Master's Degree in Computer Engineering
ACADEMIC YEAR 2020/2021
December 2021

Prototyping a cloud resource broker

Supervisor

prof. Fulvio Giovannni Ottavio RISSO

Candidate

Giuseppe ALICINO

Summary

As Kubernetes gains adoption, clusters start to be everywhere: on private data-centres, on the cloud, at the edge of the network and so on. With Liqo, a project carried out by the Computer Networks Group at Politecnico di Torino, applications and services can leverage those resources, by creating dynamic and opportunistic peerings of clusters. Starting from this list of features this thesis has the aim of doing another step. This work, carried out by the Computer Networks Group at Politecnico di Torino and developed with the collaboration of TOP-IX, tried to extend the concept of resource sharing already implemented by Liqo creating a prototype of resource brokering: a broker is a new type of cluster which has the task of aggregate all the resources shared by other normal clusters and "sell" them to every cluster which has an Incoming Peering with it in a smart way. With this new concept will be possible to have a trusted cluster to connect with which can satisfy every resources request grouping a lot of offers sent by other providers. All these features are implemented using the most important Liqo core concepts (e.g. Peering, reflection, pod offloading). This is a prototyping work so there is no explicit attention to the performances but there are some tests and measures that have been done to evaluate the impact of these new features.

Table of Contents

List of Figures

Chapter 1

Introduction

As Kubernetes gains adoption, clusters start to be everywhere: on private data-centers, on the cloud, at the edge of the network and so on. With Liqo, a project carried out by the Computer Networks Group at Politecnico di Torino, applications and services can leverage those resources, by creating dynamic and opportunistic peerings of clusters. The last version of Liqo provides:

- Automatic discovery of available clusters with Liqo installed
- Dynamic peering and resource sharing
- Support for inter-cluster connectivity with P2P parameter negotiation
- Transparent Multi-cluster pod offloading and service reconciliation
- Pod-to-pod and pod-to-service connectivity across the clusters, disregarding the installed CNI

1.1 Goal of thesis

Starting from this list of features this thesis has the aim of doing another step. This work, carried out by the Computer Networks Group at Politecnico di Torino and developed with the collaboration of TOP-IX, tried to extend the concept of resource sharing already implemented by Liqo creating a prototype of resource brokering: a broker is new type of cluster which has the task of aggregate all the resources shared by other normal clusters and "sell" them to every cluster which has an Incoming Peering with it in a smart way. The discussion is structured as follows:

- **Chapter 2**: provides a presentation of Kubernetes and Liqo, their architecture and main concepts;
- **Chapter 3**: analyzes more in deep some core features of Liqo useful for the Broker design;
- **Chapter 4**: introduces the general design of the broker prototype architecture and presents its implementation;
- **Chapter 5**: discusses some performance results and the differences with Liqo vanilla;
- **Chapter 6**: discusses all the possible improvements to design and what are the following steps to make the work production ready;

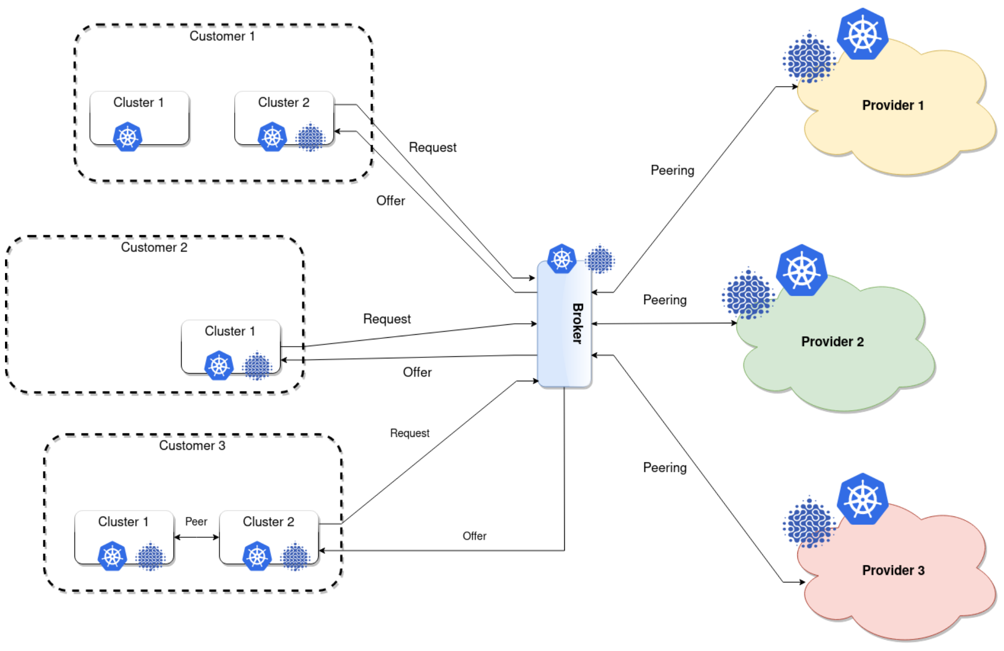

Figure 1.1: General brokering scenario

Chapter 2

Background technologies: Kubernetes and Liqo

2.1 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 [1].

The chapter continues with an introduction to other technologies and tools used to develop the solution, in particular **Virtual-Kubelet** [2], a project which allows to create virtual nodes with a particular behaviour, and **Kubebuilder** [3], a tool to build custom resources.

2.2 Kubernetes: a bit of history

Around 2004, Google created the **Borg** [4] 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" [4].

In 2013 Google announced **Omega** [5], 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) [6]. 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 [7, 8].

2.3 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.

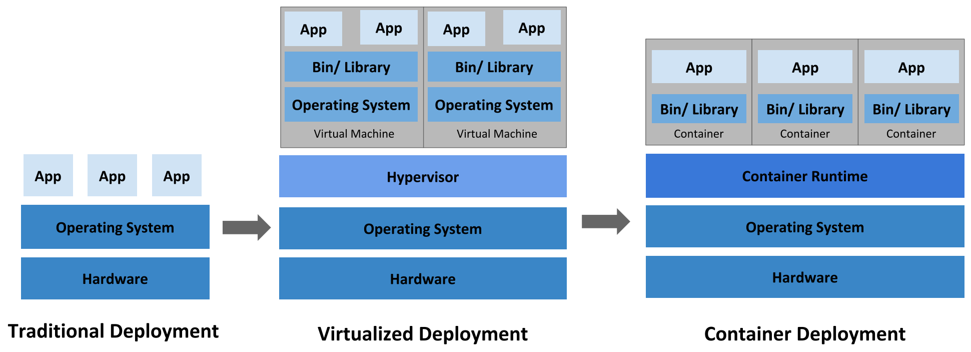

Figure 2.1: Evolution in applications deployment.

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.

2.4 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

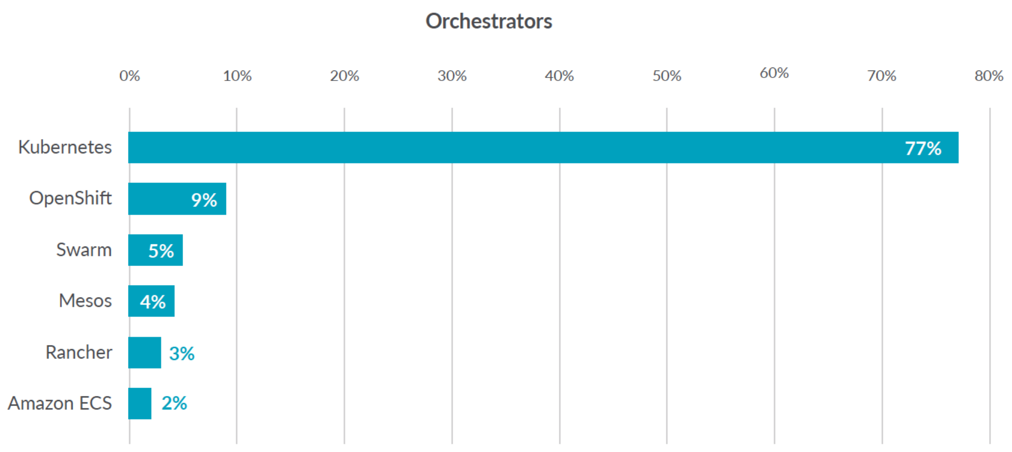

Figure 2.2: Container orchestrators use [9].

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 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.

2.5 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.

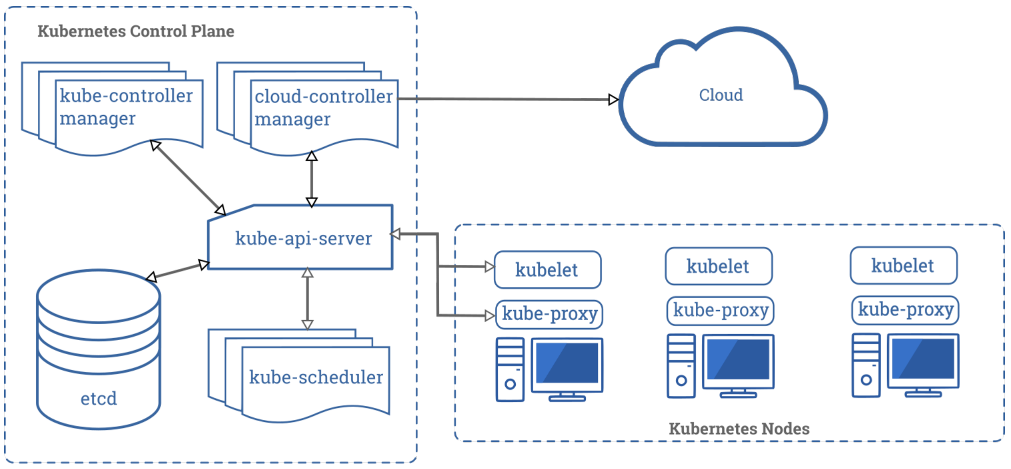

Figure 2.3: Kubernetes architecture

2.5.1 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 [10], 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.

2.5.2 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.

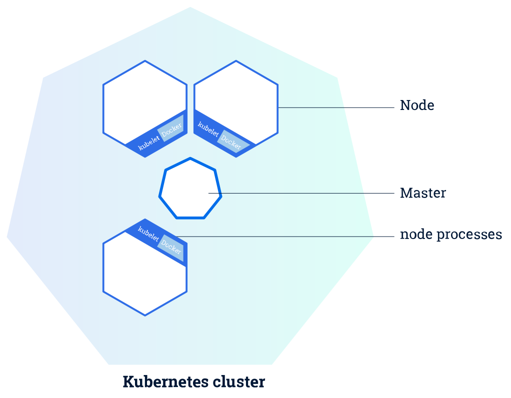

Figure 2.4: Kubernetes master and worker nodes [11].

2.6 Kubernetes objects

Kubernetes defines several types of objects, which constitutes its building blocks. Usually, a K8s resource object contains the following fields [12]:

- `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.

2.6.1 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.

2.6.2 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.

2.6.3 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.

2.6.4 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**.

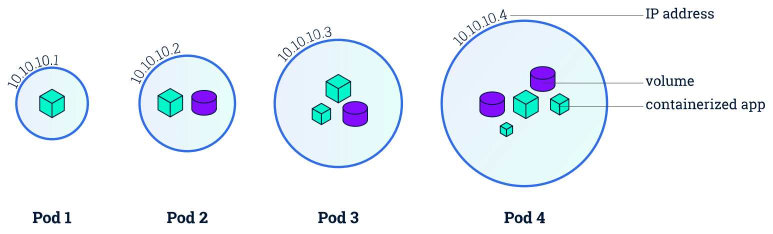

Figure 2.5: Kubernetes pods [11]

2.6.5 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 is an example of deployment.

```
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:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

`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.

2.6.6 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.

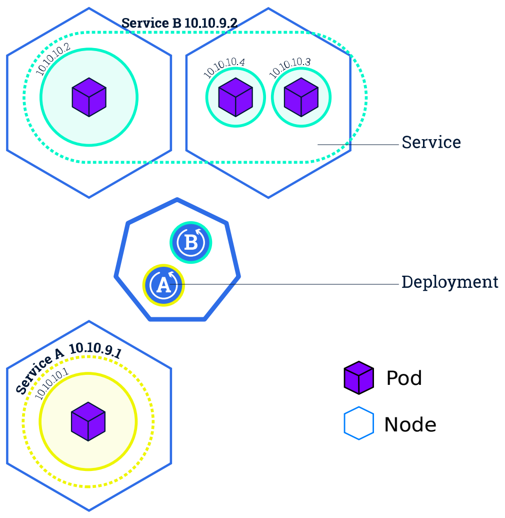

Figure 2.6: Kubernetes Services [11]

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.

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

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 [2]. 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 [2] 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".

2.8 Kubebuilder

Kubebuilder is a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs) [3].

CustomResourceDefinition is an API resource offered by Kubernetes which allows to define Custom Resources (CRs) with a name and

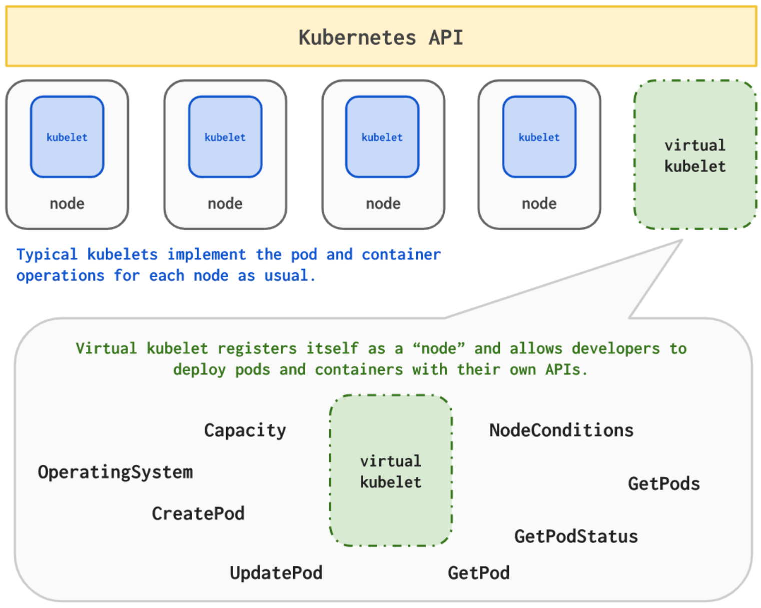
Kubernetes API
kubelet
node
kubelet
node
kubelet
node
kubelet
node
virtual kubelet
Typical kubelets implement the pod and container operations for each node as usual.
Virtual kubelet registers itself as a “node” and allows developers to deploy pods and containers with their own APIs.
Capacity
OperatingSystem
CreatePod
virtual kubelet
NodeConditions
GetPods
GetPodStatus
UpdatePod
GetPod

Figure 2.7: Virtual-Kubelet concept [2]

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 [11]. 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 [13].

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 [3]:

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.

2.9 Liqo

2.9.1 Liqo Idea

Liqo aims to create an opportunistic interconnection of multiple Kubernetes clusters allowing seamless resource and service sharing among them, creating an "endless Kubernetes ocean" where the user applications can be scheduled.

We can have a multiple cluster environment in a lot of different scenarios, both owned by the same entity or owned by different entities, These cluster may have underutilized resources because all these clusters have to have enough resources to deal with a peak of load by their own, but during the day they have moments of low load. In these moments they are wasting a part of their resources that can be available to be shared.

Liqo aims to extend the resources present in an already existent cluster using the ones currently non-occupied in neighbor clusters in an opportunistic way, so no peering and no sharing are definitive or not reversible, and it's always possible unpeer the two clusters in a simple way and return to the original state. When we extend a cluster with Liqo there is no change in the standard Kubernetes APIs, the ones described in this chapter are still valid in the new environment, and the user applications have not to be changed in order to work with Liqo.

Liqo extends the cluster by adding a new virtual node for each remote peered cluster, creating in that way a "virtual big node" where the pods can be scheduled by the default Kubernetes scheduler with no change. The Kubernetes Pods that will be scheduled on this virtual node will be took by the Virtual Kubelet and offloaded to the remote cluster.

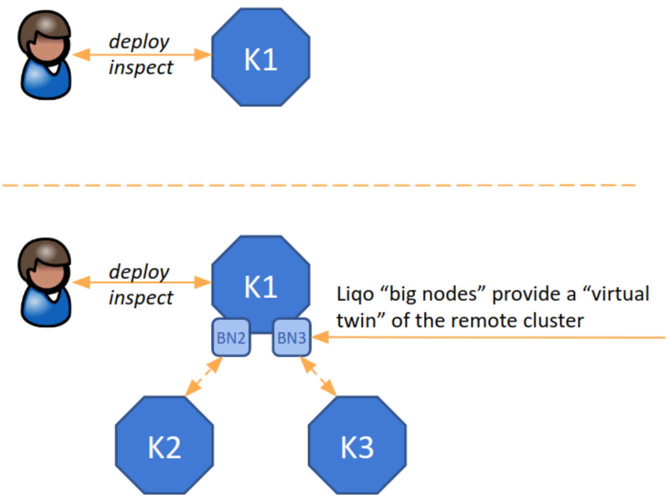
deploy
inspect
K1
deploy
inspect
K1
BN2
BN3
Liqo “big nodes” provide a “virtual twin” of the remote cluster
K2
K3

Figure 2.8: No Change in Kubernetes API

2.9.2 Cluster Management

Liqo is able to manage multiple Kubernetes clusters, allowing the user to use external resources, in a transparent way. We can describe the Liqo cluster management functionality with five pillars:

1. **Discovery**: Discover available clusters.
2. **Cluster Authentication**: Every cluster discovered authenticate using a particular token.
3. **Peering**: Establish an administrative interconnection between the clusters and negotiate the parameters.
4. **Network Interconnection**: Establish a network interconnection between the clusters.

5. **Resource Management**: Create the virtual node and make the external resources available.

6. **Usage**: Offload your pods.

2.9.3 Discovery

Liqo can dynamically discover and add new clusters to the "Big Cluster" abstraction. These clusters can be discovered in a lot of different ways, such as manually, or by an automatic configuration with DNS (on selected domains) or with mDNS (only on local area network). The discovery process will take the information from different data sources and will create a new ForeignCluster CR in the local cluster.

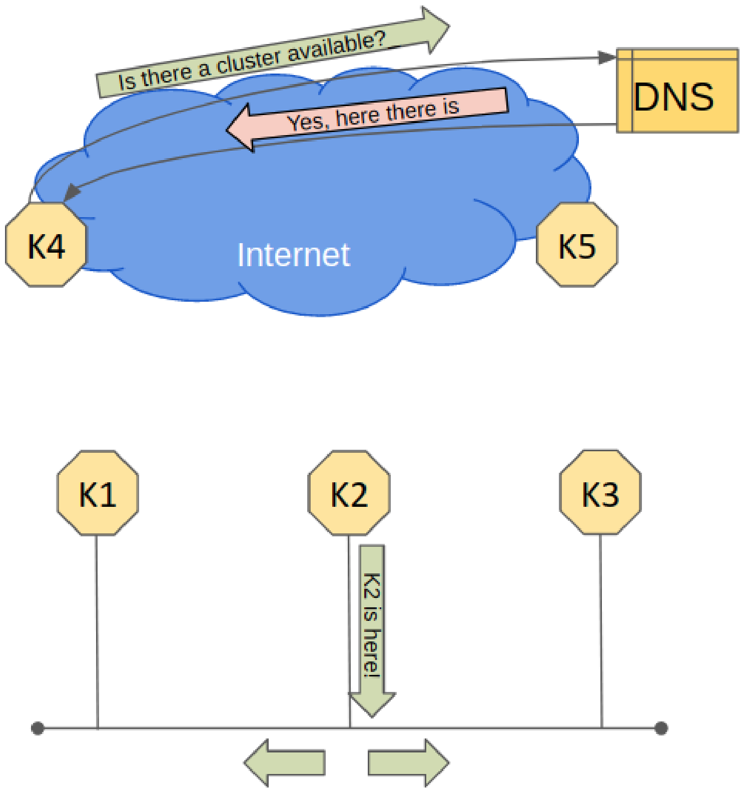

Figure 2.9: Discovery

Peering

Liqo can dynamically peer different and administratively separate clusters with a policy-driven, voluntary, and direct relationship. This connection has to be established before sharing any resources. It has a peer-to-peer architecture, so no *master* cluster is involved (this paradigm will change to adapt liqo to implement brokering features as it will be discussed in next chapters).

The Liqo peering uses the information collected during the discovery phase to contact the remote cluster and checks that both clusters that will be part of the peering are available and have accepted the interconnection.

The peering process will be deeper analyzed in Chapter 3.

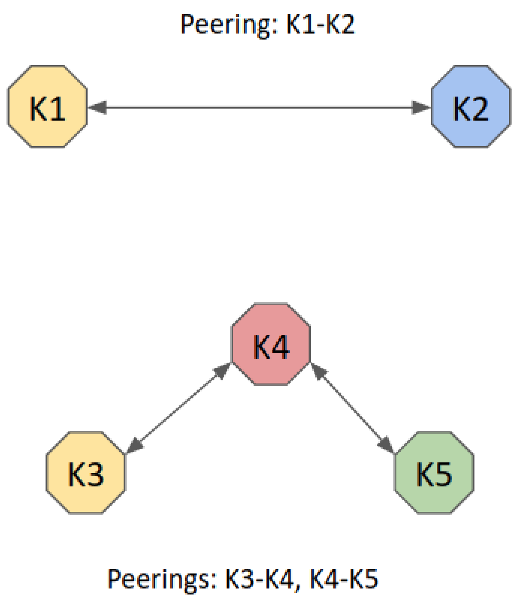

Figure 2.10: Peering

Network Interconnection

Liqo can extend the cluster network to the remote cluster, basing on the peering information. The network parameters, required to establish the VPN tunnel, are dynamically negotiated with dedicated CRD to allow Liqo to support overlapping pod CIDR in the two clusters. This let Liqo not take any assumption on the IP address space and on the networking in the peered clusters.

Liqo defines a *gateway* pod (possibly replicated) that works as a VPN terminator and allow the traffic to flow between the peered clusters. If required, it performs a double natting to allow them to communicate even if they have overlapping IP address spaces.

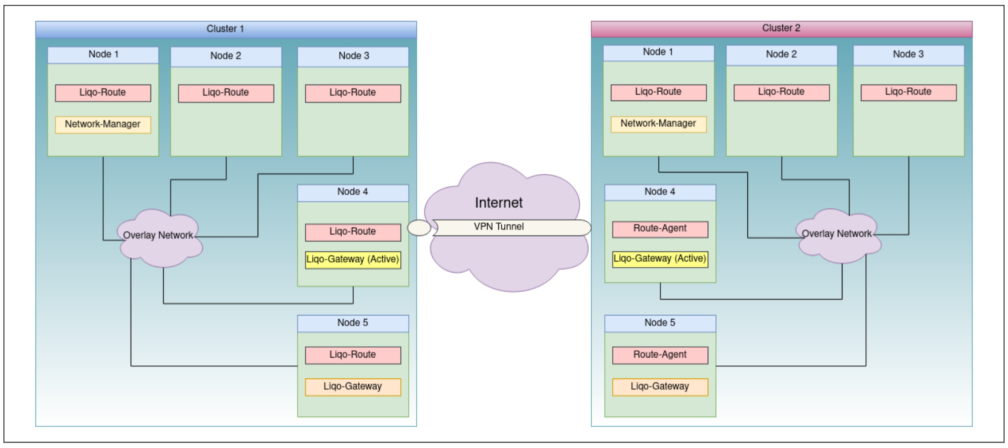

Figure 2.11: Network Interconnection

Resource Management

When a cluster accepted the peering with another one, it can offer to the second the amount of resources that it can share. If the remote cluster accepts the offer, it will create a new Virtual Node with these resources (i.e. CPU and memory). The Virtual Nodes are equivalent to physical nodes, hence can be controlled by the vanilla Kubernetes scheduler and controller-manager.

When the peering and the network interconnection is completed, the Virtual Kubelet will enable the new node setting it to ready.

Usage

When in the local cluster the new virtual node is set up and marked as ready, and the vanilla Kubernetes scheduler can schedule new pods on this node, these new pods will no see differences when accessing a service being deployed locally or remotely.

The VirtualKubelet is in charge to take the Pods scheduled on the virtual node, to offload them reflecting them in the remote cluster, and to keep the local shadow Pod Status aligned to the remote one. These Pods will be reachable from the local cluster and they can reach the services in the local cluster. Services and Endpoints are consistent on both the clusters, because of the VirtualKubelet reflection of EndpointSlices and them IP translation.

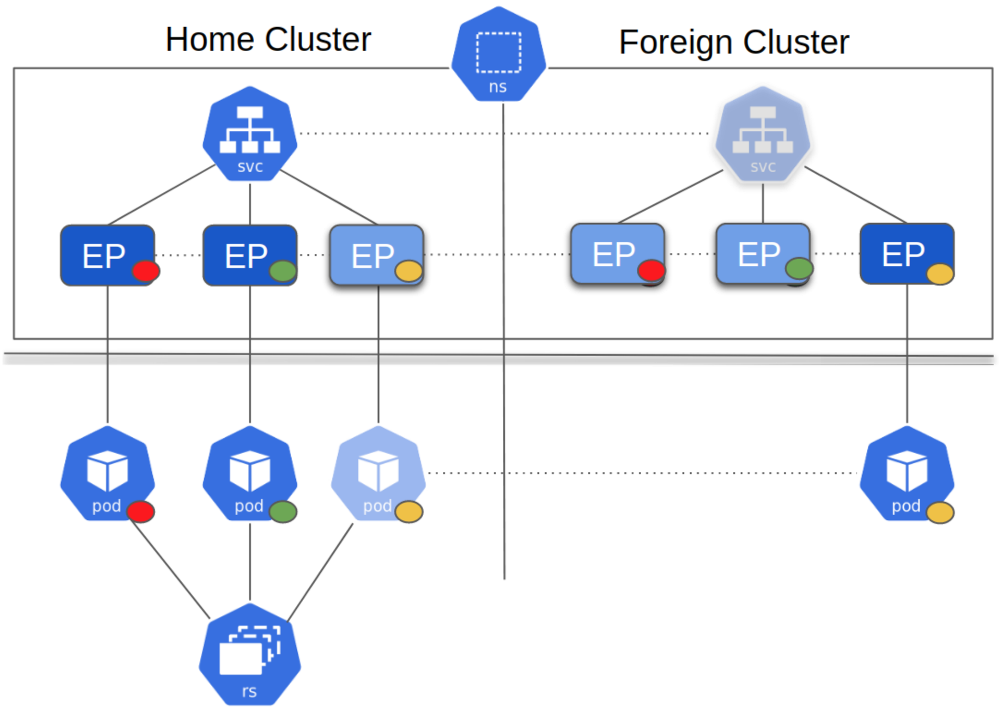

Figure 2.12: Use

Chapter 3

Towards the broker design

This chapter will describe in details some core features of Liqo which have been relevant for the broker prototype design. In the last part of this chapter will be presented how this features will be adapted for the new design. Then in the next will be presented the implementation.

3.1 Liqo peering phase

In this section will be described in details the Peering mechanism in Liqo that is fundamental mechanism to understand how the prototype will communicate with other clusters to offer resources.

3.1.1 Overview

The peering process allows to manage the control plane of the shared resources among different clusters. ResourceOffer messages embedding cluster capabilities are sent to other peers; these messages are then used to build a local virtual-node where jobs can be scheduled: if a job is assigned to a virtual-node, it will be actually sent to the respective foreign cluster. There are five main Liqo components involved in this process:

1. **ForeignCluster Controller**: this is a controller which is in charge to manage ForeignCluster CR Reconcile process. It can start the peering generating a ResourceRequest CR after Discovery and Authentication phases.

2. **ResourceRequest Controller**: this is a controller which is in charge to manage ResourceRequest CR Reconcile process and start ResourceOffer generation getting resources from the Liqo Broadcaster.

3. **Broadcaster**: this operator is fundamental for the generation of the ResourceOffers because tracks all available cluster resources in which it is running.

4. **ResourceOffer Controller**: this is a controller which is in charge to manage ResourceOffer CR Reconcile process. It has also the task to start and stop the Virtual Kubelet deployment.

5. **CRD Replicator**: this component replicates all the local CRs to remote destination cluster using Liqo Network.

Incoming Peering and Outgoing Peering

Before starting the analysis of all involved components is important to explain the difference between **Incoming Peering** and **Outgoing Peering**: the Liqo peering process is bidirectional so is fundamental to distinguish the direction and it depends form the point of view; if the whole process is considered form the point of view of the cluster that wants to start the peering it is an **Outgoing Peering** otherwise it is an **Incoming Peering**.

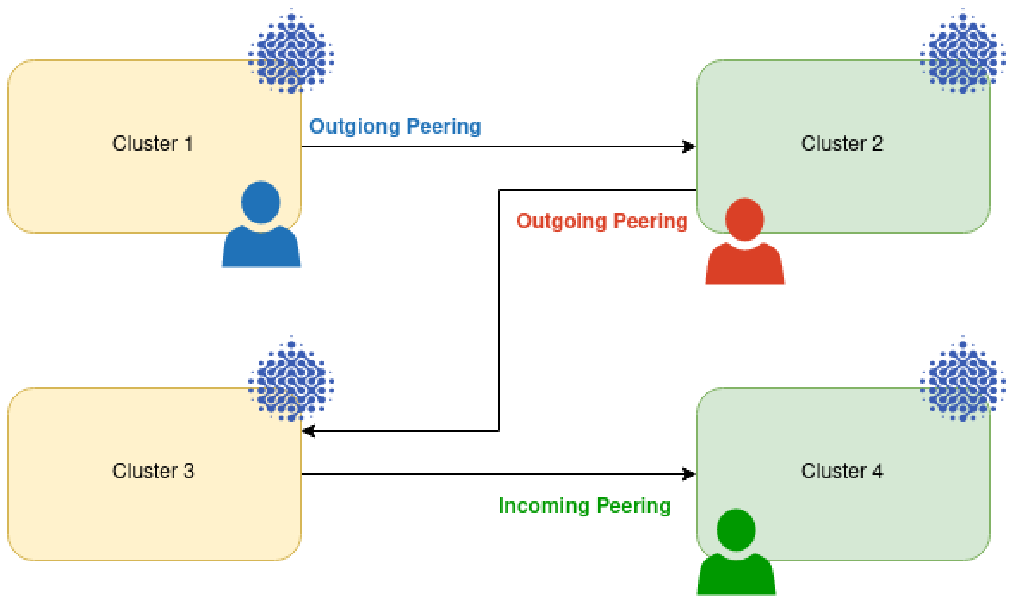

Figure 3.1: Incoming Peering and Outgiong Peering

In the following paragraph the term **peering**, if not specified, is used for both Incoming and Outgoing.

3.1.2 Liqo ForeignCluster Operator

Every peering is started from a cluster which needs more resources to deploy its applications generating a particular CR, the ResourceRequest. This resource is sent to the cluster with whom it wants to start a peering. This particular cluster is represented by another CR: the ForeignCluster. A ForeignCluster CR is generated after a new Liqo Cluster is discovered.

Listing 3.1: Example of ForeignCluster CR

```
apiVersion: discovery.liqo.io/v1alpha1
kind: ForeignCluster
metadata:
  name: my-foreign
  resourceVersion: "2031845"
  uid: 79c7e76e-7882-4774-b6f1-7535b711eb05
spec:
  clusterIdentity:
    clusterID: 619db837-69d3-40b2-963d-f74a6d762be0
    clusterName: cluster2
  foreignAuthUrl: https://<auth.service.ip>:<port>
  incomingPeeringEnabled: Auto
  insecureSkipTLSVerify: true
  outgoingPeeringEnabled: Auto
  ttl: 90
```

The code below shows a minimal example of a ForeignCluster which has name `my-foreign` and some other attributes to identify the related cluster. The ForeignCluster controller periodically watch this type of Resources to reconcile values in the spec with values in status as described in the 2.8. During this action the controller evaluate the `outgoingPeeringEnabled` attribute which can have three possible values:

- **Yes**: This is used for manually configuration. This value means that the controller is allowed to start the peering towards this foreign cluster and then generate a ResourceRequest CR which will be sent to the cluster using Liqo Network.
- **No**: This is used for manually configuration. The controller is not allowed to start the peering if it does not exist or drop it otherwise.
- **Auto**: The controller starts or not the peering automatically depending on the configurations set during Liqo installation.

If a ResourceRequest CR is sent the peering phase starts.

Listing 3.2: The structure of a ResourceRequest

```
apiVersion: discovery.liqo.io/v1alpha1
kind: ResourceRequest
metadata:
  creationTimestamp: "2021-10-14T08:26:25Z"
  finalizers:
  - crdReplicator.liqo.io
  generation: 1
  labels:
    liqo.io/remoteID: remote-cluster
    liqo.io/replication: "true"
  name: my-resource-request
  namespace: remote-namespace
  ownerReferences:
  - apiVersion: discovery.liqo.io/v1alpha1
    blockOwnerDeletion: true
    controller: true
    kind: ForeignCluster
    name: d6f44ab5-adc6-498c-9c22-4720eae70a1c
    uid: 8d2348a6-3a5a-450b-a8b6-50d71dc11712
  resourceVersion: "6471"
  uid: c9b7a936-80d5-49b8-81a1-3f304e943e17
spec:
  authUrl: https://<auth.service.ip>:<port>
  clusterIdentity:
    clusterID: my-cluster-id
    clusterName: cluster1
status:
  offerState: Created
```

Above can be seen an example of ResourceRequest named `my-resource-request` sent to a foreignCluster named `remote-cluster` by a cluster named `cluster1` and id `my-cluster-id`.

3.1.3 Liqo ResourceRequest Controller

Receiving a ResourceRequest CR triggers the ResourceRequest controller which has the task of Reconcile (2.8) the resource ensuring the correctness of all the conditions (e.g ForeignCluster existence, permissions checks...). After passing all the checks the controller is in charge to start the generation of a local ResourceOffer (see 3.1.4) that will be populated with the amount of Cluster resources that can be shared. These resources are calculated by the Liqo Broadcaster (see 3.1.4).

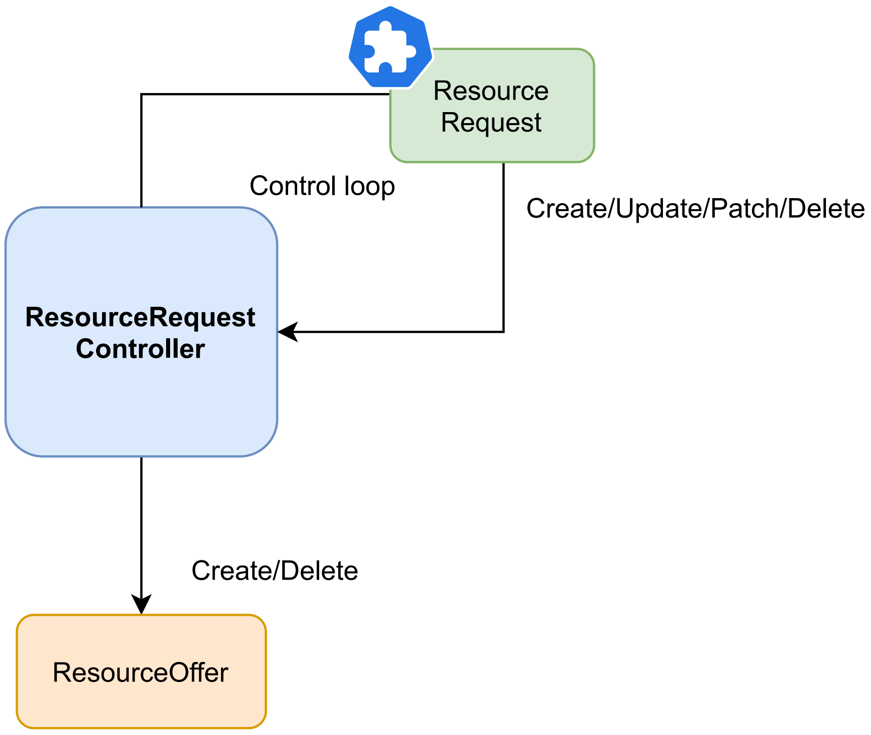

Figure 3.2: general Schema of ResourceRequest Operator

Listing 3.3: The structure of a ResourceOffer

```
apiVersion: sharing.liqo.io/v1alpha1
kind: ResourceOffer
metadata:
  creationTimestamp: "2021-10-14T08:26:25Z"
  finalizers:
 - liqo.io/virtualkubelet
```

```
  - liqo.io/node
  generation: 1
  labels:
    discovery.liqo.io/cluster-id: 48c5753c-87a1-49dd-bad2-3f3
 c3a135a6a
    liqo.io/originID: remote-cluster
    liqo.io/remoteID: cluster1
    liqo.io/replicated: "true"
    liqo.io/replication: "false"
  name: my-resource-offer
  namespace: remote-namespace
  ownerReferences:
  - apiVersion: discovery.liqo.io/v1alpha1
    blockOwnerDeletion: true
    controller: true
    kind: ForeignCluster
    name: cluster1
    uid: my-cluster-id
  resourceVersion: "6548"
  uid: 0240e1a2-94eb-4779-8588-4ca39df740ee
spec:
  clusterId: remote-cluster-id
  resourceQuota:
    hard:
      cpu: 1683m
      ephemeral-storage: 19820376Ki
      hugepages-1Gi: "0"
      hugepages-2Mi: "0"
      memory: 5843750Ki
      pods: "99"
status:
  phase: Accepted
  virtualKubeletStatus: Created
```

The Example above shows the structure of ResourceOffer sent to a ForeignCluster. The most relevant part is the `resourceQuota` attribute where are listed all the shared resources. They will become the resources assigned to the Virtual Node.

3.1.4 Liqo Broadcaster

The Liqo Broadcaster is the most important component in the peering phase. It has the task of periodically watching internal cluster resources and calculating the right amount of them to be shared with the other clusters. This mechanism is obtained by using Kubernetes runtime objects like Informers that periodically observe a particular resource to notify Write/Update/Delete actions. The Liqo Broadcaster uses two Informers to watch both Nodes and Pods resources to collect all the changes in resources. The resulting number is obtained considering the following parameters:

- **physical Nodes allocatable resources** which are summed together grouped by the tipology (e.g. cpu, memory etc.).

- **deployed pods resource requests** to be subtracted from physical nodes resources. This calculus does not consider shadow pods (they are not real pods but just reflectd images of offloaded ones) and pods offloaded by the ForeignCluster which started the peering (avoiding virtual node shrinking phenomena).

- **scaling percentage** which multiplies all the resulting value. It represents the quantity of resources the cluster owner wants to share and it is set in the cluster ConfigMap create at installation time.

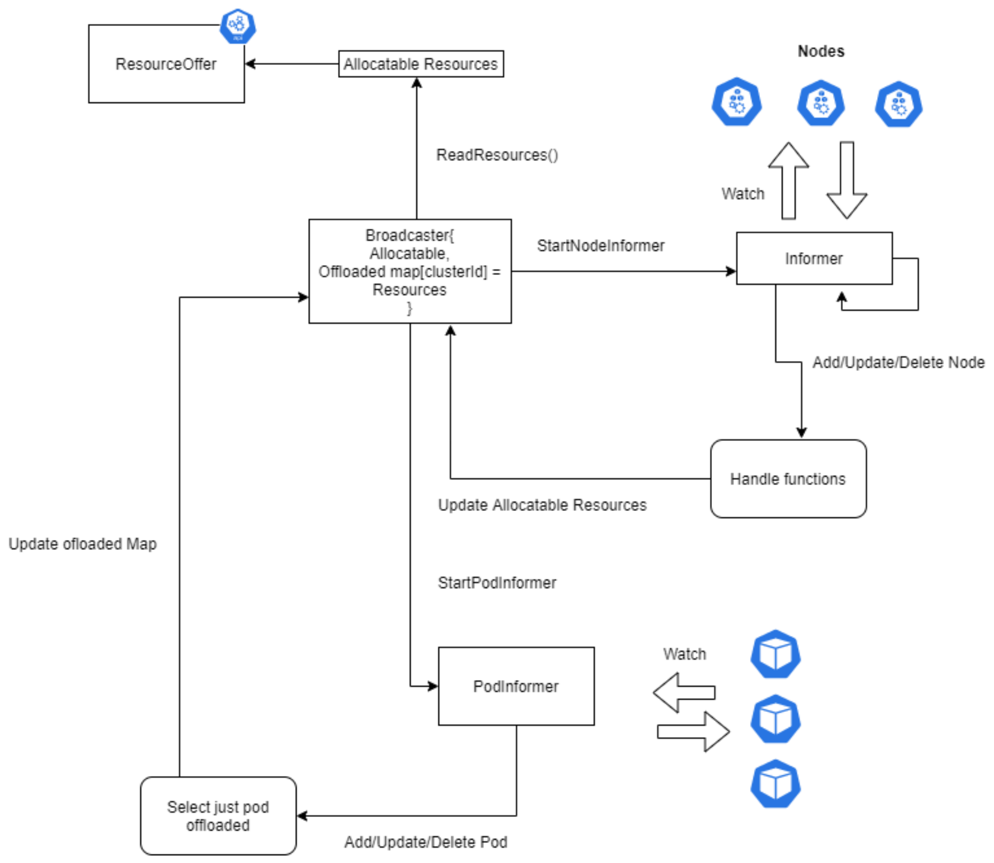

Figure 3.3: Broadcaster logic schema

This values are periodically calculated by the Broadcaster and are read by a sub-component which has the task of generating or updating all the `ResourceOffer` CRs: the **offerUpdater**.

The OfferUpdater

This is a sub-component of the Broadcaster which is in charge to generate or update all the resoureOffer resources. It works with a working queue mechanism that periodically dequeue clusterIDs and process the related ResourceOffer.

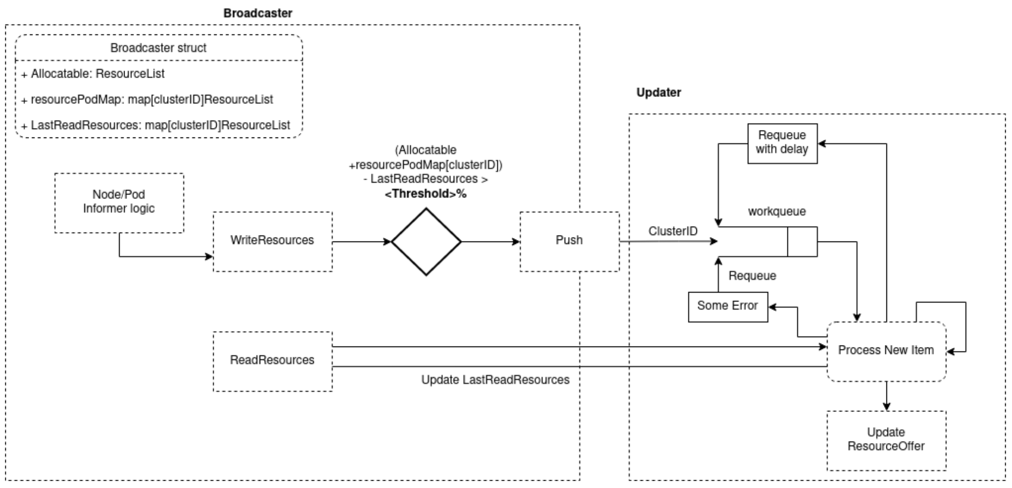

Figure 3.4: ResourceOffer Update logic schema

As represented in the above figure clusterIDs are enqueued just in case there is a resource update greater than a threshold percentage decided at installation time by the user. If some errors occur the clusterID is re-queued for further processing. There is also a re-queue with a delay after a successfully processing to avoid missing update because of little updates do not overcome the threshold.

3.1.5 Liqo ResourceOffer Controller

This controller has the task to `Reconcile` (2.8) all the `ResourceOffer` CRs received by a ForeignCluster and accept or decline it. It accepts a ResourceOffer by default and if it happens this controller creates the `Virtual Kubelet` (2.7) deployment. This controller is also responsible of destroying the Virtual Kubelet deployment if the related ResourceOffer is no more valid.

3.1.6 Liqo CRD Replicator

This is a more general component of Liqo but also very important in the peering process because it has the task of replicate all the local CRs remotely using the Liqo Network. Every process previously described at the end creates local CRs which are taken in charge by the CRD Replicator that send them to the destination cluster.

3.2 Pod Offloading and communication

In this section will be analyzed what happens at the end of a peering phase. In details will be described these concepts:

- **The role of the Liqo Virtual Kubelet**: in liqo the Virtual Kubelet has a fundamental role in the process of pods offloading;
- **Shadow pods**: another type of pod which is a reflect of his real image;
- **Namespace offloading**: how liqo creates remote namespaces to isolate more different deployments regarding other peering;
- **Pod communication and IPAM**: some hints of how the liqo network works to guarantee the pod reachability assigning right IP addresses;
- **Service Reflection and ExternalCIDR**: the mechanism which makes possible reflect kubernetes services to other clusters even running on another one.

3.2.1 The Role of the Liqo Virtual Kubelet

As said in 2.7 the Virtual kubelet masquerades a cluster as a kubelet and in Liqo has in charge of managing all the process of pod offloading. Firstly the Virtual kubelet starts the creation the Virtual Node assigning it the resources shared by the remote cluster through the ResourceOffer CR; the reconcile process of the Virtual Node is charge of the **Virtual Node Controller**.

3.2.2 Shadow pods

Shadow pods are the images of pod that have been offloaded. These are special pods because they are not directly executed on the local cluster but are useful to get all the information about the micro-service execution status.

3.2.3 Namespace Offloading

At the end of the peering phase Liqo creates a new technical namespace in order to isolate all components (e.g. pods, services) and resource (e.g. ResourceOffers) related to that remote cluster (in Liqo is used tenant concept to manage the permissions) including the Virtual Kubelet. The feature is a more general mechanism which allows to replicate remote namespaces just applying a simple resource called `NamespaceOffloading`.

Listing 3.4: Sample NamespaceOffloading structure

```
apiVersion: offloading.liqo.io/v1alpha1
kind: NamespaceOffloading
metadata:
  name: offloading
  namespace: target-namespace
spec:
  namespaceMappingStrategy: DefaultName
  podOffloadingStrategy: LocalAndRemote
  clusterSelector:
    nodeSelectorTerms:
    - matchExpressions:
      - key: liqo.io/type
        operator: In
        values:
        - virtual-node
```

This is a very powerful feature of Liqo and it will be fundamental to make the broker correct behavior.

3.2.4 Pod communication and IPAM

In the whole Liqo process, the network part is the most important one because allows communication among pods which can be part of a single application. They may need to talk to each other and this is not natively supported by Kubernetes if they run on different clusters. Thus in the following paragraphs will be shortly described some of the basic components of the Liqo networking to make more understandable some Broker implementation choices and why they have been adopted.

PodCIDR

The PodCIDR is the name of the IP addressing space used by the cluster CNI to assign IP addresses to Pods(e.g., 10.0.0.0/16). The PodCIDR is generated at cluster CNI installation time and works very well but just for a local pod scheduling policy. Nevertheless, in the Liqo context, a pod can be scheduled remotely as described in the previous paragraphs, so the addresses can have some conflicts. Liqo solves this problem using IPAM (IP Address Management).

IPAM

Embedded within the Liqo Network Manager, the IPAM (IP Address Management) is the Liqo module to avoid the previous problem but it does more. In particular IPAM is in charge of:

- **Managing networks currently in use in the home cluster**: the IPAM maintains a list of all the networks currently in use in the home cluster and when a new peering takes place, adds the network reserved to the remote cluster to the list. The network remains in the list as long as the peering is active and it is removed when the peering is terminated. In this way, after the termination of a peering, such a network becomes available for future use
- **Translating an offloaded pod's IP address**: IPAM is in charge of translating IP addresses, assigned by the remote cluster, to the corresponding IP address that is visible from the home cluster.

- **Translating Endpoints IP addresses during the Reflection**: IPAM is in charge of translating IP addresses of reflected Endpoints. Those IPs will be provided to the Virtual Kubelet, that in turn will reflect adjusted Endpoints.

Following paragraphs describe better the last two features which are the most relevant for this thesis work.

Translating an offloaded pod's IP address

Offloaded Pods are assigned an IP address by the CNI of the foreign cluster, as they actually are run there. Even if these addresses are valid on the foreign cluster, they may have no meaning in the original cluster. So the Virtual kubelet, when receiveing the pod offloaded status to update the corresponding shadow pod, consumes IPAM APIs to remap the IP assigned by the foreign cluster CNI. The IPAM module keeps track of how foreign clusters have remapped local networks and provides offloaded Pod IP addresses to the Virtual Kubelet when it has to update the Status of Shadow Pods.

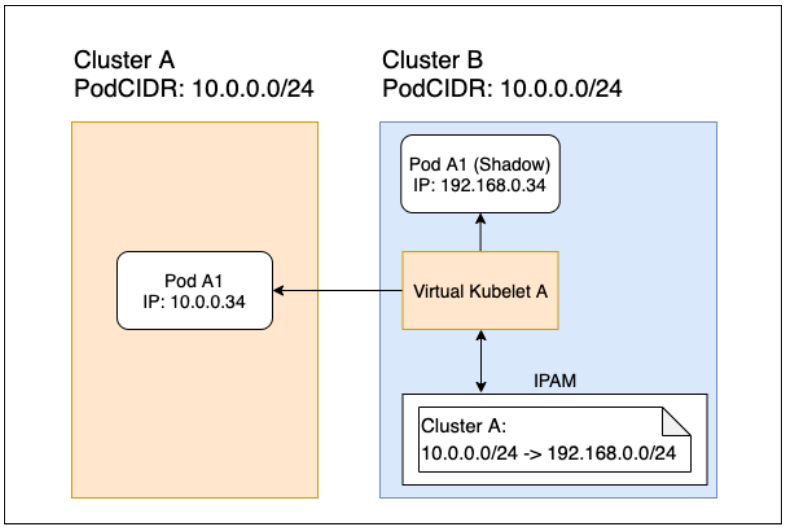

Figure 3.5: IPAM Address Remapping (from Liqo docs)

Service Reflection and ExternalCIDR

One of the most important Liqo feature is the Reflection of Services and Endpoints. The address translation of local Endpoints can be carried out thanks to the fact that the local IPAM knows if and how the local PodCIDR has been remapped in the cluster the reflection is going to take place. The problem of reflecting on cluster X an Endpoint running on cluster Y is way harder, if the home cluster is neither X nor Y. The root problems are the following:

1. cluster X and cluster Y may not have a Liqo peering, thus their Pods could not be able to talk to each other.

2. Even if such a peering existed, the home cluster would not have any information about the network interconnection between X and Y. In other words, the local cluster would not be aware of how X could have remapped the PodCIDR of Y.

The current solution to these issues is based on the usage of an additional network, called ExternalCIDR. Its name comes from the fact that addresses belonging to this network are assigned to external (i.e., non-local) Endpoints that are going to be reflected on a foreign cluster. Each Liqo cluster has its own ExternalCIDR address space which can be remapped by peered clusters, just like the PodCIDR.

Whenever the IPAM receives an Endpoint address translation request by the Virtual Kubelet and the address does not belong to the local PodCIDR, the IPAM module allocates an IP from the ExternalCIDR address space and returns it to the Virtual Kubelet. The ExternalCIDR address could be further translated if the remote cluster has remapped the local ExternalCIDR. That's not all: the IPAM stores the associations between Endpoint addresses and ExternalCIDR addresses so that future address translation requests on the same Endpoint address will use the same ExternalCIDR address. This is useful in case that Endpoint is reflected on yet another cluster, using that same ExternalCIDR address.

Suppose cluster B has a Liqo peering with clusters A and C. Therefore, cluster B can offload Pods on them. However, clusters A and C do not have an active peering and, thus, their Pods cannot connect. Furthermore, assume that B wants to reflect Service A in cluster A. The Service exposes the Shadow Pod C1, whose remote counterpart lives in C with IP address 10.1.0.5. In this particular case, if the IPAM did not translate the Endpoint's address, Pods in A would not be able to reach Service A. Pods in cluster A do not know how to connect to network 10.1.0.0/24 (and therefore to Pods in C). They can only reach Pods in B, as depicted in the IPAM block of cluster A.

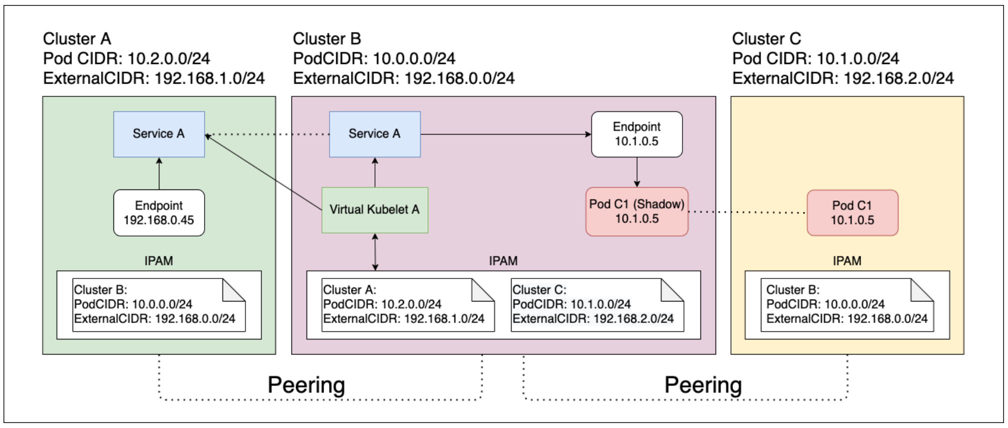

Figure 3.6: Service Reflection and ExternalCIDR (from Liqo docs)

Chapter 4

Implementation

4.1 Overview

In this chapter will be described the whole process of prototyping a Liqo Broker. It will have the following guideline:

- **The concept of broker in Liqo**: a general definition of the first broker idea as an extension of Liqo and an high level example of application context.
- **The Broker operator**: the description of the most important component of the broker and its implementation.
- **New offloading level: the Liqo network problem**: the description of a problem introduced by the broker which will represent a new offloading level not easily supported by Liqo.
- **The solution to the networking problem**: the solution to the previous problem and its implementation.

4.2 The concept of broker in Liqo

Generally speaking a broker is a person or firm who arranges transactions between a buyer and a seller for a commission when the deal is executed[wikipedia]. The Liqo broker aims to have this kind of behaviour in a multi-cluster context: a customer which owns a cluster will send a peering request to the broker which "re-sell" resources shared by a provider that have one or more clusters as represented in the figure below.

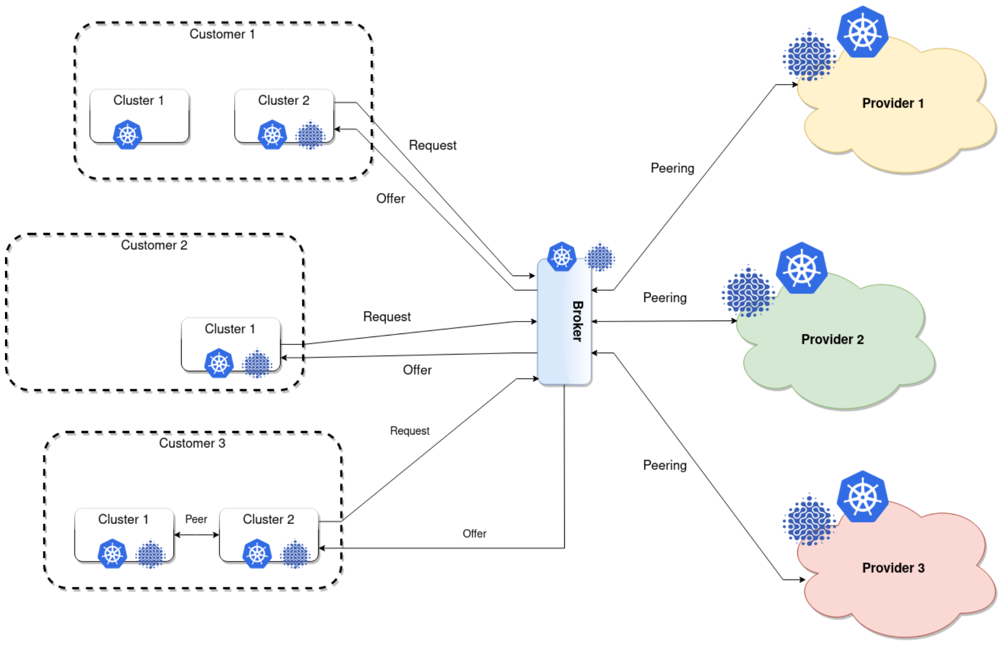

Figure 4.1: General brokering scenario

4.3 Broker Operator

The Broker operator is the core of Broker logic. It is a component started by the Liqo Controller Manager, the main Liqo operator which is in charge of starting all the control plane controllers. If a cluster has the role of broker the Liqo controller manager starts the broker operator instead of the Broadcaster as explained in the next paragraph.

4.3.1 Broker Mode

To keep the compatibility of this prototype with standard Liqo has been introduced a flag to pass at installation time which has the purpose of distinguishing the broker mode from the standard mode:

- **Broker Mode**: is activated when `broker-mode` flag has `true` value. The Liqo controller manager starts the broker operator and the cluster acts as a broker.
- **Standard Mode**: is activated when `broker-mode` flag has `false` value or is not set. The Liqo controller manager starts the broad-caster operator as usual.

Listing 4.1: controller manager broker mode managing

```go
var operator interfaces.ClusterResourceInterface
	var resourceRequestReconciler *resourceRequestOperator.ResourceRequestReconciler
	if brokerMode {
		klog.Info("Starting broker...")
		broker := &resourceRequestOperator.Broker{}
		broker.SetupBroker(*clusterID, clientset, mgr.GetScheme(), *resyncPeriod, mgr.GetClient())
		operator = broker
		resourceRequestReconciler = &resourceRequestOperator.ResourceRequestReconciler{
			Client:                 mgr.GetClient(),
			Scheme:                 mgr.GetScheme(),
			ClusterID:              *clusterID,
			Broadcaster:            broker,
			EnableIncomingPeering: *enableIncomingPeering,
			BrokerMode:             true,
		}
	} else {
		klog.Info("Starting broadcaster...")
		newBroadcaster := &resourceRequestOperator.Broadcaster{}
		updater := &resourceRequestOperator.OfferUpdater{}
		updater.Setup(*clusterID, mgr.GetScheme(), newBroadcaster, mgr.GetClient(), clusterLabels.StringMap)
		operator = newBroadcaster
		if err := newBroadcaster.SetupBroadcaster(clientset, updater, *resyncPeriod, resourceSharingPercentage.Val, offerUpdateThreshold.Val); err != nil {
			klog.Error(err)
```

```
            os.Exit(1)
        }
        resourceRequestReconciler = &resourceRequestOperator.
    ResourceRequestReconciler{
            Client:                 mgr.GetClient(),
            Scheme:                 mgr.GetScheme(),
            ClusterID:              *clusterID,
            Broadcaster:            newBroadcaster,
            EnableIncomingPeering:  *enableIncomingPeering,
            BrokerMode:             false,
        }
    }
```

The previous figure shows how is implemented the broker mode in the Liqo controller manager. However at this point is not very clear what means "starting the broker operator" so is the moment of better explain how the broker mode works and how it is implemented.

4.3.2 Broker operator structure

Listing 4.2: broker operator structure code

```
type Broker struct {
    // nodeResources holds a list of clusters ("provider")
   with the resources they offer.
    nodeResources map[string]corev1.ResourceList

    // nodeInformer reacts to changes in virtual nodes.
    // Note that we currently use an Informer on Nodes (not
   on ResourceOffers) because when ResourceOffer are created
   the
    // corresponding VirtualNode may not be ready, and thus
   we may not be able to offload workloads yet.
    nodeInformer cache.SharedIndexInformer
    // nsInformer reacts to namespaces being offloaded on the
    broker.
    nsInformer cache.SharedIndexInformer

    scheme *runtime.Scheme
    client.Client
    homeClusterID string
}
```

In figure above is reported the main structure of the Broker operator which contains the following attributes:

- **nodeResources**: is a map wich represents the bond between a virtual node and its corresponding resources. The key is the node id (clusterID) and the value all the shared resources wrapped in the ResoruceList struct. This attribute is important to keep track of all the virtual nodes informtion useful to generate a new offer towards a customer.
- **nodeInformer**: is a particuar library object to manage watchers that report all changes in the object watched. In this case this informer tracks all changes in virtual nodes to add or modify a new wntry in the noderesources map.
- **nsInformer**: is another informer used to watch the namespace resources. Is used in association with the namespaceOffloading resource to replicate a namespace from the customer to the provider every time the customer creates a new namespace to offload new jobs.
- **scheme**: library object used by the controller-runtime functions. It has not a direct application in the logic.
- **client**: the client object to make requests to the API server.
- **homeClusterID**: a string to identify the cluster in which the broker is runnung.

4.3.3 Startup process

In the next figures is explained how works the broker startup process.

Listing 4.3: setup

```
func (b *Broker) SetupBroker(clusterID string, clientset
    kubernetes.Interface, scheme *runtime.Scheme, resyncPeriod
     time.Duration, k8Client client.Client) {
```

```
    b.nodeResources = map[string]corev1.ResourceList{}
    b.Client = k8Client
    b.scheme = scheme
    b.homeClusterID = clusterID
    b.nodeInformer = informers.
  NewSharedInformerFactoryWithOptions(clientset,
  resyncPeriod,
        informers.WithTweakListOptions(virtualNodesFilter)).
  Core().V1().Nodes().Informer()
    b.nodeInformer.AddEventHandler(cache.
  ResourceEventHandlerFuncs{
        AddFunc: b.onNodeAddOrUpdate,
        UpdateFunc: func(oldObj, newObj interface{}) {
            b.onNodeAddOrUpdate(newObj)
        },
        DeleteFunc: b.onNodeDelete,
    })
    b.nsInformer = informers.NewSharedInformerFactory(
  clientset, resyncPeriod).Core().V1().Namespaces().Informer
  ()
    b.nsInformer.AddEventHandler(cache.
  ResourceEventHandlerFuncs{
        AddFunc: b.onNamespaceAdd,
        // DeleteFunc is not necessary: when the offloaded
  namespace goes away, the NamespaceOffloading will also be
  deleted
    })
}
```

The code above is used to prepare all the broker parameters before starting. The most important part is the informer setup: as previously described the informers wrap object watchers and report all the the changes to that object. This changes will be asynchronously processed through three type of handle functions:

- **AddFunc**: handle function to process the creation of a new resource.
- **UpdateFunc**: handle function to process existing resource changes.
- **DeleteFunc**: handle function to process the deletion of a resource.

The next paragraph describe all the handle functions of the broker operator.

Virtual Nodes handle functions

Listing 4.4: Virtual Node Add function

```go
func (b *Broker) onNodeAddOrUpdate(obj interface{}) {
    node := obj.(*corev1.Node)
    // Do not register the ResourceOffer until the node is ready
    if !utils.IsNodeReady(node) {
        return
    }
    clusterID := node.GetAnnotations()[consts.RemoteClusterID]
    if clusterID == "" {
        return
    }

    resources, err := b.getClusterOffer(clusterID)
    if err != nil {
        klog.Errorf("Failed to register resources for node %s: %s", node.Name, err)
        return
    }
    klog.Infof("Registering ResourceOffer for cluster %s", clusterID)
    b.nodeResources[clusterID] = resources.DeepCopy()
}
```

This is the handle function triggered at a new Virtual Node creation. Every time a new Virtual Node is created and becomes ready this function extracts all the available resources and add them to the nodeResource map.

Listing 4.5: Virtual Node Delete function

```go
func (b *Broker) onNodeDelete(obj interface{}) {
    node := obj.(*corev1.Node)
    clusterID := node.GetAnnotations()[consts.RemoteClusterID]
    if clusterID == "" {
        return
    }
    klog.Infof("Unregistering ResourceOffer for cluster %s",
   clusterID)
    delete(b.nodeResources, clusterID)
}
```

This is the handle function triggered at a new Virtual Node deletion. Every time a new Virtual Node is deleted and becomes ready this function extracts all the available resources and delete them from the nodeResource map.

Namespaces handle functions

Listing 4.6: Namespace Add function

```go
func (b *Broker) onNamespaceAdd(obj interface{}) {
    ns := obj.(*corev1.Namespace)
    clusterID := ns.Annotations[consts.RemoteNamespaceAnnotationKey]
    if clusterID == "" {
        return
    }

    _ = clusterID
    klog.Infof("Creating a NamespaceOffloading in response to new namespace %s", ns.Name)
    nsOffloading := &offv1alpha1.NamespaceOffloading{
        ObjectMeta: metav1.ObjectMeta{
            Name:      consts.DefaultNamespaceOffloadingName,
            Namespace: ns.Name,
        },
        Spec: offv1alpha1.NamespaceOffloadingSpec{
            NamespaceMappingStrategy: offv1alpha1.EnforceSameNameMappingStrategyType,
            PodOffloadingStrategy:    offv1alpha1.RemotePodOffloadingStrategyType,
            ClusterSelector: corev1.NodeSelector{
                NodeSelectorTerms: []corev1.NodeSelectorTerm{{
                    MatchExpressions: []corev1.NodeSelectorRequirement{
                        {
                            Key:      consts.TypeLabel,
                            Operator: corev1.NodeSelectorOpIn,
                            Values:   []string{consts.TypeNode},
                        },
                        {
                            Key:      consts.RemoteClusterID,
                            Operator: corev1.NodeSelectorOpNotIn,
                            Values:   []string{clusterID},
                        },
                    },
                }},
            },
        },
        Status: offv1alpha1.NamespaceOffloadingStatus{},
    }
    err := b.Client.Create(context.TODO(), nsOffloading)
    if err != nil {
        klog.Errorf("onNamespaceAdd: %s", err)
    }
}
```

This function is triggered when a new namespace is created. If this namespace is a remote one the broker create a new namespaceOffloading resource to re-offload that to the remote provider. For the namespaces there are not handle functions for update and deletion because this process is managed by the NamespaceOffloading controller.

Listing 4.7: start broker function

```
func (b *Broker) Start(ctx context.Context, group *sync.
   WaitGroup) {
    group.Add(2)
    go b.startNodeInformer(ctx, group)
    go b.startNsInformer(ctx, group)
}
func (b *Broker) startNodeInformer(ctx context.Context, group
    *sync.WaitGroup) {
    defer group.Done()
    b.nodeInformer.Run(ctx.Done())
}

func (b *Broker) startNsInformer(ctx context.Context, group *
   sync.WaitGroup) {
    defer group.Done()
    b.nsInformer.Run(ctx.Done())
}
```

The function above finally starts the broker. As the two informers have to run asynchronously they are stared in two separated threads(goroutines).

4.3.4 Peering phase

In this section will be analyzed how the broker works during the peering phase from the receipt of the ResourceRequest to the ResourceOffer generation.

Peering with providers

To make the broker work properly is necessary to collect a certain amount of offers generated by some providers it knows. So the broker administrator has to start an outgoing peering towards these providers with whom it has some agreements. In this phase the peering works exactily the same as described in the previous chapter[3]. As reported

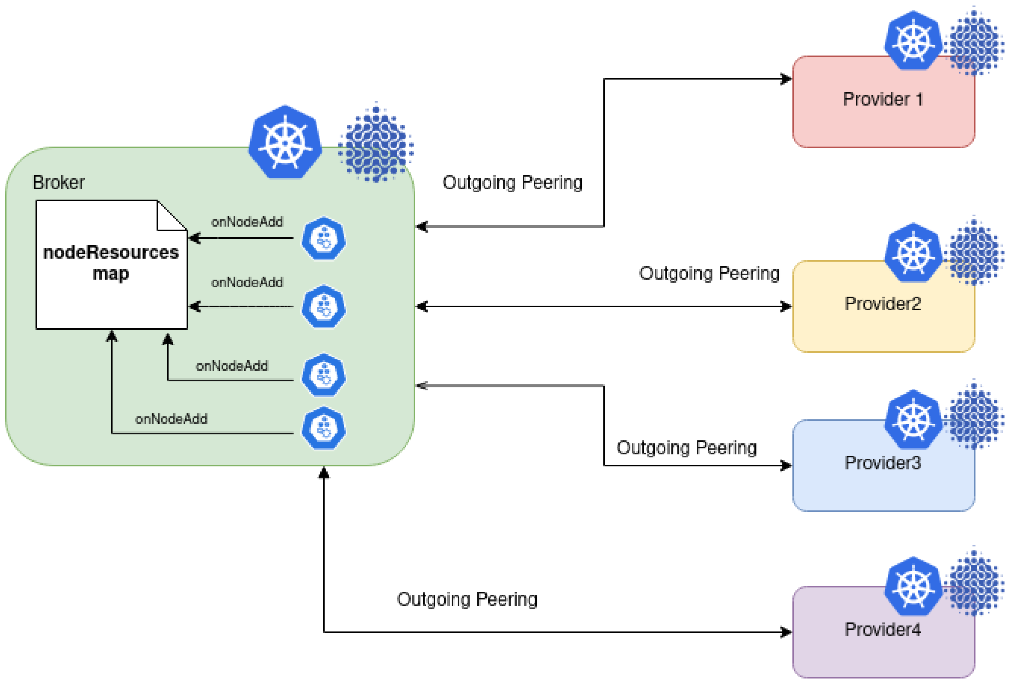

Figure 4.2: Providers Peering

in the 4.2 the handler of virtual nodes informer is triggered at the end of every peering, updating the nodeResources map.

Peering with customers

The previous phase can be described as a sort of configuration part without whom a broker cannot work properly (e.g. it cannot send offers because they are generated from existing ones). In this prototype there is a protection mechanism against this misconfiguration which consists in a simple warning message but it will be more sophisticated if necessary. However this phase is started by a customer which needs some resources so it starts a peering with the broker.

The Broker receives a ResourceRequest which triggers the ResourceRequest controller[3] which enqueue the request to be processed by the broker operator.

Listing 4.8: ResourceRequest controller enqueue the request of the customer

```go
// ensure creation, update and deletion of the related ResourceOffer
    switch resourceReqPhase {
    case allowResourceRequestPhase:
        // ensure that we are offering resources to this remote cluster
        r.Broker.EnqueueForCreationOrUpdate(remoteClusterID)
        resourceRequest.Status.OfferWithdrawalTimestamp = nil
    case denyResourceRequestPhase, deletingResourceRequestPhase:
        // ensure to invalidate any resource offered to the remote cluster
        err = r.invalidateResourceOffer(ctx, &resourceRequest)
        if err != nil {
            klog.Errorf("%s -> Error invalidating resourceOffer: %s", remoteClusterID, err)
            return ctrl.Result{}, err
        }
    }
```

The function `EnqueueForCreationOrUpdate` does this work and starts the offer generation.

Listing 4.9: the structure of enqueue function in the Broker operator

```go
func (b *Broker) EnqueueForCreationOrUpdate(clusterID string) {
    // Return aggregated resources
    totalResources, err := b.getTotalResources()
    if err != nil {
        klog.Error(err)
        return
    }

    if resourceIsEmpty(totalResources) {
        klog.Warningf("No resources found: nodeResources=%v",
    b.nodeResources)
        return
    }

    // Forward the offer to the cluster
    klog.Infof("Generating offer for cluster %s", clusterID)
    err = b.generateOffer(clusterID, totalResources)
    if err != nil {
        klog.Error(err)
        return
    }
}
```

This function get the total amount of resources available shared by all the providers and generate a new ResourceOffer with them (if there are not available resources a warning message log will be printed). At the end of this process the crd-replicator send the offer to the customer which completes the peering phase.

The offer generation

The most important task of the broker operator is the offer generation. It can have a lot of possible implementation, even very sophisticated, depending on what results are expected to have. This prototype implements a very simple approach which consists of collecting all the resources shared by the providers which are still available.

Listing 4.10: extraction of the resources and ResourceOffer generation

```
func (b *Broker) getTotalResources() (corev1.ResourceList,
   error) {
    total := make(corev1.ResourceList)
    for cluster, _ := range b.nodeResources {
        resources, err := b.getClusterOffer(cluster)
        if err != nil {
            return nil, err
        }
        mergeResources(total, resources)
    }
    return total, nil
```

```
}

func mergeResources(a, b corev1.ResourceList) {
    for key, val := range b {
        if prev, ok := a[key]; ok {
            prev.Add(val)
            a[key] = prev
        } else {
            a[key] = val.DeepCopy()
        }
    }
}

func (b *Broker) generateOffer(clusterID string, toOffer
   corev1.ResourceList) error {
    list, err := b.getResourceRequest(clusterID)
    if err != nil {
        return err
    }
    if len(list.Items) == 0 {
        return fmt.Errorf("no resource request for cluster %s
   ", clusterID)
    }
    if len(list.Items) > 1 {
        return fmt.Errorf("more than one resource request
   exists for cluster %s", clusterID)
    }
    request := list.Items[0]
    offer := &sharingv1alpha1.ResourceOffer{
        ObjectMeta: metav1.ObjectMeta{
            Namespace: request.GetNamespace(),
            Name:      offerPrefix + b.homeClusterID,
        },
    }

    op, err := controllerutil.CreateOrUpdate(context.
   Background(), b.Client, offer, func() error {
        if offer.Labels != nil {
            offer.Labels[discovery.ClusterIDLabel] = request.
   Spec.ClusterIdentity.ClusterID
            offer.Labels[consts.ReplicationRequestedLabel] =
   "true"
```

```
            offer.Labels[consts.ReplicationDestinationLabel] = request.Spec.ClusterIdentity.ClusterID
        } else {
            offer.Labels = map[string]string{
                discovery.ClusterIDLabel:             request.Spec.ClusterIdentity.ClusterID,
                consts.ReplicationRequestedLabel:     "true",
                consts.ReplicationDestinationLabel: request.Spec.ClusterIdentity.ClusterID,
            }
        }
        offer.Spec.ClusterId = b.homeClusterID
        offer.Spec.ResourceQuota.Hard = toOffer.DeepCopy()
        return controllerutil.SetControllerReference(&request, offer, b.scheme)
    })

    if err != nil {
        klog.Error(err)
        return err
    }
    klog.Infof("%s -> %s Offer: %s/%s", b.homeClusterID, op, offer.Namespace, offer.Name)
    return nil
}
```

As reported in the figure above the function `getTotalResources` collects all the available resources and passes them to the `mergeResources` function to sum all of them in a single ResourceList object which will be returned to be passed to the `generateOffer` function which creates the ResourceOffer. This is a very simple strategy but sufficient to have a working prototype. In the next chapters will be proposed some possible new strategies.
At the end of this phase the customer starts the Virtual kubelet as usual even if it will create a big node which has the resources sent by the broker.

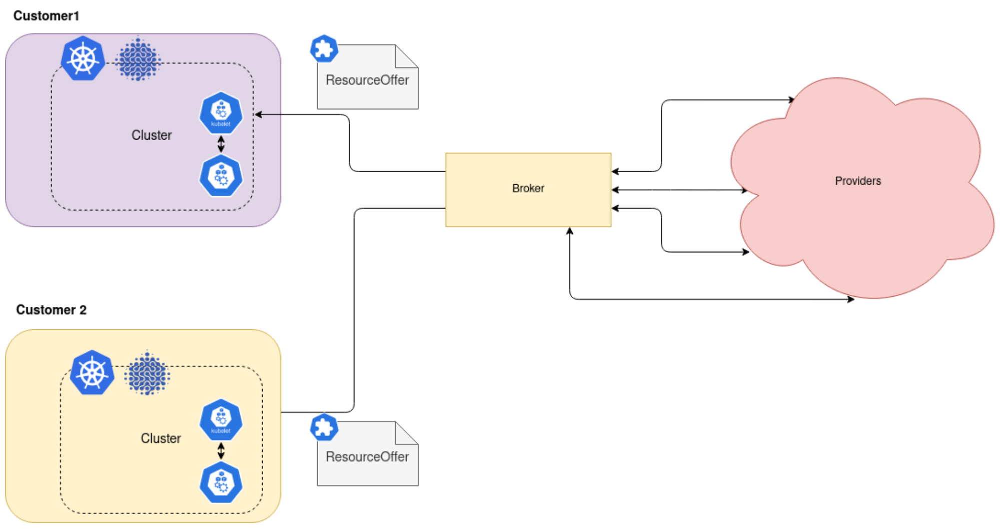

Figure 4.3: Customer peering

The following scheme shows the general flow described in this section to better understand the entire peering process.

4.4 Pod Offloading phase

At the end of peering phase the customer can start deploying its pods which may be offloaded to the broker. The broker takes the pod offloaded and schedules them on the providers Virtual Nodes where they will be executed. To make this possible a customer needs to do this three simple actions:

1. **Create a new namespace**: it will contain all the pod the customer wants to run.

2. **Enable the namespace to the offloading**: it is possible using a particular liqo label or creating a new NamespaceOffloading custom resource (see liqo docs).

3. **Deploy every application in this namespace**: the pod started may be offloaded if they will be scheduled on the virtual node.

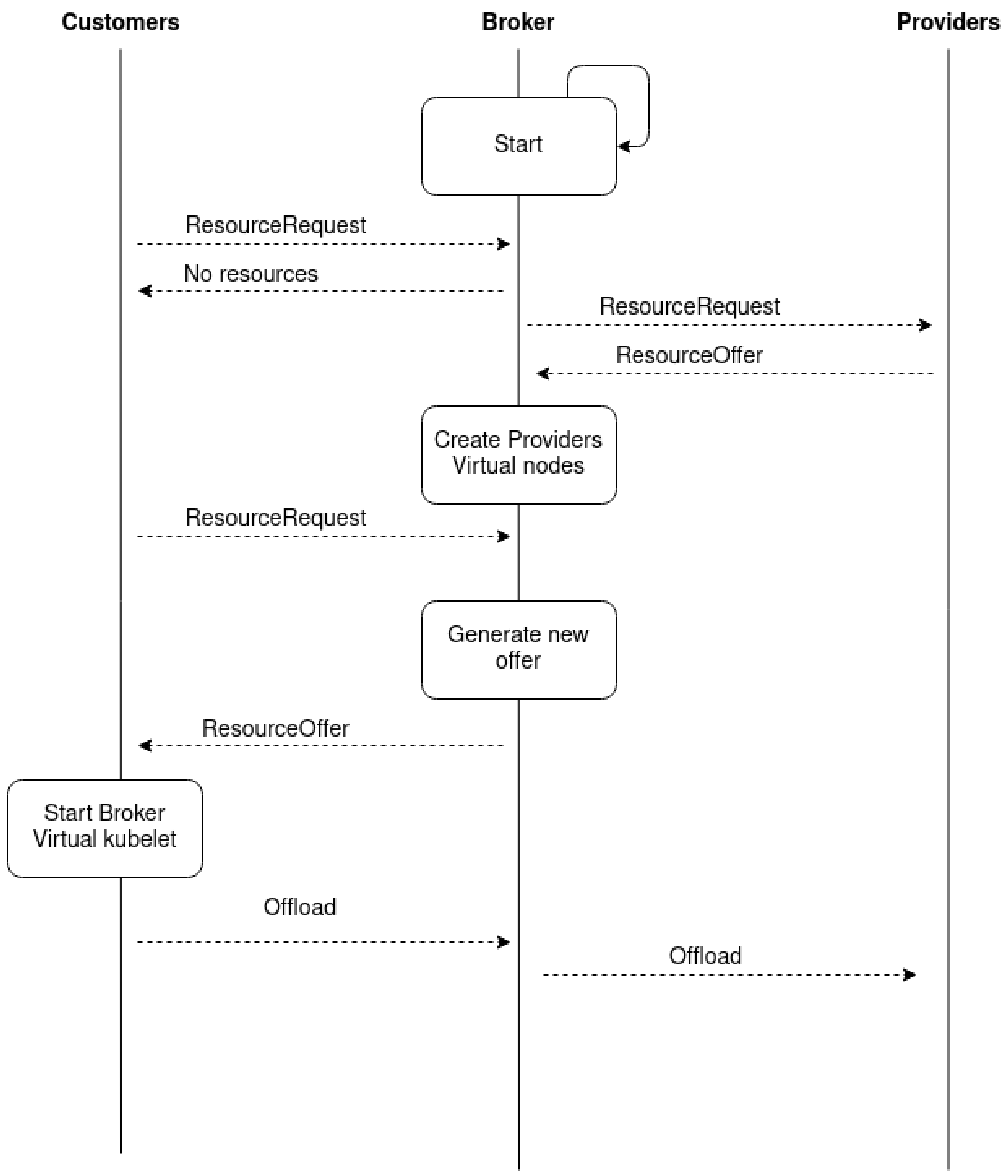

Figure 4.4: General broker flow

4.4.1 The Namespace Offloading

If the customer complete the second point in the previus paragraph correctly liqo offloads the marked namespace to the broker. At this moment the broker operator is triggered by the new namespace creation and as a result re-offload the namespace to all the providers which have an outgoing peeering marking the namespace as remote only. Every pod started on this namespace will be directly re-offloaded to the providers. This process complete what the broker operator does to make this

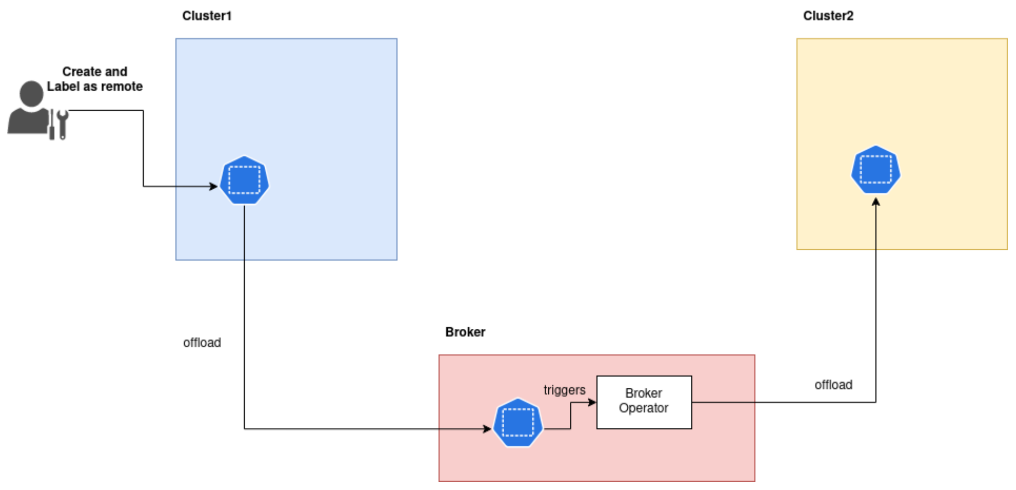

Figure 4.5: Namespace Offloading high-level scheme

cascading offloading work, so it is expected that when a pod reach any provider becomes ready and it can be reached by the liqo network. Unfortunately this is not completely true because the liqo network does not support the kind of cascading offload and the pod becomes unreachable.

4.5 The network problem

In a general configuration an offloaded pod becomes reachable thanks to the liqo network and ipam address managing [3.5].

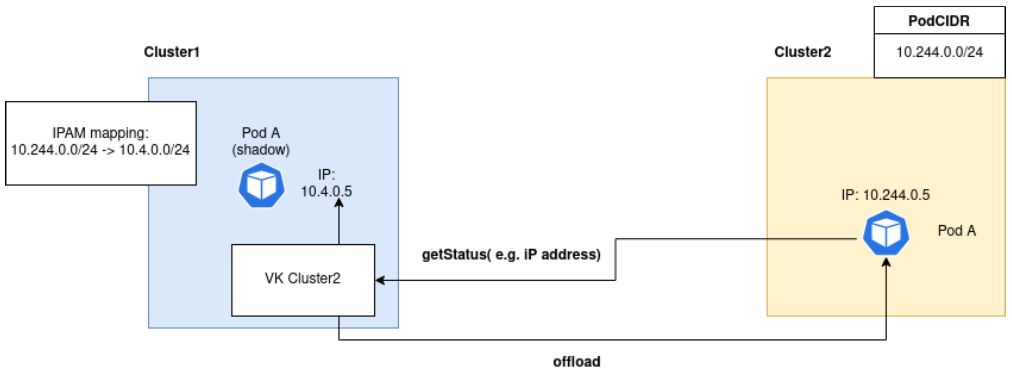

Figure 4.6: Pod address remapping in standard case

Nevertheless if we add a broker as intermediate cluster in the previous figure something go wrong and pod becomes unreachable. As reported

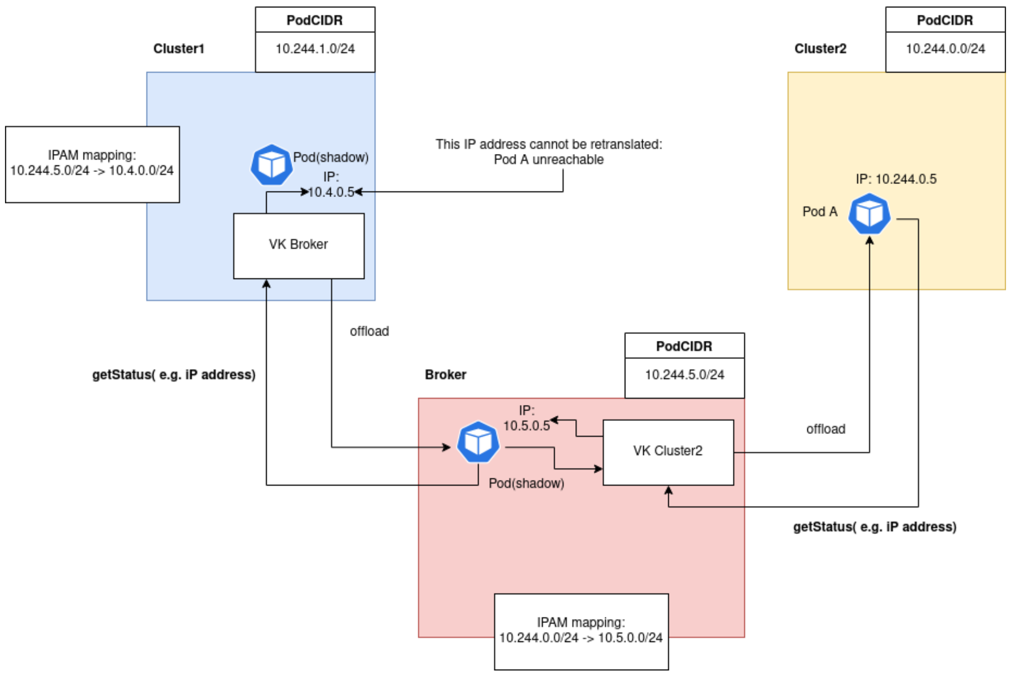

Figure 4.7: Pod unreachability scheme

in the figure there is a double address remapping done by both the broker IPAM and cluster1 IPAM which makes the final address assigned to the shadow pod not re-translatable. This is big problem because the standard liqo network is not completely compatible with the broker architecture. On the other hand is possible to use a very powerful liqo network feature to bypass this problem fixing this configuration problem: **the service reflection and the EXTERNAL CIDR**.

4.5.1 The solution

Using the serice reflection, the broker can reflect its IPAM service on the customer making possible to it asking an IP address for a shadow pod belonging to the Broker EXTERNAL CIDR net. This will be managed by the broker virtual kubelet running on the customer which is in charge to check the address of every pod of whom is getting the status:

- **The ip address belongs to POD CIDR of the remote cluster**: this is the general case; the remote cluster could not be the broker so the remapping is the same of the standard one.
- **The ip address does not belong to POD CIDR of the remote cluster**: the broker case; the Virtual kubelet asks to the Broker IPAM a new address from its EXTERNAL CIDR net.

Having this address all pods offloaded by the broker becomes reachable and the applications will work properly.

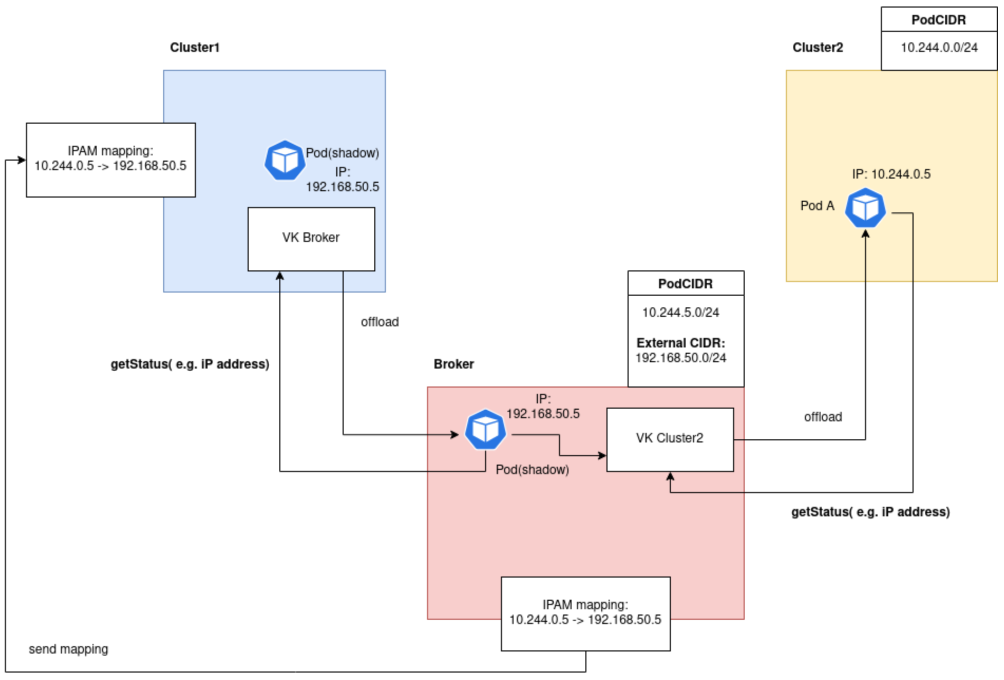

Figure 4.8: Pod unreachability solution

Chapter 5

Results

5.1 Overview

In this section will be described some performance results obtained comparing the offloading time of a standard Liqo configuration (two clusters peered in standard mode) with Liqo in brokering configuration (a cluster as customer, a cluster as broker and another as provider) in order to measure the overhead of the broker.

5.2 Performance Tests

The evaluate which impact the broker has on liqo performances it has been collected two type of data:

- **The pod offloading time in standard liqo configuration**.
- **The pod offloading time using a cluster in broker mode as intermediate cluster**

The following tables show 10 measurements of pod offloading time for each quantity of pod offloaded. The chosen quantities are 1, 5, 10, 20, 30 and 50 pods deployed at the same time. As we can see the data measured with the broker are more unstable than the standard case; this instability has two major factors to be considered:

1 pod	**5 pods**	**10 pods**	**20 pods**	**30 pods**	**50 pods**
0,5s	2,847s	6,307s	5,865s	10,15s	14,197s
2.527s	3.733s	4.199s	6.773s	7,272s	10,724s
1,436s	4,287s	6,636s	5,198s	10,54s	20,735s
1,669s	3,722s	5,802s	8,333s	9,76s	16,449s
2,713s	3,271s	5,098s	8,921s	13,276s	7,944s
1,753s	3,413s	5,942s	6,304s	10,491s	18,145s
2,269s	4,26s	5,167s	6,446s	9,321s	20,561s
2,657s	3,631s	6,977s	8,582s	12,794s	15,438s
1,828s	3,742s	4,695s	6,295s	15,348s	17,703s
2,151s	3,505s	5,082s	5,978s	14,915s	15,37s

Table 5.1: Offloading times in liqo standard configuration

1 pod	**5 pods**	**10 pods**	**20 pods**	**30 pods**	**50 pods**
2,846	3,921	9,199	22,129	40,476	27,69
1,99	4,975	17,977	9,908	14,153	14,562
0,417	3,723	9,719	14,019	22,96	26,56
2,872	4,21	8,913	18,111	17,725	22,608
2,113	7,319	7,988	14,795	15,518	20,212
16,555	4,431	9,431	11,181	11,831	20,434
6,33	16,615	8,482	12,189	16,204	23,598
2,402	18,078	6,788	13,625	14,488	22,28
16,543	4,799	8,665	28,616	15,791	22,555
2,277	4,37	18,665	16,125	25,541	26,443

Table 5.2: Offloading times in liqo with broker

1. **The scheduling**: this measures are taken from a case when the pod can be scheduled both locally and on the remote cluster and this adds a variety in time because the scheduler has no deterministic behaviour.

2. **The number of cluster crossed**: in the standard case, when

offloaded, pods are directly sent to the destination otherwise in the broker case they have to pass through the broker and then they reach the destination cluster.

If we analyze the average time of each quantity of pod to be fully ready it can be seen that the broker takes mostly twice the time taken in the standard configuration.

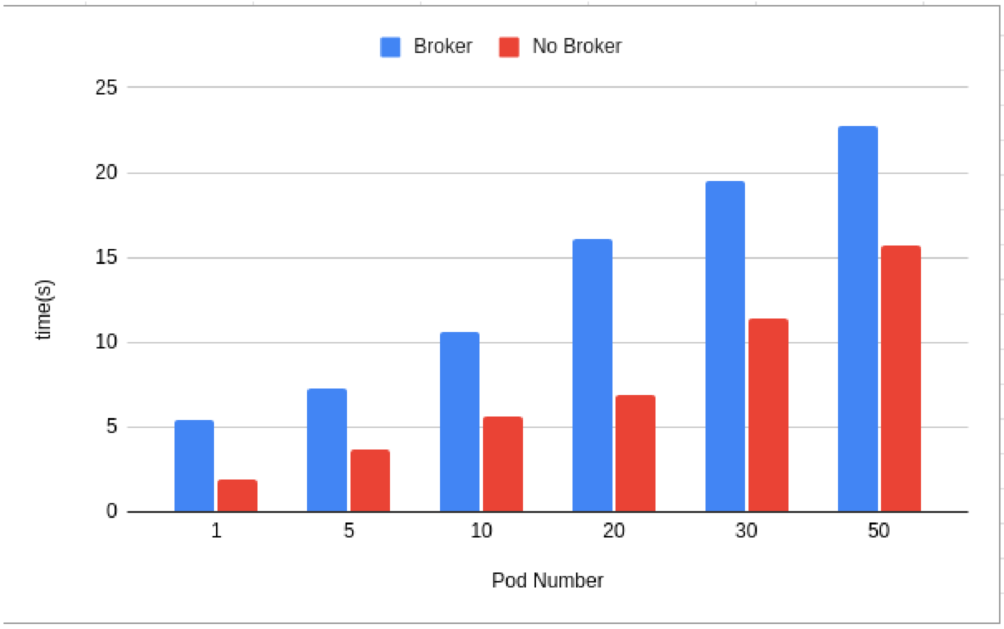

Figure 5.1: A graph with the average times comparison

5.3 Final Evaluation

In conclusion we can say that the broker adds an overhead comparable with a new level of cluster offloading. This is a prototype so it has not any optimization and some code instability which can influenced the data. However it is important also to consider the fact that the broker could become a bottleneck without the dedicated computational and network resources.

Chapter 6

Conclusions and Future work

6.1 Some General consideration

Since this thesis work is a prototyping one there are many feature that a future broker can have. This work demonstrated that is possible to implement brokering features in liqo and it is great. Nevertheless there are some points to face which can transform this prototype in a real implementation:

- **Explicit Resource Requests**: a customer can personalize its ResourceRequest resource to ask exactly what needs to the broker. It is very useful for the broker which can better select the resources and share is smarter way.

- **Offer generation algorithm**: the previous feature has this one with a direct consequence; when the broker receives a new request it is very useful to have a sophisticated algorithm to choose what resource share and then generate the offer which better satisfies the customer.

- **Over committing strategies**: this strategy consists in offer more resources than owned having more and more Broker approach.

- **hierarchical broker**: since the Broker can be a bottleneck can be very useful to have more than one broker acting together as one. This feature is very difficult to implement in liqo so needs more study.

Bibliography

[1] *Kubernetes official documentation.* URL: `https://kubernetes.io/docs/home/` (cit. on p. 3).

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

[3] *Kubebuilder git repository.* URL: `https://github.com/kubernetes-sigs/kubebuilder` (cit. on pp. 3, 18, 19).

[4] 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. 4).

[5] 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. 4).

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

[7] 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).

[8] 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).

[9] 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. 7).

[10] Diego Ongaro and John Ousterhout. «In search of an understandable consensus algorithm». In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14).* 2014, pp. 305–319 (cit. on p. 9).

[11] *Kubernetes official documentation.* URL: `https://kubernetes.io/docs/home/` (cit. on pp. 12, 15, 17, 19).

[12] *Kubernetes API official documentation.* URL: `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/` (cit. on p. 12).

[13] *Kubernetes Operator pattern.* URL: `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/` (cit. on p. 19).