POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

A Multi-Tenant federated approach to resources brokering between kubernetes clusters

Supervisors

Prof. Fulvio RISSO

Dott. Marco IORIO

Candidate

Alessandro CANNARELLA

Academic year 2021-2022

Summary

Cloud computing has been a key technology in the last two decades, enabling the digital transformation that has shaped the current technology landscape. Within this trend, Kubernetes is currently the most prominent solution, promoting a model where applications are split into many loosely coupled components, each packaged as a Docker container and configured as a microservice. This paradigm effectively decouples the infrastructure - which can be scaled up or down on demand - from the application, which is then said to be cloud-native.

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. Liqo is an open source project launched at Politecnico di Torino that enables liquid computing on top of Kubernetes: with Liqo, Kubernetes clusters can join each other in a peer-to-peer fashion to seamlessly create a larger network, with each cluster still retaining full control over its resources.

The goal of this thesis is to create a Multi-Tenant Kubernetes clusters federation, implementing a solution that also fits with the Gaia-X vision of federated cloud.The thesis aims to research and define brokering models for the Liqo ecosystem: service and resource brokers are an important player in a peer-to-peer topology, establishing trust, improving scalability and facilitating connections between providers and consumers. We propose three alternative models, among which we then further design and implement the Catalog approach, which best fits our use-case and allows to advertise and discover offers by different providers.

The Catalog can be queried by any authenticated customer, and is also integrated with Liqo, through a Catalog Connector. As far as this last one, it is implemented as a Web server, which exposes a REST API and a WebSocket interface, it is managed by a graphical UI, where it is possible to create, update, delete and join offers, as well as to start a Liqo peering connection based on a stipulated contract. Part of the work has been integrated into the Liqo project, and is currently fully functional in the latest release, instead the Catalog Broker, the Catalog Connector and its UI have been developed as a Proof-of-Concept, and are available as a separate project on GitHub.

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

B2C
: Business to Customer

B2B

Business to Business

GDPR

General Data Protection Regulation

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

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 the Catalog broker that deals with Liqo. Our work integrates feedback from TOP-IX, a commercial entity that seeks to offer brokering and networking services in the liquid computing landscape.

One of the most relevant project that it is working on is Structura-X: this is a lighthouse project of Gaia-X, the European cloud initiative. The goal of Structura-X is to create a federated infrastructure that will include both cloud providers and IXPs, capable of competing with the hyperscalers. The first demo of the project studied, developed and implemented in this thesis has been presented in Paris on November during the Gaia-X Summit 2022.

1.2.1 Thesis Structure

- **Chapter 1: Introduction** - This chapter introduces the reader to the topic of the thesis, and to the problem that we are trying to solve. It also introduces the reader to the structure of the thesis.
- **Chapter 2: Kubernetes** - This chapter introduces to the Kubernetes technology, and to the main concepts that are used in the thesis.
- **Chapter 3: Liqo** - This chapter introduces to the Liqo project, his behaviour and his architecture.
- **Chapter 4: Broker Models** - This chapter introduces to the different broker models studied in this thesis and a comparison between them. Exploring also use cases and applications.
- **Chapter 5: Catalog** - This chapter goes in depth in the Catalog model, the different components that compose it, its behaviour and its architecture.
- **Chapter 6: Implementation** - This chapter describes the implementation solution that has been chosen to implement the proposed architecture.
- **Chapter 7: Evaluation** - This chapter shows the results of the performance benchmarks that have been performed to evaluate the proposed implementation.
- **Chapter 8: Conclusions** - This chapter makes a final summary of the thesis work and of the results obtained, and it also introduces the future work that would be done.

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 [1].

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

2.1 Kubernetes: a bit of history

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

In 2013 Google announced **Omega** [5], a flexible and scalable scheduler for large compute clusters. Omega provided a "parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, in order to achieve both implementation extensibility and performance scalability" [5].

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) [6]. Since then, Kubernetes has significantly grown, achieving the CNCF graduated status and being adopted by nearly every big company and cloud provider: AWS [7], Azure [8] and Google Cloud [9] offer managed Kubernetes clusters. Nowadays, it has become the de facto standard for container orchestration [10, 11].

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.

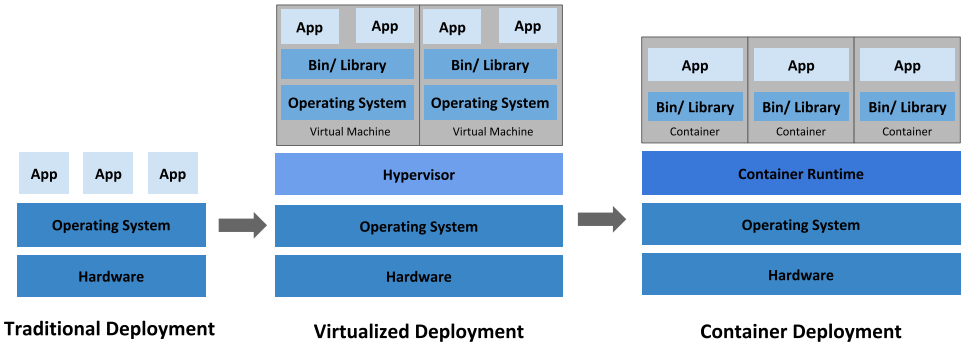

Figure 2.1: Evolution in applications deployment.

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.

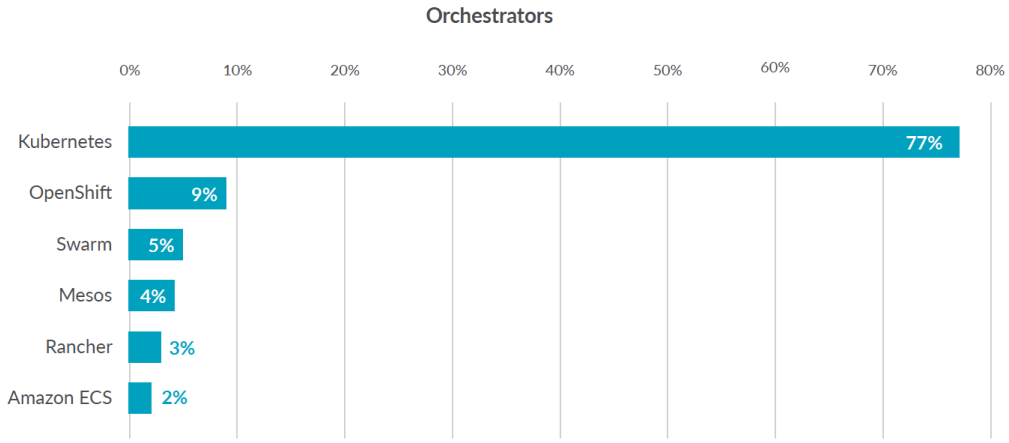

Figure 2.2: Container orchestrators use. [12]

- **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.

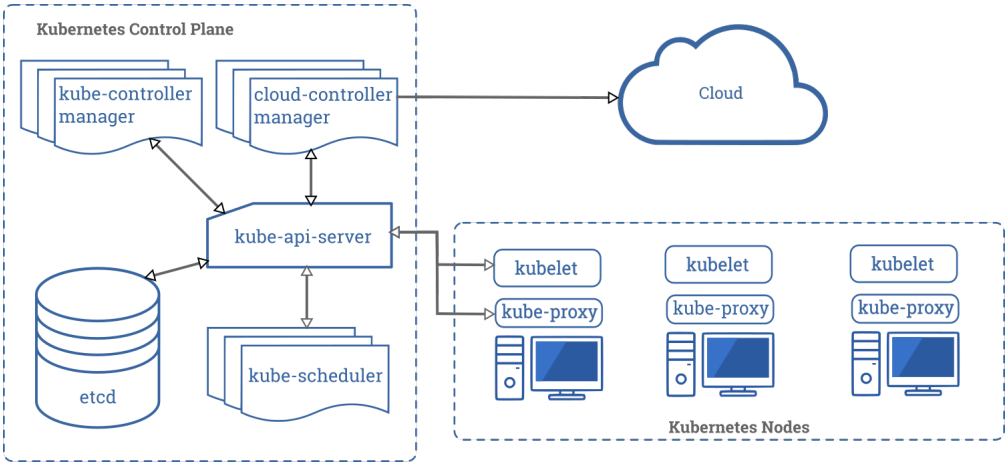

Figure 2.3: Kubernetes architecture.

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 [13], 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.

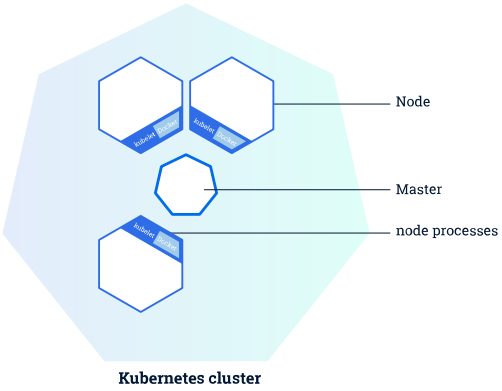

Figure 2.4: Kubernetes master and worker nodes. [1].

2.5 Kubernetes objects

Kubernetes defines several types of objects, which constitute its building blocks. A K8s resource object typically contains the following fields [14]:

- `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.

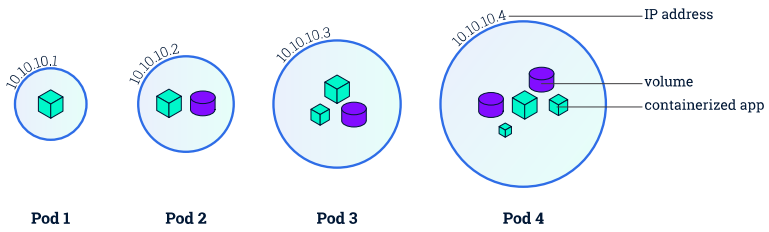

Figure 2.5: Kubernetes pods. [1]

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:
```

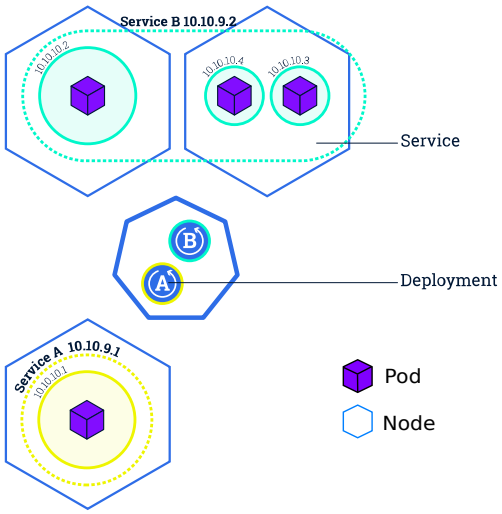

Figure 2.6: Kubernetes Services. [1]

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

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.

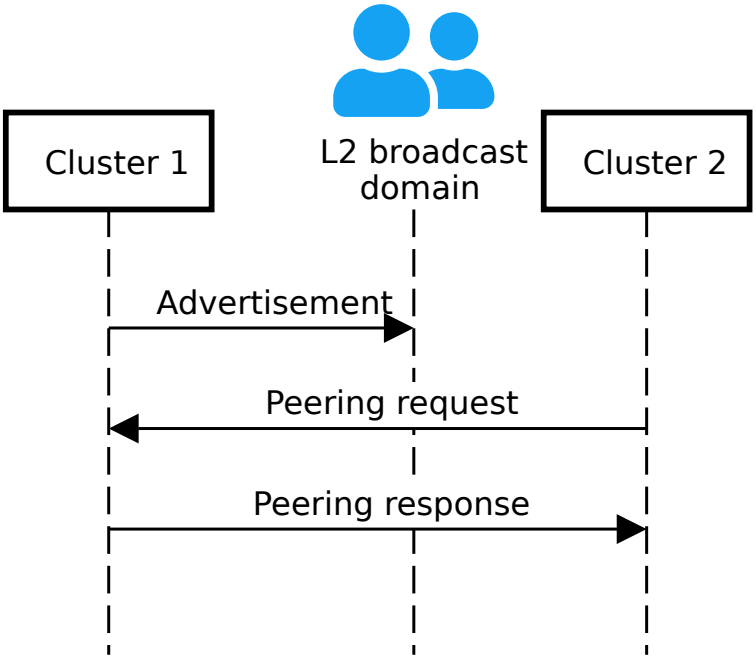

Figure 3.1: Discovery over LAN.

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.

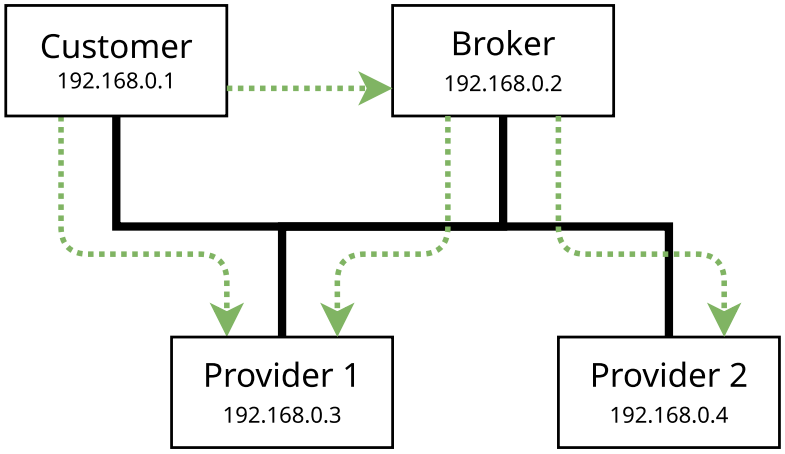

Figure 3.2: A complex peering topology over a LAN. Black: the L2 medium, green: peerings.

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.

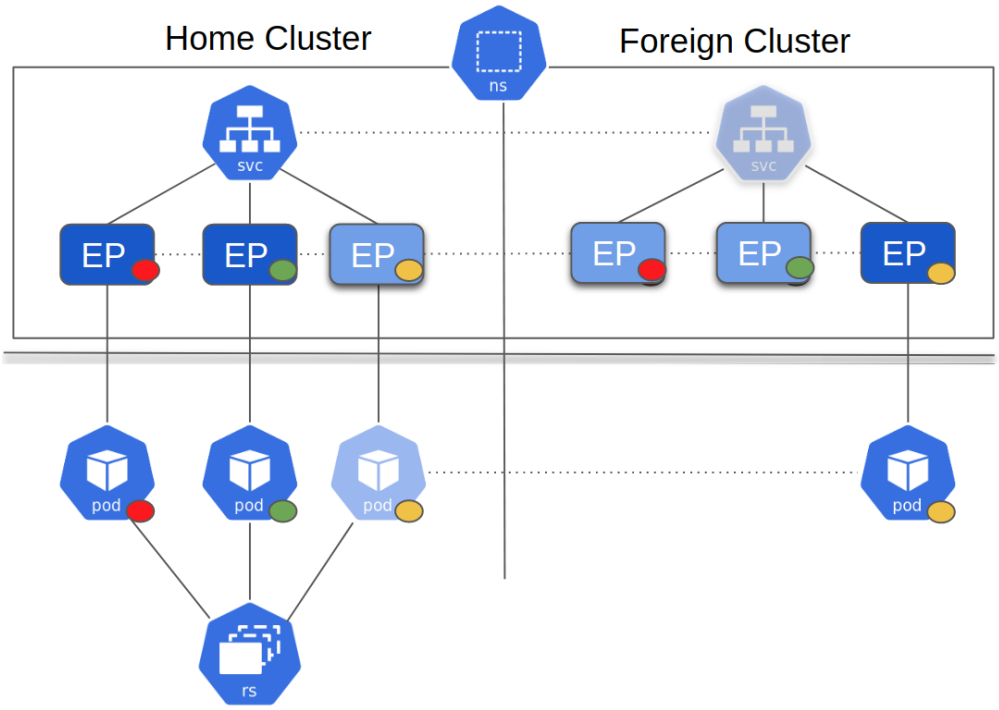

Figure 3.3: Reflection of pods and EndpointSlices.

3.3 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 [2]. 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 [2] says that "providers must provide the following functionality to be considered a supported integration with Virtual Kubelet:

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

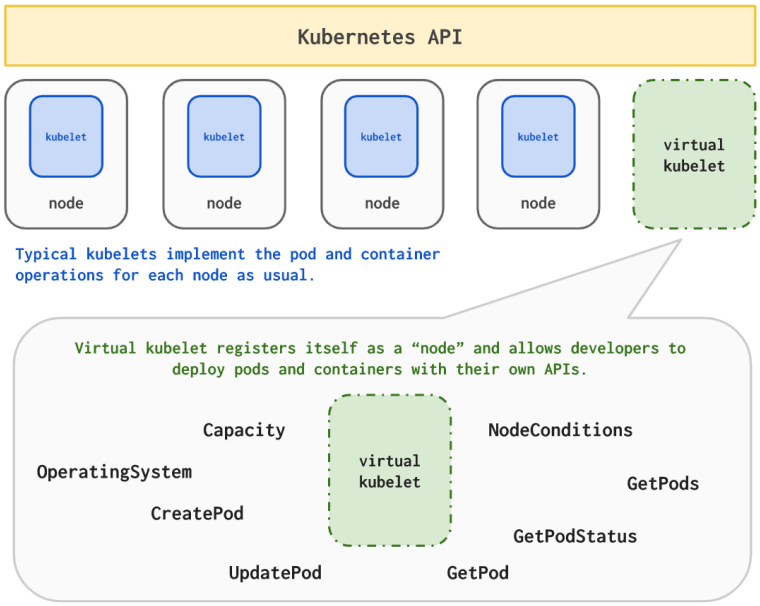

Figure 3.4: Virtual-Kubelet concept. [2]

Chapter 4

Broker models

This chapter aims to present the main concepts and the models of resource brokering on Kubernetes, giving a more punctual high-level definition of the roles and functions involved in the brokering process. We do so by firstly defining the concepts of Multi-Tenant and Multi-Cloud which cover a continuous thread throughout the thesis. Then we go through some user stories and use cases for resource brokering in cloud services federations, focusing on some real scenarios, proceeding to derive an understanding of possible added-value services, and finally devise Liqo-based solutions to offer these services, dwelling in particular on the "Catalog" model which will be studied deeply in the following chapters.

4.1 Multi-Tenancy and Multi-Cloud

To understand the reasons for some architectural choices and the problems underlying the proposed solutions, it is first of all necessary to deepen the concepts of Multi-Tenancy and Multi-Cloud that play a fundamental role in the entire research path. In the field of computer science and in particular of software applications, a Multi-tenant architecture represents a single software instance that serves multiple and heterogeneous customers. Each customer is called tenant, who has the possibility to act in a restricted functional domain.

This type of architecture is able to work because each tenant is physically integrated together with all the others, but is logically separated from them. This concept has been adopted above all in Cloud adoption and is used most with cloud computing, in particular in order to allow each tenant's data to be separated from each other. This has therefore given the possibility, for example, to use the same server to host multiple users, and ideally provide them a secure space to store data.

Of course, this approach has brought several advantages in terms of costs and scalability, but it has also introduced some disadvantages. In particular, it is clear

to imagine how the general complexity of development has grown considerably, especially from a security and resource management point of view.

In the other hand, the concept of Multi-Cloud, sometimes also used in the sense of Hybrid Cloud, is the common background. It simply means a company, or more generally an enterprise, which during the deployment process makes use of various heterogeneous cloud providers, including On-Premise, instead of applying a *one vendor business* approach. This allows the company not only to be able to differentiate the services offered and therefore improve their reliability and redundancy, but also reduces vendor lock-in, thus impacting the competitiveness of the markets and potentially also costs.

If we imagine all this in a contemporary context, in which the control and sovereignty of data plays a fundamental role both at a geopolitical level and at a strategic level in the competition between companies, it makes it easy to understand how this principle is becoming more and more fundamental in the approach to the modern Cloud

4.2 User stories

The need for resource brokers arises from creating large-scale Kubernetes clusters for the purpose of sharing computing resources. Brokers address some shortcomings of the peer-to-peer, horizontal model in scaling to large numbers of users: as the number of providers increases, it becomes increasingly harder to have full visibility over the resources and organizations present on the network - *vice versa*, each provider may no longer place the same level of trust in consumers as in a smaller network with well-known participants. Furthermore, even if each party has full visibility over the global resources, an "overseeing organization" may be able to optimize the distribution of workloads using proprietary metrics and the knowledge of each customer's needs.

From this consideration we can contextualize the broker as a commercial operator in the niche of enabling multi-cloud environments, that provides added-value services to large, pre-existing federations. As such, the broker can additionally be understood in the GAIA-X framework as a participant that fulfills Federation Services: we will see that our thesis work addresses existing GAIA-X Federation Services like Access Management, but also expands the notion of Federated Catalogues to include IaaS offerings.

As part of thesis I worked closely 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.

One of the most relevant project that we worked on is Structura-X: this project is born as lighthouse project of Gaia-X, the European cloud initiative. The goal of Structura-X is to create a federated infrastructure that will include both cloud providers and IXPs, enabling a trustful environment for the exchange of data and services. The idea is to create an ecosystem capable of competing with the hyperscalers, both in terms of networking and in terms of computing layer. The first demo of the project has been presented in Paris on November during the Gaia-X Summit 2022. In this occasion, we presented the first version of the Structura-X architecture, and also of the project studied, developed and implemented in this thesis.

4.2.1 Generic scenarios

Our research shows that there are a number of different scenarios that address different needs. We distinguish between *B2C* and *B2B*:

B2C

1. **Scenario 1 - Discovery**: there are a multitude of cloud operators, each offering different features and having their own clusters. The "cloud market" is a sprawl with little discoverability integrated into the network.
   Customer story: I want to discover what providers are available, so that I can choose the one that best fits my needs.
   Provider story: I want to advertise my resources to potential customers in a way that is integrated with the federation.

2. **Scenario 2 - Metrics and certification**: there are many more cloud operators, to the point that it is infeasible or undesirable for a customer to choose the optimal one. Furthermore, some metrics of interest (eg. latency) may not be available to the user, or may be self-certified.
   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 certify metrics like latency and uptime, so that I 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 - Data sovereignty**: there are a number of organizations that want to enable access and computation over their data while retaining control over it.
   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.

B2B

1. **Scenario 1 - Competitiveness**: there are a number of small cloud providers that are individually not competitive with larger players.
   User story: I want to access a large amount of computing resources.
   Provider 1 story: I want to join forces with other providers to enable a larger, aggregated commercial offering.
   Provider 2 story: I want to increase my visibility in the cloud market to compete with bigger players and engage new customers.

2. **Scenario 2 - Service offering extension**: there are a number of cloud providers that want to extends their service offering on demand without investing to create a suitable proprietary infrastructure and/or without losing potential customers.
   User story: I need a set of specific service not entirely offered by my cloud provider.
   Provider 1 story: I want to offer to my customers certain services through my infrastructure by engaging them on demand from other vendors.
   Provider 2 story: I want to increase my business providing directly to other vendors some service offers, exploiting infrastructural resources that would be wasted.

3. **Scenario 3 - GDPR Data compliance**: there is Data that, according with GDPR, need to be stored in a specific geographical location compliant with the rule, or that, for some reasons, cannot be taken outside some political or geographical boundaries.
   User story: I want to store my data in a place compliant with certain rules and simultaneously access it from other services hosted in a separate infrastructure of the same cloud provider.
   Provider 1 story: I want to offer to my customers a data space service that respects their needs joining an IaaS offer provided by another vendor, thus guaranteeing them a continuity of service.
   Provider 2 story: I want to increase my business providing to other vendors some specific IaaS GDPR compliant offers.

4.2.2 Real scenarios

Gaia-X

Gaia-X is a project reportedly working on the development of a federation of data infrastructure and service providers for Europe to ensure European digital sovereignty. It seeks to create a proposal for the next generation of data infrastructure for Europe, as well as foster the digital sovereignty of European cloud service users. It is reportedly based on European values of transparency, openness, data protection, and security. To accomplish this it hopes to specify common requirements for a European data infrastructure and develop a reference implementation.

Structura-X

A lighthouse project for European cloud infrastructure endeavours to enable existing Cloud Service and Infrastructure Providers (CSP) data and infrastructure services to be Gaia-X certifiable. The goal is to create an ecosystem of independent CSPs, orchestrated by a shared layer of federation certification and labelling services based on Distributed Ledger Technology (DLT). The first milestone on the Structure-X project is to provide and certify Infrastructure Service Offerings against the definitions of the Gaia-X Trust Framework to provide Transparency and Trust to all participants

4.3 Models and use cases

After analyzing the possible scenarios, it is now clear how the existence of a third entity, the *broker*, is important and necessary. It acts more or less as a centralized binder between the various subjects taken into consideration. From the user stories we can therefore develop, based on the role and the functionalities they will cover, at least three possible brokering solutions, which are mainly distinguished by the wideness of control they have in the management of the relationships between Providers and/or Customers.

These correspond to three different typologies and roles that are mostly independent with one another, which need to be developed into three software components:

1. **The Catalog:** this is represented by an endpoint that customers and providers can browse and query. The main role is that of **advertisement** and **discovery**, both from the point of view of the individual subjects existing in the federation and of the services offered and made available within the same. The goal is therefore to rely on broker to be informed of what clusters are on the network, what are their features and resources, and possibly other information

(eg. a trusted estimate of their uptime or latency). Thanks to this broker all the subjects can join available offers and establish new interconnections on demand. Each interconnection is completely independent from the broker, it is based on a *peer-to-peer model*, thus keeping the overall ecosystem completely decentralized by eliminating the possibility that the broker may somehow be a single point of failure

2. **The Orchestrator:** an active component that is in charge only of scheduling computing resources according to defined policies3. This implies a well defined splitting between control plane, up to the orchestrator, with data plane, demanded to the involved customers. So, 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.

3. **The Aggregator:** Unlike the previous two, this broker is completely opaque, acting as a single virtual cluster, in charge of totally managing control and data plane of the clusters below him. 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. We can identify this broker as a middleman that presents a unified view of resources.

4.4 Catalog

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 to accomplish these tasks, and how they were extended. We note that much of the functionality in the "peer-to-peer version" of Liqo can be reused to a large extent, at least for a proof of concept. The only parts that require substantial changes are the resource enforcement for the management of multi-tenant peerings and the customization of the peering resource reservation in such a way as to bypass the standard process in which Liqo calculates and allocates resources.

At the functional level, a catalog needs to receive peering credentials and the commercial offers of each provider cluster, store and aggregate them, and send the full list to each one wants to browse the catalog when a query is issued.

The aspect of peering credentials is arguably the simpler. As mentioned in section

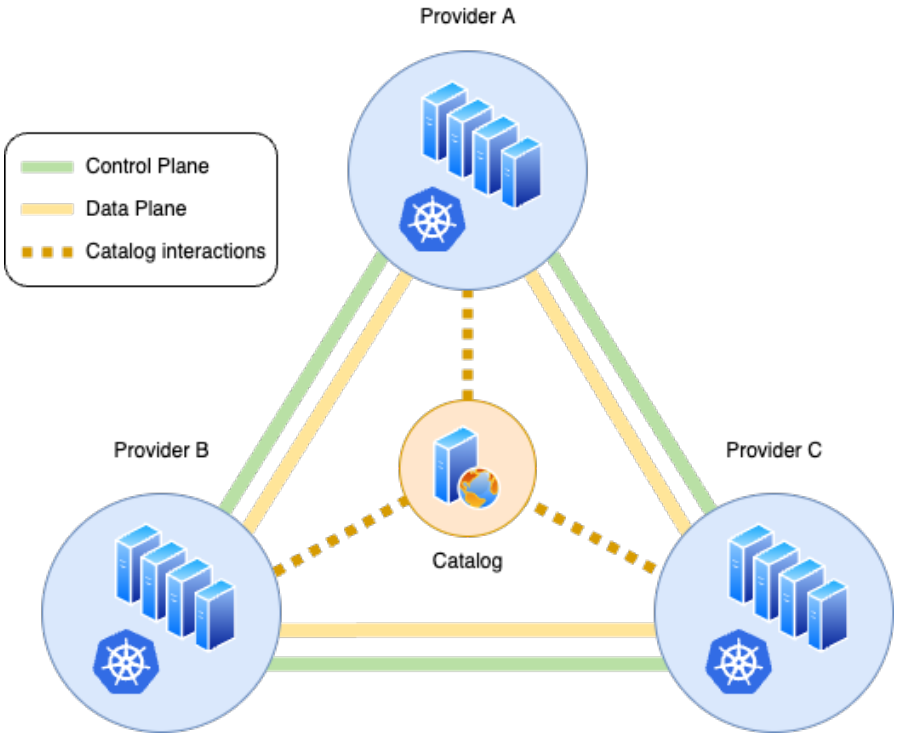

Figure 4.1: A diagram that illustrates a Catalog model example.

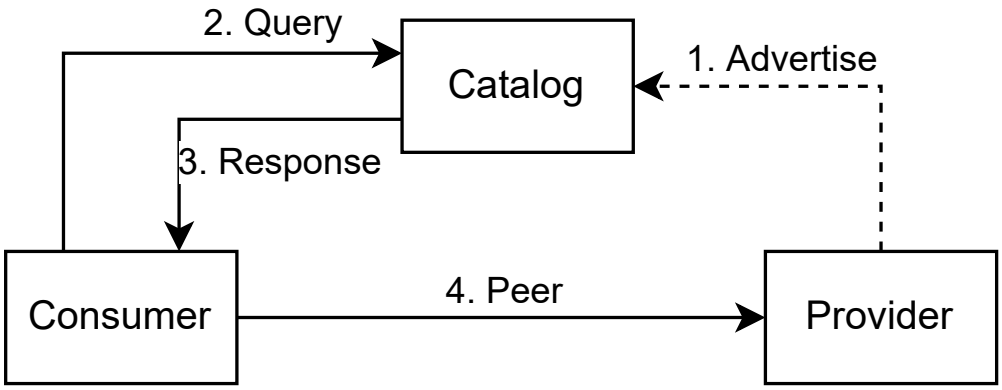

Figure 4.2: A conceptual model of how a catalog can be used.

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 provider/customer can run Liqo peering command to establish the connection, and Liqo will proceed to do the rest.

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 or in certified metrics for uptime/latency/etc. In our thesis we will concentrate only on the first case, trying to develop a IaaS environment.

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, for a better understanding we demand the reader to go back to the right section 3.2.2 where we explained this concepts. (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.

We will go in depth with the description of this model in the next chapters, where we also explain the evolution of this model in a distributed fashion and of course we describe how all the components interact each other and elaborates and/or exchange data.

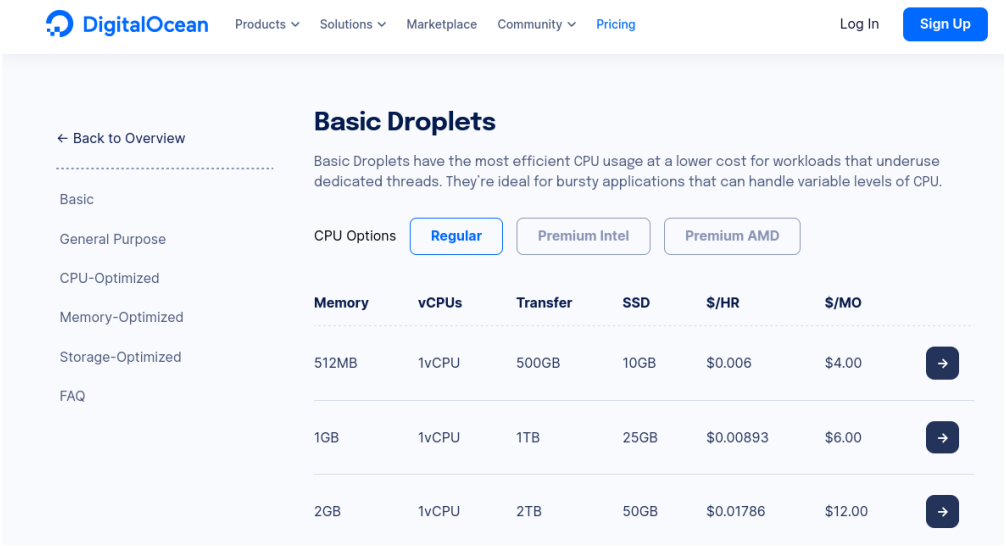

Figure 4.3: Part of the DigitalOcean catalog: note the list of offers on the left and the list of plans on the right.

4.5 Orchestrator

Recall the user story described at the beginning of this chapter (B2C - Scenario 2). 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 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.

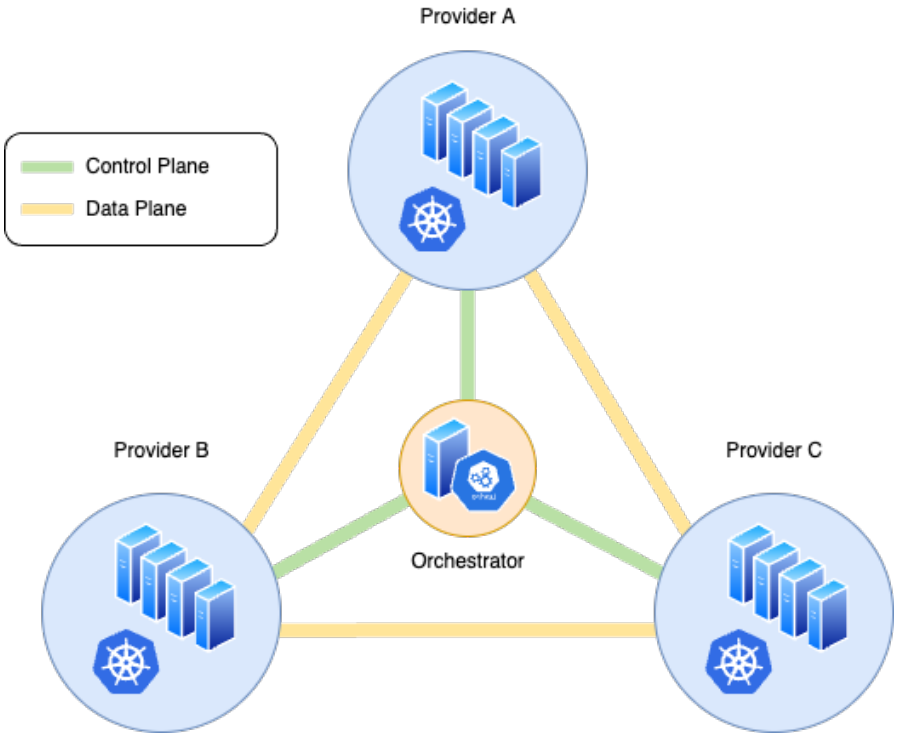

Figure 4.4: A diagram that illustrates an Orchestrator model example.

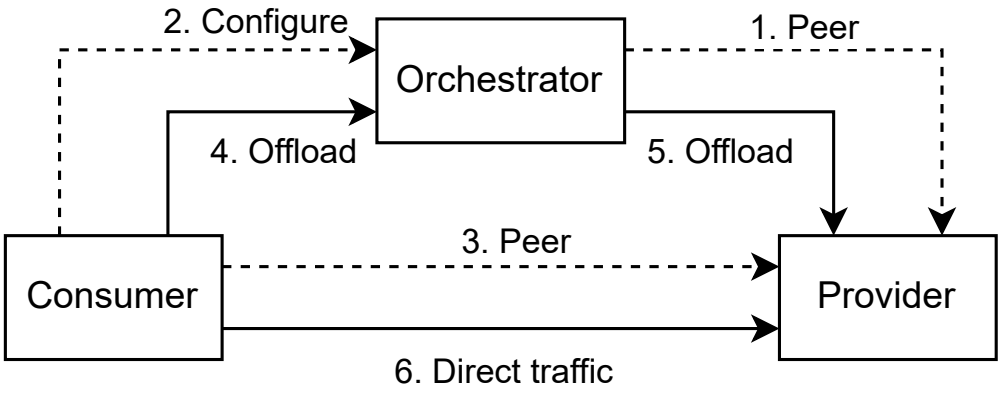

Figure 4.5: A conceptual model of how an orchestrator can be used.

At the implementation level, the *Orchestrator* is able to enforce arbitrary policies if we establish that all peerings must go through it. At the same time, it is not entirely "opaque": 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 would be ideal if the data plane

was direct between the customer and the provider, while the control plane retains the *Orchestrator* as an intermediary: if on the other hand the data plane had to pass through the broker, the latency would suffer from the extra hop, and the broker would have to allocate sufficient bandwidth for serving all of its customers. In essence, 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.

4.6 Aggregator

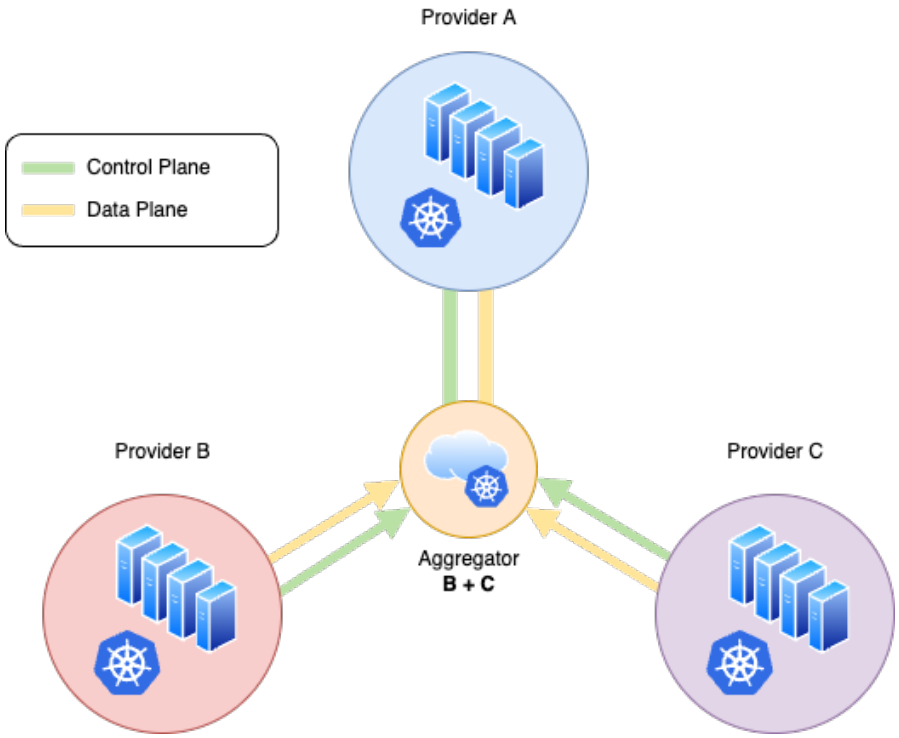

Figure 4.6: A diagram that illustrates an Aggregator model example.

At the beginning of this chapter we have explored some user stories that can be used to motivate the need for a brokering solution. In this section, in particular, we will focus on the use case (B2B - Scenario 1) of a cloud provider that wants to be competitive in the market by joining forces with other cloud providers (in the Figure 4.6 Provider B and C), to be able to offer a better service to its customers (Provider A) or just to increase its market share and visibility.

In this scenario, the brokering model that best suits the needs is the *Aggregator* model, because it allows the cloud provider to federate its resources with other

cloud providers, and to offer them in the market as a single entity. In this way, the cloud provider can offer a better service to its customers, maybe with a better price, performance, or with a better service diversification.

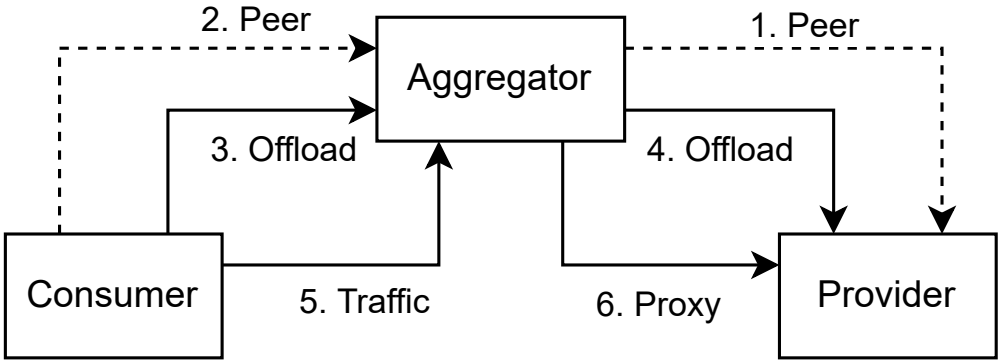

Figure 4.7: A conceptual model of how an aggregator can be used.

Unlike the *Orchestrator*, in this case both the "Control plane" and the "Data plane" of the cloud providers are proxied and controlled by the *Aggregator*, which is in charge of aggregating their resources, and of distributing the workloads to the ones that are able to handle them based on some metrics and policies. On top of that it exposes to the customers, and so to the market in an opaque way, the aggregated resources of the providers that are federated with it. The final result is that no one knows and matters about the exposition of its services and resources to the market, and how the workloads are distributed among the providers federated with the *Aggregator*. They just should define their own policies, if they wants, or let the *Aggregator* decide where based on metrics.

This model implies of course an high responsibility for the *Aggregator*, because it becomes the single point of management of the resources of the providers that are federated with it. Moreover it has to be sure that the resources of the providers are not overused, and that the workloads are distributed in a fair way among the parties. To do that, the *Aggregator* has to be able to monitor the resources and metrics of the overall environment.

From a security and reliability point of view, it should appear as a single point of failure, in fact if the *Aggregator* fails, all the providers that are federated with it will not be reachable anymore. To avoid this, a complex system of redundancy and failover should be implemented, and the *Aggregator* should be able to recover from failures in a fast way. Moreover, the providers should be able to bypass the *Aggregator* in case of failure, or provide backup peerings with other instances of the *Aggregator*.

Chapter 5

Catalog

5.1 Overview

Until now we have focused on studying the different models of brokering and the strategies to implement them with the purpose of designing and enabling an efficient and scalable architecture that fits the requirements of a Kubernetes clusters federation.

Our choice fell on the Catalog model, leaving out the other two models, because it is the most suitable for our use case, and it is also the most scalable and efficient. Moreover, it is the lightest, and the only one that does not centralize the entire infrastructure, which is one of the key requirements for a federation of clusters. This model is also the most flexible, as it allows to implement different strategies to aggregate and collect offers, to implement different policies to manage them and also because it lends itself better to being extended in the future with new features, as for example negotiation logics.

Lastly, it is the most suitable for the Gaia-X vision of federated cloud, where multiple cloud providers through heterogeneous infrastructure can join together to extend their services and resources. We do not forget that in a federation of clusters the primary goal is to avoid a loss of control over the resources, and to maintain the autonomy of each cluster. The Catalog model allows to achieve this goal, so we decided to implement it.
The above mentioned requirements can be summarized in the following points:

- The broker must allow CSPs to advertise their resources and offers to the federation.
- The broker must allow CSPs to discover and join offers from other CSPs.
- The broker must implement a standard interface that IaaS providers can subscribe to, which is an important element for application portability, preventing

vendor lock-in.

- The broker must be able to establish trust, improving scalability and facilitating connections between providers and consumers by aggregating offers and allowing for complex topologies going beyond the current point-to-point model.
- The broker must be able to manage multiple tenants.
- The broker has to be agnostic to the underlying infrastructure.
- The broker has to be agnostic about what happens between the CSPs. In fact data plane and control plane of each peering have not to be on the control of the broker.
- CSPs have to be able to create and manage contracts with other CSPs through the catalog.
- CSPs have to be able to create, update, delete and join offers through the catalog.
- CSPs have to be able to enstablish a Liqo peering connection with other CSPs based on a stipulated contract through the catalog.
- CSPs have to be able to manage the peering and revoke eventually the contract through the catalog.
- CSPs must be able to create a peering connection with an automatic assignment of resources to each one.
- CSPs must be able to continuously monitor the status of the peering connection and the resources assigned to them.
- CSPs must be able to mantain the control of their resources.
- The connections between the broker and the CSP must be secured.
- The connections between CSPs must be secured, authenticated and encrypted, and must be able to support multiple tenants.
- Each connection between CSPs must follow a peer-to-peer model, where each CSP can be both a provider and a consumer.

Analyzing the pros and cons of all models, we have identified the Catalog as the most suitable model for our use case. In fact, its lightweight nature and the fact that it is not a central point of failure, make it a good candidate for a distributed

architecture where many different tenants would like to cooperate in creating a common environment.

Moreover, the Catalog model follows all the criteria listed above, and it is also the most flexible model, since it allows to implement different strategies for the discovery and advertisement of offers. In fact, the Catalog model can be implemented in a centralized way, where all the offers are stored in a single database, or in a distributed way, where each CSP can store its offers in a local database and share them with other CSPs through a synchronization mechanism.

In this thesis we have implemented a distributed Catalog, where each CSP can store its offers and share them with other CSPs through the broker. In the following sections we will describe the Catalog architecture and the different components that compose it.

We will also describe the Broker and its APIs, which are the standard interfaces that IaaS providers could use to interact with him, and the Catalog Connector, which is the component that contains the core logic to interact with the Liqo controller and the Liqo external resource monitor through a gRPC server, and that manages the offers and contracts through a REST API and a WebSocket interface.

Finally, we will describe the Catalog Connector UI, which is a graphical interface that allows cluster admins to interact with the Catalog Connector through a web browser, allowing them to create, update, delete and join offers, creating and managing contracts with other CSPs, and to establish Liqo peering connections based on stipulated contracts.

5.2 Objects: Offers and Contracts

First of all, we have to define the objects that represent the information that the Catalog will store. We have mainly two different types of objects: Offers and Contracts.

An Offer, as already defined in the previous chapters, is an object that is composed by different metadata such as the ID, the Provider name, the creation date, the availability, the description and so on. It is attached by a collection of Plans that can be interpreted as a package of resources that a CSP is willing to share with other CSPs. Each plan can be composed of a single type of resource, or a set of different types of resources, such as CPU, memory, storage, GPU, etc. In fact, the Catalog is not tied to a specific type of resource, and it can be used to share any kind of them. Moreover, a plan can be attached to different metadata, such as the name, the price (defining also the billing strategy understood as billing period and currency), the quantity, and so on. Each offer can be created by a CSP, and it can be updated or deleted by the same. It is also possible to decide if it has to be published on subscribed brokers or only saved locally. This strategy allow

CPS admins to draft them or make them unavailable to other CSPs anytime.

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

```
{
  "offerID": "...",
  "offerName": "Storage-optimized offer",
  "offerType": "storage",
  "description": "We designed this offer for all your storage needs.",
  "created": "timestamp",
  "status": true,
  "plans": [
    {
      "planID": "..."
      "planName": "plan-basic",
      "planCost": "1.0",
      "planCostCurrency": "USD",
      "planCostPeriod": "Monthly",
      "planQuantity": 10,
      "resources": {
        "cpu": "1000m",
        "memory": "2Gi",
        "storage": "100Gi"
      }
    },
    {
      "planID": "..."
      "planName": "plan-medium",
      "planCost": "2.0",
      "planCostCurrency": "USD",
      "planCostPeriod": "Monthly",
      "planQuantity": 10,
      "resources": {
        "cpu": "2000m",
        "memory": "4Gi",
        "storage": "300Gi"
      }
    },
    {
      "planID": "..."
      "planName": "plan-performance",
      "planCost": "5.0",
      "planCostCurrency": "USD",
      "planCostPeriod": "Monthly",
```

```
      "planQuantity": 10,
      "resources": {
        "cpu": "8000m",
        "memory": "32Gi",
        "storage": "1Ti"
      }
    },
  }
}
```

Listing 5.1: An example Offer JSON object

On the other hand a contract is an object that consists of several metadata such as the buyer ID, in this case represented by the Liqo ClusterID, the Provider information, which include all the data needed to start a peering connection with the seller, the creation date, and of course the offer plan configuration which it is linked to, which describes the resources that have to be enforced during the peering.
This configuration is useful to define the resources that the buyer will be assigned and reserved to. On the other side it represents a way for the seller to control that the resource usage is not exceeding the limits defined in the contract.
Here an JSON document as example of a Contract object.

```
{
    "contractID":"0a2e226b-9d54-4b3b-9e65-80c71200e252",
    "buyer-cluster-id":"ab5764bd-5403-4de4-b1e8-9b0f4cb0b798",
    "seller":{
        "clusterPrettyName":"Cloud Service Provider example",
        "clusterContractEndpoint":"http://10.0.0.7:8010",
        "clusterName":"cloudprovider",
        "clusterID":"191a8cba-14e4-4fde-a60a-6995f812e047",
        "endpoint":"https://172.18.0.2:30425",
        "token":"3470cebaeeade3e7b190c064aaf912de08..."
    },
    "offer":{
        "offerID":"191a8cba-14e4-4fde-a60a-6995f812...",
        "offerName":"CPU Offer",
        "offerType":"computational",
        "description":"This id a CPU oriented offer",
        "plans":[
            {
                "planID":"191a8cba-14e4-4fde-a60a-6995f812...",
                "planName":"Basic",
                "planCost":10.0,
```

```
22                "planCostCurrency":"USD",
23                "planCostPeriod":"week",
24                "planQuantity":10,
25                "resources":{
26                    "cpu":"100000m"
27                }
28            }
29        ],
30        "providerPrettyName":"Cloud Service Provider example",
31        "created":1668966474385,
32        "status":true
33    },
34    "planID":"191a8cba-14e4-4fde-a60a-6995f812...",
35    "enabled":true,
36    "created":1668966490
37 }
```

Listing 5.2: An example Contract JSON object

In order to make this possible, an extension of Liqo controller manager has been implemented, which is responsible for the enforcement of the contract. This argument will be discussed in depth in the next sections, moreover, we will see how the Liqo controller manager is able to monitor the resource usage of the buyer and to enforce the contract limits by avoiding the overuse of resources.

5.3 Architecture design: distributed Catalog

The main focus of our studies was to design a solution that satisfies the requirements of a Kubernetes clusters federation and that scales well in a distributed environment. Our purpose was to design a solution that is able to manage multiple tenants and to scale horizontally, in order to support a large number of CSPs. We initially identify two possible implementations path: a full mesh peer-to-peer architecture (Figure 5.1) and a selective on demand architecture (Figure 5.2) that needs to be supported by a Broker, in particular the Catalog model widely studied in the previous chapter. The first solution is the most simple one, but it would not be a good choice for our use case because it is not scalable, both in terms of resource consumption and in terms of discoverability, so it is not efficient due it requires that all clusters should be connected to each other. The second one is more complex to be implemented and managed, but it is more scalable and efficient.
Our challenge converges on the second one: in fact, it allows to interconnect only the clusters that are really interested in sharing resources without know or peer themselves a priori. Moreover it could be possible to allow to add more brokers to the system, increasing the scalability of the environment, giving in this way the

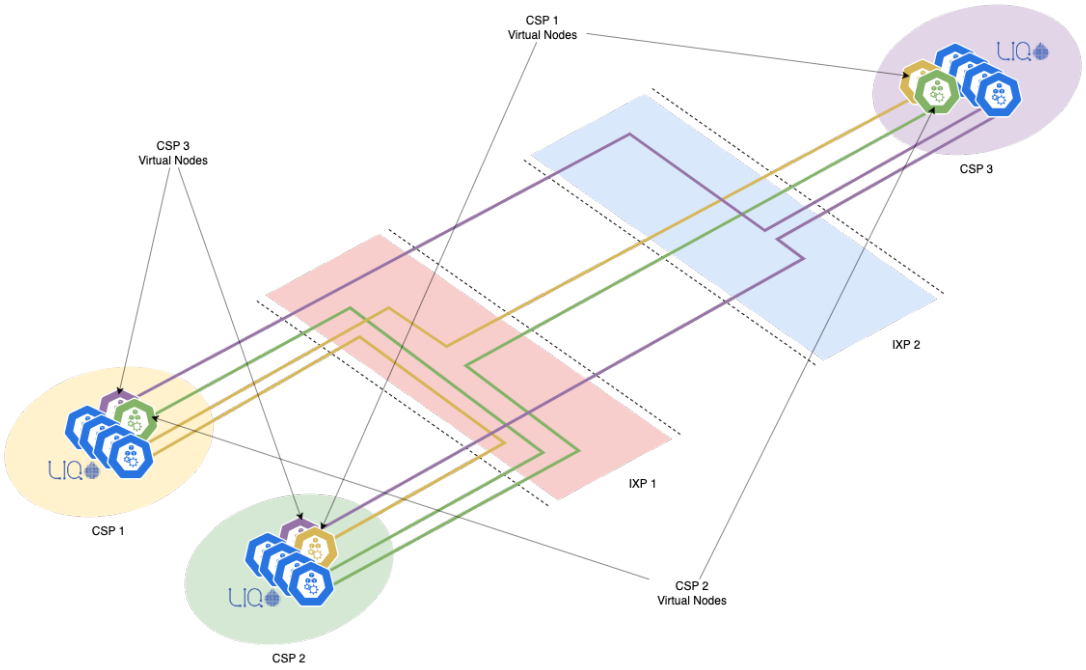

Figure 5.1: An example of a full meshed kubernetes cluster federation with Liqo

possibility to the CSPs to choose the broker that better fits their needs (dataspace topics, geographical constrains, etc.), exposing its offers with different strategies and policies.
We realized that, as explained previously, this model imposes a centralized broker, that however might have added a single point of failure to the system, so the hard work was to design an architecture where the broker maintains as much as possible a lightweight role, avoiding that its failure could affect the whole system.
Obtaining this result involved to split the broker, and so the architecture, in two different elements, that from this moment we will call *Catalog Broker* and *Catalog Connector*. So we mostly moved the complexity of the system to the *Catalog Connector*, that is the component located and run on each CSPs cluster. In this way:

- The **Catalog Broker** becomes a simple component that is responsible for the continuous synchronization and aggregation of the offers between the different CSPs and the authentication and authorization of the CSPs and the requests coming from them.
- The **Catalog Connector** is the component that is responsible for the management of the owned offers and the contracts (sold or purchased), for the communication and synchronization with the Catalog Broker, for the creation

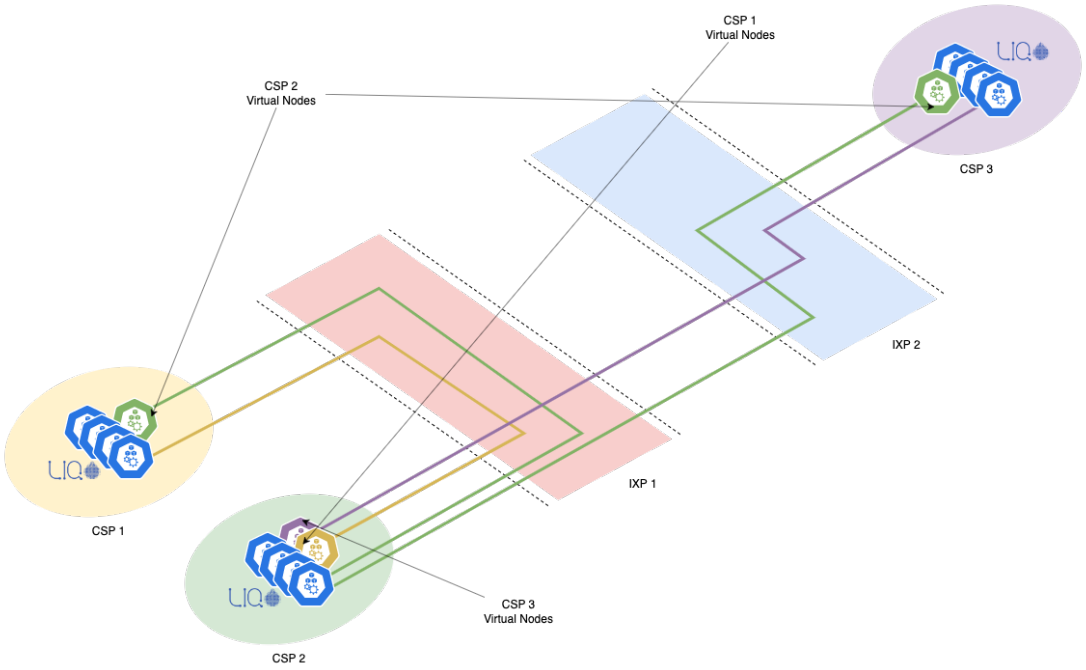

Figure 5.2: An example of a on-demand kubernetes cluster federation with Liqo

of the peering connections with other CSPs, and for the enforcement of the contracts limits when a peering connection request is received.

Basically, we can say that the strategy applied was to move on the edge all the complexity of the system.
In the Figure 5.3 we can definitely see the final architecture of the *Catalog*, that for the reasons explained above we can also define *Distributed Catalog*. On each CSP there is a *Catalog Connector*, usable through a UI[1], in charge of interact internally with the own cluster and externally with the Broker. On each IXP[2], instead, a *Catalog Broker*. Thanks to this choices we also maintained the vanilla behaviour of the Liqo peering, focusing our effort only on adding to it some fundamental missing features as the enforcement of the resources limits and multi-tenancy.

[1]User Interface

[2]Internet Exchange Provider

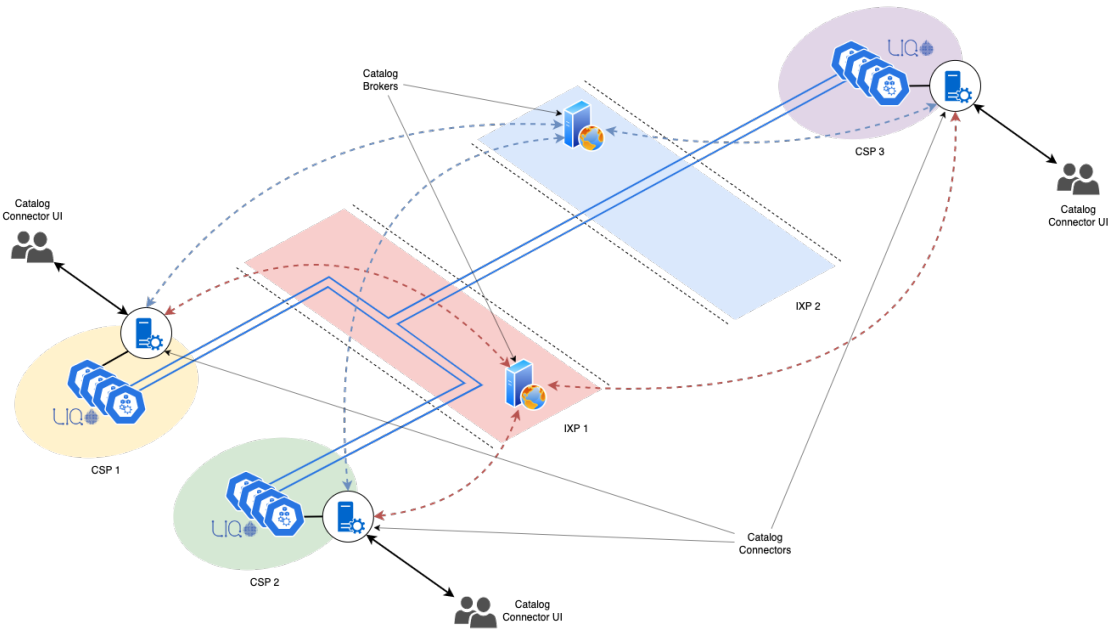

Figure 5.3: Final design of our Kubernetes cluster federation through the distributed Catalog

5.4 Catalog Broker

The Catalog Broker, as previously said, is the component that is responsible for the continuous synchronization and aggregation of the offers between the different CSPs, grouping and broadcasting them in JSON objects that we defined as Catalog (one per each CSP), the authentication of each Provider that wants to be part of the federation, dressing the role of federator, and finally the authorization of the requests coming from this federated CSPs.
It is implemented as a simple Web server that exposes a REST API for CRUD[3] operations on Offers and a WebSocket interface for the continuous broadcasting of messages to update the connected clients (CSPs). This simple design allowed it to be extended, built and deployed everywhere, on a single server, VM or on a Kubernetes cluster, moreover to be flexible, light and compatible with almost all the standard infrastructures. We did this choice because of in our idea of Federation, it is reasonable to think that the IXPs are the most suitable place to host this component, because they are probably already connected to all the CSPs, and they are the most neutral and trusted point of the network, moreover

[3]Create, Read, Update, Delete

their infrastructure is not so computing intensive, so they can host it without any problem.
Careful readers will notice that the Catalog Broker, as we defined it, requires some form of persistence, i.e. a way to store information about the CSP peering credentials and their offers. 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.

Here set of main APIs implemented in our Catalog Broker:

- `POST /authenticate`: receives peering credentials and provides a JWT
- `POST /subscribe`: enstablishes a WebSocket connection to the broker to receive catalog updates
- `GET /catalog`: gets the full catalog of offers offered by all the providers registered on broker
- `POST /offer/<id>`: creates a new offer, or updates it if it exists
- `DELETE /offer/<id>`: deletes the offer identified by an ID

The use of technologies like JSON and WebSocket, that are primarily oriented to the Web, enabled that can easily integrate with a wide range of existing tools and libraries, such as the ones used to implement the Catalog Connector. This is a very important feature, because it allows each contributor to the project to use the tools that are most familiar to them, and to focus on the specific part of the project that they are interested in, allowing the project to be easily extended and maintained in the future by a large community of developers.
Let go deeper into each aspect and related APIs in the next sections.

5.4.1 Authentication

The authentication is the first step that a CSP has to do in order to be part of the federation. For our purpose we decided to use a simple authentication mechanism

based on a JWT[4], that is a standard for representing claims securely between two parties. Of course this solution is not the most secure one, but it is the most simple and flexible one, and it is enough for our PoC[5] use case.
The token is composed of a header, a payload and a signature, and it is signed by the broker using a secret key. The payload contains the information about the CSP, such as the Cluster Name, the Liqo ClusterID, Liqo token and endpoint for the peering. This token is released by the broker to the CSP, and it is used by the Catalog Broker to authenticate all the requests coming from the CSP, this represents a security layer which prevents unauthorized users from interacting with the broker. To obtain the token, the CSP has to send a POST request to the `/authenticate` endpoint of the Catalog Broker, with the following payload:

```
{
    "clusterName": "LiqoClusterName",
    "clusterID": "LiqoClusterID",
    "endpoint":"http://123.456.78.90:35426",
    "token":"LiqoAuthToken"
}
```

The Catalog Broker will check if the CSP is already registered, and if it is not, it will generate a new token and it will store the CSP Liqo information in the database. Then it will send the token to the CSP, that will use it for all the other requests to the Catalog Broker, attaching it to the header of the request as a Bearer token.
This solution is intrinsically insecure, for the moment it represents only a placeholder ready to be implemented in the future. For instance, a secure behaviour could be the possibility to release a JWT only if the CSP is whitelisted or authenticated by a trusted third party, or if it owns a certification released previously by a certification authority, as it happens in the case of Gaia-X Federated Catalog. Of course all the requests coming from the CSPs will be authenticated by the Catalog Broker, and if the token is not valid, the request will be rejected.

5.4.2 Offers reflection

To better understand the concept of *Offers reflection*, we need to have in mind what is a Broker and what could be a Broker in the context of a Cloud Federation. Starting from the definition of a Broker, we can say that it is like an intermediary

[4]JSON Web Tokens

[5]Proof of Contept

between two or more parties, that is able to exchange information and to mediate between them. Moreover, usually, it is also responsible for what happens between the two parties, and it is the one that is responsible for the final result of the exchange.
In the context of a Cloud Federation, the Broker is, as already mentioned several times, the component that is responsible for the collection and aggregation of the offers of the different CSPs, and the distribution of the final aggregated catalog to the CSPs that are part of the federation. About this last aspect, we can say that the Broker is in charge of continuously update the parties about what is happening in the federation. Actually, different ready to use brokering solutions are already available, such as Apache Kafka, Apache Pulsar, RabbitMQ, etc. But, in our case, we decided to implement our own Broker, because we wanted to have a more flexible and customizable solution, that could be easily extended and adapted to our needs. Moreover, different functionalities could result not useful for our use case, increasing the complexity of the system, and adding some overhead. For example, in our case, we don't need a message queue, because we don't need to store the messages.
Our customized broker message flow is very simple, and it is based on a WebSocket connection, that is a bidirectional communication channel between a *client* and a *server*. The *client* can send messages to the server, and the *server* can send messages to the client. The WebSocket protocol is a standard protocol, and it is supported by all the major browsers, and it is also supported by all the major programming languages. We use this protocol because it is very simple to implement, and it is very efficient and very easy to scale, because it is based on a single TCP connection. Moreover, it is also very secure, because it is based on TLS. In our case, the broker is the server, and the CSPs are the clients. The broker is responsible for the distribution of the offers to the clients, and it is also responsible for the update of the offers, when a new offer is added, edited or removed. Due to this behaviour we decided to use this channel only uni-directional, from the server to the clients, this explains why we call this process Offers reflection. To open a WebSocket connection, the client has to send a POST request to the `/subscribe` endpoint of the Catalog Broker, and it has to attach the token (already released) to the header of the request as a Bearer token. The Catalog Broker will check if the token is valid, and if it is, start the following procedure:

1. Entablish the WebSocket connection
2. Extract the CSP Liqo information from the token
3. Get all the offers from the database excluding the ones of the CSP that is trying to connect
4. Wrap and organize the offers in a JSON object, that is an array of catalog

objects, one per each CSP in the federation that has at least one offer

5. Send the JSON object to the CSP through the WebSocket connection

The CSP will receive the JSON object, and it will parse it, extracting the offers of the other CSPs. From this point, it will be able to receive the updates of the offers of the federation, without doing any other request to the Catalog Broker. Here we can see the main advantages of this solution, that is the possibility to have real-time updates, without the need of periodically polling the Catalog Broker. Following an example of the JSON object that the CSP could receive (for the sake of simplicity, we are showing only the catalog of one CSP, but in reality at time of the connection the it will receive an array of the catalog of all the CSPs in the federation):


```
{
  "clusterID":"191a8cba-14e4-4fde-a60a-6995f812e047",
  "clusterName":"cloudprovider",
  "clusterPrettyName":"Cloud Service Provider",
  "clusterContractEndpoint":"http://contracts.example.it",
  "endpoint":"https://172.18.0.2:30425",
  "token":"3470cebaeeade3e7b190c064aaf912de08d94b06787dbd4c9523...",
  "created":1669393527206,
  "offers":[
    {
      "offerID":"191a8cba-14e4-4fde-a60a-6995f812e047_57c9090c...",
      "offerName":"CPU Offer",
      "offerType":"computational",
      "description":"here you can find the description of the offer",
      "plans":[
        {
          "planID":"191a8cba-14e4-4fde-a60a-6995f812e047_57c9090c..._1",
          "planName":"Basic Plan",
          "planCost":15,
          "planCostCurrency":"USD",
          "planCostPeriod":"week",
          "planQuantity":10,
          "resources":{
            "cpu":"100000m",
            "memory":"128Gi"
          },
        },
        {
          "planID":"191a8cba-14e4-4fde-a60a-6995f812e047_57c9090c..._2",
          "planName":"Enthusiast Plan",
```

```
31            "planCost":45,
32            "planCostCurrency":"USD",
33            "planCostPeriod":"week",
34            "planQuantity":10,
35            "resources":{
36              "cpu":"500000m",
37              "memory":"512Gi"
38            },
39          },
40        ],
41        "created":1669393827573,
42      },
43    ],
44  }
```

Listing 5.3: An example of a Catalog JSON object

At the end we can summarize our proposed mechanism to exchange information over a WebSocket transport with the following specification:

- When a *Customer* (a provider that acts as customer) first connects to the Catalog Broker over WebSocket, this sends a full list of offer catalogs.
- When a *Provider* creates, updates or deletes its offers, the updated catalog 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 catalogs is sent again.

5.4.3 Discovery and Advertisement APIs

The population of the Catalog Broker is up to the CSPs, that are responsible for the advertisement and the management of their own offers. The Catalog Broker, in this case, takes a passive role, and that does not care about for the creation of the offers and its content, but, as said previously, it is only responsible for the aggregation and the distribution of them. For this purpose, we implement the following methods on the Catalog Broker, beyond those already described:

1. `POST /offer/<id>`
2. `DELETE /offer/<id>`
3. `GET /catalog`

Mainly we can divide these in two categories: the discovery API, and the advertisement APIs. The discovery APIs are used by the CSPs, in the role that we can define as *consumer*, to discover the offers of the others in the federation. The advertisement APIs are used, in the role that we can define as *provider*, to advertise their own offers to the the federation. This behaviour is of course interchangeable anytime. On the *provider* side fall the API number 1 and 2, a set of REST APIs for CRUD operations on own offers. These APIs, as always, require authentication via JWT as a security layer which prevents unauthorized users from modifying offers on the Catalog Broker.
Through this API the provider can publish its commercial offering, update it as required, and remove it when it is no longer available. The `POST` method is used to create or update a new offer, it receives in the body of the request the offer to be added or updated, and as query parameter the ID of the offer. Internally the Catalog Broker will check if the offer already exists, and if it does, it will update it overwriting the existing, otherwise it will create a new offer. The `DELETE` method is used to remove an offer, it receives in the query parameter the ID of the offer to be removed. Both require the authentication via JWT, attached in the header of the request as a Bearer token, that will be used to check if the user is authorized and recognized by the Catalog Broker, but above all to extract the information of the CSP that is trying to perform the operation, in order to check if it is the owner of the offer.
We apply a similar reasoning for the customer-side interface. In this case, although, we only need a method to query the available offer catalogs. For this purpose we implement the `GET` method on the `/catalog` endpoint. This method does also require the same authentication system, to avoid unauthorized users, in this unauthorized CSPs, to fetch and download information that are available only for the federation members. It does not require any parameter, and it returns the catalog of the federation, that is an array of catalog objects, one per each CSP that has published at least one offer. In other words, we're looking at a producer-subscriber scheme, that coupled with the WebSocket connection, completes and makes available a lightweight and real-time communication and interaction system for the environment, avoiding wasteful polling and keeping the system as simple and reactive as possible.

5.5 Catalog Connector

As explained in the previous chapters, our goal was to move the most of the logic of our federation Broker on the edge, in the CSPs, and to keep the Catalog Broker as simple as possible allowing the environment to continue working even if the Catalog Broker is down. For this reason, we decided to implement a Catalog Connector,

with a much higher processing and functional complexity but which also allows not to affect the federation in case of failure of the single Provider. The final result is that up to this component there are most of the processes useful to the correctly working of the ecosystem. In particular it amis to:

- Communicate with the Catalog Broker through the collection of APIs described previously.
- Expose outside the cluster a `POST` method responsible of receiving the requests from the other CSPs to join/purchase a specific offer.
- Interact with the Kubernetes API server to obtain the Liqo peering informations and create the resources usefull to instantiate a peering with the other CSPs.
- Expose inside the cluster an set of APIs to be used by the UI to interact with the Catalog Connector and indirectly with the Catalog Broker. This will include also the REST methods for performing the CRUD operations on own offers and contracts stored on the Database.
- Allowing Liqo to acquire the contracts informations and the attached resources limits. These will be used by the Liqo Controller Manager to reserve the right amount of resources on the cluster to the CSP that is going to be peered.

We can resume the internal subset of module of the Catalog Connector in the following figure that we will individually explain in the following paragraphs.
As we can see, the Connector consists of different elements. First of all a Server, which contains our backend logic and it is responsible about the management of all those operations that include the manipulation of data and the instantiation/management of WebSocket connections, playing as a proxy between the UI and the Broker. Moreover it acts as controller through Liqo functionalities, as well as an API Endpoint to expose himself to internal and external requests. To this more logical elements, we add a Database, that is a MongoDB instance (the implementative choice is obviously the same of the Catalog Broker), used to store the contracts and the offers of the CSP, but in the future it cuold be extent to store the information about the peering status of the different clusters, the resources still available in own cluster and so on. Still, we add a gRPC server, queried by Liqo for the retrieving of contracts informations, and finally a User Interface that is the frontend of the Catalog Connector, and makes these operations easier and more user friendly for the administrator of the CSP.
When we had to implement this component, we found ourselves faced with the problem of having to choose whether to keep the same language used with the Catalog Broker with the disadvantage of running into greater difficulties in interacting with k8s or adding the overhead of using two different frameworks by

exploiting the implementation advantage due to the simplicity obtained thanks to wide libraries and documentation available for our use case. We decided to choose the second option, exploiting the Go language for the implementation of the Catalog Connector, using the k8s and Liqo libraries to interact with the cluster and the MongoDB driver to interact with the Database. At the end, from the frontend-side, we decided to use the React framework, based on the TypeScript language, which is the most used in the industry and which allows us to have a very simple and intuitive interface, with a very low learning curve for the users who in the future want to use and extend its functionalities. This choice was also dictated by the fact that we wanted to have a modern and responsive interface, which could be used on any device, from a smartphone to a desktop computer, without having to make any compromises on the user experience.

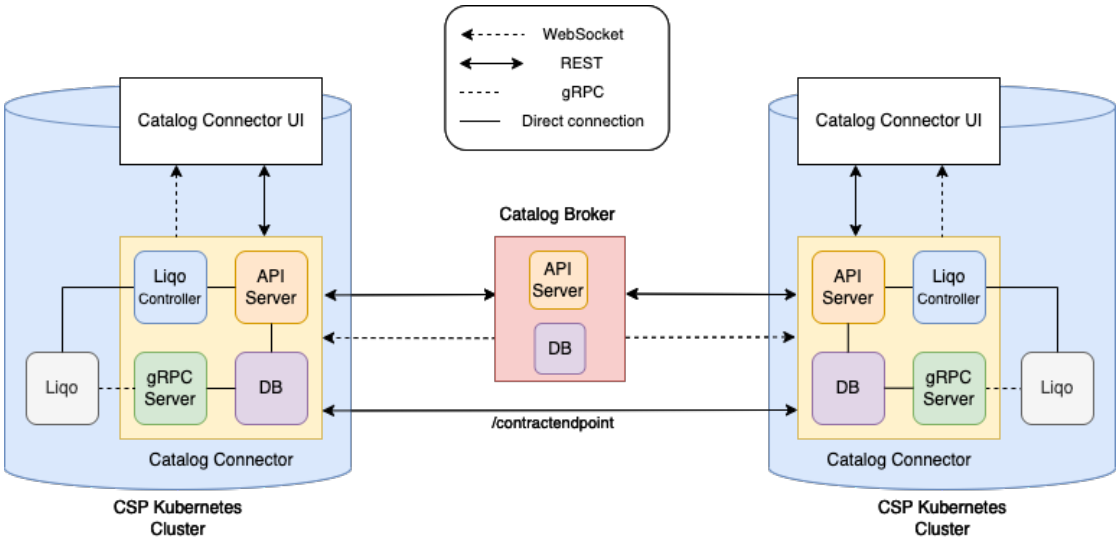

Figure 5.4: A closer and technical overview of the federation architecture

5.5.1 API Server: UI and Contracts

First aspects that we have to consider, describing the Catalog Connector, are the interactions and the underlying communication system between the UI and the backend logic. As we have seen in the previous paragraph, the UI is the frontend of the Catalog Connector, and it is the interface that the CSP administrator will use to interact with the core logic of the Server.
To design the architecture that we had in mind, we focused our effort mainly on two principles, very similar to the ones we have seen for the Catalog Broker, with the purpose of implementing a lightweight and reactive system that meets and fits the requirements of a modern and dynamic environment.

Basically, we wanted to coexist two different channel of communication, one classic and synchronous, based on the REST APIs, and one asynchronous and reactive, based on the WebSocket connection.
The first one, as we have seen, is used of course to perform the CRUD operations on the offers and contracts stored on the Database, is exploited when a new session of the UI is started to retrieve the information about the federation or when the Catalog Connector is initialized for the first time, as well as to get and set the Provider and Liqo informations.
The second one, instead, is used to notify the UI about the changes in the federation, and to allow to update the information in real time, without having to perform any polling operation to the server. In fact, in a federation, the number of changes that can occur could be very high, and the Catalog should be updated several times in a short time, so it is important to have a system that can converge as fast as possible to the new state of the federation, showing coherent and consistent information to the user.
Now that we know how Catalog Broker, Connector and UI work together, it is time to see how the Catalog Connector works internally and how it manages the above-mentioned communication channels.
In the Figure 5.4 we have shown a more technical and closer overview of the federation architecture. What immediately catches the eye is the presence of two different WebSocket Connections, one from the UI to the Server (Catalog Connector) and one from the Server to the Catalog Broker. These two connections are physically separated but logically connected thanks to a complex system of channel pools and go routines that allows the Server to receive messages from the subscribed Brokers and forward them to all the active UI sessions. In this way, for instance, if a user "A" of the CSP "A" performs the registration of a new Catalog Broker, the server receiving the whole federation catalog from him is able to forward it to all the active UI sessions of the CSP A, allowing all connected user (B, C, D, etc.) to see the update in real time. Of course this is just one of the possible corner case examples that we can make, but it is important to understand the importance and the complexity of this system. Moreover through this implementation strategy we avoid to instantiate multiple WebSocket connections from the UI to the Catalog Broker, which would be a waste of resources and a source of possible errors and problems.
After that, as already mentioned, we need to describe the synchronous communication channel, which is based on the REST APIs. Going in depth, this is the list of the APIs that we have implemented:

- `GET /api/contracts`: returns the list of all the contracts stored on the Database.
- `POST /api/contracts/buy`: creates a request to buy a specific offer plan, and

returns the status of the operation (CUSTOMER-SIDE).

- `POST /api/contracts/sell`: used to accept a request to buy a specific offer plan, and returns the status of the operation (PROVIDER-SIDE).
- `DELETE /api/contracts/:id`: deletes and revokes the contract with the specified id.
- `GET /api/offers`: returns the list of all own offers stored on the Database.
- `POST /api/offers`: creates or update an offer on the Database.
- `DELETE /api/offers/:id`: deletes the offer with the specified id.
- `GET /api/brokers`: returns the list of all the Catalog Brokers registered on the Database.
- `POST /api/brokers`: creates a Catalog Broker on the Database and try to connect to it.
- `DELETE /api/brokers/:id`: deletes the Catalog Broker with the specified id and disconnects from it.
- `PUT /api/brokers/:id`: updates the Catalog Broker subscription with the specified id.
- `GET /api/cluster/init`: returns the information about the readiness of the Catalog Connector.
- `POST /api/cluster/init`: initializes the Catalog Connector.
- `GET /api/cluster/parameters`: returns the Liqo cluster parameters.
- `GET /api/cluster/prettyname`: returns the CSP pretty name.
- `POST /api/cluster/prettynames`: updates the CSP pretty name.
- `GET /api/cluster/contractendpoint`: returns the endpoint of the CSP useful to join offers and stipulate contracts.
- `POST /api/cluster/contractendpoint`: updates the endpoint of the CSP useful to join offers and stipulate contracts.
- `GET /api/subscribe`: instantiates a WebSocket connection with the Catalog Connector from the UI.
- `GET /api/catalog`: makes a request to the Catalog Broker to retrieve the whole federation catalog.

Among these, those that deserve more attention are the ones that allow the Catalog Connector to buy and sell offers, creating contracts between the CSPs, the ones that allow to interact with Liqo to instantiate a peering with a specific CSP, and the ones that allow to makes CRUD operations on the Database. The first two groups of APIs will be described after in the next paragraphs, here we focus on the last one.
About the CRUD operations, we have to say that they are very similar to the ones we have seen for the Catalog Broker, with the only difference that the Catalog Connector is able to perform them on its database, and manipulate and save only the owned offers without saving the federation received ones. The notable concept is that it is responsible of the synchronization of the offers with the Catalog Broker, so also of the consistency of the CSP offers in the federated Catalog. Every time a new offer is created, updated or deleted, the Catalog Connector sends a request to the Catalog Broker to update the federation catalog. For this reason this process becomes very important and delicate. In fact, if the synchronization is not performed correctly, the Catalog Connector could be in a state of inconsistency with the Catalog Broker, and the offers could be not correctly updated on it.
To avoid this problem, we have implemented such a security mechanism that allows the Catalog Connector to understand if the Catalog Broker is in a consistent state or not, to emergency clear the remote catalog and to immediately stop the synchronization process, notifying the user about the problem, to let him decide what to do. For the moment we follow the strategy of using a timestamp as consistency indicator, but we are working on a more complex and reliable solution such as a cryptographic signature or hashing.

5.5.2 Offer joining and Peering

Until now we have seen all the aspects about how the Catalog Connector manages the synchronization with the Catalog Broker, how it works internally and manages Offers, and consequently how it makes CRUD operations on the Database. All these aspects represent the underlying concepts to better understand the real purpose of our study, which is the possibility to explore and join offers and to stipulate contracts between CSPs in a Kubernetes cluster federation.
As far as the the offer joining is concerned, we have to start doing an overview about what were the main problems that we have faced during the designing of this feature.
First of all, due to how we had implemented the Catalog Broker, we had to deal with the fact that it was not thought, and therefore capable, to operate as middleman for complex operations like a negotiation. This is just a design limitation, that, as we have seen in the previous paragraph, we strictly imposed from the beginning to avoid a relevant centralization of the Catalog Broker. If from one side this is

a good thing, because it allows to have a more flexible, distributed and scalable environment, from the other side it is a problem because it forces us to implement a new logic for the direct communication between two different CSPs and so two different Catalog Connectors.
To overcome this problem, we have decided to make available a couple of APIs: one that is exposed only internally to the UI and that allows to make an internal request for the creation of a contract, and one that is exposed externally to the federation and that allows to make a request to buy a specific offer plan. In the Figure 5.5 we can see the sequence diagram that describes the offer joining process with the consequent contract creation.

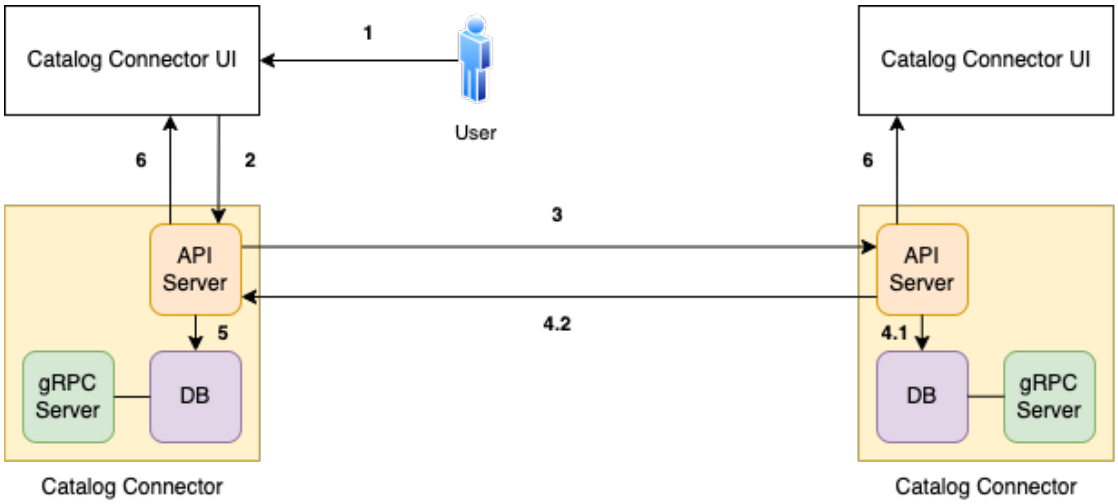

Figure 5.5: Offer joining and contract creation sequence diagram

In order, the steps are the following:

1. The user selects an Offer from the Catalog of the UI and clicks on the *Join* button of a specific Plan.

2. The UI makes a request to the Catalog Connector to buy the selected Offer Plan, specifying the offer id and the plan id.

3. The Catalog Connector, after checking the validity of the request, makes a request to the Catalog Connector of the remote CSP to buy the selected Offer Plan, specifying the offer id, the plan id and its own clusterID. To make this possible, it uses the *Contract Endpoint* hostname or IP previously provided by the remote CSP during the offers publishing phase.

4. The Catalog Connector of the remote CSP, after checking the validity of the request, answers to the Catalog Connector of the local CSP with the newly

created contract, if the request is valid, or with an error message otherwise.

5. The Catalog Connector of the local CSP, checks the validity of the response and, if it is successful, it deposits the contract in the own Database and sends a notification to the UI.

6. The UI, after receiving the notification, updates the UI with the new contract in the Contracts section.

Careful readers may have noticed that the Contract stored by the *Consumer* (the buyer) it is the same received and stored by the *Provider* (the seller). This behaviour it is clearly due to the will to have data consistency between the two CSPs, we are talking about a negotiation, and so we have to be sure that the two parties are in agreement about the same contract.
For the moment we have not implemented a logic for the negotiation, such as a payment or a billing system, because it is not the main purpose of our study. In the future, we will implement a more complex logic after some researches about the best way to implement it.

5.5.3 Liqo controller

Now that we have in mind all the process of our study, we can move our attention to the integration of our Catalog Connector with Liqo (Chapter 3).
Our goal was basically to make the Catalog Connector of the *Consumer* (the buyer) able to interact with Liqo to instantiate a peering with a specific CSP, and, viceversa, to make the Liqo Endpoint of the *Provider* (the seller) able to communicate with the Catalog Connector to obtain the needed information to answer to a peering request reserving the correct resources written in the Contract between the two CSPs. The second part of this goal was the most complex and it is deeply explained in a dedicated next section, here we will focus on the first one.
Firstly we studied how works the standard behavior of Liqo and we discovered that a Liqo peering session starts always from the *Consumer* towards the *Provider*: this workflow fits perfectly with our needs. We also understood that this process started from the creation of a new CR[6] called *ForeignCluster*, included in the Liqo framework, that is created in the *Consumer* cluster, so, to not reinvent the wheel, we decided to reuse the same process exploiting the functionalities available with the Liqo library.
Thanks to this, we have been able to implement a new controller inside the Catalog Connector that could answer the request of peering coming from the UI, forwarding it to Liqo, without applying any modification to vanilla version of Liqo.

[6]Custom Resource

A complete sequence diagram of the peering process is shown in Figure ?? in the Section 6.2.

5.6 Catalog connector UI

The main topic of this section is the UI of the Catalog Connector, which is the unique interface that the user has to interact with to manage the Catalog Connector. We already explained all the concepts and the logic behind the Catalog Connector UI, but we have not yet seen how it looks like and how it works in practice. In this section we will see the main features of the Catalog Connector UI, and we will explain how it works.
The UI is composed by four main parts: the *Catalog* section, the *Contracts* section, the *Broker* section and of course the *My Offers* section.
All this sections are equally important, because they allow the access to the main features of the Catalog Connector. Moreover, another page has been added to the UI, the *Overview* page, that for the moment is just almost totally a placeholder, but that could be used in the future to show some statistics about the Cluster, the existing peering, with its related metrics and status, and so on. Actually, the only thing that really works is the part of this page that shows the Provider information and allows to change the Provider pretty name and the Provider Contract Endpoint. Here a screenshot of the Overview page of the Catalog Connector UI

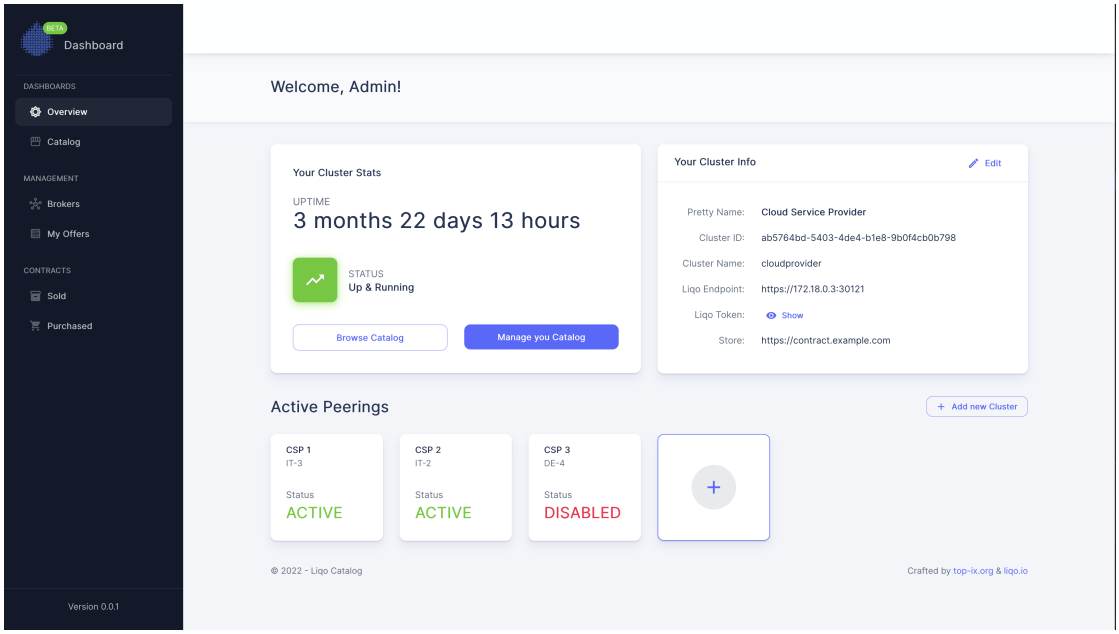

Figure 5.6: Catalog Connector UI: Overview page

When a user opens the Catalog Connector UI for the first time, it will be redirected to an initialization page, where it will be asked to insert some information, such as the Provider pretty name and the Provider Contract Endpoint. After that, the user will be redirected to the Overview page. If we move to the Catalog section, we can see that it appears empty, because the Catalog Connector is not yet connected to any Catalog Broker. To to do this, we have to move on the Broker section and click on the + (plus) button Figure 5.7, now we have to insert the Broker hostname or IP and the Broker name Figure 5.8.

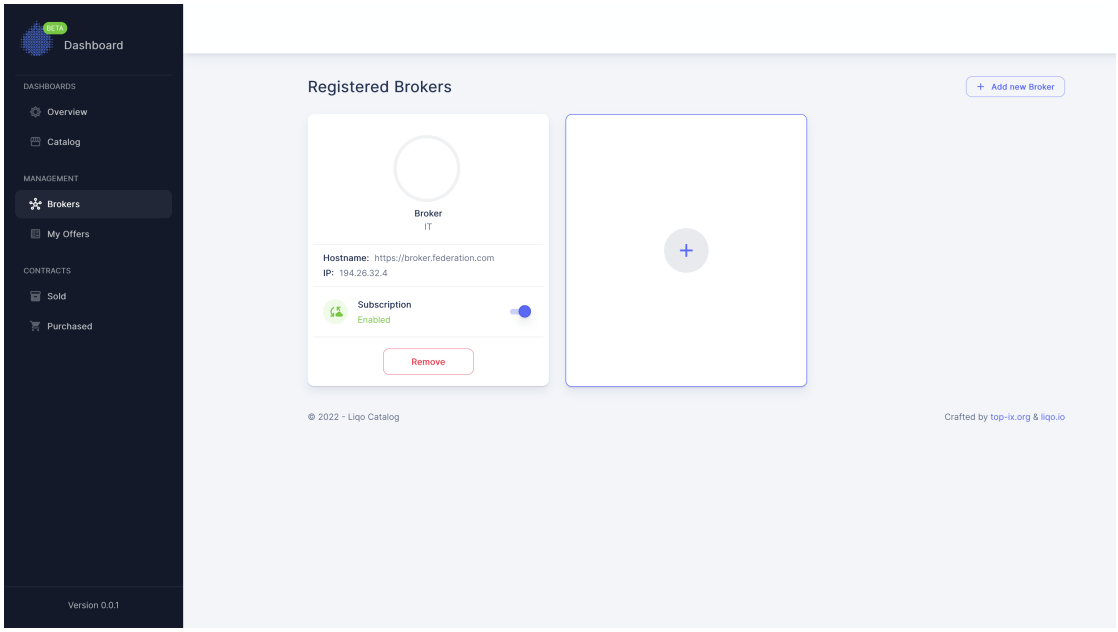

Figure 5.7: Catalog Connector UI: Broker section

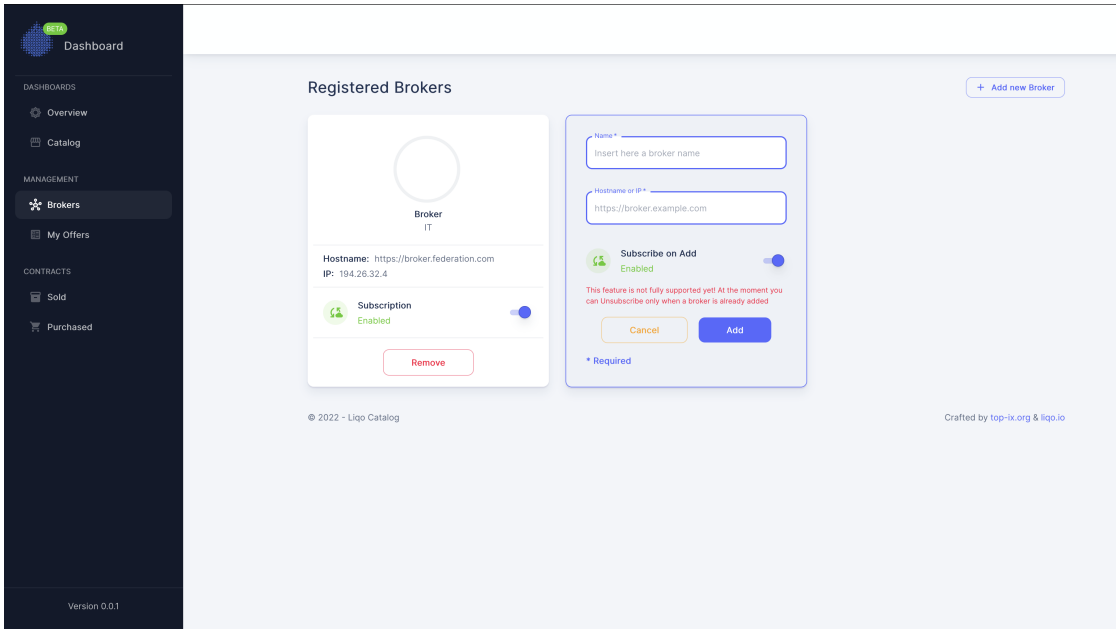

Figure 5.8: Catalog Connector UI: Broker connection

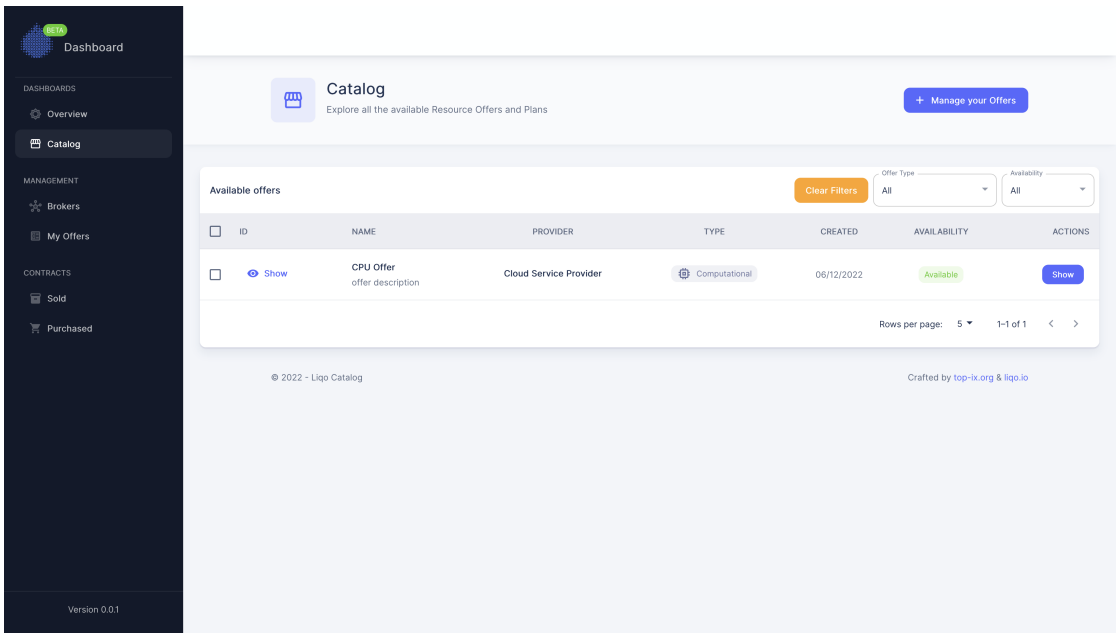

Figure 5.9: Catalog Connector UI: Catalog section

After that, the Catalog Connector will be connected to the Broker and the Catalog section will be populated with the offers published by the other CSPs on the

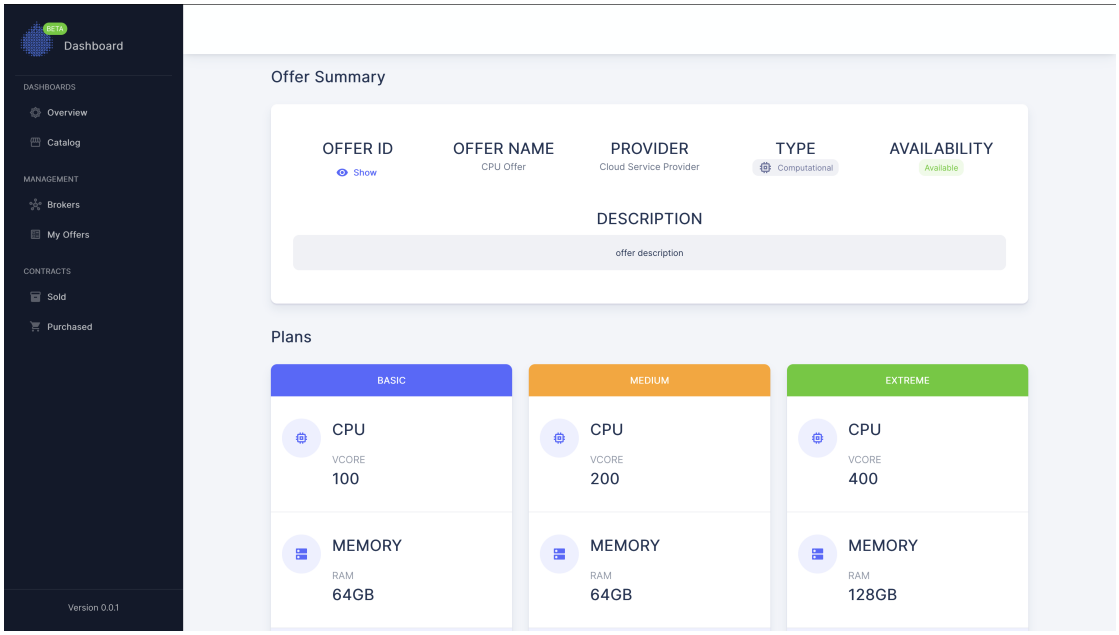

Figure 5.10: Catalog Connector UI: Offer details

federated Catalog. Now it is possible to browse the Catalog Figure 5.9, explore the offers Figure 5.10, and buy the desired offer Figure **??**, and join one of its Plans, clicking on the *Join* button Figure **??**.
When the user buys an offer, the Catalog Connector will create a new Contract between the two CSPs, and the Contract will be stored in the Purchased Contracts section. The user can also revoke the Contract, and the Catalog Connector will delete the Contract also from the corresponding remote CPS.
Of course, the user can also publish its own offers through the My Offers section Figure 5.11, and as widely explained before the Catalog Connector will publish them on the Broker. He has to click on the *Insert a New Offer* button to create a new offer, jumping on a page where he can fill a form with the desired Offer information Figure 5.12. It can create till 6 Plans for each offer, and each Plan can have a different price and billing method and a different set of resources 5.13.

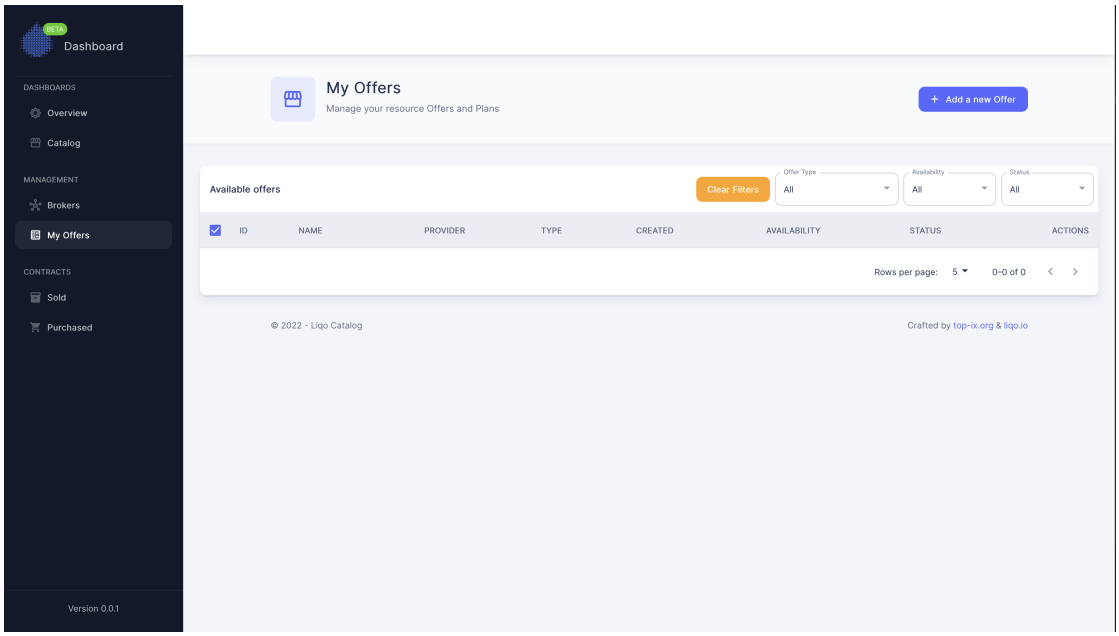

Figure 5.11: Catalog Connector UI: My Offers section

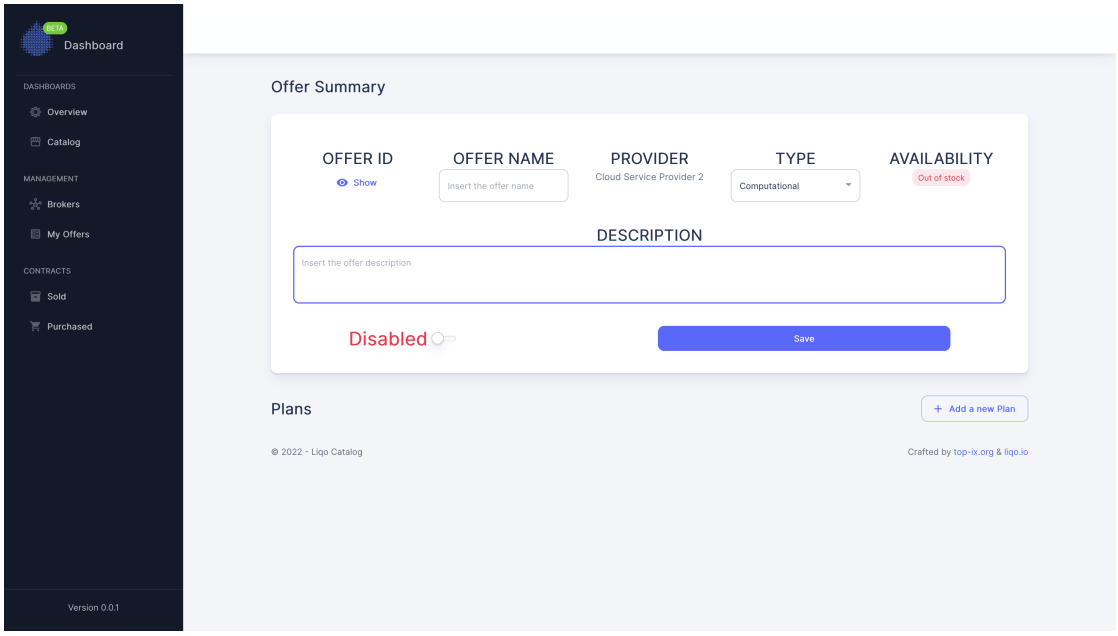

Figure 5.12: Catalog Connector UI: My Offers creation

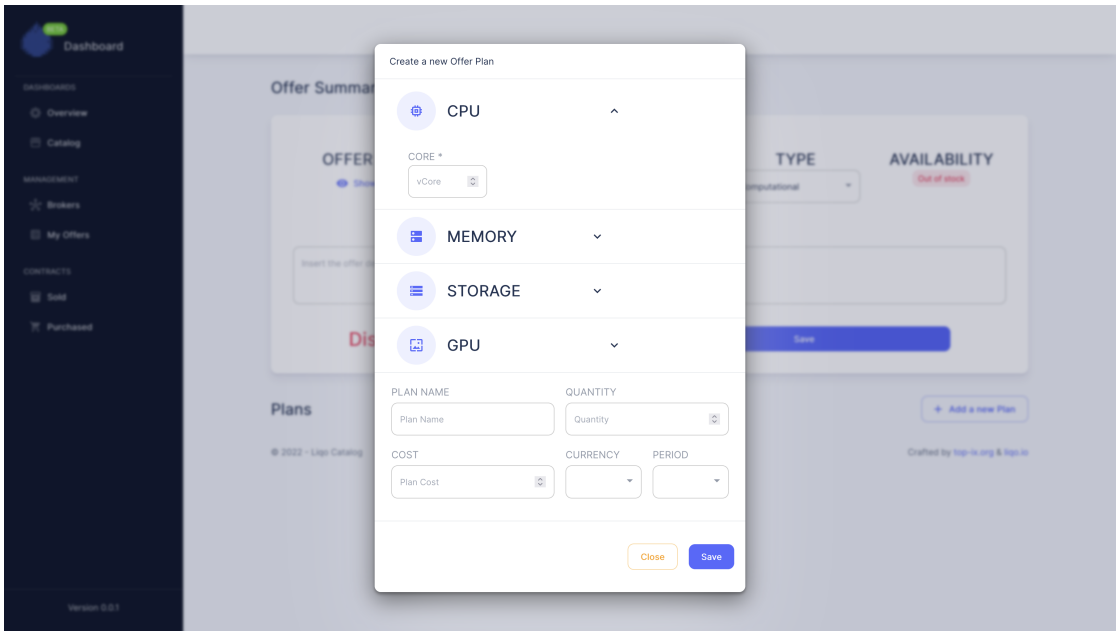

Figure 5.13: Catalog Connector UI: My Offers Plan creation

Chapter 6

Implementation

Until now, starting from the initial problem definition, we have seen how our research and study has led us to discover the various architectural solutions that could be used to reach the goals of the project, we have defined the main concepts and the requirements that should satisfy our model of a Multi-tenant federated brokering system between kubernetes clusters, designing and in the meanwhile explaining the implementation of the main components of the desired environment: the *Catalog Broker*, the *Catalog Connector* and its *UI*.
But, as we said before, the goals of this project were from one side to implement a working prototype of the proposed solution, also with the purpose of validating it with the mentioned Structura-X PoC, on the other hand we needed to implement in Liqo a system to support the multi-tenancy in a fashion that better suits our requirements of resource enforcements and workloads identity validation, enabling at the same time a way to customize the assignment of resources during the establishment of a new peering connection
We have widely explain the first goal, but we have not yet seen how we have implemented the second one, and why we have chosen to implement it in this way. This section is dedicated to this topic, because the work behind is noteworthy and has taken a lot of time and effort, and we think that it is important to explain it in detail.
We can mainly divide the work done in two parts: the first one is related to the implementation of a new feature in Liqo, that for semplicity for the moment we can call as *Liqo Multi-tenancy*, and the second one that is related to the modification of an already existing component in Liqo, the *External Resource Monitor*, that was born as an element useful for the *Aggregator* brokering model, but which we have redesigned as an extension point to implement external mechanism of resource assignments to each peering connection different from the default one, with the purpose to suit Liqo for our scenario.
If the first case it is a real new feature, design and developed from scratch, the

second one is a strengthening of the behaviour of an already existing component. This two parts can be respectively expanded in the following descriptions:

- **Liqo: Multi-tenancy**: A system that enables Liqo for a multi-tenant management of the established peering, adding a new layer of control both regarding the check and validation of the workload offloading requests for each peering and the enforcement of the resources consumption limits assigned to the corrisponding tenant.

- **Liqo: Custom peering**: A system able to intercept and modify the standard liqo peering flow to customize the assignment of the resources to each peering connection based on an external source of information without impacting the standard Liqo behaviour.

At the time of writing this document, both the features are already implemented and merged in the Liqo repository, and they are available in the latest release of the project, the version 0.6.

6.1 Liqo: Multi-Tenancy

The Liqo framework is mainly designed to able, in a single domain, to create a continuum between different Kubernetes clusters, it is also capable to work in a Multi-cloud environment, as long as this clusters are managed by the same Kubernetes administrator. From the point of view of the interconnection and the sharing/offloading of resources, Liqo is the perfect solution to fit our needs, but it is not designed, by principle, to be used in a multi-tenant environment. As far as this last point is concerned, Liqo could represent a limit, because it conflicts with the initial idea of our project.
The standard behaviour of Liqo, in fact, does not care about partitioning and isolating the usage of the resources between the different tenants, moreover it does not provide any kind of control on the resources consumption assigned to each one. This is explained because it was not intrinsically thought for this purpose.
Starting from this point, we challenged ourselves to understand how to enable Liqo to work also in this direction, and after a long period of study and research, we have found a solution by taking inspiration from the *Resource Quota* object of Kubernetes, which is a standard feature of the Kubernetes framework, and which is used to limit the resources consumption of a single namespace. It provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by workloads in that namespace.

In our case, we have decided to implement a similar system, not at the namespace level, but at the peering level. To realize this, we have decided to implement a new Liqo component, mainly based on a *Validating Admission Webhook* and a *Cache* system, that will intercept the requests of the pods offloading, in particular the *ShadowPod* Liqo CRD, making on the fly some checks and validations, and then decide if the request can be accepted or not. This behaviour is better explained in the Figure 6.1 and described in detail in the following paragraphs.

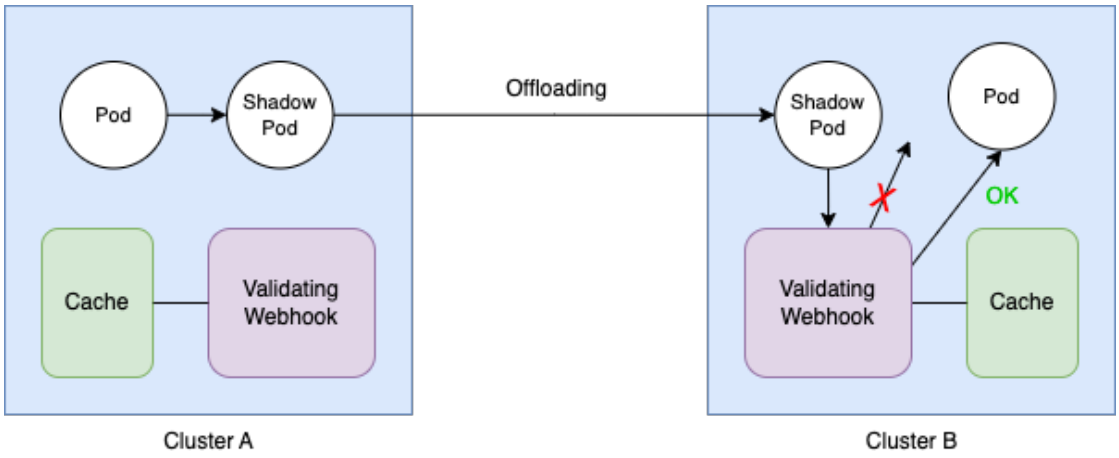

Figure 6.1: Liqo Multi-tenancy

6.1.1 ShadowPod Validating Webhook

The first step to realize the Liqo Multi-tenancy system was the implementation of a new *Validating Admission Webhook*. A Webhook is a user-defined HTTP callback that is called and triggered by an event. There are two types of Webhooks: *Mutating Webhooks* and *Validating Webhooks*. The first one is used to modify the request before it is accepted, while the second one is used with the purpose to validate the request before it is accepted.
In our case, the event is represented by the creation of a new *ShadowPod* object. Intercepting this event, our Validating Webhook will be able to make some checks and validations on the request before deciding if the request could be accepted or not.
The first check that we have implemented is the one that controls if the *ShadowPod* is created in a namespace that is allowed and its ownership corresponds to the tenant that is trying to create the resource. This check is performed by comparing the origin *ClusterID* of the *ShadowPod* with the destination *ClusterID* mapped to the namespace in which the *ShadowPod* is created. If the two *ClusterIDs* are equal, the request is accepted, otherwise it is rejected.

The second check is related to the resources consumption limits assigned to the tenant, but for a better understanding of this check, we decided to dedicate the following paragraph to it.

6.1.2 Resource Enforcement

As previously anticipated, in this paragraph we try to explain our implementation solution for the resource enforcement, which is the second check that we have implemented in the Validating Webhook.
We already mentioned that our strategy takes inspiration from the *Resource Quota* object of Kubernetes and its behaviour. In fact, it is not a coincidence that we have decided to reuse, from the Kubernates library, some of the functions that are used to manage the *Resource Quota* object, and in particular the implementation in charge of manipulating the aggregate resources consumption, adapting it to our needs, or for example the logic used to calculate the resources consumption of a single *Pod*, starting from the defined resources limits of the containers and initContainers that compose it.
Up to here everything was pretty simple, but the question we arised was: how can we know the already consumed resources by the tenant to calculate if the new request goes over the assigned limits or not, without adding a computational overhead to the system.
The answer to this question has been given by implementing a *Cache* system, which is a custom data structure that supports our Webhook and stores the information of each peering connection and the defined limits imposed by the corresponding *ResourceOffer*, as well as the resources consumption of each tenant.
This system is updated in real time by the Validating Webhook, which is the only component that has access to it, and which is the only one that can make the properly calculations. Moreover, this *Cache* allowed us to solve the problem of the possible inconsistencies that could arise trusting the system snapshots returned by the Kubernetes API server when it is queried by the *Controller Runtime* Go Client. This solution has paid off, both in term of performance, as we will see in the following section dedicated to the benchmarks, and in term of reliability. But the optimal solution have been found adding a refresh mechanism to the *Cache* system, which is responsible to periodically update the information stored in the *Cache* by querying the Kubernetes API server, in order to avoid possible inconsistencies that could arise due to an possible failure of the Validating Webhook or possible corner cases that we have not considered, consolidating the stored data.

6.2 Liqo: Custom peering

Solved the problem of the Liqo Multi-tenancy, we have decided to go further, trying to understand if it was possible to obtain a more flexible and customizable behaviour of the Liqo peering, without impacting the standard flow, and without modifying almost at all the Liqo codebase. We had clear in mind what was our goal, but all the first roads we have tried to follow, have led us to a dead end, because they all required a deep modification of Liqo, and this have been involved a lot of work, and a lot of time, going against the initial idea of our project.
One of the possible solutions, that we have tried to examine, involved the understanding of how much it would cost, in terms of time and effort, to modify the CRDs in charge to manage the peering peering process. We are talking about the *ResourceRequest* and *ResourceOffer* Liqo CRDs. In fact, these two CRDs are the ones that are used to exchange the information between the two clusters, and respectively to trigger a request of peering creation and to answer to it providing the information about the resources that are available to be shared.
The basic idea was to add some new fields to the *ResourceRequest* CRD, in order to allow the *consumer* cluster to specify the resources that it wants to get from the *provider* cluster. Besides that, as already said, this solution would have involved a lot of work, and a lot of time, in order to apply this changes we would have had to drastically modify Liqo, probably rewriting a lot of code, without the certainty of the result. Moreover, this would have been a very risky choice, because it would have been very difficult to guarantee the correct functioning of the system. And for this reason, we asked ourselves if, at the end, we were going down the right path. Fortunately, after a long period of study and research, we have found a solution that has allowed us to obtain the desired result.
We focused our attention on how Liqo performs the calculation of the resources that are available to be shared, in particular we paid attention to the *Liqo Controller Manager* component, which, among all his duties, it is in charge to demand to the *Local Resource Monitor* component to calculate, starting from some internal metrics, the resources that the *provider* cluster is able to share. We realized that there was the possibility to replace the *Local Resource Monitor* component with the *External Resource Monitor* component. This element is basically a gRPC client, that makes available a set of interfaces useful to query and communicate with an external gRPC server.
Its native purpose was to use this component to integrate an external brokering system able to run as an *Aggregator* of resources of multiple clusters, exposing them as a single entity. We already explained this concept in the previous paragraph where we explored the Brokering models (Chapter 4). What resulted interesting for us, was the possibility to modify this component to implement external mechanism that runs custom logic to forward to the *Liqo Controller Manager* the resources

that we wanted to share.
In the next paragraph we will explain how this component has been used to obtain the desired result and how it has been implemented. Here a representation of the architecture of our final solution, with the main components involved in the implementation of the Custom Peering.

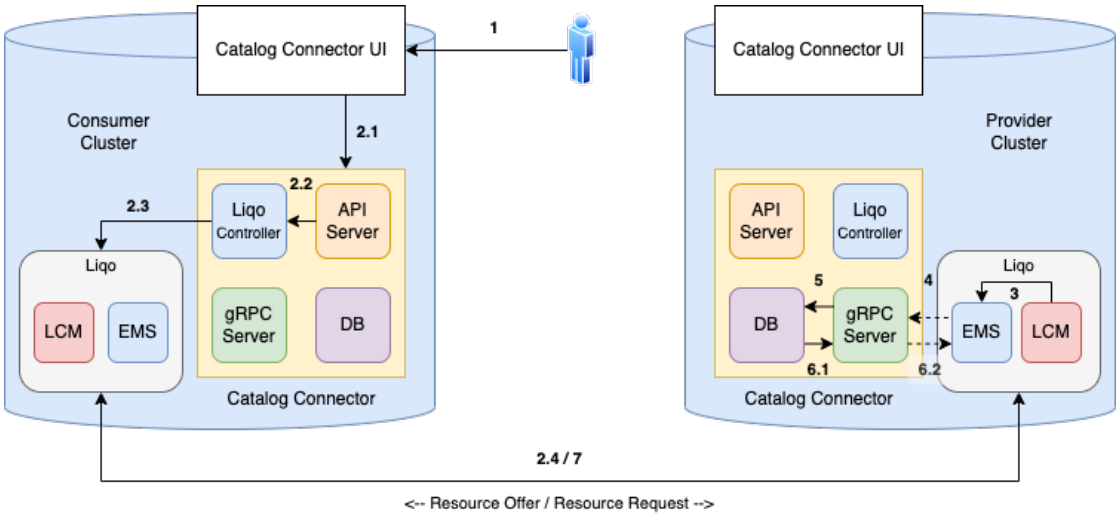

Figure 6.2: Custom Peering Architecture and flow

6.2.1 External Resource Monitor

We almost said everything about the External Resource Monitor, but we have not yet explained how it has been implemented. The External Resource Monitor is a gRPC client that declares a set of interfaces that can be used to query an external gRPC server. We have already described its purpose and its gRPC interfaces in the right paragraph, but we have not yet explained how it works. Basically, when the *Liqo Controller Manager* is instantiated to deal with the External Resource Monitor (for simplicity from now we will call it EMS), it creates a new EMS object, which wraps a gRPC client that is in charge to connect to the EMS gRPC server running on the same cluster with a specific IP/hostname. After the connection is established, the system is ready to manage the peering requests. From this moment there are three methods involved in the process:

- `func (m *ExternalResourceMonitor) Register(...)` This is the method that sets an update notifier to the EMS object, which is the function that instantiates a subscription to keep listening to possible notification.
- `func (m *ExternalResourceMonitor) ReadResources(...)` This is the

method that is called by the *Liqo Controller Manager* to ask to the gRPC server a set of resources to be shared, given a specific *ClusterID* of the *Consumer* cluster. This method is called every time a new peering request is received and it answers with a *ResourceList* kubernetes object, which is a map that contains key-value pairs, where the key is the name of the resource and the value is the quantity of that resource.

- `func (m *ExternalResourceMonitor) RemoveClusterID(...)` This is the method that is called by the *Liqo Controller Manager* to notify the EMS gRPC server that a cluster has been removed and so the peering process has been terminated.

6.2.2 gRPC Server

To complete the implementation of the Custom Peering, we have had to implement the other side of the EMS, that is mainly composed by two components: a gRPC server able to receive the requests from the EMS client, and compatible with the EMS gRPC interfaces, and a custom logic that is in charge to compute the resources that we want to assign to each cluster.
If we pay attention to the Figure 6.2 and we remember what we have seen and explained in the *Catalog Connector* (chapter 5.5), focusing on the Figure 5.4, it becomes clear that the EMS gRPC server in question is the same included in our *Catalog Connector* component. At this point it is simple to us to explain the new flow of the Liqo Peering.
Once a new *Contract* (chapter 5.2) is created and stored both in the *Provider* and *Consumer* clusters:

1. The *Consumer* admin can click and start the peering request process.

2. The *Consumer Catalog Connector* creates some internal resources that send a peering request towards the *Provider* cluster through a *ResourceRequest*.

3. The *Provider* cluster is triggered by the *ResourceRequest* and forwards the request to its *EMS*.

4. The *EMS* tries to communicate with the gRPC server inside the *Catalog Connector*, attaching the *ClusterID* of the *Consumer* cluster.

5. The *Catalog Connector* search in the *Contract* database the one that matches the *ClusterID* of the *Consumer* cluster.

6. If the *Contract* is found, the *Catalog Connector* returns the resources that are specified in the *Contract* to the *EMS*.

7. The *EMS* sends the *ResourceList* to the *Liqo Controller Manager*, which will use them to create the *ResourceOffer* that will be sent to the *Consumer* cluster.

Now the peering process is completed, the connection is established and the *Consumer* cluster can start to use the resources that are shared by the *Provider* cluster.

6.3 Conclusion

In this chapter we have seen two different approaches to solve two different problems. In the first case our solution was completely a new feature, where we designed, implemented and tested new concepts and new components in a production ready fashion. In the second case we have used the existing Liqo components, coupled with new external ones, to impose a new behaviour to the Liqo peering, without changing its core, but only studying and applying a workaround that ables us to obtain the desired result.
This two solutions, if combined together, allowed us to exploit Liqo in a new way, going beyond the architectural limits of the original project, and to obtain a new and interesting application use case of Liqo, without however affecting the performance, in terms of resource consumption and offloading latency, of the original project.

Chapter 7

Evaluation

The proposed work requires validation in terms of resources consumption and latency. The analysis separately considers the two different implementations parts explored in the previous chapter 6.
In particular, about what concerns the *Validating Webhook* and the *Resource Enforcement*, we have to benchmark how much they affect the offloading latency performance and the resource consumption, in terms of RAM, of the *Liqo Controller Manager*, considering the additional step that have been added to the offloading process, and the cache that is used to store the multi-tenant peering information.
On the other hand, about what concerns the Custom Peering, the evaluation is focused on the additional latency that has been added to the peering process using the *External Resource Monitor* rather than the *Local Resource Monitor*, considering that the big difference is not represented by a new implemented component, however by a new behaviour of the existing ones and its intrinsic latency overhead due to the communication with the external gRPC server. We will see better in the following paragraphs.

7.1 Benchmarks and Measurements

First of all let us to discover the test environment that we have used to perform the benchmarks. We have run our benchmark on a Kubernetes cluster composed by 7 nodes, 3 masters and 4 that act as worker nodes, where the master nodes are used to run the control plane components of the cluster, while the worker nodes are used to run the user applications. The nodes are connected to each other through a 10 Gbps network.
The overall cluster at our disposal offers:

- 96 vCPUs

- About 528 GB of RAM

The onboard Kubernetes version is 1.24.7 and the Liqo version is 0.6.1. The cluster is hosted on premises, in a datacenter, and it is managed by *TOP-IX Consortium*, the company that has supported my thesis. Moreover this cluster was used to test and validate the Structura-X PoC discussed in the chapter 4, where we, together with other important CSPs, Academies, System integrators in Europe, have used Liqo to simulate a Kubernetes clusters full-mesh Federation.
Now that we have seen the test environment, let us see the benchmarks that we have performed to evaluate the result of the proposed work.

7.1.1 Liqo: Single-Tenant vs Multi-Tenant

As said initially, we structured our benchmarks in two different parts. For the first one the goal was to understand how much the Validating Webhook and the Resource Enforcement affect the offloading latency performance. So we did a comparison between Liqo in single-tenant mode and Liqo in multi-tenant mode, considering the same application and the same workload, but with different configurations of the Liqo Controller Manager.
The tests were performed simulating inside our environment two different K3s clusters, one of them is the *Provider* cluster, the other one is the *Consumer* cluster. The *Provider* cluster owns 100 worker nodes and 1 master, while the *Consumer* cluster only 1 master. The steps followed to benchmark the offloading latency performance were the following:

1. Deploy the *Provider* cluster, with its worker nodes, and the *Consumer* cluster as Pods that runs inside testing environment.

2. Establish a Liqo peering between the two clusters only in the direction *Consumer -> Provider*. So the *Consumer* cluster can offload its workloads to the *Provider* cluster.

3. Inside the *Consumer* cluster create a Namespace and a NamespaceOffloading (Liqo CRD) and deploy recursively inside it a defined number of pods, automatically offloaded to the *Provider* cluster.

4. Measure the time passed from the moment the first pod is deployed to the moment the last pod is running.

5. Remove all the resources created in the previous steps.

The steps from 2 to 5 have been repeated for different numbers of pods, following the subset of 10, 100, 1000 and 10000. This test has been performed three times

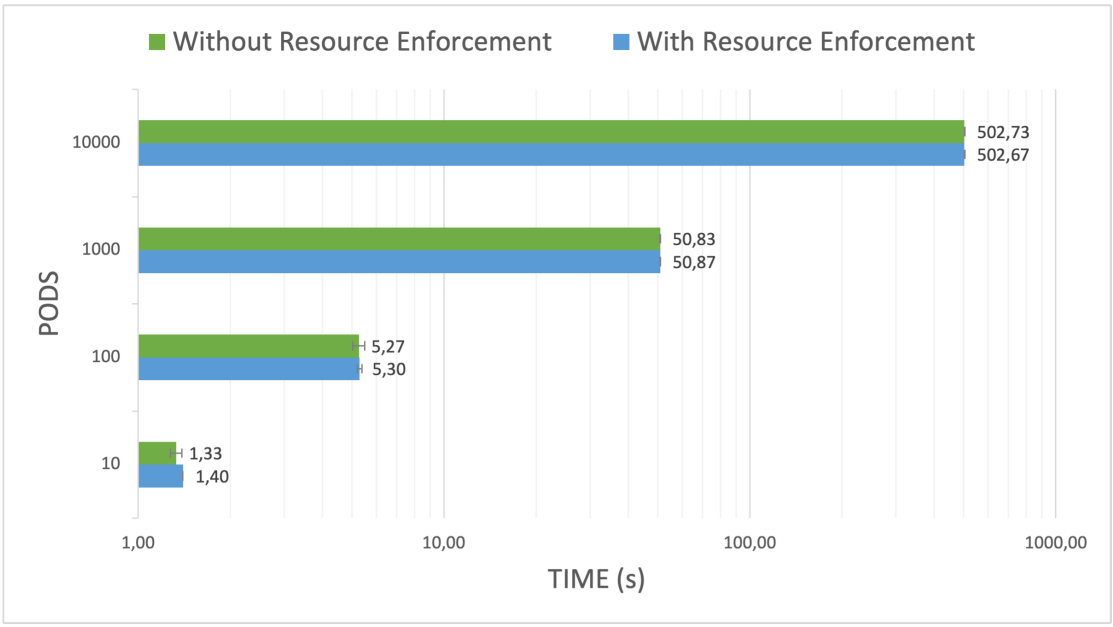

Figure 7.1: Liqo: Single-Tenant vs Multi-Tenant - Offloading Latency

for both the single-tenant and the multi-tenant mode of Liqo, and the results are shown in figure 7.1.
As we can see from this figure, the offloading latency performance is completely not affected by the multi-tenant mode of Liqo, and the difference between the two modes it is practically negligible. This is due to the fact that the Validating Webhook and the Resource Enforcement have a very low impact on the offloading latency performance, respect of capability of Kubernetes to schedule pods. Most of the time is spent by the Kubernetes to bring up the pods into a running state, and not by the Liqo components.

The same test has been done with the purpose to calculate the resource consumption of the Liqo Controller Manager, considering the additional cache that is used to store the multi-tenant peering information with the result that is not affected by the multi-tenant mode of Liqo. This is due to the fact the usage of RAM is the same already used by other existing components inside the *Liqo Controller Manager* that work on the same resources already managed by our new components, overlapping, in fact, the RAM usage made by our cache.

Chapter 8

Conclusions

This work introduces a new way to take advantage of the Liqo project, going beyond the architectural limits of the original project, and to obtain a new and interesting application use case. What we were searching for was a straightforward and intuitive way to enable a federation of Kubernetes clusters that can be found application in a real world scenario, and not only in a research environment.
With this purpose we studied what was the state of art and the technology that was already available. This bring us to choose the Liqo project as the base of our work, that perfectly fits for needs in terms of lightness and flexibility. Moreover, the Liqo project is already a mature project, with a lot of features that simplify our work, and it is already used in production by some important companies, besides that the Politecnico di Torino where, in the moment I write this thesis, it is used as system to manage Exams in the Cloud.
The model we had in mind had some strict requirements about the scalability and the performance, in fact, we would introduce a system that was able to federate a lot of clusters going beyond the already existing peer to peer full-mesh architecture. For this reason we explored different brokering models that can help the environment to scale up and to be more performing, from one side, reducing the topology complexity, and from the other side that can add a layer of trust and security in a multi-tenant environment, avoiding anyway that the broker should become a single point of failure.
So we have designed and implemented a federated architecture that includes a central brokering point that enables all the federated clusters to advertise them and their resources to the ecosystem and, viceversa, by the others to discover what is already available in the federation. To reach the expected result we split our thesis work in two parts that follow two implementation methodologies: one well tested and with a high level of coding quality which aims to extend the Liqo project adding a new feature that allows its use in a multi-tenant mode, and moreover finding a way to workaround the standard peering process to enable a different

strategy for the assignment of the shared resources.
In the other hand, in a more experimental fashion, we have implemented a PoC to validate the proposed federation model and to show the potential of this work. These two approaches allowed us to be able to explore widely the proposed goal of this thesis work, and to be able to validate it in a real world scenario, in particular in the context of the European lighthouse project Structura-X.
Together with important European partners of this project, we have been able to implement a working demo of the integration of Liqo with Structura-X, and we have presented it in Paris during the Gaia-X Summit 2022. In this occasion we had the opportunity to show the prototype of the *Catalog* developed in this thesis work as part of the Structura-X project, presenting its functionalities and its capability to federate cloud providers and IXPs. The demo has been very successful, and the implemented project have received a lot of positive feedbacks.

8.1 Next steps

Following the footsteps of the European project Gaia-X and in particular of its lighthouse Structura-X we want to carry on what have been done until now to a more solid and stable version of the proposed architecture.
Next steps will involve the implementation of a more complete and robust version of the broker, that will be also able to manage the negotiation processes between different Providers, designing and implementing advanced business logics and billing models. And since the broker is the central point of the federation, it will be also the point where the security and the trust will be managed, and where the policies will be enforced, for this reason we want to make it complaint with the requirements defined in Gaia-X to enable it to be part of the *Gaia-X Federated Catalogues*. This will imply the implementation of the already defined *Trust Framework* and the capability to became a federator point for the CSPs that will be part of the federation.
In this scenario, me and the company where I work, will continue to be involved in the contribution to the Liqo project to improve and extend its functionalities.
After that we could open our envision also to the application services, making a step forward in the direction of the *Gaia-X Federated Services*, going over the only infrastructural level, giving also to Service Providers the possibility to federate their services and to offer them in this federation environment.

Bibliography

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

[2] *Virtual-kubelet git repository*. URL: https://github.com/virtual-kubelet/virtual-kubelet (cit. on pp. 4, 23).

[3] *Kubebuilder git repository*. URL: https://github.com/kubernetes-sigs/kubebuilder (cit. on p. 4).

[4] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015 (cit. on p. 4).

[5] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. «Omega: flexible, scalable schedulers for large compute clusters». In: *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 351–364. URL: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf (cit. on p. 4).

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

[7] 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).

[8] 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).

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

[10] 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).

[11] 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).

[12] 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).

[13] 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).

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