POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Dynamic sharing of resources between different Kubernetes clusters

Supervisors

Prof. Fulvio RISSO

Candidate

Giulio MUSCARELLO

Academic year 2022-2023

Summary

Cloud computing has driven the digital transformation of the past two decades. In the framework of cloud computing the "cloud native" paradigm and the Kubernetes engine have become a prominent solution. The cloud native paradigm consists of a modular approach in which applications are divided into loosely coupled components ("microservices"), each running in a Linux container. Thus, the infrastructure - managed by a cloud provider and capable of scaling up and down - is decoupled from the application itself. This business model is typically described as IaaS (Infrastructure as a Service) or PaaS (Platform as a Service) depending on the level of control of the infrastructure owner over its resources.

Liquid computing deals specifically with the difficulties of a computing infrastructure that backs cloud-native workloads. Such an infrastructure is effectively "liquid" in that it can shift resources and applications from host to host, without the binding that was typical of earlier cloud scenarios. Liqo is an open source project launched at Politecnico di Torino that builds a liquid computing infrastructure on top of Kubernetes with a specific focus on resource sharing and multi-tenancy: with Liqo, Kubernetes clusters can federate in a peer-to-peer fashion to seamlessly create a larger network, with each cluster still retaining full control over its resources.

This thesis work aims to research, define and implement brokering models for the Liqo ecosystem. Service and resource brokers are an important player in a peer-to-peer topology, establishing trust and facilitating connections between providers and consumers by aggregating offers and allowing for complex topologies going beyond the current point-to-point model. Service brokers also create a standard interface that SaaS providers can subscribe to, which is an important element for application portability and preventing vendor lock-in.

Table of Contents

Acronyms

K8s
: Kubernetes

CNCF
: Cloud Native Computing Foundation

CRD
: Custom Resource Definition

CR
: Custom Resource

CIDR
: Classless Inter-Domain Routing

API
: Application Programming Interface

REST
: Representational State Transfer

RPC
: Remote Procedure Call

KIND
: Kubernetes IN Docker

Chapter 1

Introduction

Containers are now a defining feature of the cloud computing landscape. Cloud-native workloads feature tens or hundreds of containers across tens of hosts, and as workloads become more and more complex several solutions have emerged to automate their management. Today, Kubernetes is the framework of choice for container orchestration in medium and large companies, with infrastructure ranging from traditional data centres to smaller edge facilities. These setups involve a multitude of compute nodes managed by a single logical entity, and are thus classified as "single-tenant" clusters.

Liquid computing frameworks like Liqo take this a step further and envision a peering model where different clusters may share resources and services with each other. This creates dynamic data centres that can scale endlessly beyond what a single provider may offer: an entity may peer with a number of providers to extend their Kubernetes cluster as needed.

Envisioning a computing environment where some clusters are "providers" and others are "consumers", a **broker** is a component that facilitates peering with providers by offering a standardised, aggregated view of their resources; it may optionally establish trust in the ecosystem by endorsing specific entities, enabling secure and reliable resource sharing architectures.

1.1 Classification

The initial phase of our thesis work consisted of identifying the needs of stakeholders and categorizing them into three functional models. We note that the result of this work can also be found in [1], which starts from a common basis of broker models.

We note that the object of brokering may be **(hardware) resources** or **services**: the former effectively presents a PaaS offering, while the latter is a SaaS offering.

The role of a broker also varies in relation to its position on the control plane

and data plane. We identify the following three types:

1. **Catalog**: a component that merely collects metadata about providers, but is not otherwise a party to the peering process. Clients consume metadata from the catalog, then peer directly with a provider of their choice.
2. **Transparent broker**: a component that orchestrates the client's workloads on the providers, while on the data plane the client retains a direct connection to the provider clusters (i.e. the broker is transparent).
3. **Opaque broker**: a component that orchestrates the client's workloads on the providers, acting as a proxy on the data plane. The client is not aware of the existence of specific providers, being presented with an aggregated view of their resources, so the broker is said to be opaque.

1.2 Goal of the thesis

Brokers are an important addition to resource sharing infrastructures, establishing trust and discoverability in the ecosystem. These are important in the cloud environment, which is typically static and well-known, but the latter is especially important in edge environments with rapidly changing topologies and workloads. Furthermore, brokers can lower the "barrier to entry" of smaller cloud providers by aggregating their resources into a larger offering comparable with mainstream providers.

We observe that the Kubernetes open source ecosystem contains a standard for service brokering, the Open Service Broker API, as well as an implementation of the API. However, this implementation responds to the necessities of a single-tenant Kubernetes cluster, requiring an extension to work in the scenarios described here. On the other hand, no *resource* broker exists, again reflecting the reality of single-tenant environment with little interconnection to other providers.

This thesis aims to define technical models of resource and service brokers, exploring the capabilities of each model and the challenges that arise. We also review existing brokering solutions based on Kubernetes and demonstrate an implementation of an opaque resource broker on top of Liqo. Our work integrates feedback from TOP-IX, a commercial entity that seeks to offer brokering and networking services in the liquid computing landscape.
The analysis proceeds following this structure:

- **Chapter 2** provides an extensive presentation of Kubernetes concepts necessary to understand the implemented solutions.
- **Chapter 3** presents the Liqo architecture and some of its core concepts, especially with regards to reflection mechanism in one-to-one peerings.

- **Chapter 4** presents the general problem of resource brokering, identifies three functional models and analyses their design and implementation with the Liqo architecture.
- **Chapter 5** extends the analysis to the sector of SaaS brokering, giving an overview of existing solutions in well-established ecosystems.
- **Chapter 6** characterises the implementations in terms of latency and scalability.

Chapter 2

Kubernetes

This chapter provides an overview of the Kubernetes architecture showing its history and evolution through time. This summary lays the foundations for all the concepts 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 core concepts and components. Further details can be found in the official documentation [2].

The chapter continues with an introduction to other technologies and tools used to develop the solution, more precisely, the **Virtual Kubelet** [3] project, which allows creating virtual nodes with a particular behavior, and the **Kubebuilder** [4] tool, used to build custom resources.

2.1 Kubernetes: a bit of history

Around 2004, Google created the **Borg** [5] system, a small project with fewer than 5 people initially working on it. The project was developed in 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" [5].

In 2013 Google announced **Omega** [6], 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" [6].

In the middle of 2014, Google presented **Kubernetes** as an open-source version of Borg. Kubernetes was created by Joe Beda, Brendan Burns, Craig McLuckie, and other engineers at Google. Its development and design were heavily influenced by Borg, and many of its initial contributors 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) [7]. Since then, Kubernetes has significantly grown, achieving the CNCF graduated status and being adopted by nearly every big company and cloud provider: AWS [8], Azure [9] and Google Cloud [10] offer managed Kubernetes clusters. 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 manages the life cycle of applications and services using methods that provide consistency, scalability, and high availability.

What does the term "containerized applications" mean? In the last decades, the process of deploying applications has undergone significant changes, which are illustrated in figure 2.1.

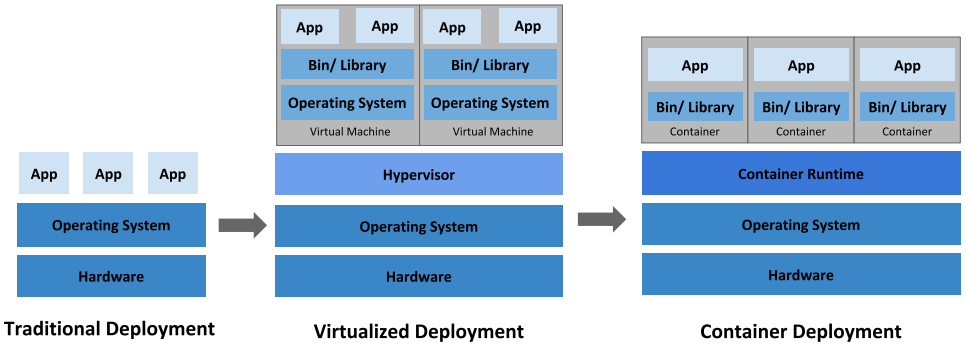

Figure 2.1: Evolution in applications deployment.

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. This 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 multiple Virtual Machines to run on a single physical server. This technique grants isolation of the applications between VMs, providing a high level of security, as the information of one application cannot be freely accessed by other applications. 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 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 operating system, on top of the virtualized hardware.

A second solution has been proposed recently: **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 filesystem, CPU, memory, process space, etc... One of the key features of containers is that they are portable. They are decoupled from the underlying infrastructure and 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 for 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. A description of this system is provided 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 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 storage, or dynamic storage supplied by public cloud providers, and more.

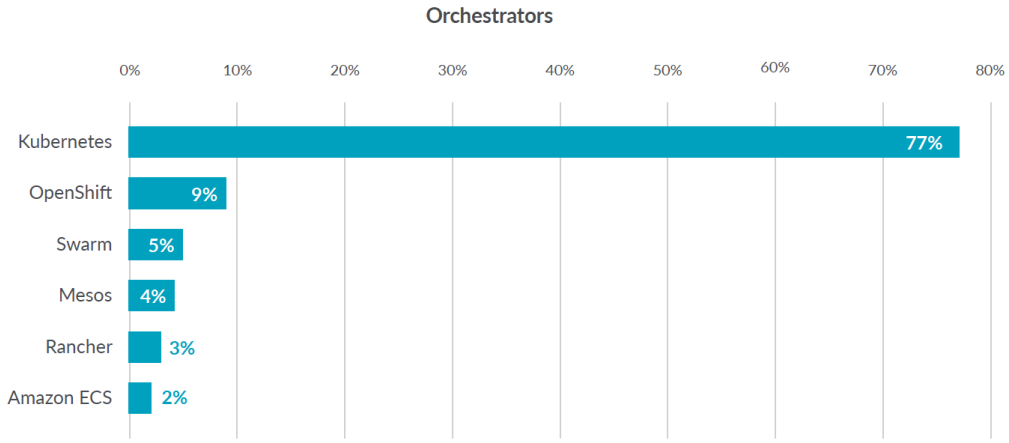

Figure 2.2: Container orchestrators use. [13]

- **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 creation of new containers, remove existing ones and adopt all their resources to the new containers.

- **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 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 application components. 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.

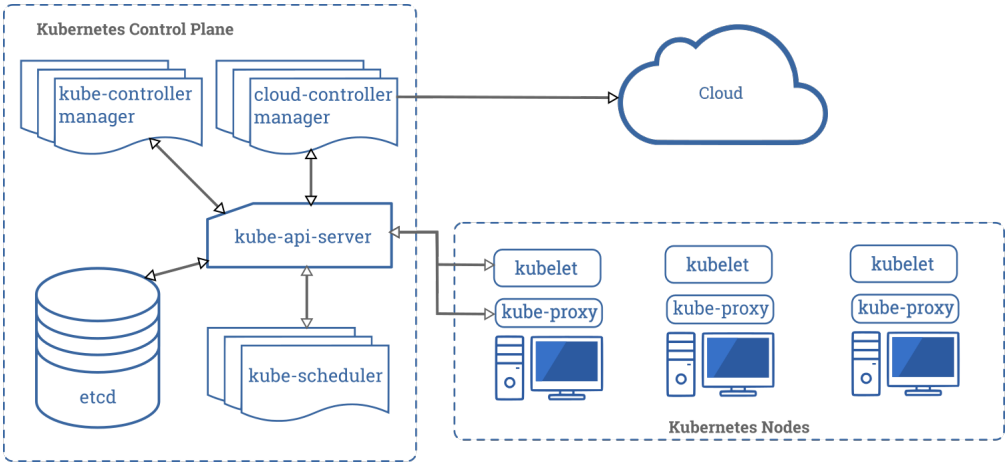

Figure 2.3: Kubernetes architecture.

2.4.1 Control plane components

The control plane's components take 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, they are typically executed on the same machine, which does not run user containers.

API server

The API server is a control plane component that exposes the Kubernetes REST API and constitutes the front-end for the Kubernetes control plane. Its function is to intercept REST requests, 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 operate with high redundancy by running several instances and balancing 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 [14], which allows different machines to work as a coherent group and

survive the breakdown of one of its members. `etcd` can be stacked in the master node or be external, installed on a dedicated host. Only the API server can communicate with it.

Scheduler

The scheduler is the control plane component responsible for 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 yet assigned to a node and selects one for them to run on. To take its decisions, it considers single and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

Kube-controller-manager

The kube-controller-manager is a 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). From a logical point of view, 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 link 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.

The 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 the 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**: responsible 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

The kubelet is an agent that runs on each node of the cluster, making sure that containers are running in the node's pods. This agent 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

The 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

The Addons are features and functionalities not yet available natively in Kubernetes but implemented by third parties pods. Some examples are DNS, the dashboard (a web UI), monitoring, and logging.

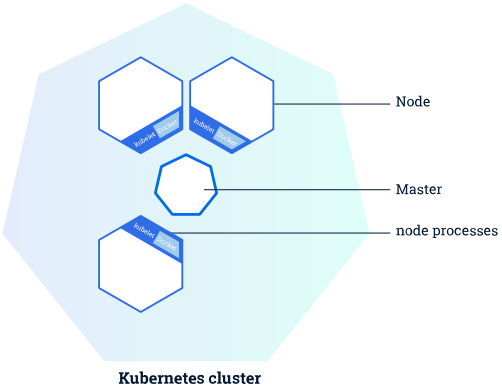

Figure 2.4: Kubernetes master and worker nodes. [2].

2.5 Kubernetes objects

Kubernetes defines several types of objects, which constitute its building blocks. A K8s resource object typically contains the following fields [15]:

- `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 standard 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.

The following list illustrates the main objects needed in the next chapters.

2.5.1 Namespace

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

- **kube-system**: it contains objects created by the 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 public cluster information;
- **kube-node-lease**: it maintains objects for heartbeat data from nodes.

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

2.5.2 Pod

Pods are the basic processing units in Kubernetes. A pod is a collection of one or more containers that share the same network and storage and are scheduled together. Pods are ephemeral and have no auto-repair capability. For these reasons, they are usually managed by a controller which handles replication, fault-tolerance, self-healing, etc.

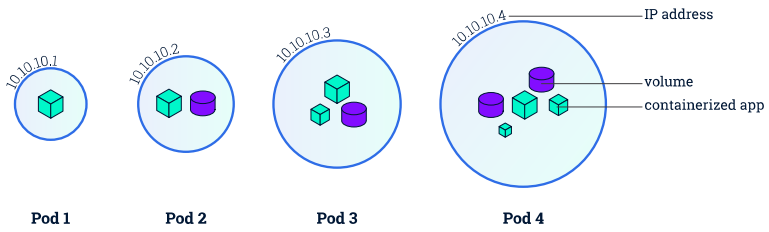

Figure 2.5: Kubernetes pods. [2]

The Kubernetes scheduler assigns pods to nodes automatically depending on a number of factors including resource requirements/availability, node characteristics and topology spread. An important feature widely used in Liqo as well as in this thesis is the possibility to add constraints, called "affinities", on where a pod can run. Kubernetes features two types of affinities:

- **Node affinities**, to select nodes by their labels;
- **Pod affinities**, to constrain pods against labels on other pods.

Additionally, an affinity may be "required" (meaning that if it can't be satisfied, the pod will not be scheduled) or "preferred" (meaning that unsatisfied affinities will not prevent scheduling).

In this thesis we present two use cases for affinities:

- Required node affinities are used in Liqo to offload pods on specific virtual nodes, effectively leveraging Kubernetes to control how pods are distributed on different clusters;
- Preferred pod affinities are used to favour scheduling pods on the same virtual node, preventing situations where a workload is deployed across geographically distant clusters causing high service latencies.

Here's an example of a pod that uses node affinity:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: pod-with-node-affinity
5 spec:
6   affinity:
```

```
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/disk-type
            operator: In
            values:
            - ssd
```

The `requiredDuringSchedulingIgnoredDuringExecution` field means that these constraints must be enforced during the pod scheduling, and they are mandatory ("required"). In this case, the pod could only be scheduled on nodes with a SSD. Only the nodes that expose exactly the `kubernetes.io/disk-type` label can be chosen by the scheduler.

2.5.3 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. ReplicaSets are usually not used directly: a higher-level concept, called **Deployment**, is provided by Kubernetes.

2.5.4 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 difference between ReplicaSets and Deployments is that Deployments allow for declarative updates to pods: when a Deployment is edited, a new ReplicaSet is created and the old one is destroyed. This listing is an example of a 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
```

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 labeled as `app:nginx`. The template field shows information about the created pods: they are labeled as `app:nginx`, and they run in one container the `nginx` DockerHub image on port 80.

2.5.5 Service

A Service is an abstract way to expose an application running on a set of Pods as a network service. The network service 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.

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

```
apiVersion: v1
kind: Service
metadata:
```

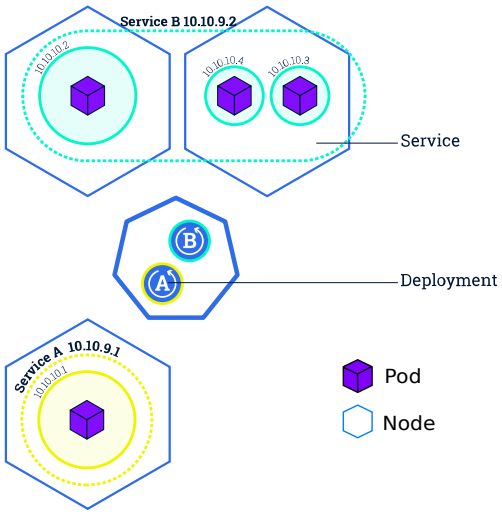

Figure 2.6: Kubernetes Services. [2]

```
4   name: my-service
5 spec:
6   selector:
7     app: myApp
8   ports:
9     - protocol: TCP
10      port: 80
11      targetPort: 9376
```

2.6 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 connecting Kubernetes to other APIs [3]. Virtual Kubelet is a Cloud Native Computing Foundation sandbox project.

The project offers a provider interface that developers need to implement to use it. The official documentation [3] 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".

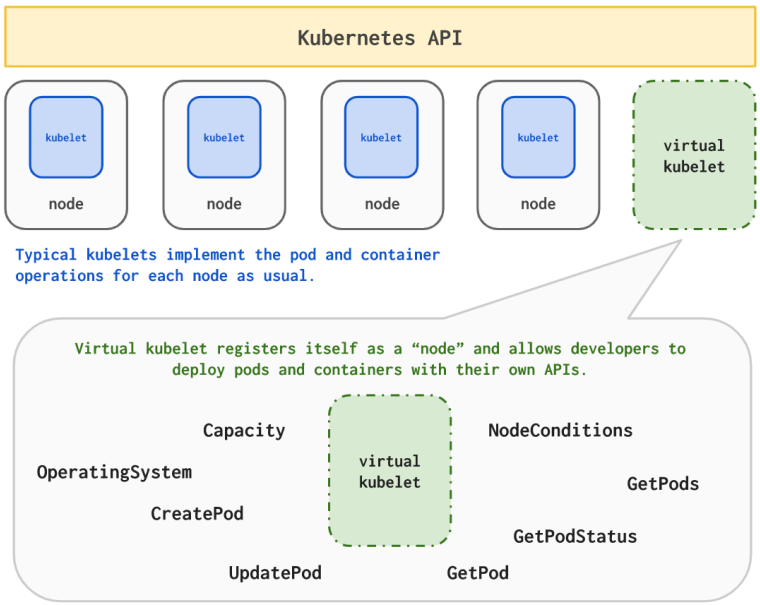

Figure 2.7: Virtual-Kubelet concept. [3]

2.7 Kubebuilder

Kubebuilder is a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs) [4].

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 generated, the Kubernetes API server instantiates a new RESTful resource path. The CRD can be either namespaced or cluster-scoped, and its name 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 [2]. To have more powerful management, you also need to provide a custom controller which executes a control loop over the custom resource: this behavior is called Operator pattern [16].

Kubebuilder helps a developer in defining his Custom Resources taking basic decisions, and writing a lot of scaffolded code. These are the main actions operated

by Kubebuilder [4]:

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 checking new fields and business logic.
6. Build and publish a container from the provided Dockerfile.

Chapter 3

Liqo

In this chapter we present Liqo, an open source project started at Politecnico di Torino that allows Kubernetes to seamlessly and securely share resources and services. We give an overview of its architecture with particular regard to some key features that enabled the development of this thesis.

3.1 Introduction

Computing load on Kubernetes clusters is typically not constant, with peaks and lows depending on the time of day, business necessities and other factors. For this reason they are provisioned with an excess of compute resources, so that they may see full utilization at peak demand. However, this also implies that they often have spare resources that they are unable to use and that could be shared with other organizations that are in demand.

Liqo interconnects clusters in a liquid computing fashion (hence the name), sharing compute resources and services among each other. It creates so-called "opportunistic data centers" where clusters can offer their resources at any time, lowering the cost of infrastructure for its peers and creating new opportunities in the field of edge computing. To do so it leverages the well-known paradigm of peering that allows for a variety of topologies, both centralized and decentralized. This also means that at a basic level individual clusters retain full control over what resources they share and with whom.

It is important to note that Liqo extends the standard Kubernetes APIs in a way that is transparent to applications and, to some extent, to Kubernetes administrators. In fact, we will see that the resources described in Chapter 2 are still valid in the new environment and are often augmented for the purposes of Liqo. As a result, user applications do not require changes to work with Liqo.

3.2 Liqo concepts

3.2.1 Discovery

Liqo communicates with clusters over IP. Clusters may be discovered in a number of ways: the user can add clusters manually by their IP address, but Liqo can also advertise its presence via mDNS on a local network, or use DNS records that specify the cluster IPs for a given domain. Manual configuration is the most flexible method, not requiring any configuration on the other cluster's part; mDNS discovery is particularly appropriate for automating the setup of a Liqo federation on a LAN; and DNS discovery is meant for use cases where an organization has multiple clusters that may be provisioned dynamically.

No matter how clusters are discovered, the end result is the creation of a custom resource called ForeignCluster in the local cluster. It represents the remote cluster and holds information about it.

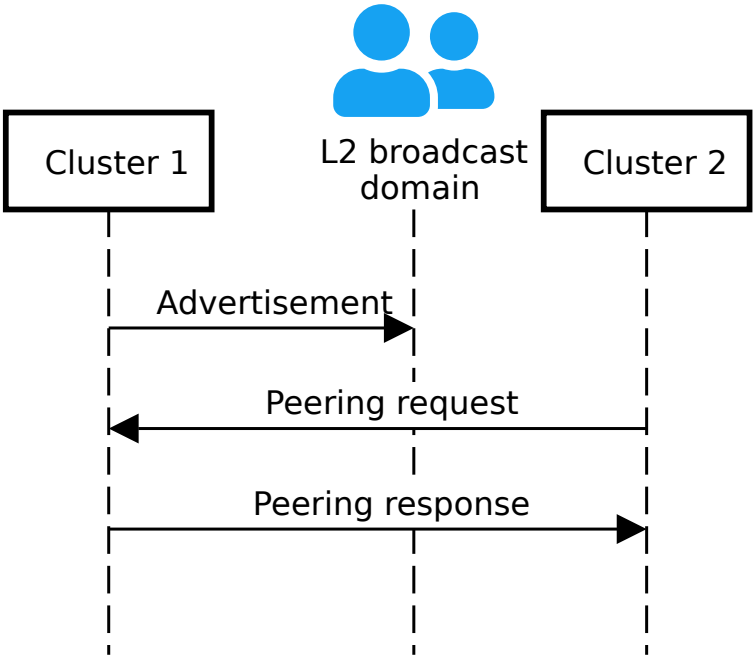

Figure 3.1: Discovery over LAN.

3.2.2 Peering

As mentioned in the introduction, Liqo makes use of the peering model to represent and regulate the relationship between different, administratively separate clusters. Peering is a process by which a connection is created between two clusters, one that

requests resources and one that offers them. It comes after the discovery process, because it uses the IP endpoint found previously. It consists of three steps:

- Authentication, by which the clusters validate each other's identity;
- Networking, by which the clusters discover each other's IP ranges and configure NAT rules;
- Resource sharing, by which the clusters communicate the amount and type of resources to exchange.

Let us review the last two steps, which will be key to understanding the implementation of brokers.

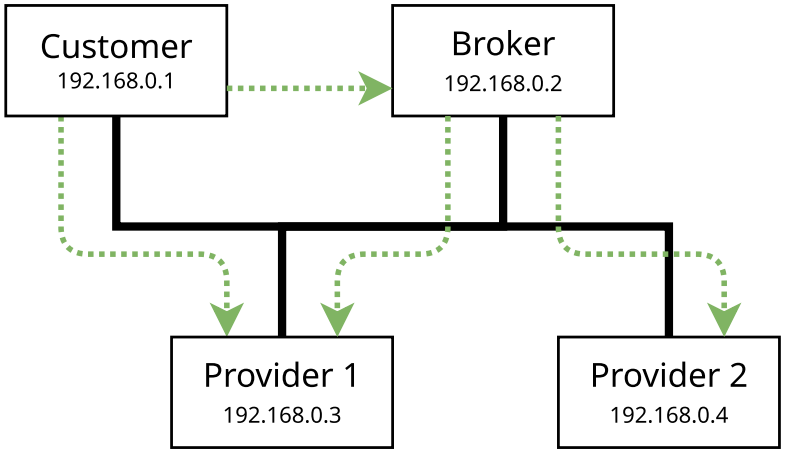

Figure 3.2: A complex peering topology over a LAN. Black: the L2 medium, green: peerings.

Networking

In the Kubernetes networking model, the cluster administrator defines a "pod CIDR" and a "service CIDR". These are private subnets (for instance, the default values on K3s are respectively 10.42.0.0/16 and 10.43.0.0/16) from which IP addresses are assigned to each pod or service. These IPs are guaranteed to be unique inside the cluster, and reachable from every node that belongs to the cluster. The implementation is left to the network plugin, for which there are tens of alternatives based on many different technologies - the most common ones are Flannel, based on a VXLAN overlay, Calico, based on BGP, and Cilium, also based on VXLAN.

This model was built to work on a single cluster, and does not translate directly to a setup with several clusters as there is no guarantee that one's pod CIDR

does not overlap with its peers'. Liqo tackles this problem using Network Address Translation: as part of the peering process, the IPAM (IP Address Management) module reserves a new subnet that maps to the peer's pod CIDR by means of an iptables rule. Packets addressed to remote clusters are then tunneled via a Wireguard VPN.

We see here an example of a NAT configuration that allocates the subnet 10.45.0.0/16 for the remote cluster's pod CIDR 10.42.0.0/16:

```
apiVersion: net.liqo.io/v1alpha1
kind: NetworkConfig
metadata:
  labels:
    liqo.io/remoteID: d14e610e-4c1b-402c-a5f1-5ef6f39c0490
    liqo.io/replication: "true"
  name: polito-labs-9b173a
  namespace: liqo-tenant-polito-labs-9b173a
spec:
  backend_config:
    port: "30020"
    publicKey: +CiHH3SJp2CQIj/Hu8jlyDWJOn7P40MQZfHfadODu0g=
  backendType: wireguard
  cluster:
    clusterID: d14e610e-4c1b-402c-a5f1-5ef6f39c0490
    clusterName: polito-labs
  endpointIP: 194.116.77.110
  podCIDR: 10.42.0.0/16
status:
  podCIDRNAT: 10.45.0.0/16
  processed: true
```

Resource sharing

An important part of peering is determining what resources to share. Liqo implements a simple request-response model, in that the consumer requests a list of resources (a ResourceRequest) and the provider responds with an offer for a certain amount (a ResourceOffer). Note that at the time of writing it is not supported to ask for specific resources, just empty generic ResourceRequests may be sent.

Let us present an example of a resource request/offer pair:

```
apiVersion: discovery.liqo.io/v1alpha1
kind: ResourceRequest
```

```
3  metadata:
4    labels:
5      liqo.io/remoteID: 4b22f032-9bd7-4afb-a168-91a845e2be50
6      liqo.io/replication: "true"
7    name: liqo-consumer
8    namespace: liqo-tenant-topix-broker-8ef341
9  spec:
10   authUrl: https://194.116.77.110:31466
11 status:
12   offerState: Created
13 ---
14 apiVersion: sharing.liqo.io/v1alpha1
15 kind: ResourceOffer
16 metadata:
17   labels:
18     liqo.io/originID: 4b22f032-9bd7-4afb-a168-91a845e2be50
19     liqo.io/remoteID: 6376f896-8ad0-45b8-b98e-a78e0d6d7ff5
20     liqo.io/replicated: "true"
21   name: topix-broker
22   namespace: liqo-tenant-topix-broker-8ef341
23 spec:
24   clusterId: 4b22f032-9bd7-4afb-a168-91a845e2be50
25   resourceQuota:
26     hard:
27       cpu: 1908m
28       ephemeral-storage: "35913494528"
29       hugepages-1Gi: "0"
30       hugepages-2Mi: "0"
31       memory: "2664000000"
32       pods: "99"
33   storageClasses:
34   - default: true
35     storageClassName: local-path
36   - storageClassName: liqo
37 status:
38   phase: Accepted
39   virtualKubeletStatus: Created
```

In this example, a cluster named `liqo-consumer` requests resources from a cluster named `topix-broker`, which offers approx. 2 CPUs, 36 GB of disk storage and 2.7 GB of RAM. It also offers some storage classes that Persistent Volumes (PVs) can use.

3.2.3 Virtual nodes

The final step is to allow Kubernetes to offload pods to the remote cluster. This is achieved by creating a "virtual node", i.e. a Node resource that does not correspond to a physical node on the cluster. Kubernetes will be able to schedule pods on it as if it were a normal node, but Liqo will intercept pods scheduled on it and reflect them on the remote cluster. The specific component responsible for reflecting the pods is the virtual kubelet, and it synchronizes the local "shadow pod" with the remote pod. It also reflects EndpointSlices to allow the local cluster to reach pods and services that point to the remote cluster.

However, we do not want Kubernetes to treat a virtual node *exactly the same* as a physical node - offloading a pod incurs additional latency and possibly costs, so we want Kubernetes to prevent scheduling on these nodes by default. This is implemented by adding a taint to the virtual nodes, i.e. a condition that pods must explicitly "tolerate" to be eligible for being scheduled on the tainted node. When the user wants to offload a pod, Liqo automatically adds a taint toleration using a webhook.

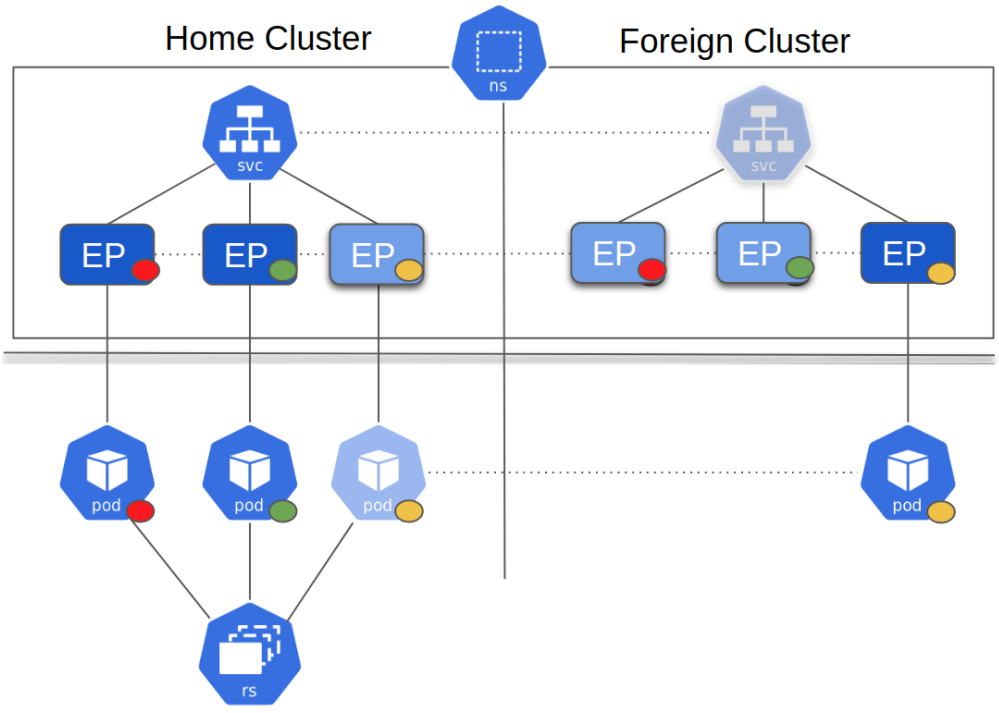

Figure 3.3: Reflection of pods and EndpointSlices.

Chapter 4

Resource brokering

This chapter focuses on explaining the core ideas behind resource brokering on Kubernetes. It provides a clear, high-level explanation of the different roles and functions that play a part in the brokering process. To do this, we start by illustrating user stories and practical scenarios for resource brokering within federated cloud services. From there, we explore potential additional services that can be offered, and subsequently, we develop Liqo-based solutions to provide these services effectively. This way, we aim to offer a comprehensive view of the resource brokering landscape.

4.1 User stories

Resource brokers become essential when establishing extensive Kubernetes clusters designed for resource sharing. These brokers address certain limitations of the peer-to-peer, horizontal model, especially when it comes to scaling up to accommodate a large user base. As the number of providers grows, maintaining comprehensive visibility over the resources and organizations within the network becomes progressively challenging. Conversely, providers might find it harder to place the same level of trust in consumers as they would in a smaller, more familiar network. Additionally, even if all parties possess complete visibility of the global resources, an "overseeing organization" could further enhance workload distribution by utilizing specialized metrics and a deep understanding of each customer's specific requirements. This underscores the significance of resource brokers in optimizing resource allocation within expansive Kubernetes environments.

Taking this into account, we can view the broker as a business entity specializing in facilitating multi-cloud environments. It offers added-value services to well-established, extensive federations. In the context of the GAIA-X framework, the broker serves as an active participant in providing Federation Services. Our thesis

work not only tackles existing GAIA-X Federation Services like Access Management but also broadens the scope by incorporating IaaS offerings into the concept of Federated Catalogues.

As part of our thesis work we held meetings with TOP-IX (Torino Piemonte Internet eXchange), the Internet Exchange Point for north-western Italy. TOP-IX is seeking to expand its business in the direction of being a neutral, trusted intermediary to data exchange and cloud computing, extending its role consistently with the historical nature of IXPs. Our research shows that there are a number of different scenarios that address different needs:

1. **Scenario 1**: there are several cloud operators, each with distinct features and their own clusters. The "cloud market" is sparsely connected, and it is hard to discover participants.
   Customer story: I want to discover the full list of providers and their offerings to choose the best one for my needs.
   Provider story: I want to advertise my resources to potential customers in the federated network.

2. **Scenario 2**: there are many more cloud operators, and it becomes difficult for an individual customer to pick out the "best" one according to some metric. Also, some metrics like latency may not be available to the user, or may be self-certified - either way, they are not readily accessible for scheduling decisions.
   Customer story: I want to choose the optimal cluster, but I don't have the time/data to do it myself.
   Broker story: I want to act as a certifying authority for metrics like latency and uptime, and provide a value-added service.
   Provider story: I want to advertise my resources, and compete with other providers with certified, reliable metrics.

3. **Scenario 3**: there are a number of organizations provide access to data or compute facilities, and want to retain control over who is given access (eg. via specific policies).
   Customer story: I want to run computations over sensitive data (eg. healthcare databases).
   Provider story: I want to make my data available, but I also want to make sure it is in safe hands and in compliance with regulations.
   Broker story: I want to certify customers and workloads, so that I provide a value-added service.

4. **Scenario 4**: there are a number of cloud providers, each individually with a small commercial offer that is not competitive with larger players.

User story: I want to access a large amount of computing resources.
Provider story: I want to pool with other providers to enable a larger, aggregated commercial offering.

4.2 Requirements

We identify three needs that emerge from our user stories. These correspond to three different roles that are mostly independent with one another, which we will develop into three software components.

1. the **need to discover** providers: users rely on brokers to be informed of what clusters are on the network, what are their features and resources, and possibly other informations (eg. a trusted estimate of their uptime or latency).
   The broker takes the **role of catalog**, an endpoint that customers may browse and query.

2. the **need to orchestrate** workloads: users rely on brokers to orchestrate their workloads, either because they do not want to deal with the complexity of orchestration (for an extreme example, a customer may not even use Kubernetes, instead deploying Helm charts from the broker) or because the broker has access to proprietary information that can provide for an optimal orchestration. This process can also go the other way: providers can require customers to go through a trusted orchestrator, that acts as a security gateway to inspect the users' identities or their workloads.
   The broker takes the **role of orchestrator**, an active component that schedules computing resources according to defined policies.

3. the **need to aggregate** resources: providers rely on brokers to present their resources and those of their partners in aggregated form, creating a commercial offering that can compete with larger and more established providers. Unlike the previous two, this broker is completely opaque, acting as a single virtual cluster.
   The broker takes the **role of aggregator**, a middleman that presents a unified view of resources.

Of course, depending on the scenario the broker may enable more than one brokering service. For instance, an exchange point like TOP-IX may act both as a catalog for larger clusters (leaving peering directly to the customer and the provider) and as an orchestrator for smaller ones (centralizing concerns like networking or billing).

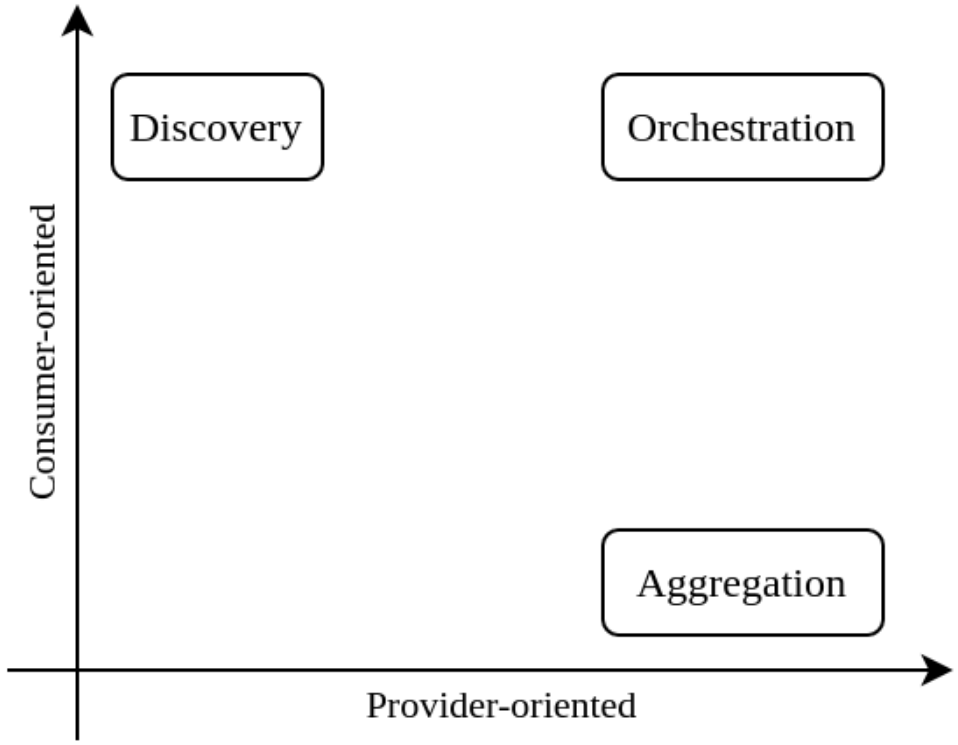

Figure 4.1: A diagram that illustrates how brokering services relate to the needs of providers and consumers.

4.3 Design and architecture

Now that we have a clearer picture of our design goals, let us understand how the standard Kubernetes resources and the Liqo paradigm can be used and extended to accomplish these tasks. The goal of this section is to provide a theoretical architecture that addresses the previously identified use cases and that will guide the actual implementation of the proof-of-concept.

We already mentioned that the requirements naturally map to a set of three roles: catalog, orchestrator and aggregator. We define each of these as a stand-alone software component.

4.3.1 Catalog

We conceptualize the catalog as a database of clusters. The top-level unit of this database is a "cluster document" that uniquely identifies and completely describes the cluster. It is a collection of fields with different functions:

- Administrative information: name, description
- Peering credentials
- Resources
- Metrics (optionally)

Through an administrative procedure, providers are allowed to authenticate themselves against the catalog (eg. with a username/password pair) and advertise on it. On the technical side this translates to creating a minimal "cluster document" containing the administrative information and the peering credentials. The "resources" field can then be pulled directly from the provider's Liqo instance and automatically updated through an appropriate protocol; alternatively, providers can manually enter their commercial offering for a lower "barrier to entry". Finally, the catalog will deploy monitoring pods on the provider's cluster to populate the "metrics" fields.

From the consumer's point of view, the catalog is simply a read-only database. We envisage an appropriate API model such as REST or GraphQL through which the user can query this database (with varying levels of complexity, from "list all clusters" to "list clusters with a complex filter condition"), get a list of providers that satisfy their needs, and then privately peer with them using the peering credentials contained in the database.

We note that because this model is essentially a document database with a well-defined schema and protocols, it can rely to a large extent on state-of-the-art solutions for matters of reliability, scalability, monitoring and others. For example, the underlying DBMS may support replication out of the box, allowing the broker to scale to large amounts of consumers and providers with little additional technical effort.

Ancillary services can also be envisaged besides the already mentioned authentication service. To name a few, consumers will most likely benefit from a Web-based UI where they can interactively browse the commercial offerings, and broker administrators may be interested in collecting statistics from the DBMS into a business intelligence solution to understand how stakeholders interact with the broker. Overall, designing the "catalog" broker around a document database makes it very easy to develop a commercial and production-ready solution.

4.3.2 Orchestrator

Recall the user story described at the beginning of this chapter. To the consumer, orchestration is a value-added service by which optimal providers are chosen according to some (possibly private) metrics; to the provider, it is a tool for

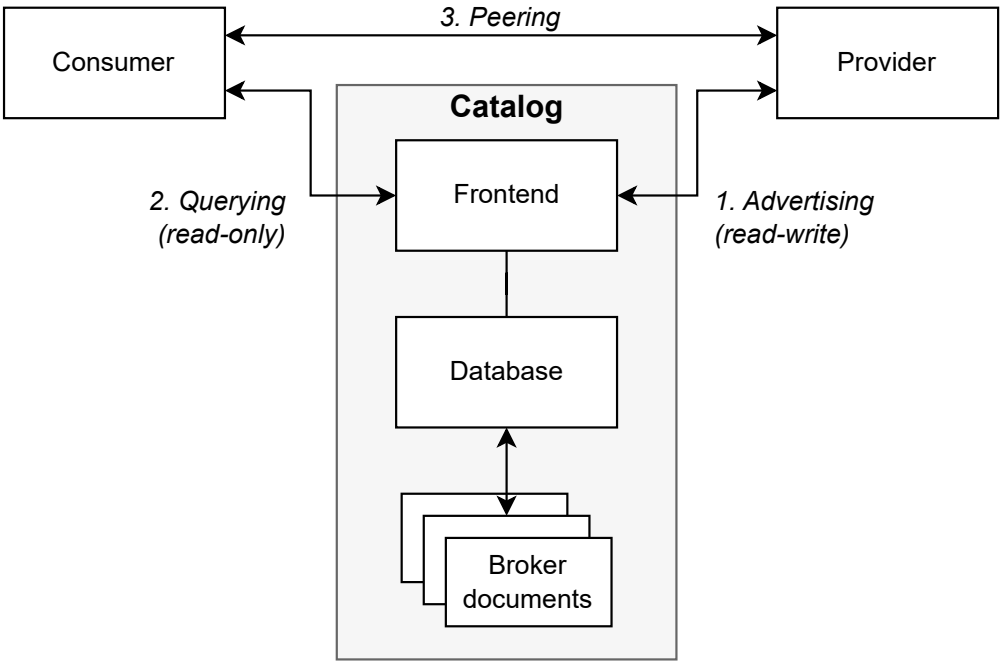

Figure 4.2: A conceptual model of the catalog.

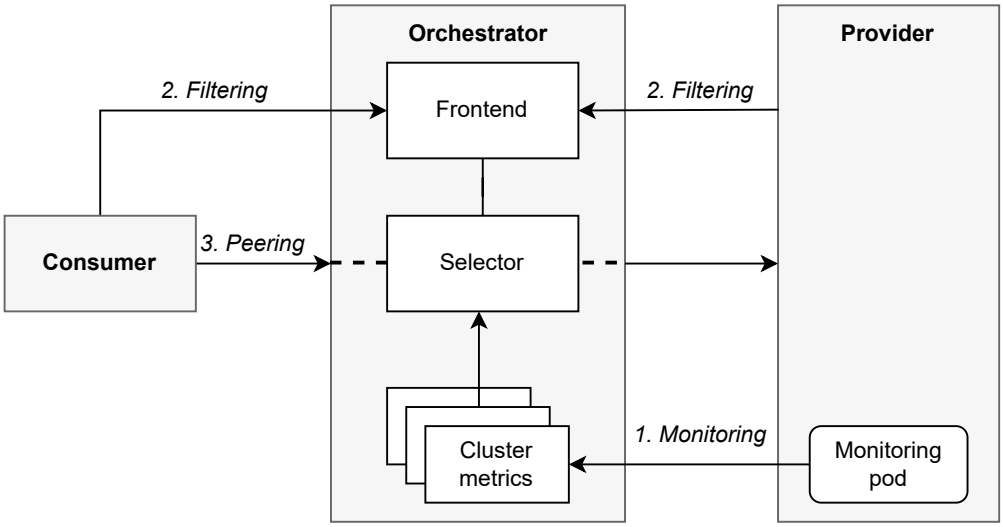

Figure 4.3: A conceptual model of the orchestrator.

competing with certified metrics, as well as - potentially - a security filter in front of consumers.

We can define an orchestrator as an intermediary that operates on the control

plane, with the primary function of enforcing some policy in the peering process and in the distribution of workloads. This policy may take a number of forms, ranging from optimizing metrics to authentication and authorization constraints, but a common feature is that the orchestrator takes the additional responsibility in actuating a policy; compare this with the catalog, which is merely a passive component that provides non-binding suggestions.

Thus, the key component of the orchestrator is the selector, a man-in-the-middle to Liqo peerings that "reflects" the consumer's peering onto a number of providers. It is the material point of enforcement, as policies are applied here to determine what providers will receive the peering.

Policies may be defined on the provider's side, on the consumer's side, or both. A provider will typically define *authorization* policies, that restrict consumers based on identity information (for example, a provider may only be open to peers from an approved organizational entity, or it may require compliance with a specific standard). For this reason, consumers should be able to provide these metadata by means of an authentication stage. As with the orchestrator, part of this process will be rooted in administrative practices, while the technical solution is rather simple - a document-based database with standardized keys may suffice, although more complex solutions like OAuth authentication may also be viable.

Based on discussions with possible stakeholders for Liqo brokering we posit that policies on the consumer's side will be either *compliance* policies or *performance* policies. For an example of the former, an organization may require servers to be based in the European Union to comply with privacy regulations; these policies also pertain to cluster metadata, and can be addressed as already discussed. Performance policies like filtering for a minimum SLA on the other hand need active monitoring. This value-added service can be provided by having the broker offload a monitoring solution onto each provider. This solution can range from simple, custom-built monitors to more complex and state-of-the-art solutions like Prometheus depending on the commercial needs of the broker. No matter the solution, the broker will need to store and continuously update these metrics in an internal database, that the selector will need access to in order to enforce the policies.

Recall that the orchestrator model is not entirely "opaque". While the selection and policy enforcement processes are handled by the broker, we want customers to retain visibility into what providers their pods are being offloaded to, and vice versa, we want a provider to know what customers it is serving. In fact, performance-wise it is ideal for the data plane to be direct between the customer and the provider, without going through the additional hop and potential bottleneck at the broker. If we want to develop a scalable brokering solution it is a prerequisite that we decouple the control plane from the data plane, so that only the control plane may be proxied. For this reason, in the orchestrator design we envisage that customers are peered – at least at the networking level – with the providers that serve them,

which in turn are determined by an API that the orchestrator exposes.

From the point of view of reliability and, partly, performance, the orchestrator is the critical point in this design, concentrating all control plane traffic. The implementation needs to take into account strong reliability and performance needs, and the impact of this additional hop in Kubernetes orchestration (for example, when scaling up or down workloads) needs to be benchmarked in terms of latency. A reliable, production-ready brokering solution should thus evaluate the use of highly-available software components in the entire Liqo stack.

Finally, a frontend should allow cluster operators to define the peering policies they wish to apply. We envisage this frontend as a simple Web application and REST interface to the internal database, again with authentication and authorization features that restrict access to one's own peerings.

4.3.3 Aggregator

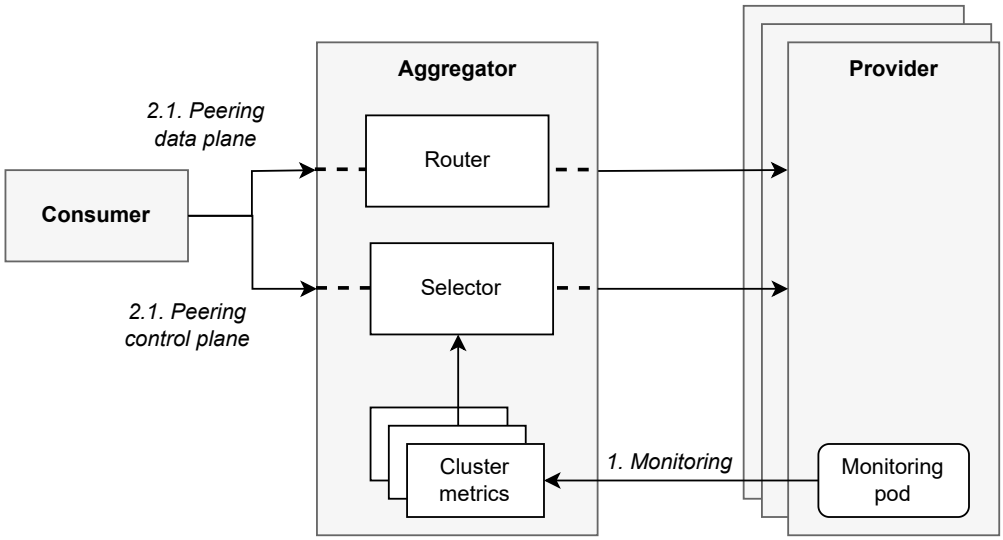

Figure 4.4: A conceptual model of the aggregator. Note the resemblance with the orchestrator architecture, with the difference that the data plane is also routed by the broker.

The aggregator functionality responds to the stakeholders' need for a way to present a unified commercial offering that is the sum - an *aggregate* - of contributions of individual entities affiliated with the aggregator. Unlike the catalog and the orchestrator, which in a sense look like transparent "marketplaces" where one can browse individual providers, the aggregator is an opaque broker, presenting one virtual Liqo cluster.

If we dig deeper into the functional level, this means that the aggregator must be opaque both on the control plane and on the data plane. In this sense, it borrows from the control plane logic of the orchestrator (in both cases, we want a deployment to be reflected from the broker to providers), while the data plane logic must be such that traffic is proxied by the broker cluster and routed to the correct provider (while in an orchestrator setup we want traffic to bypass the broker entirely).

In addition to reflecting pods on providers, we must also aggregate the resource description. Recall that Liqo has a ResourceRequest/ResourceOffer mechanism for representing the amount of resources available in a cluster or engaged in a peering: we want the aggregator to advertise not its own resources, but the sum of the providers' resources. For this reason, an inner "aggregator" component must exist that polls the providers' resources and exposes the aggregate.

Generally speaking, the aggregator must be a Liqo cluster itself, as it will need to carry out the typical functions of a Liqo cluster - authentication, peering, replication and so on. In addition, the aggregator needs to maintain a list of the providers that it aggregates; ideally, this would be as simple as maintaining an outgoing peering towards them. Finally, a production-ready design would need to have monitoring capabilities to understand what clients are peered in and what workloads are being accepted: generally speaking it should be possible to retrieve this data inspecting Liqo objects from a Web frontend, with no modifications to Liqo components.

We note that although the default scheduling algorithm will work correctly, an aggregator has more information available to better inform scheduling decisions, notably the location of provider clusters and inter-cluster latencies. For this reason, the implementation may want to enhance the scheduling algorithm. Collecting eg. network latency metrics may also help to schedule pods in an efficient manner.

4.4 Implementation

We sought to develop an initial, proof-of-concept implementation of the designs in section 4.3. The goal is to have a functional prototype that can be assessed together with stakeholders to determine the suitability for specific edge cases, identify design flaws early and conduct preliminary tests to spot performance bottlenecks.

We envisage brokering not as a core Liqo feature - after all, many people and institutions nowadays use the decentralized peer-to-peer model. Rather, it should be an optional component running on top. For this reason, we set some implementation goals to minimize the dependency on specific Liqo versions and features:

- Rely primarily on Liqo CRDs, rather than Liqo APIs, to interact with Liqo;
- Minimize changes to Liqo core code;

- Where changes to Liqo are needed, design them to be of general interest, beyond the broker user story;
- Develop tests to identify breaking changes in new Liqo releases.

We developed and tested our implementation with Liqo v0.5.0, the latest version available at the time.

4.4.1 Catalog

At the functional level, a catalog needs to receive peering credentials and the commercial offer of each provider cluster, store it, and send the aggregate list to consumers when they issue a query.

The aspect of peering credentials is arguably the simpler of the two. As mentioned in section 3, peering takes place over IP with an optional authentication step based on a token. As such, it is sufficient for a cluster to know the provider's IP address, its cluster ID, and its authentication token. With this information the customer can run `liqoctl add` to establish a peering, and Liqo will proceed to do the rest.

We note that while recent versions of Liqo offer both in-band and out-of-band peering, the former requires some degree of mutual access between the peered clusters (typically, sharing kubeconfigs), while the latter may be more appropriate for such a public-facing scenario. For this reason we adopt the out-of-band credentials (cluster ID, authentication URL, authentication token).

The representation of cluster resources is more complex depending on the specific business requirements: Liqo supports sharing hardware resources with a format that encapsulates Kubernetes' ResourceQuotas, but a customer may also be interested in SaaS offerings (discussed in Chapter 5) or in certified metrics for uptime/latency/etc. Thankfully, the ResourceQuota representation is flexible enough that in principle it can also represent immaterial resources. Indeed, ResourceQuotas are a key-value structure with a well-defined representation for numerical values, and it is sufficient to define a key scheme ("labels") for custom resources: for instance, a provider may offer `"brokering.liqo.io/redis": true`, as long as customers recognize `"brokering.liqo.io/redis"` as a meaningful resource.

We might be tempted to reuse parts of the Liqo peering logic in our catalog. In a one-to-one peering, sharing the list of hardware resources is a substantial part of creating the peering: after authentication is carried out and networking is set up, the consumer creates a ResourceRequest in the provider cluster, which in turn creates a ResourceOffer in the customer cluster to signal acceptance. (This architecture is intended to support a resource negotiation process, as hinted by the CR naming, but at the time of writing it is not possible to encode queries in the ResourceRequest or to limit the resources offered.) A catalog would then need to collect ResourceOffers from all providers, but in doing so it will need to open one

peering for each provider. This makes for a rather unwieldy solution in terms of resource usage, not to mention that the broker needs to run a Liqo instance (and possibly a Kubernetes stack) exclusively for collecting ResourceOffers.

We propose an alternative, light-weight solution where we decouple the resource advertisement from the peering process. Indeed, if we define a stand-alone protocol for advertisements and queries, the catalog only needs to support our protocol. Additionally, decoupling these two processes enables a range of complex resource negotiation logics, ranging from unconditional acceptance (which might be desired in simple, one-to-one peerings where negotiations happen on paper) to extensible and dynamic marketplaces (which we envisage in a wide computing federation).

Our protocol conceptualizes resource offers as a multitude of "packages", each with a set of hardware resources as well as, potentially, SaaS and other immaterial resources. We call each "package" a **plan**, and a collection of plans is an **offer**. This reflects the offer structure of commercial computing providers like Amazon AWS, Azure or DigitalOcean: there are a number of offers optimized for different workloads, and each offer has a number of plans that determine the size.

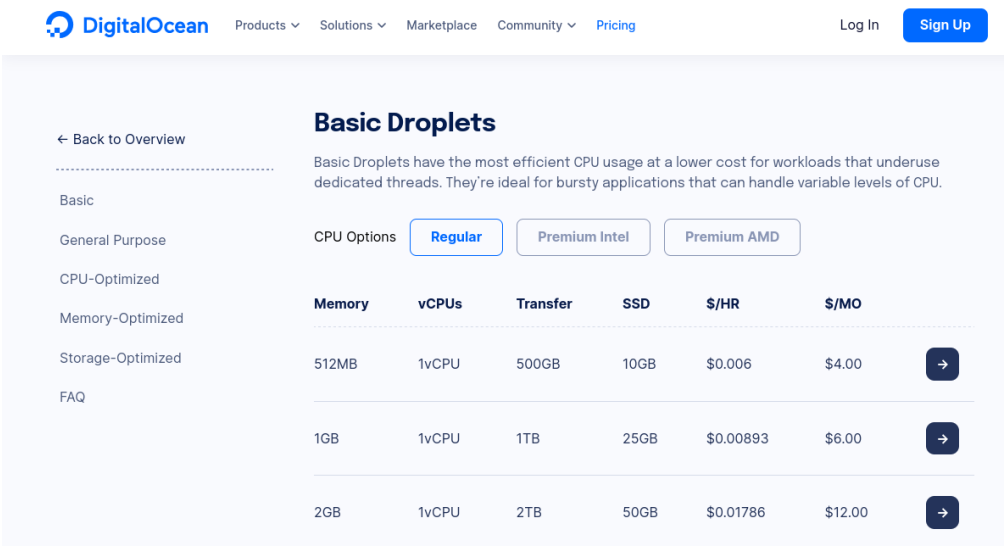

Figure 4.5: Part of the DigitalOcean catalog: note the list of offers on the left and the list of plans on the right.

Our catalog implements, on the provider side, a REST API for CRUD[1] operations on plans. The API requires authentication via JWT[2] as a security layer which

[1]Create, Read, Update, Delete

[2]JSON Web Tokens

prevents unauthorized users from modifying offers. Through this API the provider can publish its commercial offering and update it as required. We implement the following methods:

- `POST /authenticate`: receives peering credentials and provides a JWT
- `GET /offers`: gets a list of offers offered by the provider
- `GET /offer/<id>`: gets the offer identified by an ID
- `POST /offer/<id>`: creates a new offer, or updates it if it exists
- `DELETE /offer/<id>`: deletes the offer identified by an ID

Offers are encoded with a simple JSON object. We report an example offer with three plans:

```
{
  "name": "Storage-optimized offer",
  "type": "storage",
  "description": "We designed this offer for all your storage needs.",
  "plans": {
    "basic": {
      "name": "plan-basic",
      "cost": "1.0",
      "resources": {
        "cpu": "1000m",
        "memory": "2G",
        "storage": "100G"
      }
    },
    "medium": {
      "planName": "plan-medium",
      "planID": "medium",
      "cost": "2.0",
      "resources": {
        "cpu": "2000m",
        "memory": "4G",
        "storage": "300G"
      }
    },
    "performance": {
      "planName": "plan-performance",
      "planID": "performance",
      "cost": "5.0",
```

```
29        "resources": {
30          "cpu": "8000m",
31          "memory": "32G",
32          "storage": "1T"
33        }
34      }
35    }
36  }
```

We apply a similar reasoning for the customer-side interface. A traditional approach would emulate the ResourceRequest/ResourceOffer negotiation as part of a peering, again requiring a Liqo instance on the broker; if we decouple the peering process we only need a mechanism for clients to query the resources available and be informed of any changes of them. In other words, we're looking at a producer-subscriber scheme. We propose a simple mechanism to exchange JSON representations of offers over a WebSocket transport with the following specification:

- When a customer first connects to the broker over WebSocket, the broker sends a full list of providers and offers;
- When a provider creates, updates or deletes its offers, the updated list of offers is broadcast to all subscribed customers;
- No state is associated with subscribers, so if they disconnect and reconnect (eg. because they restart the application, or due to a temporary network failure) the full list of offers is sent again.

As mentioned in the design stage, the catalog needs a document-based database for persistence. The choice of persistence engine fell on NoSQL technologies, and specifically on MongoDB. The main reason for choosing NoSQL databases is that the schema is potentially flexible: a provider may offer resources with arbitrary labels that may not be known to the broker, one may want to extend the project to support more structured offers (eg. including pricing information or constraints), and even the schema for peering credentials may conceivably change in the future. This is complemented by the fact that we need very simple CRUD operations on offers that do not need the complex features of SQL. Finally, we envision the possibility for the broker to run simple queries (but still beyond the scope of CRUD), for example to select all plans below some price threshold: MongoDB already implements support for queries, which we wouldn't have if for example we used simple files for persistence.

We bridged the JSON representation of resources with the native Liqo mechanisms by defining a software component that allows for resources to be read from an arbitrary source, be it a software component or a gRPC source. Especially the

gRPC mechanism allows for very flexible and interoperable setups, not limited to the catalog scenario: one can envisage integration with existing resource monitoring solutions, or applying complex policies of filtering/transformation/etc., so long as one exposes it with a simple gRPC server. On the Go side we decoupled the resource monitoring algorithms and defined a Go interface called `ResourceReader` between it and the Liqo controller manager:

```
1  // ResourceUpdateNotifier represents an interface for OfferUpdater to
       receive resource updates.
2  type ResourceUpdateNotifier interface {
3    // NotifyChange signals that a change in resources may have occurred.
4    NotifyChange()
5  }
6  // ResourceReader represents an interface to read the available
       resources in this cluster.
7  type ResourceReader interface {
8    // ReadResources returns the resources available for usage by the
       given cluster.
9    ReadResources(clusterID string) corev1.ResourceList
10   // Register sets the component that will be notified of changes.
11   Register(ResourceUpdateNotifier)
12   // RemoveClusterID removes the given clusterID from all internal
       structures.
13   RemoveClusterID(clusterID string)
14 }
```

This interface is two-way, allowing the list of resources to be "pulled" regularly via `ReadResources` but also allowing the implementation to "push" updates via `ResourceUpdateNotifier.NotifyChange()` when it detects a change in resources. It is also structured in such a way that it can be chained, so that for example one can have a resource monitor followed by a "quota" stage (like "reserve 20% of the total resources") and a "filtering" stage (for example, a whitelist of clusters that can request GPUs). In general, by chaining ResourceReaders we can have arbitrarily complex mechanisms for reading resources in software.

Another key component for interoperability, as already mentioned, is gRPC support. We call `LocalResourceMonitor` the `ResourceReader` implementation that reads the hardware resources present in the cluster, and `ExternalResourceMonitor` the `ResourceReader` implementation that bridges Liqo with a gRPC server. The Protobuf schema definition looks very similar to the Go interface, where the `ResourceUpdateNotifier` has been replaced with a gRPC stream:

```
1  syntax="proto3";
2  option go_package = "./resourcemonitors";
3
4  // This interface is a gRPC translation of the ResourceReader Go
       interface.
5  service resource_reader {
6    rpc ReadResources (ReadRequest) returns (ReadResponse);
7    rpc RemoveCluster (RemoveRequest) returns (RemoveResponse);
8    rpc Subscribe (SubscribeRequest) returns (stream UpdateNotification);
9  }
10
11 // A request to read resources to be offered to a cluster. The cluster
       ID is passed so that we don't offer a cluster's
12 // resources to itself.
13 message ReadRequest {
14   string originator = 1;
15 }
16
17 // A response representing a Kubernetes ResourceList. Quantities are
       represented as string values (eg. "ram": "1Gi").
18 message ReadResponse {
19   map<string, string> resources = 1;
20 }
21
22 // A request to remove a cluster from a reader's data structures.
23 message RemoveRequest {
24   string cluster = 1;
25 }
26
27 message RemoveResponse {
28 }
29
30 // A request to subscribe to a stream of notifications representing
       possible changes in resources.
31 message SubscribeRequest {
32 }
33
34 message UpdateNotification {
35 }
```

Our work on a headless catalog implementation was complemented by the development of a Web-based frontend by my colleague Alessandro Cannarella in [1]. The use of technologies like JSON and WebSocket that are primarily oriented to the Web enabled the extension, which is written in TypeScript with the React framework, to access the APIs with very little code.

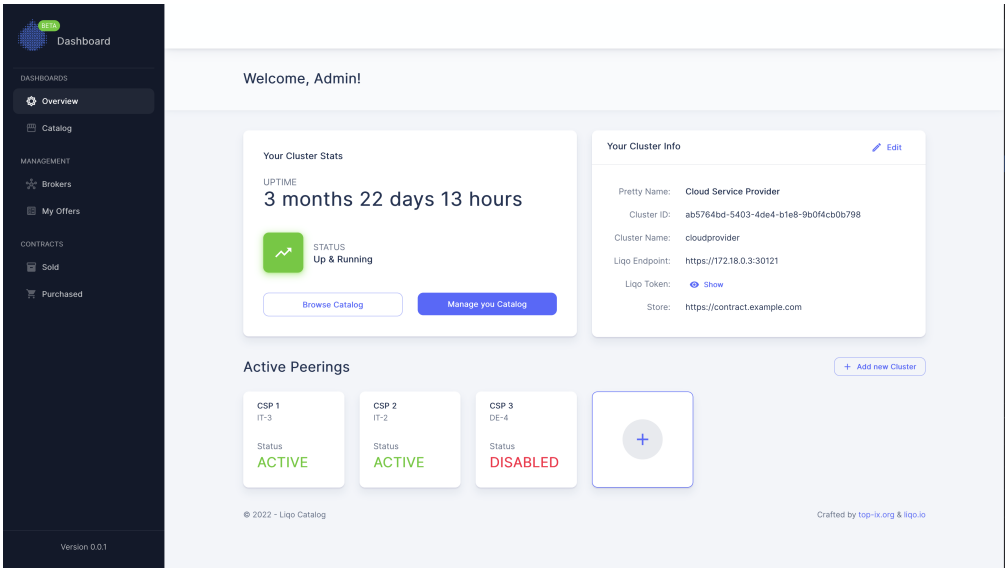

Figure 4.6: The Web frontend for the catalog, courtesy of Alessandro Cannarella.

4.4.2 Orchestrator

The core feature of the orchestrator is to accept workloads and offload them in turn to select providers. If we seek to achieve this result with minimal complexity, we can take advantage of the fact that offloading is itself a core Liqo feature: ideally, we should be able to merely configure Liqo using existing CRDs, without introducing additional custom logic in the control plane except for the choice of providers.

At the same time, it is a defining feature of the orchestrator that the customer's peering is decoupled, with the control plane being mediated by the orchestrator and subject to orchestration policies, and the data plane being a direct connection to the provider. If we look at the Liqo architecture we see that this decoupling is partly in place in the form of having a software component dedicated to Network Address Translation, the network manager. But let us proceed in order.

It is simple enough to accept workloads from consumers and re-offload them to select providers. The logical solution, a "daisy chain" of NamespaceOffloading objects (one on the consumer cluster offloading to the orchestrator, and one on the orchestrator reflecting to providers), works out of the box in Liqo: we just need a software component to detect when new namespaces are offloaded to the broker cluster, and create a matching NamespaceOffloading with an appropriate selector. Specifically, this component will use the orchestrator brokering algorithm (in our case, a simple whitelist) to get a list of cluster IDs that are suitable for re-offloading a namespace, and encode that into the selector for the NamespaceOffloading.

Now that offloading is in place we need to set up routing between the consumer

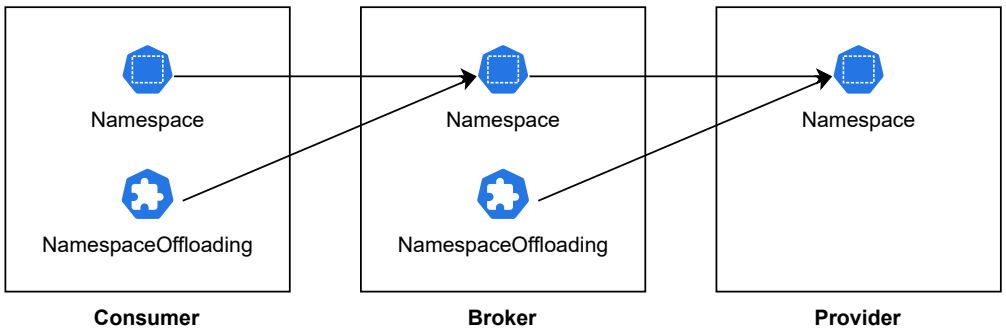

Figure 4.7: A daisy chain of NamespaceOffloadings.

and the provider's pods. Let us review how routing works in Liqo.

A Liqo cluster where pods are running will assign them IPs from a subnet called the "pod CIDR". This is effectively a local subnet (indeed, it is 10.0.0.0/24 by default): it has no meaning outside of the cluster, and isn't publically routable. At the same time, we want a cluster to be able to route packets to a pod that it has offloaded. This is resolved by performing Network Address Translation in the IPAM plugin: a cluster's network manager will allocate a new private subnet and map it to each peered cluster's pod CIDR. This mapping is stored in the appropriately named NatMapping CRD. This ensures that the cluster can refer unambiguously to any pod, local or offloaded.

When a pod is offloaded, the virtual kubelet creates a representation of this pod in the local cluster in the form of a ShadowPod. The IP address assigned to the ShadowPod is given by querying the network manager with the remote cluster ID and the remote IP, and the translation is a simple matter of replacing the network prefix with the local pod CIDR. Vice versa, when packets are routed the network manager will perform the inverse translation, rewriting the local CIDR with the remote one.

For example, say we are peered with a cluster `clusterID=f2c3...` whose remote pod CIDR is 10.0.0.0/24. The network manager may assign the local pod CIDR 10.0.1.0/24, and associate the mapping "source = 10.0.1.0/24, destination = 10.0.0.0/24, cluster ID = `f2c3...`" CIDR with the cluster ID. When we offload a pod, the remote cluster will assign it (for example) the IP 10.0.0.18, the virtual kubelet will ask the network manager to translate `IP=10.0.0.18` in the context of `clusterID=f2c3...`, and the network manager will reply with 10.0.1.18, as it will replace the 24 subnet bits with 10.0.1.

More precisely, the IPAM plugin exposes a gRPC API on the local network manager. Let us look at the API methods:

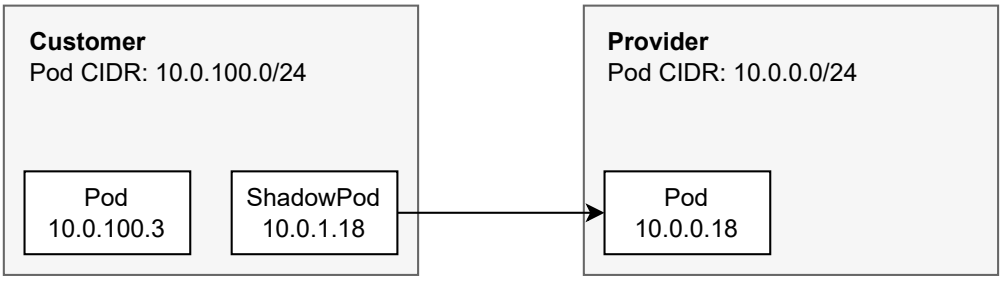

Figure 4.8: Address translation between two clusters.

```
1  syntax="proto3";
2  option go_package = "./ipam";
3
4  service ipam {
5      rpc MapEndpointIP (MapRequest) returns (MapResponse);
6      rpc UnmapEndpointIP (UnmapRequest) returns (UnmapResponse);
7      // ...
8  }
9
10 message MapRequest {
11     string clusterID = 1;
12     string ip = 2;
13 }
14
15 message MapResponse {
16     string ip = 1;
17 }
18
19 message UnmapRequest {
20     string clusterID = 1;
21     string ip = 2;
22 }
23
24 message UnmapResponse {}
```

So far we described how address translation works in a traditional, peer-to-peer setup. In such cases, the local network manager has full visibility over the peering topology, so it can handle address translation correctly. But what happens when we introduce a control-plane intermediary?

If we try to use the local network manager as-is we will quickly find that address translation fails. For clarity, let us describe an example scenario and trace the network request step by step. We have a customer cluster A, an orchestrator cluster

B, and a provider cluster C, where the customer offloads a pod on B and the orchestrator unconditionally offloads it to C. Each of these clusters has a pod CIDR of 10.0.0.0/24; the network manager of A remaps B's pod CIDR to 10.0.1.0/24, and likewise the network manager of B remaps C's pod CIDR to 10.0.1.0/24.

1. The "real" pod, the actual binary, runs on C. Its assigned pod IP is 10.0.0.1.

2. The virtual kubelet running on C, in the namespace liqo-tenant-B, queries C's network manager, which translates 10.0.0.1 to 10.0.1.1.

3. The same virtual kubelet assigns the IP 10.0.1.1 to the local shadow pod. The virtual kubelet running on B then queries B's network manager for 10.0.1.1, which throws an error because the IP does not belong to the pod CIDR of B. The problem is that when we remap an IP we are again using a local CIDR which is not valid in other clusters.

We will present a solution to this problem later in this thesis, but for now let us take a step back. There is a fundamental problem: we introduced two layers of address translation with the orchestrator as a data plane intermediary, which is exactly the scenario we seek to avoid. We must therefore translate the address only once.

We need to have direct Liqo networking between the customer and the provider, which comes with creating a peering between the two: the customer's network manager will then know about the provider's pod CIDR and be able to translate from and to it. However, we are not interested in the full feature set of peerings, just the networking - we won't directly offload pods to this cluster, as the orchestrator will take care of that for us.

We are almost done - we have now achieved separation of the control plane (through the peering to the broker) and the data plane (through the direct peering), and it was simply a matter of reusing existing Liqo resources. We need two more pieces to this puzzle: first, a way to automatically create the direct peering to providers, and second, a way to tell Liqo to use the correct data plane.

As for peering with providers we can imagine a number of solutions. Out of the box Liqo provides some control over the creation of peerings, largely preferring a manual approach: it implements a discovery API through which new clusters can be detected either on a local network (via mDNS) or over the Internet (via DNS), and can automatically peer with these clusters via a configuration flag. In our testbed scenario with a small number of clusters we used mDNS-based discovery with automatic joining, but we note that this approach does not scale well as it effectively results in a full mesh topology that does not scale well as more clusters are added. A possible direction for extending on our work would be to leverage the DNS discovery API so that clients are dynamically peered to only those providers

that they actually need; this approach also comes with its problems, as the interface does not include a method to stop peerings, and for this reason the development of a new light-weight protocol is suggested.

We also note that at the time of writing authentication is stubbed at the peering level. While this aspect can be overlooked while investigating the resource brokering concept, a production environment needs to have solid authentication in place, which can be another possible direction for extending Liqo to multi-tenant use cases.

As for the the choice of data plane, this ultimately comes down to making IPAM aware of which cluster the pod is actually running on. Where a standard Liqo assumes that the "next hop" (i.e. the other side of a peering) is the one where the pod runs, we add a mechanism to detect the provider's coordinates and use those for address translation. More precisely, for each offloaded pod we set a label `broker.liqo.io/root-cid` that defines the "final" cluster ID, and a label `broker.liqo.io/root-ip`, that defines the pod IP. This label is only set by the virtual kubelet if the pod is "real", i.e. not a shadow pod. From there on, the existing Liqo code propagates this label upstream and makes it reach the user's cluster. The user runs a modified version of the virtual kubelet with a mechanism that detects this label, and when it is present it is explicitly passed to the IPAM. Thus we achieved support for brokering - in fact, "brokering chains" of arbitrary depth, where the label can bubble upstream as many times as needed - while keeping "backwards compatibility" with 1-to-1 peerings.

Algorithm 1 Label propagation in the Orchestrator VK

```
const rootIPLabel ← "brokering.liqo.io/root-ip"
const rootCIDLabel ← "brokering.liqo.io/root-cid"
if remotePod is of type ShadowPod then
    localPod.Labels[rootIPLabel] ←remotePod.Labels[rootIPLabel]
    localPod.Labels[rootCIDLabel] ←remotePod.Labels[rootCIDLabel]
else
    localPod.Labels[rootIPLabel] ←remotePod.PodIP
    localPod.Labels[rootCIDLabel] ←remoteCluster.ID
end if
```

4.4.3 Aggregator

We note that the aggregator's main external function, the aggregation of resources, is in fact also a problem that we already solved: we can leverage the ResourceReader API (and specifically, the gRPC API) to create a custom component that computes the sum of the providers' resources, optionally with additional policies (eg. quotas,

oversubscription). The component need only broadcast the read requests to all providers, and vice versa join each provider's push channels into one. In principle, this operation can be carried out by a separate component without the need for Liqo, exactly because we decoupled the resource interfaces from Liqo operation.

On the other hand, reusing the orchestrator's architecture means that we must run Liqo on the broker cluster: recall that at its core an orchestrator works by accepting pods and rescheduling them on other clusters by means of `NamespaceOffloadings`. We can keep this architecture in place, and only deal with the change in the address translation mechanism. This requires some thought, as we now need two layers of address translation where we previously only had one.

We introduced this problem in 4.3.2. At its core, the problem occurs in the address translation stage that is carried out in the virtual kubelet. The virtual kubelet must be able to distinguish between pods that are "local", i.e. running directly on the peered cluster, and those that aren't, because they are further offloaded. Fortunately, this distinction can be carried out at the networking stage, without querying the Kubernetes API: recall that Kubernetes distributions define a "pod CIDR" from which pods are assigned IPs, and the Liqo IPAM is aware of its own cluster's CIDR. This means that at a high level we can modify the address translation logic to perform the following check: "if the pod is not part of the (peer's) pod CIDR, then resolve it in the peer's IPAM, otherwise, resolve it in the local IPAM as usual."

Algorithm 2 Address translation in the Aggregator VK

```
if originalIP ∈ peerIPAM.podCIDR then
    translatedIP ← peerIPAM.Translate(originalIP, localCluster.ID)
else
    translatedIP ← localIPAM.Translate(originalIP, peerCluster.ID)
end if
```

Note that Algorithm 2 requires us to have a connection to the peer IPAM service, which the peer must expose to its clients' virtual kubelets. The peer IPAM must also implement a function `BelongsToPodCIDR`, as the pod CIDR by itself is only known in the local cluster and isn't shared. Other than that, this implementation reuses the existing address translation functions, as well as the gRPC interface to invoke them.

4.5 Future work

In this chapter we laid the conceptual groundwork for Liqo-based Kubernetes brokering, as well as implementations for each of the three kinds of brokers.

These implementations are at a proof-of-concept level, demonstrating the key ideas behind brokering: one possible direction for future work is refining them for production usage, but they also open the door to many interesting improvements and advancements.

4.5.1 Catalog

The catalog being the simplest solution of the three, it is also the closest to a production-ready implementation. We note that in a federation users will have to agree on a common format to represent their computing resources: the schema we suggest can represent common resources like CPU percentage and RAM usage with the Kubernetes standard, but more exotic resources like GPUs or FPGA accelerators do not have a standard representation.

We also note that the Web dashboard that we developed does not currently feature authentication or authorization. On the provider side, because Liqo partially provides a public key infrastructure (PKI) it is possible in principle for a cluster to authenticate itself against the catalog and demonstrate its identity; on the consumer side such a feature may be trivially implemented in the Web interface for purposes like billing and providing paid access to premium features.

The catalog also relies on self-reported information, which is often enough for a marketplace but not always: as mentioned in the introduction users may also be interested in certified measurements and KPI. With the current architecture it is quite simple to add such a feature, as the catalog may offload pods acting as "metrics agents" on the remote clusters. Such pods would read available resources but also for example bandwidth, latency and uptime, and report them to the central catalog.

4.5.2 Orchestrator

As already mentioned in the respective chapter, the orchestrator is structured in such a way that a full mesh of point-to-point VPNs is created. This greatly hinders scalability to more than a few tens of hosts, causing a lot of "background noise" on the underlying network. An immediate improvement would be thus to design a protocol (eg. a simple exchange of cluster credentials over Protobuf) for the orchestrator to communicate which providers are hosting the customer's pods and thus should be peered with the customer, avoiding unused connections. It would also be interesting to investigate how to move from a point-to-point paradigm to a VPN "concentrator" where the client uses one VPN for multiple providers, reducing the communication and encryption overhead in using multiple clusters. However, such a work would need to be done at the Liqo level, as it is more fundamental than the specific use case of brokering.

We also note that the propagation of labels upstream from the provider to the customer relies on an undocumented behavior of the virtual kubelet which also causes a lot of back-and-forth changes in the Kubernetes API server. Again, this behaviour is acceptable in a proof-of-concept deployment with a limited number of hosts and pods, but it is an inefficient patch that hinders scalability past a certain number of offloaded resources. In a real deployment the virtual kubelet should be optimized for this mode of operation that was not previously exploited.

Finally, we note that our proof of concept demonstrates a simple usage of the orchestrator. An interesting showcase of the strengths of the orchestrator model would be to demonstrate its usage in enforcing security policies, for example by authenticating and authorizing consumers or by whitelisting permissible workloads.

4.5.3 Aggregator

The aggregator is implemented using existing Liqo primitives. While this makes it relatively easy to overlay on top of an existing Liqo installation, it also limits the visibility that the aggregator has into the resources that it provides, reducing each provider cluster to the big-node abstraction. A design more specialized for aggregator use cases would be able to take into account the internal topology of these clusters in optimizing the allocation of pods and resources, for example by reducing the latency between tightly coupled pods or by ensuring that critical pods are replicated across different availability zones.

In this regard, we note that the Kubernetes scheduler `kube-scheduler` could also be extended with plugins to make it "Liqo-aware", i.e. take into consideration performance metrics from Liqo when determining the best node for a pod.

In conclusion, our proof of concept demonstrates the fundamental mechanism of multi-cluster orchestration and proves that it can be done in the framework of Liqo. Although more optimal implementations can be conceived, they come with a trade-off in terms of impact to the Liqo codebase. As the Liqo project has taken the direction of decoupling its components to allow for exotic and custom use cases, we foresee that a reimplementation of the broker in light of this new architecture may be both cleaner and more performant.

Chapter 5

Service brokering

In this chapter we aim to approach the general topic of brokering from a different perspective. In Chapter 4 we looked at how one can use Kubernetes to broker *resources*, more specifically computing resources - either raw (some amount of CPU time, RAM, etc.) or packaged (as is the case with the broker dashboard). There exists a trend in the Kubernetes, and in the wider cloud computing ecosystem, to solve a similar problem at a higher level, notably *service brokering*.

The "30.000 foot view" of this process is that it provides an easy way to bridge SaaS-like commercial offerings with the customers' workloads in a unified, provider-agnostic way. Thus, it represents a way to interconnect the global ecosystem of services which is currently fragmented by a multitude of technologies and provider-specific APIs by virtue of a simple, generic interface.

Within the general topic of service brokering we can find a number of solutions with various degrees of maturity:

- GAIA-X Federation Services (GXFS);
- International Data Spaces (IDS);
- Open Service Broker API (OSBAPI).

In this chapter we will present and evaluate each of these solutions, highlighting their strengths and weaknesses, then give an overview of how these concepts can be integrated in the Liqo software to offer a full range of brokering capabilities.

5.1 Use cases

The cloud computing ecosystem features a vast number of SaaS providers. For example, if we look at a common RDBMS like MySQL/MariaDB, we can choose between:

- Amazon RDS (Relational Database Service) for MySQL **??**;
- Azure Database for MySQL **??**;
- Google Cloud SQL for MySQL **??**;
- Various offerings from smaller cloud providers like Alicloud, OVH and others.

Each of these services has a different way to create a new instance: all of them support this via an HTML interface, AWS does this programmatically via its CloudFormation product, Azure has the Azure Resource Manager, GCP has an API and CLI of its own, and so on. Even so-called "infrastructure as code" solutions often don't have a unified model of hosted resources, and are instead designed around the principle of managing one's own databases: for instance, the popular IoC solution Terraform by Hashicorp has a specific resource type for Amazon RDS and one for Azure Database.

The absence of a common API is an important factor in the phenomenon of "vendor lock-in", where a consumer that wants to acquire compute resources is pushed towards using a single vendor for the entire stack due to the fact that each component is tightly coupled to its vendor's implementation and has poor interoperability. This can make it difficult to switch vendors or migrate to a different platform, as the customer must either recreate their entire stack from scratch or risk data loss and other disruptions. Vendor lock-in is a major issue for those looking to invest in SaaS solutions, and can significantly limit the flexibility of their operations.

Interoperability in computing resources has thus become a key point in the cloud computing strategy of the European Union commonly known as GAIA-X. This initiative is driven by the need for the EU to have sovereignty over its data and digital infrastructure, and it aims to create an open and trusted digital ecosystem for data and services. To this end, GAIA-X supports the development of open APIs for cloud services, which will allow for vendors to build plug-and-play cloud services that can be used across vendors, thus enabling users to switch providers with minimal disruption.

5.2 GAIA-X Federation Services (GXFS)

The GAIA-X paradigm is based on digital ecosystems called "GAIA-X Federations". In these federated systems, multiple actors are interconnected and can exchange data and services in a safe and trustworthy manner. There are multiple facets to this interconnection and multiple facilitators: key features of GAIA-X Federations range from compliance to identity management to data sovereignty. Such features as a whole are called GAIA-X Federation Services, or GXFS for short: they are

functionalities provided by the federation to facilitate the exchange of data between participants.

The GAIA-X Federation Services whitepaper **??** identifies a first set of fundamental services, notably:

- Identity and Trust Services, enabling the concepts of authentication and authorization in a decentralized environment;
- Federated Catalogue, enabling the discovery and selection of data providers within the local federation;
- Data Sovereignty Services, encoding data exchanges as transactions that can be logged and negotiated in the form of data contracts;
- Compliance, ensuring that participants adhere to a common framework defined by the federator.

The Federated Catalogue is one of the more active areas of research in this field. Driven by the need to interconnect and discover services in a federation, the GAIA-X project has produced a solid specification for federated catalogues. Notably, the Federated Catalogue is based on the concept of "self-descriptions", which are a standardized way for participants to describe their identity and features.

At the functional level, self-descriptions are specific "claims" about one's service that are validated and optionally monitored by a trusted party (respectively creating a Verifiable Credential and a Verifiable Presentation). At the implementation level, Verifiable Credentials are subject-predicate-object triples about RDF entities, cryptographically signed as a proof graph and wrapped in a JSON-LD object. This object, in turn, is part of a knowledge graph that resides on the catalogue and may be consumed by participants using a REST API.

Here is an example of a fully fleshed out JSON self-description, courtesy of GAIA-X:

```
1  {
2    "@context": ["https://www.w3.org/2018/credentials/v1", "https://
       registry.gaia-x.eu/v2206/api/shape"],
3    "type": ["VerifiableCredential", "LegalPerson"],
4    "id": "https://compliance.gaia-x.eu/.well-known/participant.json",
5    "issuer": "did:web:compliance.gaia-x.eu",
6    "issuanceDate": "2022-09-23T23:23:23.235Z",
7    "credentialSubject": {
8      "id": "did:web:compliance.gaia-x.eu",
9      "gx-participant:name": "Gaia-X AISBL",
10     "gx-participant:legalName": "Gaia-X European Association for Data
       and Cloud AISBL",
```

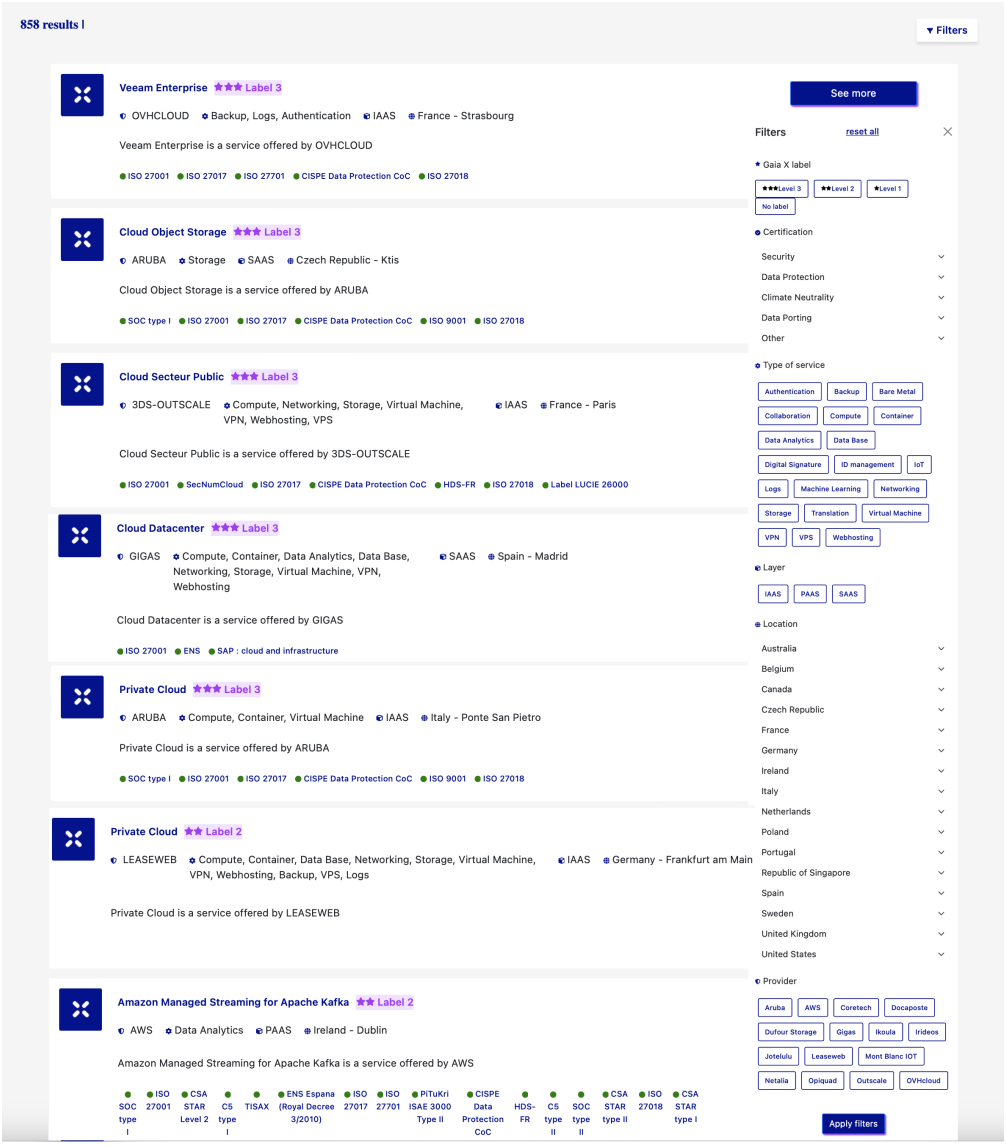

Figure 5.1: A graphical frontend for a GAIA-X Federated Catalogue developed by French Gaia-X Federated Services (GXFS-FR) initiative. The catalogue features more than 170 services across 69 locations and 16 countries.

```
    "gx-participant:registrationNumber": {
      "gx-participant:registrationNumberType": "local",
      "gx-participant:registrationNumberNumber": "0762747721"
    },
    "gx-participant:headquarterAddress": {
      "gx-participant:addressCountryCode": "BE",
```

```
      "gx-participant:addressCode": "BE-BRU",
      "gx-participant:streetAddress": "Avenue des Arts 6-9",
      "gx-participant:postalCode": "1210"
    },
    "gx-participant:legalAddress": {
      "gx-participant:addressCountryCode": "BE",
      "gx-participant:addressCode": "BE-BRU",
      "gx-participant:streetAddress": "Avenue des Arts 6-9",
      "gx-participant:postalCode": "1210"
    },
    "gx-participant:termsAndConditions": "70
    c1d713215f95191a11d38fe2341faed27d19e083917bc8732ca4fea4976700"
  },
  "proof": {
    "type": "JsonWebSignature2020",
    "created": "2022-10-01T13:02:09.771Z",
    "proofPurpose": "assertionMethod",
    "verificationMethod": "did:web:compliance.gaia-x.eu",
    "jws": "eyJhb..."
  }
}
```

We see that this self-description asserts some machine-readable facts about a `credentialSubject` (notably, its legal identity) and provides a cryptographic signature in the form of a JWS (JSON Web Signature; in this case, it wraps a SHA256 hash and an RSA signature).

We observe that the concept of the GAIA-X Federated Catalogue is extremely extensible, and can provide a variety of brokering functionalities to a GAIA-X ecosystem. Participants use a well-defined, standard API to assert arbitrary facts about themselves, and the catalogue federates them to enable discoverability. In this sense, the Federated Catalogue is much more powerful than a service broker, being able to represent arbitrary relations between entities in its ontology and to verify them with trusted parties using cryptographic signing. However, at the time of writing there are regrettably no open source implementations of this federation service that we could deploy to demonstrate these concepts in practice.

In conclusion we look to GAIA-X Federation Services as a promising and flexible platform that offers both an important use case for brokering as well as a strong theoretical groundwork regarding the role of federators and their interfaces. Integrating brokering features with the notion of Self-Descriptions and of Federated Catalogue are a key challenge for federators that want to compete in the GAIA-X landscape, which today features many major players of the cloud economy in the European Union.

5.3 International Data Spaces (IDS)

The International Data Spaces Association (IDSA) is a non-profit organization that includes more than 130 companies based in 22 countries across the European Union and the world. It is the primary developer of the International Data Spaces model for secure and trusted data sharing, providing reference architectures and implementations, a governance model and continuous feedback based on use cases from the industry.

The founding value of the IDS model is data sovereignty, that is, an infrastructure in which data exchange is trusted and bound by usage restrictions with a view to enabling a data economy. Each datum is accompanied by metadata that describes its nature and the usage policies that apply to it. At all levels of the data value chain, a technical infrastructure facilitates contractual agreement on the exchange of data and enforces the policies specified in contracts. These may regulate operations like the processing, linkage or analysis of the data referred therein, as well as its visibility.

At a technical level, we regard inter-cluster brokering as a building block that can enable data sovereignty solutions. Notably, we envision a broker that acts not only as an active exchange of computing-related metrics and as a marketplace, but also as a "firewall" of sorts that can apply intelligent policies to select which consumers are paired with which providers. The broker becomes a key part to the data space ecosystem that can apply security policies, enhancing the data space with a stronger security posture and increased trust in the system. For a concrete example, imagine a data space where a hospital seeks to share sensitive health data with select academic users for the purposes of carrying out a scientific study. The broker can enforce a whitelist of institutions that are in compliance with the appropriate legislative and technical requirements, thus making it simpler to fulfull compliance obligations in the distribution of sensitive data. Likewise, the broker can act as a smart filter for parameters like data privacy, licensing permissions and so on.

The author carried out a review of the IDS ecosystem in April 2019 as part of an internship at TOP-IX (Torino Piemonte Internet Exchange, the IXP for Northwest Italy) focused on practical use cases of data spaces. A minimal, internal deployment (so-called "Minimum Viable Data Space") was successfully carried out, with a file resource being exchanged between two entities mediated by a broker and a certification authority, using reference software implementations developed by IDSA; we note that other implementations exist, eg. the file exchange gateway has a mainstream implementation called Eclipse Dataspace Connector ("Connector" is the official name for the file exchange gateway software that is run by all participants), but these implementations were not tested as the scope of the demonstration was rather to assess the theoretical capabilities of the IDS architecture. Following the

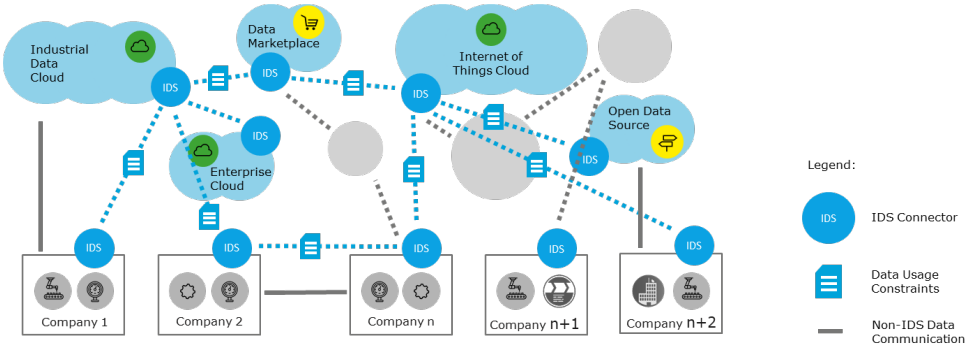

Figure 5.2: A representation of the International Data Services architecture. Data spaces enable multiple companies and cloud to exchange data with one another, creating complex topologies of data exchanges that give rise to data value chains. This high-level overview does not include intermediaries such as authentication providers and brokers. Source: International Data Spaces Reference Architecture Model. Licensed under CC-BY 4.0.

demonstration, the author had the opportunity to meet with representatives of IDSA to exchange feedback related to the pilot deployment as well as the current and future state of the IDS architecture. TOP-IX also took part to these meetings as an IDSA consortium member and stakeholder.

The IDS ecosystem features a "broker" entity that acts as a registry of assets available in the local data space. Constituted of a MongoDB database with an HTTP frontend, the broker allows machines and (optionally) humans to alternatively register a list of resources available on some Connector in the network, or browse the list of resources and fetch connection details to request the resource from the Connector where it is stored. We note that there were some technical difficulties in deploying the reference implementation of the broker, but the demonstration was eventually successful in exercising both the request and the response methods in the Minimum Viable Data Space. No other implementations of the broker appear to be publicly available.

One major shortcoming of the IDS architecture was identified in the lack of substantial support for representing dynamic data such as the result of queries: while this scenario is outlined in the IDS whitepaper, it is poorly defined and not supported in any publicly available implementation of the Connector. This reflects the general design of data spaces as a technical infrastructure that augments traditional file sharing services with machine-readable metadata like descriptions, licenses and usage policies - a design that in its current form poorly suits the modern need for dynamic data like refreshable documents or the results of queries. Indeed, such dynamic data endpoints are shoehorned into the existing structure as being

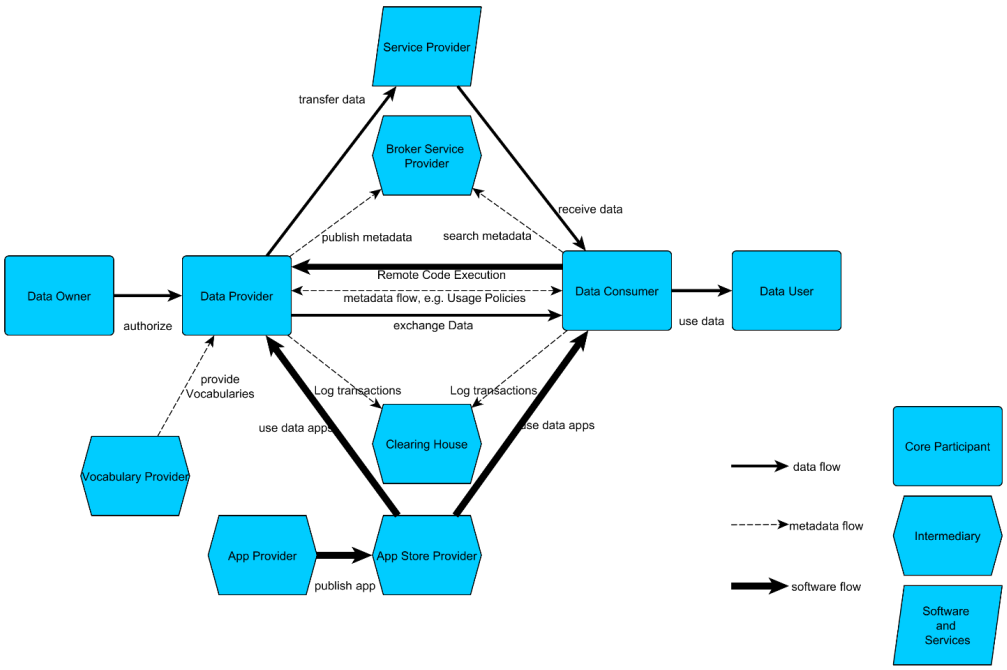

Figure 5.3: A closer look into how two entities, a "data owner" and a "data user", exchange data over International Data Spaces. Each entity uses a Connector to communicate with IDS participants and intermediaries, of which there are several types. This chapter looks at the "broker service provider", which data owners can publish metadata to and data users can search metadata on. Source: International Data Spaces Reference Architecture Model. Licensed under CC-BY 4.0.

static assets that contain configuration and credentials to access an external API endpoint (not proxied via the Connector). There are some obvious disadvantages to this, notably that the Connector is effectively unable to exercise continuous control over the consumption of data, which is one of the key proposition of data spaces. We note however that the IDS architecture may be organically extended by adding a new type of datum that properly represents dynamic data and that is effectively controlled by the Connector..

The possibility of regulating the data exchange at the network layer in IDS data spaces was also investigated by TOP-IX. The goal is to provide network-level guarantees, such as ensuring that the traffic does not leave the European Union or that it is routed through well-known hosts and Internet exchange points. The IDS architecture does not allow for such a role in its current iteration, as it encodes data usage policies at ISO/OSI layer 7 ("Application") and enforces them at layer 4 ("Transport") with no control over IP routing. In practice this means that data exchanges take place over HTTPS with best-effort routing, Connectors aren't able

to communicate routing policies, and there is no intermediary with the specific goal of enabling routing policies. Recalling that data spaces exchange either files or URLs with metadata, we note that such a mechanism could be introduced by hosting assets on IPs that are only routable over a trusted VLAN/VPN or similar solution, but this is a non-standard mechanism that requires specific client-side support and is not generally supported by the IDS ecosystem. Rather, such a solution would have to be discussed with IDS stakeholders for possible inclusion in a future iteration of the IDS whitepaper.

5.4 Open Service Broker

The Open Service Broker API is somewhat heterogeneous to the other two solutions proposed here. Unlike GAIA-X and International Data Spaces, which are wide-ranging projects that envision an ecosystem of some kind (a computing ecosystem for GAIA-X and a data ecosystem for IDS), the Open Service Broker API is more modest, defining an HTTPS interface with which SaaS solutions can be purchased and used interoperably.

The Open Service Broker API exists to address the ever-increasing variance in APIs for cloud-native solutions. By this, we mean that although many PaaS and SaaS platforms may provide the same core service, each has its own interface to access service metadata, provision and tear down the resources they offer, often with each interface being incompatible with another provider's. This contributes to vendor lock-in, which can increase operational costs by hindering interoperability with cheaper providers, as well as increase the inherent complexity of the cloud-native software ecosystem. For example, suppose that we want to provision a small MySQL database on four major cloud providers from a Python API. Our scripts may look like this:

```
# Amazon RDS

import boto3

rds = boto3.client('rds')

rds.create_db_instance(
    DBInstanceIdentifier='my-mysql-db',
    Engine='mysql',
    DBInstanceClass='db.t2.micro',
    MasterUsername='admin',
    MasterUserPassword='mypassword',
    AllocatedStorage=20
```

```
)
```

```
# Google Cloud Platform

from google.cloud import sql_v1beta4
from google.oauth2 import service_account

creds = service_account.Credentials.from_service_account_file('path/to/creds.json')
client = sql_v1beta4.CloudSqlInstancesServiceClient(credentials=creds)

instance_body = {
    "region": "us-central1",
    "database_version": "MYSQL_5_7",
    "settings": {
        "tier": "db-f1-micro",
        "backupConfiguration": {
            "enabled": True,
            "binaryLogEnabled": False
        }
    }
}
instance = client.instances().insert(project='my-project', body=instance_body).execute()
```

```
# Microsoft Azure

import os
from azure.identity import DefaultAzureCredential
from azure.mgmt.rdbms.mysql import MySQLManagementClient
from azure.mgmt.rdbms.mysql.models import ServerForCreate

credential = DefaultAzureCredential()
client = MySQLManagementClient(credential, subscription_id='my-subscription-id')

server = ServerForCreate(
    name='my-mysql-server',
    administrator_login='admin',
    administrator_login_password='mypassword',
    location='eastus',
    sku={
```

```
        'name': 'GP_Gen5_2',
        'tier': 'GeneralPurpose',
        'family': 'Gen5',
        'capacity': 2
    },
    storage_profile={
        'storage_mb': 5120
    }
)

client.servers.create('my-resource-group', 'my-mysql-server', server)
```

```
# AliCloud

import openapi_client
from openapi_client.api_client import ApiClient
from openapi_client.configuration import Configuration
from openapi_client.api import rds_api

config = Configuration()
config.access_key = 'my-access-key'
config.access_secret = 'my-access-secret'
config.endpoint = 'https://rds.aliyuncs.com/'

api_client = ApiClient(configuration=config)
api_instance = rds_api.RdsApi(api_client)

instance_body = {
    'DBInstanceDescription': 'My MySQL database',
    'Engine': 'MySQL',
    'EngineVersion': '5.7',
    'DBInstanceClass': 'rds.mysql.t1.small',
    'DBInstanceStorage': 20,
    'PayType': 'Postpaid'
}
api_instance.create_db_instance(body=instance_body)
```

We clearly see that even a simple, common operation like provisioning a database instance takes very different forms - not due to conceptual differences, but merely due to different boilerplate. A business with an Azure stack may not easily extend its scripts to provision AWS resources, and vice versa.

The goal of the Open Service Broker API is to overcome these formal differences by defining a common API for service providers. Through this API, consumers can

browse the commercial offer of several SaaS providers at the same time, provision and deprovision instances, and connect these with their stack. The provider-side endpoint is called the Service Broker. This creates a naming inconsistency with the broker role described so far, i.e. an aggregator of metadata about computing providers: throughout this chapter we will refer to the former as a Service Broker, and the latter as an aggregator broker. Because Service Broker descriptions are interoperable, an aggregator broker is a viable and desirable role, fulfilling the need for a marketplace of SaaS offers.

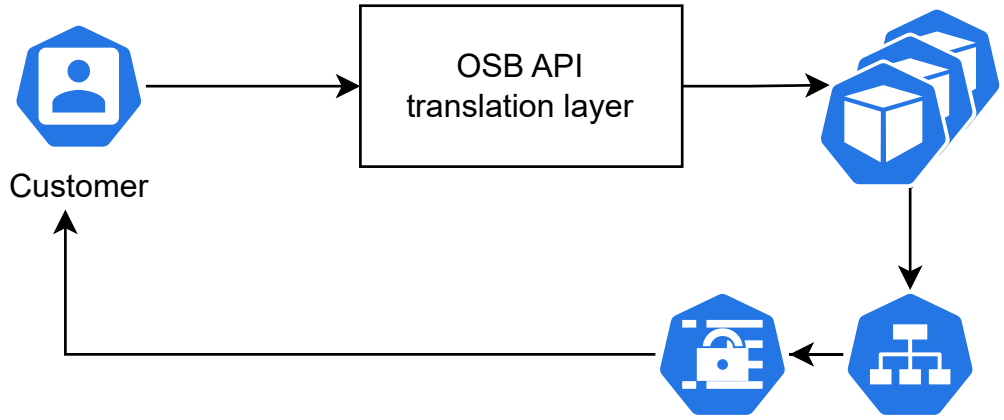

Figure 5.4: A representation of the Open Service Broker architecture, represented here with Kubernetes resources. Through the Open Service Broker API, customers can provision resources (pods) that are exposed in the form of services with credentials (secrets).

Let us briefly go through the main methods of the Open Service Broker API. The lifecycle looks like this, where each method is a REST call with a well-defined JSON schema:

1. Fetching the catalog from the service broker;
2. Provisioning new instances;
3. Connecting and disconnecting instances from applications;
4. Deprovisioning instances.

At a high level, catalogs are collections of services (an application with some functionality, e.g. a MySQL database) which themselves are in a one-to-many relationship with plans (a size for the application, eg. "small: 1 GB"). Both human-readable and machine-readable metadata is associated with each service and plan, which can be either descriptions or pricing information.

New instances of a service can be purchased and provisioned or deprovisioned with a PUT and DELETE call respectively. This instructs the provider to instantiate the actual resource that provides a service, eg. a virtual machine or a pod.

Finally, consumers are able to connect to service instances by a process called "binding". Consumers can instantiate a binding for a resource and receive what is typically an endpoint together with a set of credentials for the underlying protocol spoken by the application (eg. the MySQL client/server protocol in our case).

With this generic mechanism, consumers and providers are effectively able to discover and purchase SaaS resources in a provider-agnostic way, with seamless integration with potentially hundreds of providers and a rather simple API.

There are a number of products implementing the Open Service Broker API. Of particular interest to us is the Kubernetes Special Interest Group for the Service Catalog. The Service Catalog implements the Open Service Broker API translating API calls into deployments (to provision instances), services and secrets (to connect external application to the provisioned instances). We note that although the project saw some momentum and adoption by large customers like SAP and Orange France, it was discontinued by the Kubernetes project in May 2022.

Chapter 6

Evaluation

Having described the functional aspects of the proposed architectures and implementations of resource brokers, we now wish to characterize their performance profile. Specifically, during the design phase we noticed that some broker architectures introduce an additional hop on the control and the data plane; we wish to characterize the performance penalty thus introduced, both in terms of apparent scalability and of quantitative changes in latency on specific operations.

Because we are mainly interested in network latency measurements, we use the simplest broker topology - one provider, one broker, one consumer, on three different virtual machines. We note that tests with a larger number of providers, brokers or consumers may be more of interest when benchmarking brokering algorithms (for example, for the orchestrator to select suitable clusters given some cost function and conditions), but as this thesis merely concerns itself with the general software architecture, we leave this task for future developments in the direction of broker scheduling algorithms.

The virtual machines had two Intel Skylake cores, 2 GB of RAM, and ran Ubuntu 20.04 LTS. We would like to thank the Crownlabs team for the technical support.

6.1 Orchestrator benchmarking

We wish to quantify the impact of the orchestrator intermediary in typical Kubernetes operations. For this reason, we devise a benchmark that consists of creating a deployment with a large number of simple pods and measure the time between the first pod transitioning to "Initialized" and the last pod transitioning to "Ready". Thanks to the JSON output of kubectl, these measurements can be easily automated, and we report here the bash commands in the interest of replicability:

```
# Time of first transition to Initialized
k get pod -n hello-world -o json | jq '.items | .[].status.conditions |
    .[] | select(.type == "Initialized") | .lastTransitionTime' | sort |
    head -n 1

# Time of last transition to Ready
k get pod -n hello-world -o json | jq '.items | .[].status.conditions |
    .[] | select(.type == "Ready") | .lastTransitionTime' | sort | tail -
    n 1
```

Regretfully, Kubernetes stores timestamps with only seconds precision, so some quantization noise is present in our measurements.

We time this procedure for an increasingly large number N of pods, always starting from rest. Then, we compare the measurements between creating this deployment on the same cluster, on a 1-to-1 peering and going via the orchestrator.

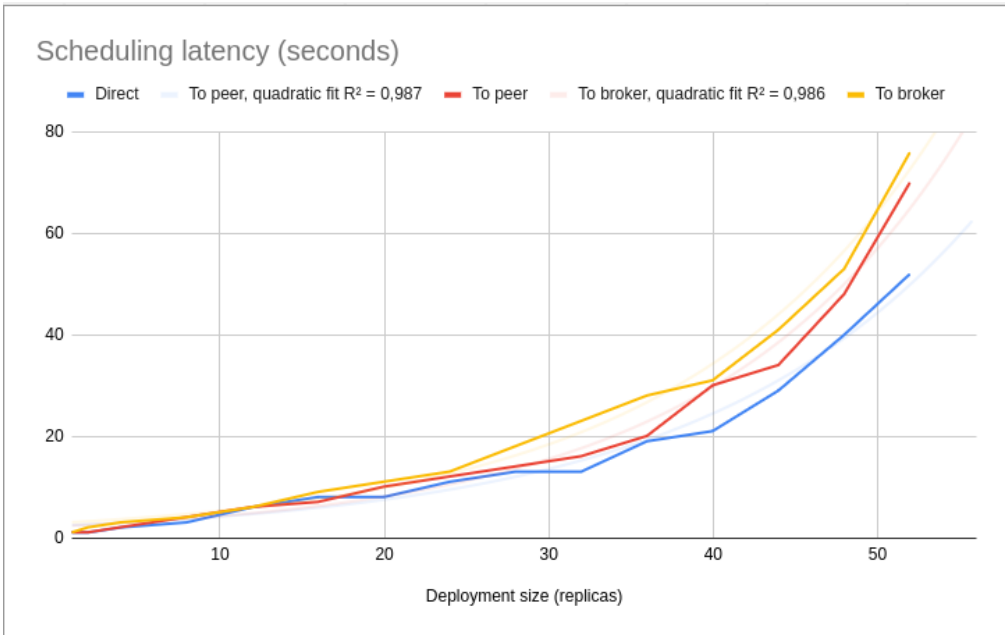

Figure 6.1: Scheduling latency for the orchestrator.

We observe that all three measurements - direct, to an immediate peer and to the orchestrator broker - fit rather well a quadratic curve. We note, however, that because times are measured with a second-level resolution, some "quantization error" must be taken into account.

Our initial hypothesis, that the scheduling latency measurably increases when the orchestrator is in use, proves correct. However, while we expected a latency

overhead increasing linearly with the number of "scheduling hops", we observe that the relative difference in scheduling latency varies a lot. We posit that this effect is partly due to the coarse resolution in the dataset, as a visual analysis of Figure 6.1 shows some small discontinuities. We indicate a better statistical analysis of this data as a possible line of development for further work.

6.2 Aggregator benchmarking

Recall that a defining feature of the aggregator is that it introduces an additional hop on the data place. It makes sense to measure the impact of this additional hop on the network latency of offloaded applications. Naturally, the aggregator also introduces a bandwidth bottleneck at the broker, but this characteristic is harder to quantify.

Let us deploy a simple HTTP server that performs minimal work, to reduce the impact of CPU load on the measurements. For this benchmark, we used the Docker image `registry.k8s.io/e2e-test-images/agnhost:2.39`, which responds with today's date. We used `ab`, the Apache HTTP server benchmarking tool [17], to take measurements.

We created and offloaded a deployment with one such pod, exposed the pod with a service, and used the following `ab` command:

```
1 ab -n 1000 http://<cluster-ip>:8080/
```

As a baseline of raw network performance on the Crownlabs platform, we observe an inter-cluster ICMP latency (i.e. a ping that takes place outside of the VPN link) of 0.436 ms, $\sigma = 0.111$ ms, with a maximum latency of 0.930 ms.

Source	Average latency	p99 (99th percentile)
Provider	0.315 ms	1. ms
Aggregator	1.083 ms	3. ms
Consumer	2.021 ms	6. ms

Thanks to a much finer resolution and a larger number of measurements, we observe much more consistent and linear results. The difference in network latency observed on the consumer-aggregator link is almost exactly the same as the difference seen on the aggregator-provider link, confirming our initial hypothesis that this latency is additive with each additional hop. Furthermore, we observe that on a fast link such as the one between Crownlabs clusters the increase in network latency is minimal, with an average latency of 2 ms. This evidences that the aggregator topology strongly depends on the network characteristics, and broker operators

that want to optimize their response times need to work on the raw network speed between their cluster and the providers'.

Source	Average latency	p99 (99th percentile)
Provider	0.315 ms	1. ms
Orchestrator	1.071 ms	3. ms
Consumer	1.030 ms	3. ms

With a simple test we can also confirm our value proposition that the orchestrator guarantees the same data-plane performance as a direct peering. Except for small variations that we attribute to random statistical variations and minor fluctuations in CPU usage, the network performance seen by an orchestrator's consumer is the same as other direct peerings.

6.3 Conclusions

This work proposes multiple solutions to centralize the sharing of information in a topology of Liqo clusters. With minimal changes to the Liqo core, this work builds on top of solid Liqo primitives, allowing for greater robustness and simpler maintenance of the codebase. The corresponding architectures necessarily introduce overheads on the control and the data plane, which we characterize here. Our quantitative analysis shows that this overhead appears to scale linearly with the number of steps. While the increase in network latency is well manageable in the presence of fast network links, we note that the increase in scheduling latency imposes a limit on the maximum number of resources deployed before set timeouts are reached.

6.3.1 Future developments

The brokering mechanisms presented in this thesis are meant to be proofs of concept with substantial reuse of the Liqo core code but minimal changes to it, in order to keep core features as generic as possible. As a result, these implementations are functional but not fully mature, and many implementation aspects can be improved on to achieve better performance through lower overhead.

The larger overhead of the two at this moment is the orchestration overhead, which must be profiled and can certainly be improved through a more clever replication logic, perhaps exploiting an already existing peering to the final cluster. The aggregator, while exhibiting a good latency response, does not scale well to the extent that it requires a dense (but not necessarily full) mesh of peerings between consumers and providers on a federation. A more clever solution could make use, for example, of a VPN aggregator to maximize connection reuse and minimize

network chatter. Generally speaking, the effects of scaling to multiple clusters, especially in the presence of complex brokering policies.

Finally, as discussed in chapter 4, this thesis work does not cover complex brokering policies. This could be an interesting future development with both a strong theoretical side and a practical one, with a lot of overlap on topics like scheduling theory. On the practical side, the topic of monitoring and certifying metrics used for scheduling is also important to make these brokering solutions production-ready.

In conclusion, this thesis as laying the functional groundwork for resource brokering solutions on Liqo. We envisage future work on this topic as looking closer at the interaction with Liqo to reduce the performance overheads and maximize scalability, as well as expanding the implementations' features in cooperation with academic and commercial Liqo stakeholders.

Bibliography

[1] Alessandro Cannarella. «Multi-Tenant federated approach to resources brokering between Kubernetes clusters». In: (cit. on pp. 1, 39).

[2] *Kubernetes official documentation*. URL: `https://kubernetes.io/docs/home/` (cit. on pp. 4, 11, 13, 16, 17).

[3] *Virtual-kubelet git repository*. URL: `https://github.com/virtual-kubelet/virtual-kubelet` (cit. on pp. 4, 16, 17).

[4] *Kubebuilder git repository*. URL: `https://github.com/kubernetes-sigs/kubebuilder` (cit. on pp. 4, 17, 18).

[5] 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).

[6] 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).

[7] Ferenc Hámori. *The History of Kubernetes on a Timeline*. June 2018. URL: `https://blog.risingstack.com/the-history-of-kubernetes/` (cit. on p. 5).

[8] Jeff Barr. *Amazon EKS – Now Generally Available*. June 2018. URL: `https://aws.amazon.com/blogs/aws/amazon-eks-now-generally-available/` (cit. on p. 5).

[9] Brendan Burns. *Azure Kubernetes Service (AKS) GA – New regions, more features, increased productivity*. June 2018. URL: `https://azure.microsoft.com/en-us/blog/azure-kubernetes-service-aks-ga-new-regions-new-features-new-productivity/` (cit. on p. 5).

[10] *GKE release notes*. URL: `https://cloud.google.com/kubernetes-engine/docs/release-notes` (cit. on p. 5).

[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. 5).

[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. 5).

[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. 7).

[14] Diego Ongaro and John Ousterhout. «In search of an understandable consensus algorithm». In: *2014 {USENIX} Annual Technical Conference*. 2014, pp. 305–319 (cit. on p. 8).

[15] *Kubernetes API official documentation*. URL: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/ (cit. on p. 11).

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

[17] The Apache software foundation. *Apache HTTP server benchmarking tool*. URL: https://httpd.apache.org/docs/2.4/programs/ab.html (cit. on p. 63).