunnelendpoints** instance. If they are the same then does nothing, if they have changed then the existing route is removed and the new one is inserted.

Listing 5.20: EnsureRoutesPerCluster function.

```
1  func (r *RouteController) ensureRoutesPerCluster(tep *netv1alpha1.
       TunnelEndpoint) error {
2  ...
3      if r.IsGateway {
4          existing, ok := r.RoutesPerRemoteCluster[clusterID]
5          if ok {
6              if existing.LinkIndex == tep.Status.TunnelIFaceIndex &&
       existing.Gw.String() == "" && existing.Dst.String() ==
       remotePodCIDR {
7                  return nil
8              }
9              err := r.NetLink.DelRoute(existing)
10             if err != nil {
11                 klog.Errorf("%s -> unable to remove old route '%s': %
       s", clusterID, remotePodCIDR, err)
12                 return err
13             }
14         }
15         route, err := r.NetLink.AddRoute(remotePodCIDR, "", tep.
       Status.TunnelIFaceName, false)
16 ...
17         r.RoutesPerRemoteCluster[clusterID] = route
18     } else {
19         existing, ok := r.RoutesPerRemoteCluster[clusterID]
20         if ok {
21             if existing.Gw.String() == r.GatewayVxlanIP && existing.
       Dst.String() == remotePodCIDR {
22                 return nil
23             }
24             err := r.NetLink.DelRoute(existing)
25 ...
26
27         route, err := r.NetLink.AddRoute(remotePodCIDR, r.
       GatewayVxlanIP, r.VxlanIfaceName, false)
28 ...
29         r.RoutesPerRemoteCluster[clusterID] = route
30     }
31     return nil
32 }
```

5.6.3 IPtables rules

The IPtables rules are used for two reasons, the first one is to resolve the network address spaces conflicts when the peering clusters' **PodCIDRs** overlap. The second one is to prevent the CNIs, for example such **Calico** to NAT the pod traffic when the destination address does not belong to a local network. The operator manages rules and chains only for the `nat` and `filter` tables. The chains involved are the `POSTROUTING`, `PREROUTING`, `INPUT` and `FORWARD`. We distinct two kind of IPtables rules: the generic ones, and the per cluster rules.

Generic rules and chains

At start-up time the operator executes a go-routing (listing 5.21) which creates the chains listed below:

- LIQO-POSTROUTING;
- LIQO-PREROUTING;
- LIQO-FORWARD;
- LIQO-INPUT;

Listing 5.21: Go-routine that ensures generic rules.

```
go func() {
    for {
        if err := r.CreateAndEnsureIPTablesChains(); err != nil {
            klog.Error(err)
        }
        select {
            case <-quit:
            klog.Infof("stopping go routing that ensure liqo iptables rules")
            return
            case <-time.After(liqonetOperators.ResyncPeriod):
        }
    }
}()
```

The go routine creates also four rules that references the chains:

- -A INPUT -p udp -m udp -j LIQO-INPUT;
- -A FORWARD -j LIQO-FORWARD;
- -A PREROUTING -j LIQO-PREROUTING;

- -A POSTROUTING -j LIQO-POSTROUTING

It checks periodically for the existence of the chains and that the rules referencing them are at the first position. This is need to capture all the traffic and send it to the rules inserted by the **route-operator** before the packets are processed by the local CNI.

Per cluster rules and chains

Given a peering cluster the operator creates four chains one for each of the following **IPtables's** chains: `INPUT` and `FORWARD` in table `filter` and `PREROUTING` and `POSTROUTING` in table `nat`. Those chains are then referenced by rules in the general chains discussed above. The presence of the `PREROUTING` chain depends if the **PodCIDR** of the local cluster has been remapped by the remote cluster. If so we need to `SNAT` all the traffic received from the remote cluster. The same is valid for the `POSTROUTING` chain, it is inserted only if the **PodCIDR** of the remote cluster has been remapped by the local cluster. For more clarity we present the rules that references the chains:

- -A LIQO-POSTROUTING -d remotePodCIDR -j LIQO-PSTRT-CLS-(clusterID)
- -A LIQO-PREROUTING -d localRemappedPodCIDR -j LIQO-PRRT-CLS-(clusterID)
- -A LIQO-FORWARD -d remotePodCIDR -j LIQO-FRWD-CLS-(clusterID)
- -A LIQO-INPUT -d remotePodCIDR -j LIQO-INPT-CLS-(clusterID)

The chains are populated with rules that handles properly the outgoing and incoming traffic for the specific remote cluster. The functions that manages the life-cycle of these rules are showed in listing 5.22

Listing 5.22: Functions used to ensure IPtables rules for a remote cluster.

```
func (r *RouteController) ensurePostroutingRules(tep *netv1alpha1.
    TunnelEndpoint) error {
...
    existingRules, err := r.ListRulesInChain(NatTable,
    postRoutingChain)
    if err != nil {
        klog.Errorf("%s -> unable to list rules for chain %s in table
    %s: %s", clusterID, postRoutingChain, NatTable, err)
        return err
    }
    return r.UpdateRulesPerChain(clusterID, postRoutingChain,
    NatTable, existingRules, rules)
}
```

```
10
11 func (r *RouteController) ensurePreroutingRules(tep *netv1alpha1.
    TunnelEndpoint) error {
12     //if the node is not a gateway node then return
13     if !r.IsGateway {
14         return nil
15     }
16     localRemappedPodCIDR, _ := r.GetPodCIDRS(tep)
17     if localRemappedPodCIDR == defaultPodCIDRValue {
18         return nil
19     }
20 ...
21     existingRules, err := r.ListRulesInChain(NatTable,
    preRoutingChain)
22 ...
23     rules := []string{
24         strings.Join([]string{"-d", localRemappedPodCIDR, "-i",
    tunnelIFace, "-j", "NETMAP", "--to", r.ClusterPodCIDR}, " "),
25     }
26     return r.UpdateRulesPerChain(clusterID, preRoutingChain, NatTable
    , existingRules, rules)
27 }
28
29 func (r *RouteController) ensureForwardRules(tep *netv1alpha1.
    TunnelEndpoint) error {
30 ...
31     existingRules, err := r.ListRulesInChain(FilterTable,
    forwardChain)
32 ...
33     rules := []string{
34         strings.Join([]string{"-d", remotePodCIDR, "-j", "ACCEPT"}, "
     "),
35     }
36     return r.UpdateRulesPerChain(clusterID, forwardChain, FilterTable
    , existingRules, rules)
37 }
38
39 func (r *RouteController) ensureInputRules(tep *netv1alpha1.
    TunnelEndpoint) error {
40 ...
41     existingRules, err := r.ListRulesInChain(FilterTable, inputChain)
42 ...
43     rules := []string{
44         strings.Join([]string{"-s", r.ClusterPodCIDR, "-d",
    remotePodCIDR, "-j", "ACCEPT"}, " "),
45     }
46     return r.UpdateRulesPerChain(clusterID, inputChain, FilterTable,
    existingRules, rules)
47 }
```

Chapter 6

Experimental evaluation

Now we will describe how the implementation presented in the previous chapter has been tested.

6.1 Functional Tests

In order to check that the solution works as it should, end-to-end tests have been written to check that the network connectivity is established between two peering clusters where our solution is deployed.

6.1.1 Test environment

Ansible, which is an open source software provisioning, configuration management and application deployment tool, has been used to provision the Kubernetes clusters used to test the implementation.The virtual machines used to deploy the Kubernetes ecosystem has been provisioned using **Vagrant**. It is a solution to build and maintain portable virtual machines for development environments. Each cluster consists on three nodes:

- one master node where the control plane of kubernetes is deployed;
- two worker nodes used to run the workloads.

Each virtual machine has 4 GB of **RAM**, 40 GB of **disc storage** and two **CPU** cores.

After that the virtual machines have been configured and Kubernetes has been installed the operators are deployed.

6.1.2 Tests

Here are the steps we performed to test the implementation:

1. create two Kubernetes clusters having each one `master` and two `worker` nodes;
2. deploy our operators on both the clusters;
3. create two pods on one the two clusters. One to be deployed in the local cluster, the other to be offloaded on the peering cluster;
4. create a `NodePort` service for the remote pod offloaded on the remote cluster;
5. check that all local nodes can reach the remote pod through the `NodePort` service and that the local pod can also reach the offloaded one;
6. repeat the steps 3, 4 and 5 for the remaining cluster.

Pod to (remote)pod communication

The function listed in 6.1 does the following:

1. deploys an **NGINX** instance in the local cluster and another one in the peering cluster;
2. waits for the pods to be ready;
3. checks if the local instance of **NGINX** can reach the remote one

This way we check that the pod to pod communication works fine between the two peering clusters.

Listing 6.1: Function used to test pod-to-pod communication.

```
func ConnectivityCheckPodToPodCluster1ToCluster2(con *util.Tester, t *testing.T) {
...
    podRemote := DeployRemotePod(image, podTesterRemoteCl1, ns.Name)
    _, err = con.Client1.CoreV1().Pods(ns.Name).Create(context.TODO(), podRemote, metav1.CreateOptions{})
    if err != nil {
        klog.Error(err)
        t.Fail()
    }
    podLocal := DeployLocalPod(image, podTesterLocalCl1, ns.Name)
    _, err = con.Client1.CoreV1().Pods(ns.Name).Create(context.TODO(), podLocal, metav1.CreateOptions{})
    if err != nil {
        klog.Error(err)
```

```
13          t.Fail()
14      }
15      if !util.WaitForPodToBeReady(con.Client1, waitTime, con.
    ClusterID1, podLocal.Namespace, podLocal.Name) {
16          t.Fail()
17      }
18      if !util.WaitForPodToBeReady(con.Client1, waitTime, con.
    ClusterID1, podRemote.Namespace, podRemote.Name) {
19          t.Fail()
20      }
21      if !util.WaitForPodToBeReady(con.Client2, waitTime, con.
    ClusterID2, reflectedNamespace, podRemote.Name) {
22          t.Fail()
23      }
24 ...
25      assert.True(t, isContained(remoteNodes, podRemoteUpdateCluster2.
    Spec.NodeName), "remotepod should be running on one of the local
    nodes")
26      assert.True(t, isContained(localNodes, podLocalUpdate.Spec.
    NodeName), "localpod should be running on one of the remote pods")
27      cmd := command + podRemoteUpdateCluster1.Status.PodIP
28      stdout, _, err := util.ExecCmd(con.Config1, con.Client1,
    podLocalUpdate.Name, podLocalUpdate.Namespace, cmd)
29      assert.Equal(t, "200", stdout, "status code should be 200")
30      if err != nil {
31          t.Fail()
32      }
33 }
```

Node to (remote)pod communication

We need also to check that the local nodes can communicate with the remote instance of **NGINX**. As shown by the code in listing 6.2 a service of type `NodePort` is created for the pod running on the remote cluster. Then for each node on the local cluster it checks that con reach the **NGINX** instance running on the peering cluster.

Listing 6.2: Function used to test node-to-pod communication.

```
1 func ConnectivityCheckNodeToPodCluster1ToCluster2(con *util.Tester, t
     *testing.T) {
2     nodePort, err := util.CreateNodePort(con.Client1, con.ClusterID1,
     podTesterRemoteCl1, "nodeport-cl1", namespaceNameCl1)
3     if err != nil {
4         t.Fail()
5     }
6     localNodes, err := util.GetNodes(con.Client1, con.ClusterID1,
    labelSelectorNodes)
```

```
7       if err != nil {
8           t.Fail()
9       }
10      time.Sleep(10 * time.Second)
11      for _, node := range localNodes.Items {
12          cmd := command + node.Status.Addresses[0].Address + ":" +
    strconv.Itoa(int(nodePort.Spec.Ports[0].NodePort))
13          c := exec.Command("sh", "-c", cmd)
14          output := &bytes.Buffer{}
15          errput := &bytes.Buffer{}
16          c.Stdout = output
17          c.Stderr = errput
18          klog.Infof("running command %s", cmd)
19          err := c.Run()
20          if err != nil {
21              klog.Error(err)
22              klog.Infof(errput.String())
23              t.Fail()
24          }
25          assert.Nil(t, err, "error should be nil")
26          assert.Equal(t, "200", output.String(), "status code should
    be 200")
27      }
28  }
```

6.1.3 Functional tests results

Three different **CNIs**: **Calico**, **Flannel** and **Canal** have been tested. The clusters has been configured in order to test each combination of the **CNIs** cited above. The values used for the **PodCIDRs** of the clusters has permit to test the following configurations:

- no NAT configured between the two clusters;
- SNAT when one of the clusters is remapped by the other one;
- double NAT when both the clusters need to remap the remote one.

6.2 Performance and scalability tests

The aim of the tests presented in this section is show how the solution behaves when it has to handle multiple peering session with remote clusters.

Test environment

We could not run tests on real clusters because of the difficulties in the setup of the required hardware. Even using `Kind` clusters, which is an easy way and resource efficient to create Kubernetes ecosystems, it would require a huge amount of hardware resources. Each node requires 2 **CPU** cores and 1 GB of **RAM**. Kubebuilder offers `envtest` a package that helps write tests for the controllers by setting up and starting an instance of `etcd` and the Kubernetes `API server`. The `envtest` and the operators has been setted up in a machine with `Intel® Core i7-4770 CPU @ 3.40GHz × 8` processor and 32 GB of **RAM**.

6.2.1 Tests

We want to test how the operators behaves under heavy load. We simulate the scenario where the network has to be configured for multiple peering clusters, and the number of the clusters changes. We want to measure the time that each component shown in figure 6.1 takes to perform his task:

- for the **TunnelEndpointCreator** we measure the time it spends processing the remote NetworkConfig and the time to create the TunnelEndpoint resource for a remote cluster;
- for the **Tunnel-Operator** we measure the time needed to install the **GRE** tunnel for a remote cluster;
- for the **Route-Operator** we are interested to know how much time does it take to configure the routing tables and the iptables rules.

We simulate five scenarios with number of remote clusters set to: **10**, **50**, **100**, **150** and **200**. The steps are the following:

1. set the number of remote clusters;
2. create local **NetworkConfigs** for each remote cluster;
3. create remote **NetworkConfigs** for each remote cluster;
4. each operator measures the time it takes to perform his task for each remote cluster(figure 6.1;
5. for each operator the time measures are save to a file.

The network parameters for the remote clusters has been forged in such a way that their **PodCIDR** has to be always remapped and the **Double NAT** has to be configured by the **Route-Operator**. At the same time the **Route-Operator** has been configured as it should run in a **Gateway** node, in order to simulate the scenario that stresses it the most.

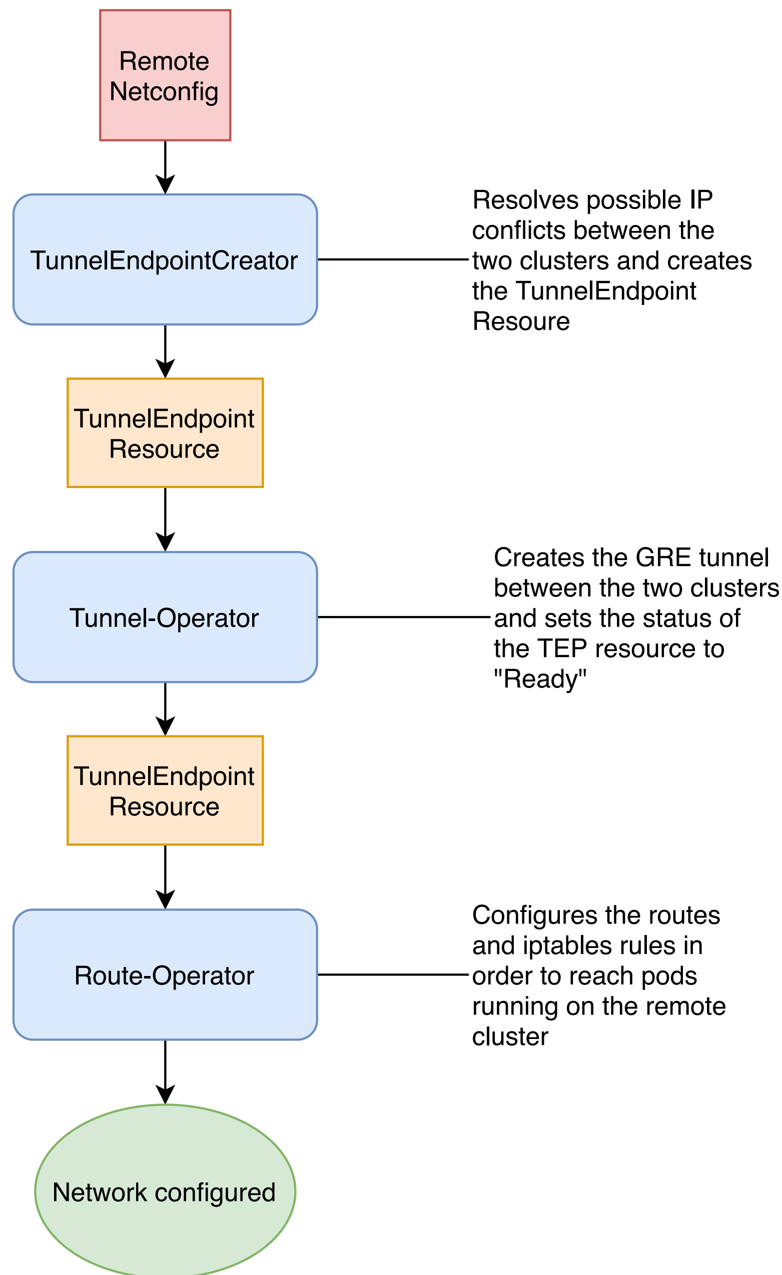

Figure 6.1: Processing chain.

6.2.2 Performance test results

The first bar plot in figure 6.2 shows the time required by each operator to process the network information coming from the remote clusters and to configure the required resources.

Checking the case with 200 clusters the **TunnelEndpointCreator** requires less than 1 second to process all the network parameters for the remote clusters. The average time to process a remote cluster is 3.7 milliseconds. The **Tunnel-Operator** requires 0.73 seconds to create and configure 200 hundred **GRE** network interfaces and to update the status of **TunnelEndponts** resources. Looking at the times for the **Route-Operator** we see that is the slowest one. That is because it has to create multiple **IPtables** chains and to populate them with **rulespecs**. Doing so, it makes multiple calls to the **IPtables** kernel module, and the context switching adds a considerable overhead. The average time required to configure the routes and **IPtables** rules is 90.86 milliseconds.

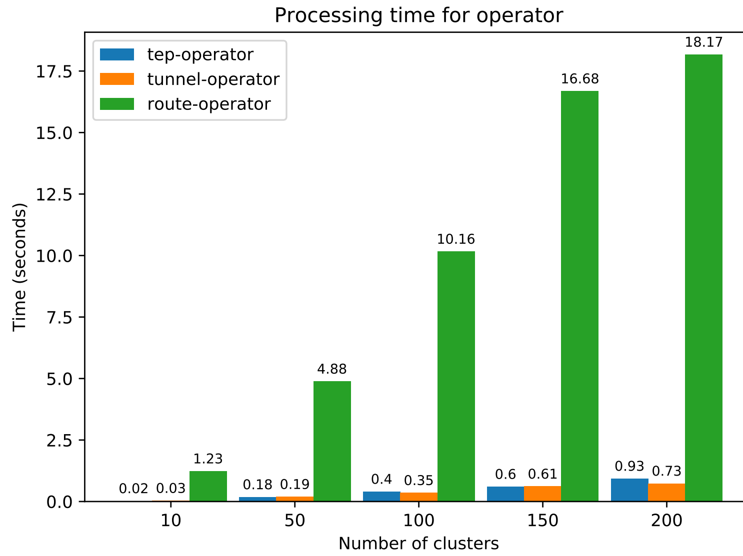

Figure 6.2: Processing time for each operator.

The figure 6.3 shows the total processing time required to configure the network connection on a cluster that has peering sessions with **10**, **50**, **100**, **150**, **200** clusters. The average time to establish a network connection with a remote cluster when receiving multiple peering requests is given by dividing the overall processing time by the number of the clusters:

$$averageTime = totalProcessingTime/numberOfClusters$$

In the case of 200 hundred cluster it takes in average 99.15 milliseconds to configure the network for each peering cluster.

The tests shows that our implementation scales in the number of peering clusters that want to establish a peering session.

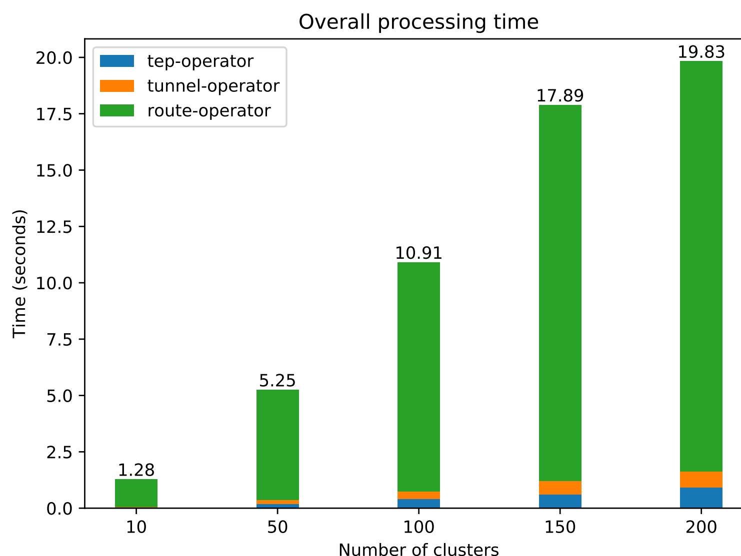

Figure 6.3: Time required to establish network connections.

6.3 Test limitations

The tests described as "scalability tests" are limited. The operators have not been containerized and deployed on a real Kubernetes cluster. The **API server** used for the tests is only used by the tested operators and cannot be compared to the Kubernetes' **API** server which has to handle the **API** requests of the **control plane** operators. The time required to exchange the network parameters by the peering clusters have been not accounted by the tests described above. The operators interacts with **API** server: the time to configure and establish a network connection between two clusters could be severely impacted by the load on the server. But we can say that these tests are a first achievement and show that the **business** logic of the operators does scale.

Chapter 7

Conclusions and future work

This work aims at proposing a network plug-in to extend the network connectivity of Kubernetes cluster to other remote clusters. In the first implementation we succeeded to:

- implement an operator which by extending the API exposed by Kubernetes using Custom Resource Definition, allows to exchange network parameters between two clusters;
- implement a solution to automatically detect and resolve network address spaces conflicts between clusters.
- implement a control plane that configures the nodes and pods of a cluster to communicate with remote resources running on a remote cluster.

In chapter 6 we evaluated the compatibility of our solution with the most used CNIs. And also obtained some results showing how the operators performs when processing multiple peering requests from remote clusters.

In the future work on the network plugin we aim to:

- implement the **southbound driver** described in section 4.2;
- improve the **Route-Operator** to make reachable only the resources deployed by a local cluster on a remote cluster and not all the PodCIDR network address space;
- support the CNI based on the **eBPF** technology such as Cilium;

Bibliography

[1] *8 facts about real-world container use*. URL: https://www.datadoghq.com/container-report/ (cit. on p. 1).

[2] Joan Engebretson. *Will Kubernetes Be the Operating System for 5G? AT&T News Suggests Yes*. Feb. 2019. URL: https://www.telecompetitor.com/will-kubernetes-be-the-operating-system-for-5g-att-news-suggests-yes/ (cit. on p. 1).

[3] *Minikube project git repository*. URL: https://github.com/kubernetes/minikube (cit. on p. 1).

[4] *Kubernetes Federation git repository*. URL: https://github.com/kubernetes-sigs/kubefed (cit. on p. 2).

[5] *Kubernetes official documentation*. URL: https://kubernetes.io/docs/home/ (cit. on pp. 3, 10, 12–15, 20).

[6] *Virtual-kubelet git repository*. URL: https://github.com/virtual-kubelet/virtual-kubelet (cit. on pp. 3, 18, 19).

[7] *Kubebuilder git repository*. URL: https://github.com/kubernetes-sigs/kubebuilder (cit. on pp. 3, 19, 20).

[8] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015 (cit. on p. 3).

[9] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. «Omega: flexible, scalable schedulers for large compute clusters». In: *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 351–364. URL: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf (cit. on p. 3).

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

[11] Steven J. Vaughan-Nichols. *The five reasons Kubernetes won the container orchestration wars.* Jan. 2019. URL: https://blogs.dxc.technology/2019/01/28/the-five-reasons-kubernetes-won-the-container-orchestration-wars/ (cit. on p. 4).

[12] Kalyan Ramanathan. *5 business reasons why every CIO should consider Kubernetes.* Oct. 2019. URL: https://www.sumologic.com/blog/why-use-kubernetes/ (cit. on p. 4).

[13] Eric Carter. *Sysdig 2019 Container Usage Report: New Kubernetes and security insights.* Oct. 2019. URL: https://sysdig.com/blog/sysdig-2019-container-usage-report/ (cit. on p. 6).

[14] *k8s Network Model.* URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-model (cit. on p. 15).

[15] *k8s CNI.* URL: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/ (cit. on p. 18).

[16] *k8s Services.* URL: https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/ (cit. on p. 18).

[17] *Kubernetes Operator pattern.* URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (cit. on p. 20).

[18] *submariner architecture.* URL: https://submariner.io/architecture/ (cit. on pp. 21–23).

[19] *cilium github page.* URL: https://github.com/cilium/cilium (cit. on pp. 23, 24).

[20] $cilimu_blog_post$. URL: https://cilium.io/blog/2019/03/12/clustermesh/ (cit. on pp. 25, 26).