POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Enabling Multi-Provider Service Composition Among Different Kubernetes Clusters

Supervisor

Prof. Fulvio RISSO

Dott. Alessandro CANNARELLA

Candidate

Andrea COLLI-VIGNARELLI

July 2023

Summary

The digital landscape has witnessed a shift towards cloud-based computing, making cloud computing an integral part of the modern era. This transition has led to increased demand for managing computational resources in the cloud. Kubernetes has emerged as a game-changer in meeting this demand. By breaking applications into microservices, Kubernetes ensures high availability and accessibility across multiple replicas, decoupling applications from specific infrastructures and enabling workload distribution across hosts.

The Liqo project, an open-source initiative from Politecnico di Torino, complements Kubernetes by connecting disparate cloud systems. It allows the aggregation and movement of resources across various cloud environments, facilitating resource and application sharing. With Liqo, multiple Kubernetes clusters can participate, enabling an application hosted on one cluster to be utilized by another.

This innovative approach enables different Kubernetes clusters to incorporate services offered by independent providers into their ecosystems. This is particularly useful in cluster federation scenarios where entities operate independently but need to utilize third-party services while maintaining control over their data.

The objective of the present work is to establish a functional model for offering, purchasing, and deploying services across multiple Kubernetes clusters. The OpenService Broker API specifications provide a similar solution, but their application is limited to single-cluster, single-user scenarios.

To overcome these limitations, a custom catalog software was developed, enhancing and customizing the OpenService Broker API specifications. This catalog software operates as a web server, extending the REST APIs with additional endpoints for managing service purchasing and Liqo peering with specific clusters.

To ensure a secure environment, the OpenID Connect protocol was integrated, enabling a multi-user scenario within the same catalog. Additionally, an automated mechanism was incorporated to link purchased and deployed services to microservices. This streamlined approach empowers users to maximize the potential of their cloud-based operations.

Ringraziamenti

Vorrei dedicare un momento speciale per ringraziare tutte le persone incredibili che hanno reso possibile il raggiungimento di questo traguardo. Innanzitutto, il professor Fulvio Risso, il mio relatore, per avermi guidato nella tesi ed avermi trasmesso la sua passione.

Ringrazio anche TOP-IX, l'azienda che mi ha accolto per questo lavoro e il mio supervisore aziendale, Alessandro Cannarella, il quale mi ha guidato passo dopo passo nell'intero percorso.

Sicuramente, non posso lasciare da parte le persone che sono sempre state al mio fianco, quelle che mi hanno sostenuto nei momenti più belli e anche in quelli più difficili della mia vita, quelle che mi hanno ispirato e motivato a dare sempre il massimo: Mamma e Papà, grazie di essere sempre stati lì per me. Grazie di cuore anche al resto della mia famiglia, compresi Ale, Giulia e Davide, siete stati una guida e una fonte di ispirazione in tutti questi anni.

Un grazie speciale ai miei amici e coinquilini, Daniel e Pagus, che hanno condiviso con me questa avventura per anni, sapendomi regalare momenti di gioia anche negli attimi più difficili. E non posso dimenticare di ringraziare tutti i miei amici e compagni di università che mi hanno ispirato ed aiutato in questi anni, in particolare Teo, colui che mi ha accompagnato passo dopo passo in ogni avventura universitaria.

Table of Contents

List of Tables

List of Figures

Acronyms

K8s
: Kubernetes

CRD
: Custom Resource Definition

CIDR
: Classless Inter-Domain Routing

CR
: Custom Resource

HPS
: Horizontal Pod Autoscaler

DNS
: Domain Name System

mDNS
: Multicast Domain Name System

LAN
: Local Area Network

NAT
: Network Address Translation

IPAM
: IP Address Management

REST

Representational state transfer

API

Application Programming Interface

OSBAPI

Open Service Broker API

KIND

Kubernetes IN Docker

QoS

Quality of Service

OIDC

OpenID Connect

Chapter 1

Introduction

In recent years, the digital landscape has gradually transitioned towards a cloud-based computing model, making cloud computing an essential part of the modern digital era. The trend of software shifting towards cloud platforms has opened up new avenues for user interaction through any network-connected device. This evolution has amplified the demand for managing computational resources in the cloud.

Amongst potential solutions to meet this demand, Kubernetes has emerged as a game-changer. Kubernetes revolutionizes how applications are managed in the cloud by breaking them down into multiple microservices, thereby ensuring high availability and accessibility across various replicas. The beauty of this approach is that applications are no longer bound to specific infrastructure but are tethered to the cloud. Consequently, workloads can be distributed across numerous hosts and moved as needed.

Complementing this approach, the Liqo project, an open-source initiative from Politecnico di Torino, allows the aggregation and movement of resources across various cloud environments. This innovative tool connects disparate cloud systems, enabling the sharing of resources and applications. For instance, multiple Kubernetes clusters can participate, allowing an application hosted on one cluster to be utilized by another cluster via Liqo.

1.1 The issue

Starting with the Gaia-X project, the TOP-IX consortium with which this thesis work was carried out, an attempt was made to understand how in a federation of clusters, these can have useful interactions. The need for various entities to communicate between their separate clusters, ignoring the existence of each other, was thus observed. This communication is designed not only to share resources,

but also to facilitate the transfer of applications or services and microservices. This need drives the search for effective ways to connect these clusters and make this sharing of **services** possible.

The premise of this concept lies in the interaction between two primary players: customers and service providers. Customers, in search of specific services, peruse through a platform presenting a multitude of offerings from various providers, whose identities are initially unknown. Once the most fitting service is pinpointed, customers purchase it and seek immediate integration into their respective clusters.

This process necessitates a standardized procedure to craft and deliver the desired application in a cluster that the provider doesn't own. The primary concern is the communication between these different parties, but another critical issue is finding a method that permits an actor to install a service in an unowned cluster. Liqo emerges as an effective tool to indirectly facilitate this action.

The entire operation calls for standardization and simplification to ensure seamless and uncomplicated interaction between the involved parties. All the while, it must be remembered that the interaction involves parties unfamiliar to each other and sharing minimal data is of utmost importance.

There is undoubtedly a requirement for a shared platform to promote service providers and function as an intermediary, facilitating the interaction between supply and demand. Essentially, this platform would serve as a marketplace.

Finally, it is crucial to develop a secure and reliable specification, leveraging security standards tailored to a distributed model involving mutually unfamiliar entities.

1.2 Thesis structure

The thesis is structured into several chapters which are the following:

- **Chapter 2 - Kubernetes**: an overview of Kubernetes, the technology that provided the orchestration and the deployment of any application compatible in the cloud-native standards
- **Chapter 3 - Liqo**: an overview of Liqo, its main concepts and components
- **Chapter 4 - OpenService Broker API**: the introduction of the OpenService Broker API standard and specifications, its components and main concepts
- **Chapter 5 - Resolution Characteristics**: the characteristics of the resolution to the main goal of the thesis
- **Chapter 6 - Implementation**: the effective implementation developed to demonstrate the theoretical resolution

- **Chapter 7 - Measurements**: the results of the performances and resource consumption of the solution developed
- **Chapter 8 - Conclusion**: introduction to the possible next steps of the work

Chapter 2

Kubernetes

Kubernetes, colloquially known as K8s, has redefined containerization, providing a scalable platform for managing and deploying applications. It leverages a container-centric infrastructure, simplifying the complexities of development and deployment processes.

Following this introduction, we will examine K8s' architecture, networking model, components and security features in detail. By doing so, we will gain an in-depth understanding of this tool's significance in the sphere of modern application development.

2.1 History

K8s is an open-source container orchestration platform that has revolutionized the way applications are deployed and managed at scale. The history of K8s begins with Google's internal system, Borg, which was developed to manage their vast number of applications [1]. In 2014, Google released a paper on Borg, which caught the attention of the technology community and served as a basis for the development of K8s.

K8s was officially launched in June 2014 by Google, in collaboration with other major industry players like Red Hat, IBM and Microsoft, as part of the Cloud Native Computing Foundation (CNCF). The goal was to create a portable and extensible platform that would enable developers to automate the deployment, scaling and management of containerized applications across various environments.

The initial codebase of K8s was largely influenced by Google's experience with Borg, but it was redesigned to be more modular and accessible to a wider audience. The project quickly gained momentum and attracted a vibrant community of contributors, who added new features, improved scalability and enhanced security.

K8s provides a powerful set of features, including automatic scaling, service

discovery and load balancing, rolling updates and self-healing capabilities. It abstracts the underlying infrastructure and provides a declarative approach to managing applications, allowing developers to define the desired state of their applications and letting K8s handle the details of deployment and maintenance.

Since its launch, K8s has become the de facto standard for container orchestration, adopted by organizations of all sizes and industries. Its popularity can be attributed to its ability to simplify application deployment and management, increase resource utilization and enable seamless scaling of applications.

Numerous case studies and success stories highlight the benefits of K8s in real-world scenarios. For example, Spotify migrated its infrastructure to K8s, which enabled them to reduce deployment time and increase developer productivity [2]. Similarly, The New York Times adopted K8s to streamline their deployment processes and improve scalability [3].

In conclusion, K8s has emerged as a game-changer in the world of container orchestration, providing a scalable and flexible platform for managing containerized applications. Its journey from Google's internal system to a widely adopted open-source project has been marked by collaboration, innovation and community-driven development.

2.2 Kubernetes Architecture

K8s is an open-source container orchestration platform designed to automate the deployment, scaling and management of containerized applications [4]. Its architecture is based on a distributed system composed of a master node, called the Control Plane and multiple worker nodes, called simply Nodes.

2.2.1 Control Plane

The Control Plane is responsible for maintaining the desired state of the K8s cluster, such as the number of deployed replicas of an application, the network configuration and other global settings. It consists of several components that work together to achieve this goal:

- **API Server:** The Kubernetes API server is the main entry point for all administrative tasks in the cluster. It exposes the Kubernetes API, which allows users to interact with the cluster and manage its resources.

- **etcd:** A distributed key-value store that stores the configuration data of the cluster, representing the overall state of the system. etcd is used by the K8s components to persistently store and retrieve data.

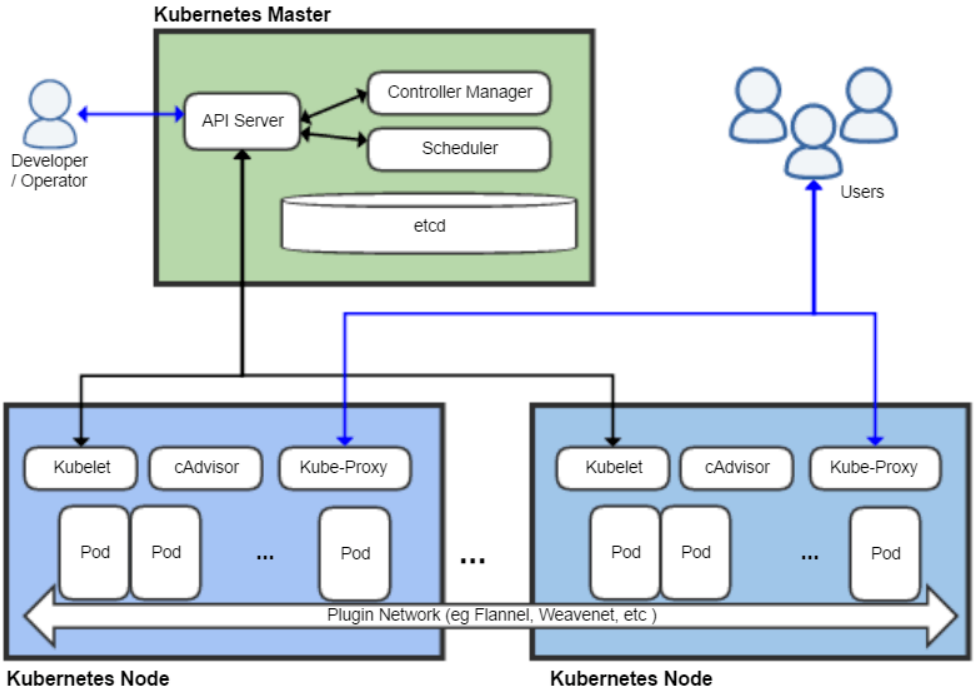

Figure 2.1: Kubernetes architecture scheme

- **Controller Manager:** A daemon that runs various controllers responsible for managing the state of the cluster. These controllers monitor the desired state of resources and take corrective actions whenever necessary .
- **Scheduler:** Responsible for allocating resources and assigning newly created Pods to Nodes based on resource availability and other constraints. The Scheduler ensures that each Pod is placed on the most suitable Node for optimal performance.

2.2.2 Nodes

Nodes are the worker machines that run containerized applications in a Kubernetes cluster. Each Node is responsible for running the containers, monitoring their health and reporting back to the Control Plane. Several key components are present on each Node:

- **Kubelet:** An agent that runs on every Node in the cluster, ensuring that containers are running in a Pod and monitoring their status. It communicates

with the Control Plane to report the status of the Node and receive instructions.

- **Container Runtime:** The software responsible for running containers on the Node. Kubernetes supports several container runtimes, including Docker, containerd and CRI-O.

- **Kube-proxy:** A network proxy that runs on each Node and maintains network rules for communication between Pods and external clients. It ensures that the network traffic is properly routed and load-balanced across the Pods in the cluster.

In summary, Kubernetes architecture relies on a distributed system comprising the Control Plane and Nodes. The Control Plane is responsible for managing the global state of the cluster, while Nodes host the containerized applications and ensure their proper execution.

2.3 Kubernetes Fundamentals

Kubernetes concepts will be the focus of this section. We will delve into the fundamental units of K8s, known as resources, as well as its internal networking structures. By doing so, we will be able to comprehend the practical operation and the rationale behind the existence of specific resources.

2.3.1 Resources

In K8s, resources are the computing units that are allocated to different components of an application. These resources can include CPU, memory, storage and network bandwidth. By properly managing and allocating resources, Kubernetes ensures optimal utilization and performance of the cluster.

K8s allows you to define the resource requirements and limits for pods, deployments and other components. Resource requirements specify the minimum amount of resources that a component needs to function properly, while limits define the maximum amount of resources that a component can consume. By setting these values appropriately, K8s can schedule and allocate resources effectively, preventing resource starvation or overutilization.

Additionally, K8s provides features like horizontal scaling and autoscaling, which allow applications to dynamically adjust their resource allocations based on demand. This ensures that resources are efficiently distributed among the running components, providing scalability and high availability.

Pods

Pods are the smallest and simplest unit in K8s. A pod represents a running process in the cluster and can contain one or more containers that share storage and network resources.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Listing 2.1: An example of a Kubernetes pod in a YAML file

Deployments

Deployments in K8s allow you to declare the containers that should be run on your cluster. By specifying the Deployment specifications, K8s can automatically handle changes and service disruptions for your containers.

```yaml
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:1.14.2
        ports:
        - containerPort: 80
```

Listing 2.2: An example of a Kubernetes deployment in a YAML file

Services

A Service in K8s is an abstraction that defines a logical set of pods and a policy to access them, sometimes referred to as a microservice. Services can be exposed in different ways by specifying a type policy in the ServiceSpec: ClusterIP, NodePort, LoadBalancer and ExternalName.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Listing 2.3: An example of a Kubernetes service in a YAML file

Secrets

A Secret is an object that holds a small amount of sensitive data, such as a password, token, or key. This information can be injected into a Pod for use by running applications/containers.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
```

```
15      secret:
16        secretName: mysecret
17        optional: true
```

Listing 2.4: An example of a Kubernetes secret in a YAML file

Namespaces

Namespaces provide a way to divide cluster resources among multiple users (via resource quotas). In terms of development environments, a namespace can be considered as a virtual environment that segregates application resources, users and environments.

2.3.2 Networking

Networking in K8s plays a crucial role in ensuring seamless connectivity and reliable communication between containers, services and other resources. In this section, we will explore how networking works in K8s, the different types of networks involved and how they interact with each other.

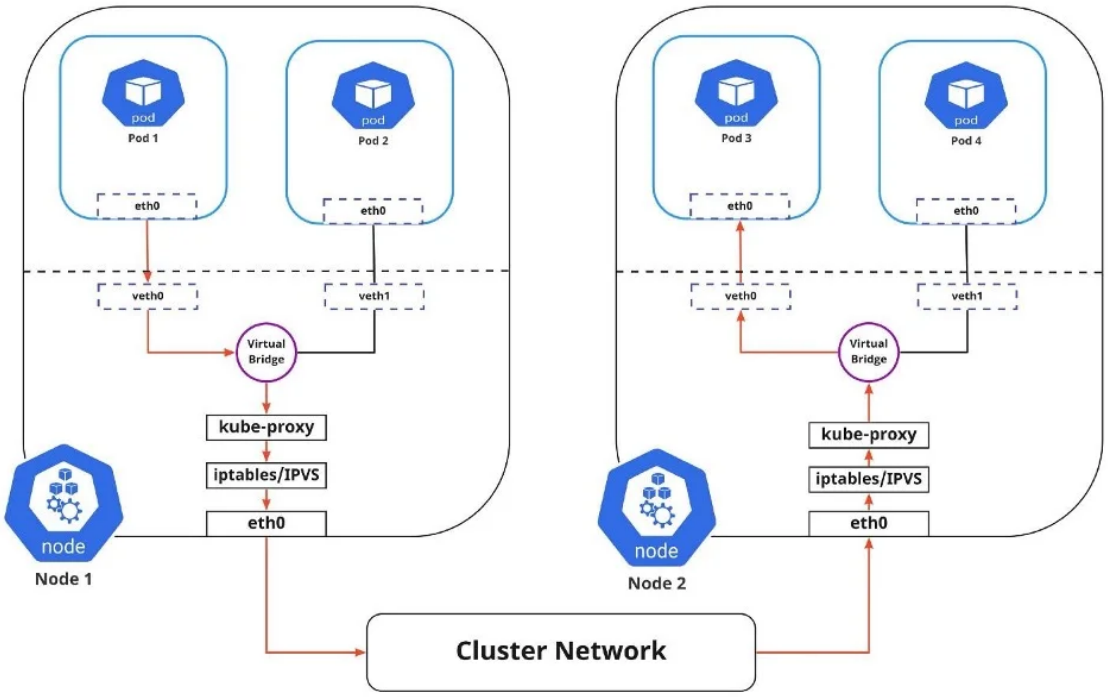

Figure 2.2: Communication between pods in different nodes

Cluster Network

The cluster network in K8s is responsible for communication between various nodes within the cluster. It allows pods running on different nodes to reach each other. The cluster network is typically implemented using a network plugin or a container network interface (CNI) plugin. Some popular network plugins for K8s include Calico, Flannel, Weave and Cilium [5, 6, 7, 8]. These plugins provide networking features such as IP address management, network isolation and routing.

Pod Network

Pods are the basic units of deployment in K8s and each pod has its own unique IP address. The pod network facilitates communication between pods running on the same node or different nodes within the cluster. It allows pods to discover and communicate with each other using their IP addresses. Pod networks are usually implemented using overlay networks, which encapsulate pod traffic within virtual networks.

Service Network

Services in K8s provide a stable endpoint for accessing a set of pods. The service network allows communication between services and pods. When a service is created, K8s assigns it a unique IP address. This IP address is used as a stable endpoint to access the service, regardless of the underlying pods' IP addresses. K8s uses a service discovery mechanism to dynamically route traffic to the pods associated with a service.

Network Policies

Network policies in K8s provide fine-grained control over network traffic within the cluster. They allow administrators to define rules that govern which pods can communicate with each other and the types of traffic that are allowed or denied. Network policies operate at the pod level and are enforced by the network plugin or CNI plugin used in the cluster.

DNS Resolution

K8s provides a built-in DNS service that enables pods to discover other services and pods using their DNS names. Each service created in K8s is assigned a DNS name, which can be resolved to the corresponding service IP address. This DNS resolution mechanism simplifies service discovery and allows pods to communicate with each other using meaningful names rather than IP addresses [9].

These are some of the key aspects of networking in K8s. Understanding how the different networks in K8s interact and how they can be configured and secured is essential for building scalable and resilient applications on the platform.

2.4 Custom Resources, CRDs and Operators

In the realm of container orchestration, Kubernetes has emerged as the undisputed leader, offering an array of features that handle the deployment, scaling and management of containerized applications. One of the strengths of Kubernetes lies in its extensibility, a core aspect of which is the use of Custom Resources, CustomResourceDefinitions (CRDs) and Operators. This section delves into these three key concepts, their purpose and the interaction among them in extending the Kubernetes API.

2.4.1 Custom Resources in Kubernetes

Custom Resources in Kubernetes provide an extension mechanism that allows you to define your own resources, which the Kubernetes API server can then handle [10]. By leveraging this mechanism, developers can model and manage their applications according to their needs, without requiring modifications to the core Kubernetes codebase.

Custom resources are a powerful and flexible feature that come into play when the built-in resources of Kubernetes are insufficient to implement the desired functionality. For instance, a developer might create a custom resource to represent an application's components, with attributes such as versioning or replication that are not covered by built-in resources.

2.4.2 CustomResourceDefinitions (CRDs)

While Custom Resources represent the instances of a new resource type, CustomResourceDefinitions (CRDs) define the schema and characteristics of these new resource types. CRDs are a particularly powerful feature in Kubernetes, enabling developers to declare new resource types with the same degree of freedom as the built-in types.

Once a CRD is created, the Kubernetes API server starts serving the defined new resource. This new resource can then be managed through kubectl, just like built-in resource types. The CRD mechanism provides a rich ecosystem for extending Kubernetes, allowing for greater adaptability to various application deployment scenarios.

2.4.3 Kubernetes Operators

Operators in Kubernetes are a design pattern meant to manage complex, stateful applications. They are built using Custom Resources and leverage the inherent Kubernetes mechanisms to manage services and perform automated tasks.

An Operator extends Kubernetes to automate the management of the entire lifecycle of a particular application, service, or component. It encapsulates the operational knowledge typically provided by human operators into software, enabling Kubernetes to automatically handle tasks such as deployment, scaling, upgrades, backups and recovery [11].

2.5 Security

Kubernetes places considerable emphasis on security, offering mechanisms to control and manage access to resources within a cluster. This section delves deeper into two vital elements of Kubernetes security: Service Accounts and Role-Based Access Control (RBAC) via Roles and ClusterRoles.

2.5.1 Service account

Service accounts in Kubernetes signify a specialized type of account meant for processes, such as pods, that are running inside the Kubernetes cluster. This distinguishes them from regular user accounts, which are intended for human users. Service accounts are namespaced, implying that they exist and can be assigned to pods within a specific namespace. If a pod doesn't explicitly associate with a service account, the 'default' service account of its namespace is automatically assigned to it at creation time.

The significance of service accounts extends to managing permissions and controlling access within a Kubernetes cluster. These accounts play a vital role in providing identity for applications running within the cluster, thus determining what these applications can or cannot do. For instance, a service account could be used to control an application's permissions to read from or write to a Kubernetes API. It is through service accounts that applications can authenticate to the API server and consequently, the API server can both authenticate the incoming requests and enforce policies that limit the actions that can be performed by different applications.

More specifically, service accounts are tied with secrets that contain credentials for API authentication. These secrets can be automatically mounted into the pods at specific paths, allowing applications running inside the pods to use them to interact securely with the API server. This forms a fundamental part of Kubernetes security

architecture, ensuring that only authorized entities can access and manipulate the cluster's state [12].

2.5.2 Role and ClusterRole

Role-Based Access Control (RBAC) is the method used by Kubernetes to regulate the permissions of different entities, including users, groups and service accounts within the cluster. This is achieved through two Kubernetes objects: Role and ClusterRole [13].

A Role is used to grant permissions to resources within a particular namespace. A Role can outline which operations are allowed (such as get, list, create, update and delete) on which resources within the namespace. These resources can encompass a wide variety of Kubernetes objects, including pods, services, secrets and others. It is important to note that Roles grant permissions to perform actions only within the same namespace, limiting the potential for unauthorized access or actions across different parts of the cluster.

ClusterRole, conversely, works at the cluster level, applying across all namespaces. A ClusterRole can grant the same permissions as a Role but, as it is not restricted to a specific namespace, it can also provide access to cluster-scoped resources (like nodes) or non-resource endpoints (such as '/healthz'). This can be particularly useful for defining permissions for cluster-wide administrative tasks or for applications that need to interact with resources in multiple namespaces or with the cluster itself.

RoleBinding and ClusterRoleBinding are used respectively with Role and ClusterRole to bind those permissions to certain subjects. These bindings connect the set of permissions defined in a Role or ClusterRole with one or more subjects, enabling fine-grained control over who or what can perform which actions within the cluster.

2.6 Managing Load and Scaling

One of the prominent benefits of a cluster environment is its capacity for effective load management. This capability becomes particularly valuable when an application starts to expand in scope and user requests begin to surge. Under such circumstances, the single application might start to require more resources. The typical response to this situation is scaling, which is an umbrella term encompassing a couple of different strategies, namely, vertical and horizontal scaling [14].

2.6.1 Vertical and Horizontal Scaling

Vertical scaling refers to an approach where resources are increased on the single instance of an application. Essentially, the amount of assigned CPUs or memory is increased until it reaches the maximum quantity available on the node. However, this strategy may have its limitations, especially when resources are constrained or the application demands exceed the node's capacity.

Horizontal scaling, on the other hand, emerges as a more viable solution in a cloud environment. Instead of focusing on a single instance of the application, the cloud orchestrator initiates and manages multiple instances of the same application, effectively distributing the load amongst them. This strategy significantly enhances the system's scalability and resilience.

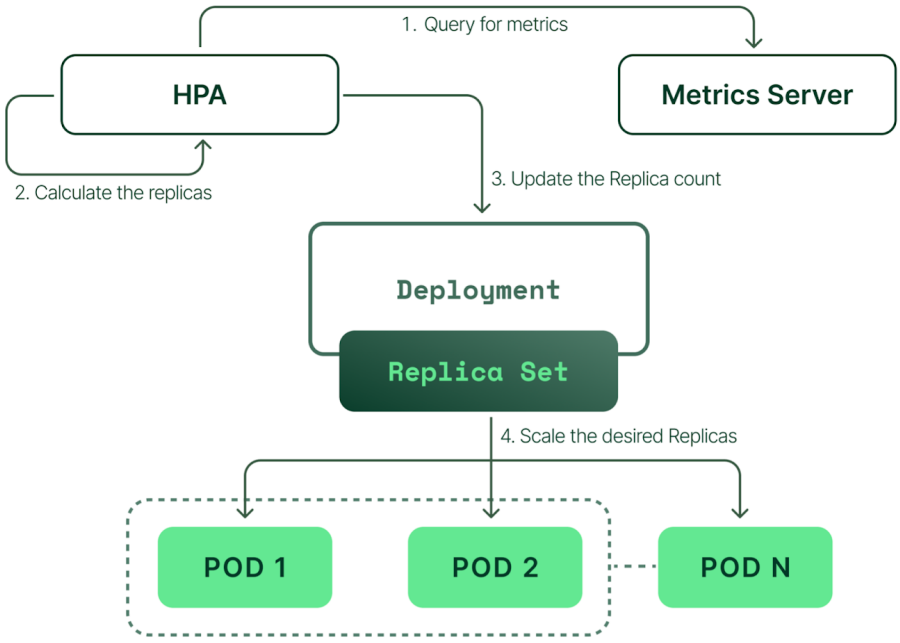

Figure 2.3: Horizontal Pod Autoscaler schema

2.6.2 The Concept of Replicas

At the heart of the horizontal scaling model lies the concept of replicas. The application that needs to be scaled is "replicated" into multiple instances, effectively

amplifying its processing capabilities. The strength of this model in a cluster environment is the ability to distribute the replicas across different nodes, rather than being restricted to the resources of a single node.

In the Kubernetes platform, for example, the individual replica is a concept encapsulated within the "Replica set" resource. This resource, in turn, is a part of the "Deployment" resource (as mentioned previously, 2.3.1). This replica set can be established with a fixed starting amount and a maximum limit of replicas. The scaling process can be handled manually, or it can be automated via a mechanism called a horizontal pod autoscaler (HPA).

The Horizontal Pod Autoscaler (HPA) is designed to respond to an increasing load on a deployment by scaling the load across new replicas. Kubernetes' scheduler will allocate these replicas across the most suitable nodes. It's possible to set specific limits for the deployment regarding CPU and memory usage. Once these limits are surpassed, the HPA will take over and scale the deployment across additional replicas.

2.6.3 Microservices Management

Modern web applications tend to be designed using a microservice architecture. This consists of a collection of microservices, ideally stateless, each responsible for handling smaller, more specific tasks. This model aligns with the "Divide and Conquer" strategy. Instead of one large monolithic system, a collection of services collaboratively completes tasks, passing necessary information from one to another [15].

In the context of Kubernetes, this model translates to multiple pods that communicate with each other. Each individual microservice can be encapsulated within a containerized application inside a Kubernetes pod. Kubernetes' service resource offers a single entry point that can be accessed by different pods within the cluster. Various microservices can be deployed into separate Kubernetes namespaces. Thanks to Kubernetes' integrated DNS, it's straightforward to reach services within different namespaces.

Chapter 3

Liqo

In this chapter, we introduce Liqo, an innovative open source project that was born out of the Politecnico di Torino. This unique project paves the way for Kubernetes to share resources and services smoothly and securely. We will be delving into its architectural components, shedding light on some key features that were pivotal in the formulation of this thesis.

3.1 Introduction

In the world of Kubernetes clusters, there exists an inherent variability in computing load. It fluctuates, characterized by peaks and lows that depend on several factors, such as the time of day, the demands of the business and other determinants. As a consequence, these clusters are often provisioned with an abundance of computing resources to ensure that full utilization can be achieved during peak demand periods.

However, a significant drawback is the emergence of spare resources that remain untapped and unutilized. These resources, unfortunately, cannot be leveraged by the cluster and could potentially be shared with other organizations who are in need.

This is where Liqo comes into play. Inspired by the concept of liquid computing, Liqo seeks to interconnect clusters, allowing them to share computing resources and services amongst themselves. The outcome is what we refer to as "opportunistic data centers". In these centers, clusters can offer their unused resources at any given time, reducing infrastructure costs for their peers and in turn, opening new possibilities in the realm of edge computing.

The underlying modus operandi of Liqo is the well-established paradigm of peering. This paradigm permits a diverse array of topologies, whether centralized or decentralized. What this means is that individual clusters, at the most basic

level, maintain absolute control over the resources they choose to share and the entities with whom they decide to share.

One of the notable attributes of Liqo is its capability to expand the standard Kubernetes APIs in a manner that remains transparent to applications and, to a certain extent, Kubernetes administrators. Indeed, as we will demonstrate, the resources that were detailed in Chapter 2 are not only applicable in this new environment but are often enhanced to suit the specific objectives of Liqo. As a consequence, user applications need not undergo any alterations to operate in conjunction with Liqo.

3.2 Concepts and Fundamental Mechanics

This section delves into the core principles and mechanics that underpin the functionality of the Liqo project. The two fundamental aspects to be discussed include the Discovery and Peering concepts that underscore the Liqo project's operation.

3.2.1 Discovery

The Discovery aspect of Liqo forms the bedrock of the intercommunication capability between clusters [16]. Clusters can be discovered through several methods. The most straightforward method is the manual addition of clusters via their IP addresses. This method allows users complete control over the process, though it might require a higher degree of technical knowledge and effort.

However, Liqo aims for convenience and user-friendly experiences as well. It achieves this by advertising its presence over a local network through the use of Multicast DNS (mDNS). This feature is particularly useful in setting up a Liqo federation in a Local Area Network (LAN) environment as it automates the discovery process.

Yet another method Liqo employs for discovery is through Domain Name System (DNS) records that specify the cluster IPs for a specific domain. This method is particularly relevant in scenarios where an organization manages multiple clusters, which may be provisioned dynamically.

Regardless of the method employed for cluster discovery, the end objective is the creation of a custom resource, referred to as ForeignCluster, within the local cluster. This ForeignCluster resource symbolizes the remote cluster and holds essential information about it, forming a crucial part of the discovery process.

3.2.2 Peering

The Peering model is another fundamental pillar of Liqo. As mentioned in the introduction, Liqo utilizes this model to delineate and manage the relationships between distinct, administratively separate clusters.

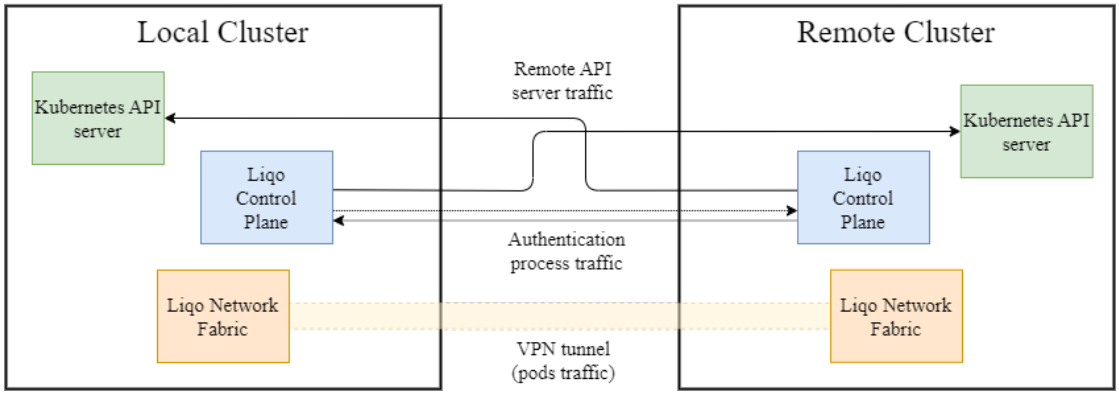

Figure 3.1: Liqo peering scheme

Peering is a process that establishes a connection between two clusters, each with a different role: one requesting resources and the other offering them. This mechanism comes into play after the discovery process, leveraging the IP endpoint found in the preceding stage. The peering process can be subdivided into three essential steps:

- *Authentication*: In this step, clusters validate each other's identity. This ensures security and trust between the clusters.
- *Networking*: Here, the clusters discover each other's IP ranges and configure Network Address Translation (NAT) rules. This forms the foundation of inter-cluster communication.
- *Resource Sharing*: This final step is where clusters communicate the quantity and type of resources they wish to exchange. This process underlies the core functionality of Liqo.

Authentication

The initial step in the peering process is the validation of identities of the clusters. In this step, the clusters verify each other's identity to ensure the security and integrity of the subsequent processes. This is crucial to prevent unauthorized access and to maintain the integrity of the data and resources that will be shared.

Networking

Following successful authentication, the clusters proceed with the discovery of each other's IP ranges and the configuration of Network Address Translation (NAT) rules. NAT rules help in routing the packets correctly between the different clusters, thereby maintaining an efficient communication network.

The foundation for this process is the Kubernetes networking model. In this 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.

The IPs assigned from these CIDRs are unique within the cluster and reachable from every node within the cluster. However, this model, designed for a single cluster, faces challenges when used in a setup with multiple clusters. This is because there is no guarantee that the pod CIDR of one cluster does not overlap with that of its peers. To address this issue, Liqo uses Network Address Translation as part of the peering process. The IP Address Management (IPAM) 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.

Resource Sharing

The final stage of peering is the determination of resources to be shared between the clusters. For this, Liqo implements a request-response model. In this model, the consumer cluster requests a list of resources (a ResourceRequest) and the provider cluster responds with an offer for a certain amount (a ResourceOffer).

At present, it is not possible to ask for specific resources, implying that only generic ResourceRequests can be sent. However, despite this limitation, the model ensures that the right amount of resources can be provisioned, maintaining the overall efficiency and performance of the federated cloud platform.

Service Offloading

With the establishment of a Liqo federation, Kubernetes services created in one cluster become accessible to pods in the peer cluster. This mechanism is due to a unique feature of Liqo known as 'service offloading'.

In service offloading, when a service is created within a Kubernetes namespace that is offloaded on a remote cluster, a "shadow" copy of the service is also created on the remote cluster. Thus, when a pod in the remote cluster wishes to communicate with the service, it communicates with the "shadow" service. This communication is then forwarded to the original service in the source cluster, effectively enabling

communication between the pod and the service, even though they are in different clusters.

This offloading feature essentially extends the reach of services beyond the confines of a single cluster and into the federated clusters, thereby further enhancing the interoperability and seamless integration capabilities of the Liqo system.

3.3 Main components

In the architecture of Liqo, several key components enable its unique functionality of connecting disparate Kubernetes clusters. This section delves into the technical specifics of these major components: the Virtual Kubelet, the Foreign Cluster, the Virtual Node and Namespace Offloading. Each of these plays a crucial role in providing the seamless experience of federated clusters.

3.3.1 Virtual Kubelet

The Virtual Kubelet is a core component of the Liqo system. Acting as a "virtual node," the Virtual Kubelet, by implementing the Kubelet API, masquerades as a node in the Kubernetes system. This allows the Virtual Kubelet to perform operations in the cluster as if it were a node, including creating, managing and deleting pods and other resources. This gives Liqo the ability to control the cluster resources in a granular and efficient manner, increasing the overall utility of the federated clusters.

It's crucial to note that while the Virtual Kubelet presents itself as a node to the Kubernetes API server, it doesn't manage any physical machine itself. Instead, it delegates this task to other software components, enabling an additional layer of abstraction and versatility in handling resources.

3.3.2 Foreign Cluster

The Foreign Cluster component in Liqo serves as a representation of a remote cluster in a local cluster's context. It holds the necessary information about the remote cluster, which the local cluster requires to establish a connection and maintain communication. This includes the network details of the remote cluster, the resources it offers and the status of the peering relationship between the two clusters.

A Foreign Cluster resource is automatically created in the local cluster during the discovery process. The resource is then used during the peering process to establish and maintain the connection between the local and remote clusters. It provides a single point of reference for the remote cluster, facilitating the effective management of federated resources.

3.3.3 Virtual Node

In the context of Liqo, a Virtual Node is a representation of a remote cluster in the local cluster's context. Each Virtual Node is associated with a Foreign Cluster resource and provides a node-like abstraction for the remote cluster's resources in the local cluster. This allows the local cluster's scheduler to schedule pods on the remote cluster, as if it were scheduling them on a node of its own.

A Virtual Node is created during the peering process and serves as a bridge between the local and remote clusters. It translates the local cluster's commands into actions in the remote cluster, providing seamless interaction and resource management across the federated clusters.

3.3.4 Namespace Offloading

Namespace Offloading is a process in Liqo that enables the transparent execution of pods in remote clusters. When a namespace is marked for offloading, Liqo ensures that any pod scheduled in that namespace is executed in one of the connected remote clusters.

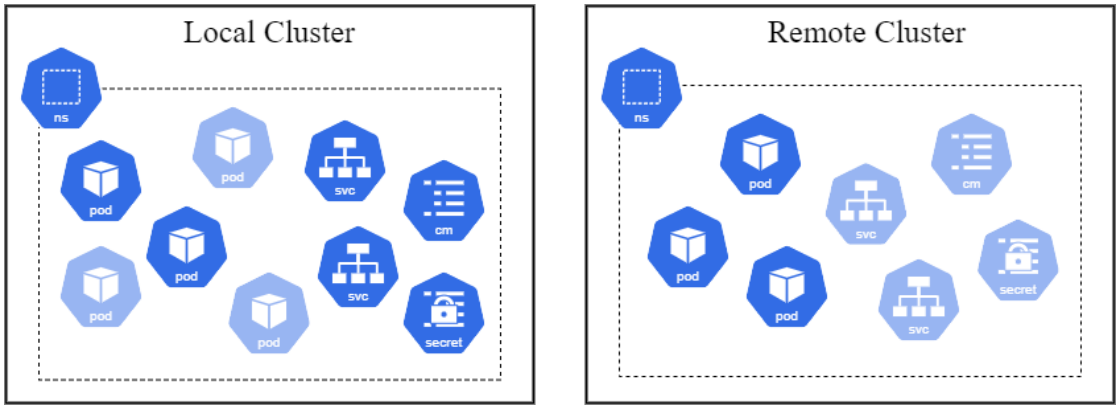

Figure 3.2: Namespace offloading scheme between local and remote clusters

This mechanism enables transparent scalability and resource optimization across the federated clusters. Pods can be offloaded to clusters that have more available resources, thereby enhancing the efficiency and performance of the overall system. This feature, combined with Liqo's ability to dynamically discover and peer with clusters, provides a powerful tool for creating a truly distributed and scalable Kubernetes environment.

Chapter 4

Open Service Broker API

This chapter provides an in-depth exploration of the Open Service Broker API (OSBAPI), a fundamental protocol that served as the foundational model for the development of the thesis solution. Although highly functional in its original form, it was found that certain adaptations of this protocol were necessary to adequately cater to the specific requirements of our work. Throughout this chapter, we will discuss the details of OSBAPI, its essential concepts, operations, use cases and the modifications made to customize it for our unique context.

4.1 Overview

The Open Service Broker API (OSBAPI) is an open and standardized protocol for managing services in the marketplace of a cloud platform [17]. Originally designed by the Cloud Foundry community, it has been widely adopted by many platforms and service providers due to its platform agnosticism. It provides a unified way of delivering services to applications regardless of the underlying cloud platform.

The core concept of OSBAPI is to simplify the process of delivering services to applications running in the cloud. By providing a standardized interface between cloud platforms and service providers, OSBAPI allows developers to consume services without needing to understand details about how services are provisioned and managed.

The API is built around a set of well-defined **objects**, such as *services*, *service plans* and *service instances*, as well as **operations** on these objects, like *provision*, *bind*, *unbind* and *deprovision*.

- **Services** are the software or infrastructure components that applications can consume. Examples of services include databases, message queues and other backend APIs.

- **Service Plans** define different tiers of a service, such as the difference between a free tier and a premium tier of a database service.
- **Service Instances** are provisioned plans that applications can consume. An instance represents a reserved resource on the service for the application.

In the next sections, we will delve deeper into the specifications, use cases and core concepts of the Open Service Broker API.

4.2 Specifications

The Open Service Broker API is designed around RESTful principles, with resources and methods defined for provisioning, deprovisioning, binding and unbinding service instances [17]. The API makes use of HTTP methods such as *GET*, *PUT*, *DELETE* and *PATCH* to interact with these resources. These endpoints are all implemented inside a REST API server called **service broker**.

The main resources involved in the API are:

- **Catalogs**: Collections of services that a service broker offers.
- **Service Instances**: Representations of provisioned services that applications can use.
- **Service Bindings**: Representations of the credentials and connection details an application needs to use a service instance.

To start the provisioning of a service instance, the API consumer sends a *PUT* request to the service instance resource. This request includes details of the chosen service plan and any parameters required for provisioning. The service broker then provisions the service instance and returns a response. This response can either be synchronous, providing the status of the operation immediately, or asynchronous, providing a way for the API consumer to poll for the status of the operation [18].

Binding a service instance is similar, with a *PUT* request being sent to the service binding resource. The service broker returns the connection details and credentials needed to use the service instance.

For deprovisioning and unbinding, *DELETE* requests are used. It is also possible to update a service instance using a *PATCH* request.

The API also includes error handling mechanisms, with standardized HTTP status codes and error messages being returned when things go wrong.

This flexible and extensible architecture of the Open Service Broker API allowed us to adapt it to the specific needs of our work. By extending the API, we were able to introduce new operations and resources, providing greater control and flexibility over service management.

Here we provide a detailed list of the API endpoints:

- **/readyz**: This endpoint is used by the platform to fetch the status of the server.

  ```
  GET /readyz
  ```

- **/v2/catalog**: This endpoint is used by the platform to fetch the service catalog of the broker. It is a *GET* request that returns a list of all services and their plans that the broker offers.

  ```
  GET /v2/catalog
  ```

- **/v2/service_instances/{instance_id}**: This endpoint is used for provisioning (*PUT*), updating (*PATCH*) and deprovisioning (*DELETE*) service instances. The *instance_id* is a unique identifier provided by the platform for the instance to be provisioned.

  ```
  PUT /v2/service_instances/{instance_id}
  ```

- **/v2/service_instances/{instance_id}/service_bindings/{binding_id}**: This endpoint is used for creating (*PUT*) and deleting (*DELETE*) service bindings. The *binding_id* is a unique identifier provided by the platform for the binding to be created. The *PUT* request returns the connection details and credentials needed to use the service instance.

  ```
  PUT /v2/service_instances/{instance_id}/service_bindings/{binding_id}
  ```

- **/v2/service_instances/{instance_id}/last_operation**: This endpoint is used for getting (*GET*) the status of a specific operation. The operation id to insert in the query URL is retrieved from a previous PUT call on service instance or service binding endpoint.

  ```
  GET /v2/service_instances/{instance_id}/last_operation
  ```

4.3 Use Cases

There are numerous use cases for the Open Service Broker API, particularly in environments where services need to be provisioned and consumed in a standardized and controlled way [19].

One primary use case is in Platform as a Service (PaaS) environments, like Cloud Foundry and Kubernetes. Here, developers deploy applications that need to consume services like databases, messaging systems, or other APIs. Using the Open Service Broker API, these services can be provisioned and managed in a standardized way across different cloud environments, improving portability and reducing vendor lock-in.

Another use case is in providing Software as a Service (SaaS) offerings. A SaaS provider can implement the Open Service Broker API to allow their customers to provision and manage instances of the SaaS offering. This can be integrated into the customer's existing service management infrastructure, providing a seamless user experience.

In a multi-cloud environment, the Open Service Broker API provides a standardized way of provisioning and managing services across different cloud providers. This can simplify operations and improve interoperability in multi-cloud deployments.

For our specific work, we utilized and adapted the Open Service Broker API to manage the provisioning and consumption of services in a complex, multi-component architecture. By extending the API, we were able to implement specific controls and operations needed for our use case, demonstrating the flexibility and extensibility of the API.

4.4 Concepts

The Open Service Broker API is built around several key concepts that are essential to understand for effective use of the API. These concepts include the Catalog, Service and Plan.

4.4.1 Catalog

The Catalog is a fundamental concept in the Open Service Broker API. It represents a comprehensive list of all the services that a service broker can provide [18]. The catalog acts as a central repository of service offerings, allowing platform users to browse and select the services they need for their applications.

Each service listed in the catalog is characterized by a set of attributes. These attributes typically include a name, a unique ID, a description and a list of service plans. The catalog should be designed in a self-descriptive manner, providing clear

and concise information about each service. It serves as a crucial resource for platforms to understand the capabilities and offerings of a particular service broker.

The catalog is fetched by making a *GET* request to the */v2/catalog* endpoint.

4.4.2 Service

In the Open Service Broker API, a Service represents a manageable piece of software that can be provisioned and bound to an application [18]. It is an abstract representation of a functional component that an application may depend on, such as a database, a message queue, an API gateway, or any other service that provides specific functionalities.

A Service is uniquely identified within the service broker's catalog and offers a range of plans. Each plan corresponds to a specific configuration or tier of the service, allowing users to choose the most suitable option for their application requirements. Services can also be associated with tags, facilitating categorization and discovery.

When a service instance is provisioned, the service broker allocates the necessary resources and provides the application with a URL or other relevant access information. Binding a service instance enables the application to securely access the service by providing the required credentials.

4.4.3 Plan

A Plan represents a particular tier or configuration offered by a service in the catalog [18]. It enables users to select the desired level of resources, features, or pricing options associated with a specific service.

Plans are defined within the catalog and are uniquely identified. Each plan has a name, a description and potentially additional metadata, such as cost or trial period information. By offering different plans, service brokers provide users with flexibility and choice in tailoring their service usage according to their specific needs.

When requesting the provisioning of a service instance, users must specify the ID of the desired plan. This information allows the service broker to provision the instance with the appropriate configuration and resource allocation.

Understanding the concepts of Catalog, Service and Plan is crucial for effectively utilizing the Open Service Broker API. These concepts form the foundation of the API's service management capabilities, providing a structured and standardized approach for offering, provisioning and managing services.

By leveraging the power of these concepts, platforms can streamline the integration and consumption of services, facilitating seamless interoperability and enhancing the overall user experience.

4.5 Operations

The Open Service Broker API defines a set of operations that enable the management and lifecycle of services and their instances. These operations provide a standardized way to interact with the service broker and enable seamless integration with various platforms.

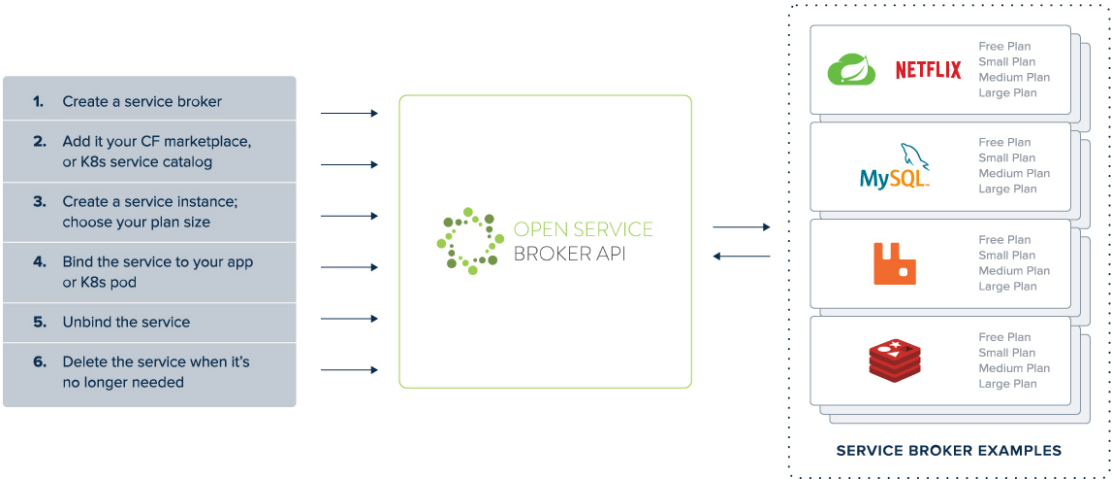

Figure 4.1: General Open Service Broker API operating schema

4.5.1 Provisioning

Provisioning is the process of creating and allocating resources for a service instance. It involves requesting the service broker to instantiate a new instance of a particular service and configure it according to the specified plan and parameters. The provisioning operation typically requires the following information:

- **Service Details**: The unique ID of the service from the catalog, the desired plan ID and any additional parameters required for provisioning.
- **Contextual Information**: Contextual details such as the organization, space and user requesting the provisioning operation.

Once the provisioning request is initiated, the service broker handles the necessary steps to create and configure the service instance. The response may include the provisioned service instance details, including connection information and any additional metadata.

4.5.2 Updating

The updating operation allows for modifying the configuration and settings of a service instance. It enables users to adapt the service instance based on changing requirements or evolving business needs. The updating operation typically involves providing new configuration details or parameters for the service instance.

Users can send a request to the service broker with the updated information, including the unique identifier of the service instance to be modified and the desired changes. The service broker processes the request and updates the service instance accordingly. The response may include confirmation of the successful update and any relevant updated information.

4.5.3 Deprovisioning

Deprovisioning refers to the process of removing a service instance and releasing the associated resources. It is the reverse operation of provisioning and involves cleaning up and deallocating any resources allocated to the service instance. Deprovisioning can be triggered by a user or an automated process.

To deprovision a service instance, users send a request to the service broker with the unique identifier of the instance to be deprovisioned. The service broker handles the deprovisioning process, ensuring the release of resources and cleaning up any associated artifacts. The response confirms the successful deprovisioning of the service instance.

4.5.4 Binding

Binding is the process of establishing a connection between a service instance and an application. It enables the application to access the resources and functionalities provided by the service. During the binding operation, the service broker generates credentials and connection details specific to the bound application.

To bind a service instance, users send a request to the service broker, specifying the unique identifiers of both the service instance and the application. The service broker generates and provides the necessary credentials, such as usernames, passwords, or access tokens, along with any connection details or configuration specific to the bound application.

4.5.5 Unbinding

Unbinding refers to the process of disconnecting an application from a previously bound service instance. It revokes the credentials and connection details associated with the application's access to the service. Unbinding can be triggered by a user or an automated process.

Users send a request to the service broker with the unique identifiers of the service instance and the application to be unbound. The service broker handles the unbinding process, invalidating the credentials and revoking the application's access to the service. The response confirms the successful unbinding of the application from the service instance.

Understanding and utilizing these operations empowers users to manage the lifecycle of service instances effectively. The Open Service Broker API provides a standardized way to perform these operations, ensuring consistency, interoperability and ease of integration with various platforms and service brokers.

Chapter 5

Design

This chapter aims to provide a comprehensive and technical examination of the problem previously introduced, proposing different design ideas and solutions. We will delve into the details of the problem, further breaking it down into sub-problems for more focused examination.

5.1 Contextual Overview

In a typical cluster environment, an application is composed of multiple components that work in unison. These include databases, front-end interfaces, message brokers and more. All these components can be considered as microservices, functioning as integrated services within a larger application structure.

However, there may be instances where hosting or deploying a service is not feasible or desirable. This could be due to various reasons: limited computational resources, excessive costs, or the proprietary nature of a service which the provider wants to maintain control over. Therefore, there arises a necessity for a service which, while not residing in your cluster environment, remains under your partial control.

In this complex network, the same service may be utilized by multiple applications, or a single service may leverage other services provided by different entities. While this chain of interdependencies can potentially extend indefinitely, the fundamental challenge lies in integrating an external component within your environment and application - a component over which direct control may not be possible and whose availability is subject to specific commercial agreements.

5.2 Potential Use Case

This context sets the stage for a potential use case. Imagine a user, representing either a major corporation or a fledgling startup, operating their own cluster. The user develops a web application that requires a database for data storage. Now, the user is faced with a few choices. They could:

- Deploy a database within their own cluster, thus utilizing their existing resources,
- Opt for a serverless solution, which would entail sending all data to a third-party database with limited visibility on the data processing operations.

Another scenario presents itself where a service provider offers a proprietary database (possibly with unique optimizations or features) for a fee. Following a purchase agreement, this database can be deployed directly within the user's cluster. However, the provider wishes to retain a certain level of control over the service offered, while also granting the user visibility into the service's operation. This situation forms the crux of the use case we aim to resolve.

5.3 Overview of the Challenges

In the course of our comprehensive exploration and through the illustration of a particularly pragmatic use-case scenario, a series of formidable obstacles requiring our attention and effort have crystallized. The challenges that stand out and need to be addressed are as follows:

- Developing a sophisticated, yet user-friendly, mechanism that effectively facilitates the purchase and sale of services within the ecosystem. This not only involves the transaction itself but also includes aspects such as service discovery, comparison and selection.
- Building a robust, trustworthy framework that retains complete transparency over the services sold by the service provider. This transparency is a crucial element that boosts user trust and confidence in the system. Alongside this, we must also consider the user's vested interest in understanding how their data is being handled, stored and processed, thereby ensuring privacy and data security.
- Devising an efficient protocol that allows seamless interaction and communication between the provider's cluster and the consumer's cluster. This area presents a triad of critical sub-issues that need special focus:

  - **Security** - Our communication protocol must prioritize the security of data transmission, guaranteeing that the exchanged information is safe from potential breaches or attacks.
  - **Reliability** - A reliable system is a trustworthy one. Hence, our communication protocol should be built to ensure consistent performance, stability and dependability in the communication process.
  - **Latency and Speed** - In the era of high-speed communication, delays can be costly. Therefore, our protocol must ensure efficient and swift transmission of data with minimal latency.

- Ensuring that the procured service can interact smoothly, seamlessly and with minimal interference or intervention with the user's application. This essentially involves developing standards, interfaces and procedures that can integrate the new service within the existing user application framework without causing disruption or requiring extensive user interaction.

The aforementioned challenges, having been identified and outlined, will be dissected and examined in greater detail in the following sections of this document.

5.4 Mechanisms of Transaction

Our proposed model must integrate a mechanism that can effectively associate an application with a user, an action that occurs subsequent to a successful transaction. This association is fundamental in ensuring the correct delivery and utilization of the procured services. It is important to underline that our model operates within a commercial setting, where services are traded, often in a competitive environment. The pricing of these services is a critical aspect and is typically determined by the provider.

The pricing strategy can be influenced by numerous factors such as market demand, cost of provision and competitive landscape. It could potentially be a usage-based cost, predicated on measurable parameters like data traffic or the number of service API calls. Alternatively, it could adopt a subscription model, constituting a fixed cost linked to time, such as a monthly or an annual subscription. Or, it might just be a one-time, flat initial cost, often seen in software licensing scenarios.

To facilitate the user's purchase decision, the service that the user is interested in procuring must be adequately advertised and made known to the potential user. For this, we require a system that can disseminate relevant information about the impending application purchase. This may include details about the price, available plans, configuration information, user testimonials and so on.

Essentially, this would require the development of a comprehensive service catalogue that not only lists the available services but also presents associated information in a clear, concise and accessible manner. This catalogue would play a vital role in guiding user decision-making and hence, must be designed with a user-centric approach.

It is not enough to merely provide the necessary functionalities and information; the manner in which they are presented and accessed by the user is equally important. Our aim should be to deliver all the discussed elements through a user-friendly, intuitive and engaging user experience.

Indeed, the development of an interface that can effectively showcase all offered services, their features, benefits and pricing would be the ideal scenario. This interface should aim to simplify the decision-making process for the user, making it easier to compare, select and purchase the services.

Our ultimate vision should be to create a dynamic platform that serves as a bridge between supply and demand, effectively connecting providers with potential consumers. This platform, if executed well, can not only facilitate service transactions but also play a critical role in shaping user perceptions, fostering relationships and driving user loyalty towards the service providers and the platform itself.

5.5 The Essentiality of Control over Services and Data

In the digital era, data has emerged as a critical asset, often regarded as the lifeblood of businesses. The operational model under discussion is situated within a complex interplay of two independent entities. These entities, though agnostic of each other's internal workings, are tied together by a shared business interest. They aim to establish a mutually beneficial business relationship, each striving to minimize the disclosure of proprietary information and maintain the highest degree of control over their respective assets.

To create a conducive environment for this exchange, a nuanced understanding of both entities' perspectives is imperative. The task at hand involves reconciling fundamentally contrasting viewpoints. Arriving at a model that satisfies both parties is a complex negotiation process that necessitates a willingness to accommodate and sometimes relinquish certain demands.

5.5.1 Ownership of Services: A Provider's Perspective

Envision a service provider with a state-of-the-art application ready to make a mark in the competitive market landscape. This application, potentially a carefully crafted and protected product, could house proprietary knowledge or

unique functionalities that set it apart from competitors. Such information becomes accessible to the user only after a successful purchase operation.

To safeguard the proprietary knowledge and the competitive advantage that it offers, the service provider would ideally want to keep the application's codebase and its operational intricacies confidential. The user is expected to interact with the application's functionalities without encroaching upon its inner workings. This ensures the provider's intellectual property remains secure, while users still derive value from the application.

However, the service provider's control needs extend beyond mere intellectual property protection. They often have technical requirements that necessitate maintaining a certain degree of authority over the service even post-sale. These requirements could be driven by factors such as the need to guarantee Quality of Service (QoS), ensure consistent application availability, or maintain the application's overall operational efficiency. Such control over the deployed environment is crucial for the provider to deliver a seamless user experience, manage updates and mitigate any potential issues, thereby translating to an indirect form of control over the service.

In an ideal world, the application remains completely under the control of its creator and provider even after being sold. The customer's interaction with the service is confined to predefined and approved touchpoints for data input, preventing unauthorized access or tampering.

5.5.2 Data Privacy: A User's Perspective

Switching perspectives, consider a customer or user who decides to leverage a cluster service procured from a third party. The act of utilizing the service implicitly involves entrusting their data to the third-party service for processing. The service must accept this data input and process it as per the user's needs.

However, data privacy concerns often surface when the data in question is highly sensitive. It could be personally identifiable information, financial data, or any information the user does not wish to be stored or managed by third parties, let alone be transmitted beyond their control. In an age where data breaches are unfortunately commonplace, these concerns are far from unfounded.

In such situations, the user would ideally prefer to have absolute control over the procured service. The service should operate within an environment that allows the customer to oversee and control all operations being performed. This ensures that the user's data doesn't disappear into a technological 'black box', but is processed transparently and in accordance with the user's wishes. This not only mitigates the risk of unauthorized data access but also ensures the user remains in the driver's seat when it comes to their data.

5.6 Communication and Deployment

The progression of current technological advancements is undeniably shifting towards a distributed model. The emphasis within this framework rests notably on effective and efficient communication strategies. One of the key challenges herein lies in the necessity for the consumer and producer to identify a safe and reliable mode of interaction.

Consider the Internet, an omnipresent yet potentially perilous medium for such interaction. Here, the question is not whether it can be used, but rather, how to harness its potential while ensuring secure communication. Given this dilemma, it becomes prudent to consider an arrangement akin to a *Virtual Private Network* (VPN).

A VPN offers the allure of secure communication, even within an insecure medium. It cleverly mimics a local network environment, allowing two entities to interact as though they were physically connected on the same local network. This setup provides a significant layer of security, but it does not eliminate the need for security protocols during communication.

In fact, these protocols should be implemented even outside the VPN when necessary and it's highly recommended within it. Establishing such practices aids in reinforcing the security of the communication, thereby mitigating any potential risks.

An additional matter to consider in the context of deployment methodology pertains to the connection between the purchased service and the application intended to use it. The conceivable scenarios are numerous, each with its unique implications. Ideally, placing both service and application within the same cluster significantly simplifies communication. Yet, even if they reside in different clusters, a connection between the two entities is feasible. However, in such instances, the need for a secure form of communication, as discussed previously, becomes even more critical.

Given the above considerations, a range of potential scenarios emerges for thorough analysis. These are not merely for defining a communication model, but also for striking a balance between the stated requirements.

1. The use of VPN: A system resembling a VPN can create a pseudo-local network that offers increased security. However, this is not a foolproof solution and additional security measures are necessary.

2. The implementation of security protocols: These should be used both within and outside the VPN as required to further enhance communication security.

3. Interconnecting service and application: Ideally, these should reside within the same cluster. However, secure communication protocols must be in place

if they are in different clusters.

These considerations highlight the complex nature of the task at hand and the need for a comprehensive analysis to establish a viable model for secure and efficient communication and deployment methodology.

5.6.1 The Customer-centric Model

This model places the customer at the heart of the service deployment process. Two clusters are involved in this model: one belonging to the service provider and another that is owned by the customer who purchases the service.

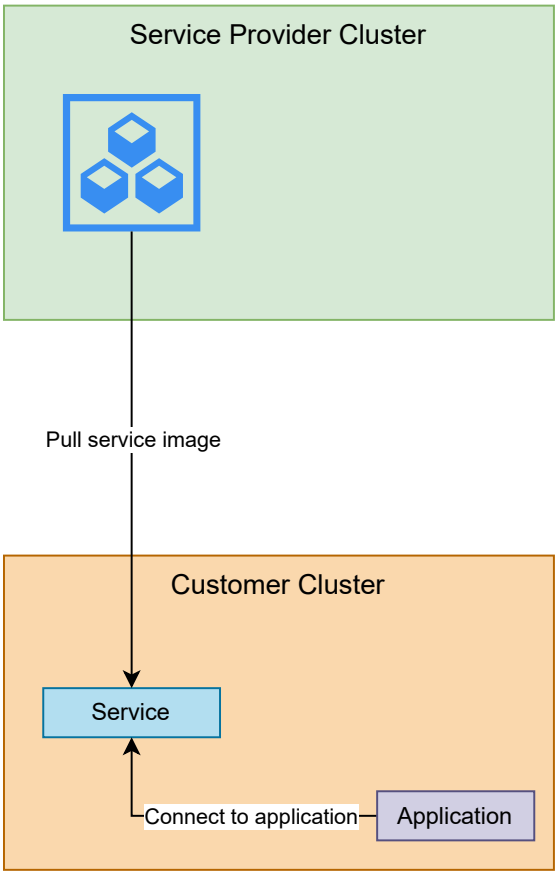

Figure 5.1: Schema of the customer-centric model

Service Deployment and Hosting

Upon purchasing, the customer is entrusted with creating and hosting the service in their cluster. They acquire an 'image' of the service from the provider. This

image is essentially a snapshot of the service that can be loaded onto the customer's system. Only authorised customers, who have completed a purchase, can acquire this image. After acquisition, the customer is responsible for instantiating this image in their environment, essentially creating a working copy of the service on their own cluster.

This model requires the customer to prepare their environment to host the purchased service. This preparation is primarily related to the resources that the service will consume. For instance, if the purchased service is a microservice such as a database, the required computational resources may be quite manageable. However, more complex services, such as machine learning software, may necessitate a much larger and more capable hosting environment, which may not be feasible for all customers.

Service Connection

Beyond hosting, the customer is also in charge of connecting the instantiated service with their application. This connection process is a manual one, as the exact location and creation method of the service cannot be known in advance. This further places the responsibility for successful service integration on the customer.

Role of the Provider

From the provider's perspective, the responsibilities are considerably less under this model. The provider's primary task is to ensure that only authorised users can obtain a copy of the purchased service. One potential method to achieve this is via a Docker repository that contains an image of the service. While the provider may offer some level of support, their ability to do so is limited, as they do not have control over the customer's hosting environment and hence cannot rectify all potential issues.

Despite these restrictions, the provider can attempt to provide as much support as possible, even if they cannot control all the aspects that may lead to potential malfunctions. However, a significant amount of the responsibility for successfully deploying and running the service lies with the customer.

Pros and Cons

This customer-centric model has several advantages and disadvantages for both parties, but the primary focus is the customer.

The main benefits involve data privacy and secure communication. Since the customer directly creates the service within their cluster, communication between their application and the service is secure. The customer maintains full control over their data, as all communication occurs within the customer's cluster. Furthermore,

	PROS	CONS
Secure connection & communication	Traffic is all inside the customer cluster	
Control over service ownership		Service provider after pull operation loses all the control
Control over private data	Data remain within the customer cluster	
Service guarantees	Can be ensure with typical K8s techniques (HPA, replicas, load balancer, etc)	Difficulties to determine service problems if not working (i.e. firewall outside service provider control)

Table 5.1: Summary of pros and cons of the customer-centric model

if there are concerns about the service sending data externally, the customer can restrict the service's connectivity to their cluster only.

However, there are also certain downsides to this model. Once the service software is transmitted, the provider loses visibility into how it's used, meaning they cannot understand or analyse the usage patterns of the service. Furthermore, given the lack of knowledge about the customer's environment, the provider's ability to offer support for the service is severely limited. All prerequisites required by the software may not be available or met in the customer's environment. Consequently, the success of the service largely depends on the customer's resources and abilities.

In conclusion, while this model grants the customer greater control over their data and enhances the security of communication, it also places a greater responsibility on them to host, deploy and connect the service successfully. Thus, it may be more suitable for technically adept customers with sufficient resources. On the other hand, providers have lesser responsibilities but also face limitations in providing comprehensive support and gaining insights into the service usage.

5.6.2 The Service Provider-centric Model

In the second model under consideration, we again have two distinct entities: the service provider's cluster and the customer's cluster. This model presents a different arrangement, in that the purchased service resides exclusively within the provider's cluster.

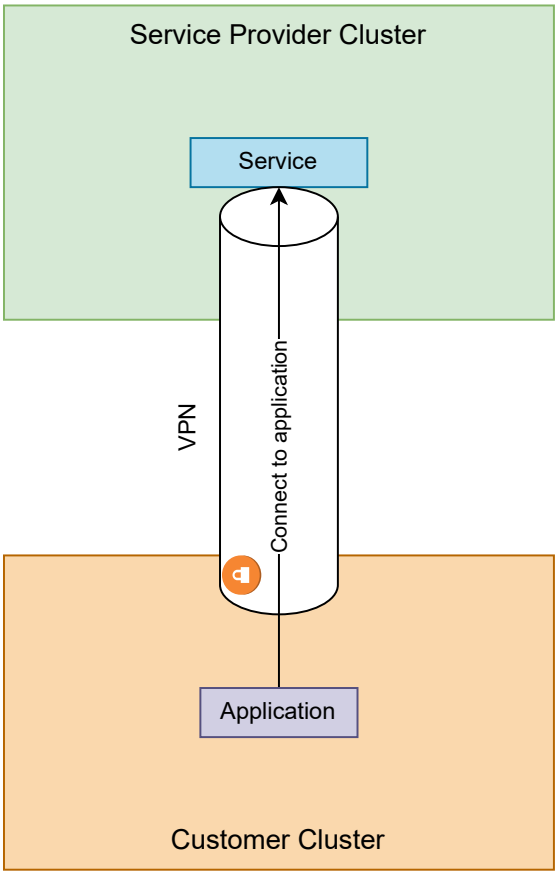

Figure 5.2: Schema of the service provider-model

Service Access and Communication

Once the service has been purchased, the customer gains 'access' to it. However, this access differs fundamentally from that of the customer-centric model. Instead of receiving an image to instantiate on their own cluster, the customer now interacts with the service as it resides on the provider's cluster.

This arrangement brings to the fore a crucial issue: communication. From the customer's perspective, the service they wish to utilise is situated in an unfamiliar environment. More importantly, this environment is different from the one in which their application resides. It is therefore imperative to establish a secure communication channel, possibly a **VPN**, between the customer's application and the service. The responsibility for the creation of this VPN rests with both parties, but it is particularly crucial for the service provider to ensure the integrity of the communication channel.

Provider Responsibilities

The service provider, in this model, finds themselves in complete control of the provided software. This control, however, comes with its own set of responsibilities. The provider must establish secure and controlled access to the service. In terms of firewall configuration, it becomes critical to expose the service to the customer but only through the VPN communication channel established earlier.

Moreover, any issues related to the service, such as quality of service (QoS), availability and the installation of prerequisites for the proper functioning of the software, fall squarely on the service provider's shoulders. As the service resides entirely under the provider's control, they bear full responsibility for its smooth operation.

Impact on Provider and Customer

This model effectively flips the roles seen in the customer-centric model. The burden that once fell upon the customer now rests with the provider, including the resource consumption of hosting the service software. As each sale may correspond to a deployment of a dedicated service for each user, the hosting burden could become unsustainable as the number of users grows over time. This could lead to considerable consumption of the provider's computational resources.

The advantages and disadvantages of this model are pronounced. On the one hand, the provider gains full control over the service, enabling them to ensure its optimal functioning. On the other hand, the situation becomes darker from the customer's perspective.

Privacy and Connectivity Concerns

Given that the customer's data needs to be sent to the provider's service for processing, the service might appear as a 'blackbox' to the customer. As such, the customer relinquishes control over their data, which could lead to potential privacy concerns.

Additionally, connectivity and communication complications may arise. A VPN between the two clusters could secure the communication and with appropriate tools, satisfactory performance could be achieved. However, the fact that the service is located on a cluster at the other end of the VPN from the customer implies that the communication channel is always active. As this channel operates outside the customer's cluster, outgoing traffic could be a concern. If traffic is heavy, the associated costs could pose a significant disadvantage for the customer.

In summary, this service provider-centric model offers the advantage of total control over the service for the provider, leading to potential quality improvements. However, it increases the responsibilities of the provider and could pose privacy and

	PROS	CONS
Secure connection & communication	Traffic is inside a VPN	Traffic to service is external the cluster (possible traffic cost problems)
Control over service ownership	Service provider completely owns and controls the service, all inside its cluster. It can be created as it wants.	
Control over private data		Data are sent to service provider's cluster, no control
Service guarantees	Service provider guarantees with its standard the service and its availability	

Table 5.2: Summary of pros and cons of the service provider-centric model

communication challenges for the customer. Hence, the suitability of this model could vary depending on the specific needs and capabilities of the customers and providers.

5.6.3 The Hybrid Model: A Comprehensive Approach

Two common models have been analysed: the customer-side and service provider-side models. Each model has its distinct advantages and challenges. However, neither provides an all-encompassing solution, hence the exploration of a third model that merges the benefits of both. This final proposal aims to establish a hybrid model, a balanced paradigm that provides a certain degree of control over the service to the provider and concurrently maintains transparency for the customer regarding their data usage.

Utilising Liqo for Seamless Integration

In order to facilitate the seamless operation of this hybrid model, we turn to a technology that has shown immense potential in bridging the gap between disparate clusters: **Liqo**. This innovative tool enables communication between two distinct

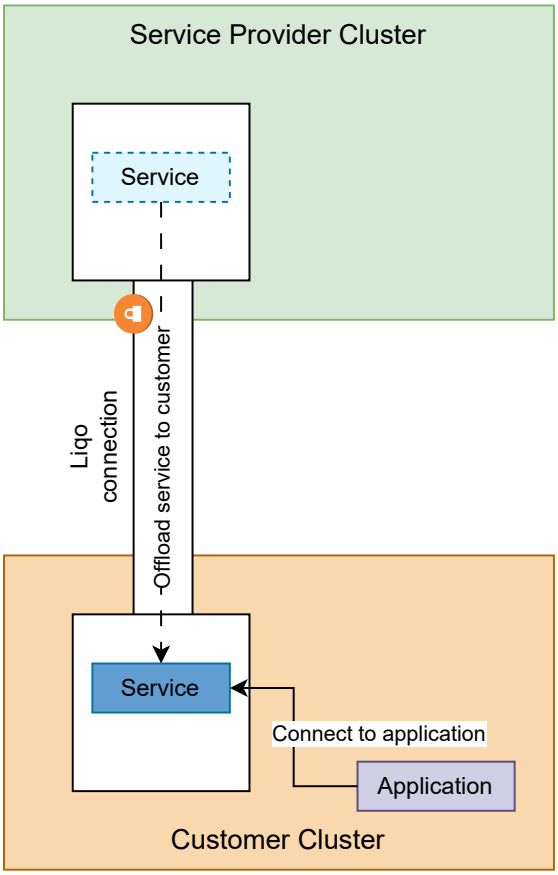

Figure 5.3: Schema of the hybrid model

clusters - in our case, the customer's and service provider's. Specifically, the service provider's cluster initiates a peering operation towards the client's cluster. This creates a secure and effective communication channel, inherent in Liqo technology.

Within this secure environment, the service provider creates a namespace within its cluster. This namespace essentially serves as an isolated environment for the provider to deploy the service. Importantly, Liqo offers the distinct capability for the provider to delegate the resource consumption that is typically associated with the namespace to the client's cluster. With the correct policies in place, the service provider's cluster generates the service, but the actual instantiation and resource consumption occurs within the client's cluster.

Balancing Resource Management and Service Accessibility

This hybrid approach, although promising, is not without its potential drawbacks. A key consideration is the sustainability of significant resource consumption on the

client side. Liqo's unique technology, however, offers flexibility in adjusting the distribution of resource consumption between the client and the service provider. Despite the physical location of resource consumption, the service is designed to remain accessible as if it were within the client's cluster. This negates the need for creating or maintaining an additional secure connection, as the secure connection is intrinsic to the Liqo communication channel.

Navigating Control and Visibility

The balancing act within this hybrid model extends beyond resource consumption. There is also a balance to be found in the control and visibility of the service. On the one hand, the service remains under the provider's control and can be created and manipulated as if it were within its cluster. This alleviates the provider's concerns around prerequisites or other requirements on the client side. Moreover, the provider maintains the capability to discontinue the service for the user as they see fit, with Liqo recognising the service as owned by another cluster.

From the client's perspective, this hybrid model provides an essential measure of visibility and control. By executing the service within their cluster, customers are granted a clear view of its various components. Theoretically, this should ensure that data remains within their cluster. However, this is not an entirely perfect solution, as any resources that cannot be instantiated within the customer's cluster might necessitate a transfer of data outside the customer's purview.

Assuring Quality and Embracing Scalability

A notable advantage of this hybrid model is its adherence to the quality standards set by the provider. By leveraging the powerful capabilities of K8s and its scheduler, the service can utilise inbuilt tools such as replicas, the HPA and more. This not only ensures a robust, scalable service but also allows the provider to maintain quality in accordance with their standards.

In summary, the hybrid model offers a balanced pathway for both customers and providers, merging control, visibility, resource management and quality assurance into a single paradigm. Although not without potential challenges, it represents a significant step towards achieving a flexible, balanced and effective service provision model that meets the evolving demands of customers and providers alike.

5.7 Liqo: An Equitable Selection

After extensive and meticulous analysis of all three presented models, the final selection was made in favor of the third model, known as Liqo. This decision was made with a profound understanding that every solution comes with its unique set

	PROS	CONS
Secure connection & communication	Traffic is all inside the customer cluster Liqo VPN for cluster-to-cluster communication	
Control over service ownership	Service provider keeps control over service resources visible to customer	
Control over private data	Data should remain inside customer cluster	Some data may leak outside the control of customer. Smart network analyser needed to enforce this requirement
Service guarantees	Can be ensured with typical K8s techniques (HPA, replicas, load balancer, etc.)	Difficulties to determine service problems if not working (i.e. firewall outside service provider control)

Table 5.3: Summary of pros and cons of the hybrid model

of pros and cons. It is inherently impossible to devise a solution that caters solely to the benefits of both parties without making certain compromises.

Therefore, rather than adhering strictly to one end of the spectrum, an intermediate approach was adopted – a balanced compromise which neither tilts extremely towards one party nor the other. The crux of the matter lies in finding a solution that upholds the needs and interests of both the user and the service provider in an equitable manner.

The resulting solution, hybrid through the use of Liqo, is a model that does not undermine either party's importance. It balances two pivotal aspects: firstly, the value of the user's data and secondly, the service provider's rights and ownership over the service.

- **Customer-centric approach**: Liqo offers a transparent environment where

users have full visibility of their purchased service. Users can monitor the service in real-time, understanding how their data is being utilized and can also observe the effects and results of the service.

- **Service provider-oriented approach**: Concurrently, Liqo offers service providers the ability to maintain an oversight of the service that they have sold, without entirely relinquishing control. Providers can track usage, optimize service delivery and respond swiftly to issues, even while the user maintains a degree of autonomy.

In conclusion, Liqo harmoniously merges the benefits derived from both user-centric and service provider-centric models, delivering a comprehensive solution that caters to the needs of both parties effectively.

5.8 Service to Application Binding Automation

In the conventional model we propose, we postulate that a client procures a service intending to leverage it within its proprietary application. This application could, for instance, be a web application requiring interaction with a database (the procured service). A sequence of information becomes necessary for effective communication between these two entities. As the scenario is described currently, the client manually executes this step, primarily focusing on identifying the information requisite for service connection. Upon extraction, this data is input into the application to establish a connection.

The primary ambition of this thesis is to automate, to the greatest extent possible, the process described above. The underlying assumption here is that we possess a service, potentially deployed within the client's cluster and a client application desiring to exploit it. In isolation, these software entities are agnostic of each other, implying a conspicuous need for an intermediary that can establish a connection between them. This intermediary can provide the application with the necessary guidance on where and how to establish a connection with the intended service.

However, this approach is not without complications:

- Not all services can connect in a standard manner.
- More importantly, not every application requires the same set of information to establish a connection.

One plausible solution to these challenges is to have a shared resource accessible from both sides, for instance, a *K8s Secret*. By maintaining the connection

information within this shared resource, the client application can reference it, potentially even before the actual service is created.

Expanding on this idea, the process of integrating the service with the application can be broken down into several distinct steps:

1. The client identifies the necessary connection information for the service.
2. This information is extracted and input into the application to facilitate a connection.
3. An intermediary is introduced that knows how and where to instruct the application to connect to the service.
4. Connection details are stored in a shared resource, like a *K8s Secret*.
5. The client application references the shared resource, even possibly before the service is actually instantiated.

This method aims to automate the process of binding services to applications, thereby minimizing the manual effort required and ensuring a seamless integration process. While the practical implementation might encounter certain complications, addressing them could pave the way for a more streamlined and efficient interaction between applications and services.

Chapter 6

Implementation

This chapter is dedicated to the practical application of the theoretical model previously discussed. It aims to delve into the tangible aspects of the proposed solution, which hitherto has been dissected only from an analytical standpoint. This materialisation of theory into practice is far from being a mere juxtaposition of elements, but a systematized assemblage informed by the concepts and hypotheses discussed until now.

The system at hand is distributed in nature, a feature that endows it with unique characteristics and challenges. To fully understand this construct, it is essential to scrutinize its constituent parts and their mutual interaction.

- **Component Description:** Each component of the system will be defined in detail, providing an in-depth understanding of their function and relevance within the larger construct.
- **Interactions:** The system's components do not exist in isolation. Thus, their interaction will be detailed, exploring the effect of various interactions on the overall system behaviour.
- **System Behaviour:** By observing the outcome of these interactions, we can infer the system's behavioural traits, enabling us to understand the overall system dynamics.

Further, as the system is fortified with security protocols, the complexities regarding its integration are worth mentioning. This includes the technical nuances and challenges faced during the implementation of these security measures.

In conclusion, this chapter serves to translate theory into practice, bringing into focus the actual dynamics of the proposed solution. A comprehensive understanding of these aspects would provide valuable insights into the practical implications and challenges of implementing such a distributed system.

6.1 Elements of the System

The foundational assumption of the system under study is the existence of at least two autonomous and independent entities: a service provider and a customer. These entities, distinguished by their unique roles and completely agnostic of each other's internal operations, each possess a minimum of one Kubernetes (K8s) cluster. These clusters are capable of interfacing with the wider Internet and serve as the critical environments where the applications that are central to our study are deployed. These applications consist of the service on the provider's end and the client application on the customer's end. In an effort to simplify the development and testing process during this research study, these K8s clusters were substituted with more manageable single node clusters implemented using Kubernetes IN Docker (KIND).

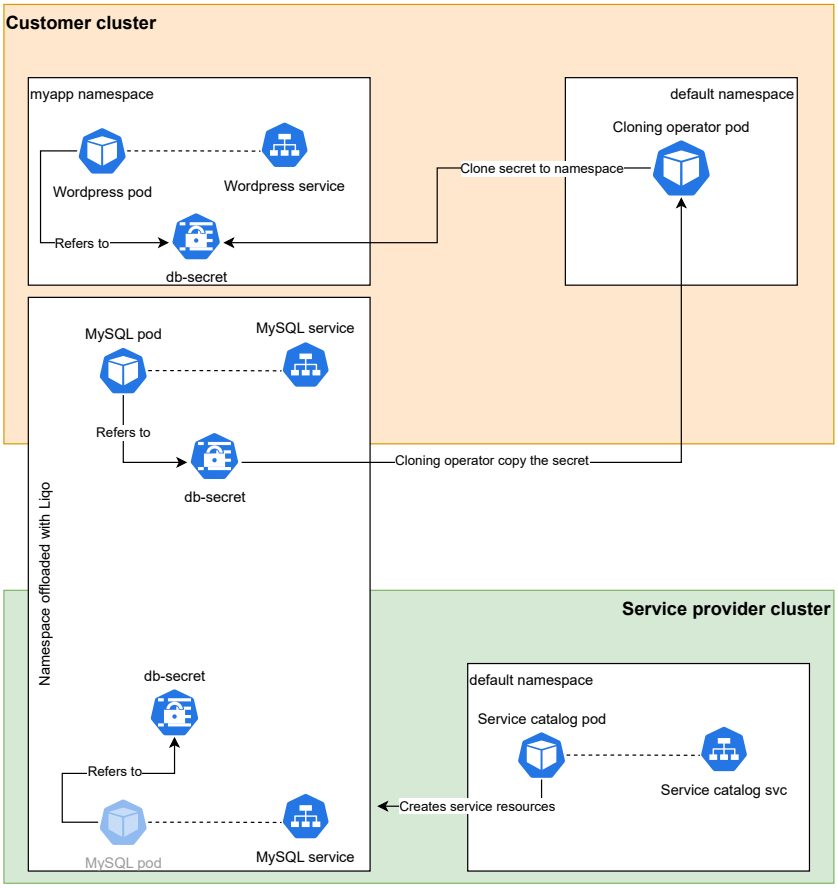

Figure 6.1: Schema of all the elements involved in the implementation and their main interactions

6.1.1 Customer Elements

In more specific terms, the K8s cluster belonging to the customer contains a namespace. This namespace serves as a designated environment within which the customer deploys its proprietary application. For the purposes of this discussion, this application is referred to as "myapp". At the outset, "myapp" is designed to be dependent on a service that it will purchase at a later time. Despite this future dependency, the application is capable of referencing this yet-to-be-acquired service by setting the necessary credentials via a secret. This secret, however, is yet to be instantiated, which adds a layer of complexity to the initial deployment process.

The deployment of "myapp" is conducted within a K8s Pod. Once deployed, this pod may be exposed to the wider Internet via a K8s Service for example. During this deployment and exposure process, the Pod enters an error state due to its inability to retrieve the necessary credentials from the non-existent secret. While this may appear to be an anomaly at first glance, this behavior is fully anticipated and indeed desired as it provides valuable diagnostic information. The resultant error condition is defined as ***CreateContainerConfigError*** in the system terminology.

In addition to the namespace containing "myapp", the customer's K8s cluster also contains a secondary namespace which houses a specific component referred to as the "Cloning operator". This operator is a K8s operator that is designed to perform a very specific task: to automatically copy certain secrets within the system. We will delve deeper into the workings and implications of this operator later in the discussion.

6.1.2 Service Provider Elements

Transitioning over to the service provider side, a separate K8s cluster exists. This cluster hosts a critical system component known as the "Catalog server". The Catalog server is the central figure in the creation of the desired service. It provides the interface through which peering with Liqo is established, enabling the purchase and deployment of services.

The Catalog server is implemented within a K8s Pod. To ensure seamless interaction with all the components and actors involved in the system, it is crucial that the Catalog server be accessible remotely. Specifically, it must be reachable from the Internet, which necessitates the exposure of the Catalog server through a K8s Service.

6.1.3 Common Elements: Liqo

A key component common to both the service provider and customer clusters is Liqo. Liqo plays a crucial role in the system's operation as it facilitates the peering

between the service provider's cluster and the customer's cluster when needed.

Upon successful installation of Liqo and establishment of peering, a bridging element between the two clusters is created. This element is a specifically designed namespace, created by the service provider in its cluster and accompanied by the addition of the Liqo NamespaceOffloading. This resource must report with a specific selection of the client cluster, providing a bridge for interactions between the two parties.

The implemented model of this system envisions a unique peering for each client. Consequently, for every service chosen by a client to be deployed, a unique K8s Namespace and the corresponding NamespaceOffloading are created by the service provider, ensuring a tailored service delivery for each client.

6.2 Catalog Server

The Catalog Server is a critical component in the system, responsible for managing and coordinating all operations. It serves as the central hub, leveraging the OSBAPI protocol integrated within the Kubernetes (K8s) environment. By combining the power of the OSBAPI protocol with the flexibility of K8s, the Catalog Server provides a robust and efficient solution for creating and managing services within a K8s cluster.

6.2.1 Deployment within the K8s Cluster

To ensure seamless integration with the K8s cluster, the Catalog Server is deployed as a pod within the cluster. It is exposed through a K8s service, allowing it to communicate with other components in the cluster. By utilizing a dedicated library, the Catalog Server can interact with the K8s API, enabling it to perform various operations within the cluster. These operations include creating, updating and deleting services, as well as managing the lifecycle of service instances and service bindings. In order to execute these actions, the Catalog Server requires appropriate permissions, which are granted through the deployment of a K8s Service Account and associated ClusterRoles. These roles define the level of access and control that the Catalog Server has over the K8s cluster, ensuring that it can perform its duties effectively.

6.2.2 A CRD for configuration: ServiceBrokerConfig

The configuration of the Catalog Server is defined using the ServiceBrokerConfig custom resource, which is created based on the corresponding Custom Resource Definition (CRD). This resource encapsulates the settings and parameters that govern the behavior of the Catalog Server.

Catalog Configuration

One of the key aspects of the ServiceBrokerConfig resource is the catalog configuration. The catalog represents a collection of services that are available for provisioning within the K8s cluster. Each service in the catalog is accompanied by a set of plans, which define different configurations or offerings for that particular service. The catalog configuration within the ServiceBrokerConfig resource includes details such as the service ID, plan ID and other metadata associated with each service and plan. This information is essential for users to understand and select the appropriate service and plan during the provisioning process.

The catalog configuration also allows for the specification of parameters required for each service and plan. In particular, parameters for the creation of the service instance and for the service binding may be required and inserted inside the service broker configuration file.

```
serviceBinding:
 create:
   parameters:
     $schema: 'http://json-schema.org/draft-07/schema#'
     type: object
     required:
       - User name
       - Database credentials secret name
       - Destination namespace
     properties:
       Root password:
         description: Administrator password
         type: string
       User name:
         description: User name
         type: string
       User password:
         description: User password
         type: string
       Database name:
         description: Database name
         type: string
       Port:
         description: Port
         type: integer
       Database credentials secret name:
         description: >-
           Name of the secret that will contain the database
    credentials.
         type: string
```

```
30          maxLength: 253
31          pattern: '^[a-z0-9][a-z0-9-.]*[a-z0-9]$'
32        Destination namespace:
33          description: >-
34            Namespace where client application is running.
35          type: string
```

Listing 6.1: Example of parameters schema for service binding creation

These parameters define the customizable options that users can choose when provisioning a service. For example, a database service may have parameters such as storage size, replication factor, or backup frequency. The ServiceBrokerConfig resource captures these parameters and their corresponding values, allowing for dynamic and flexible service provisioning.

Template Specification

Another crucial aspect of the ServiceBrokerConfig resource is the template specification. Templates are used to define the resources and configurations required for each service and plan. They provide a standardized way to create and manage the necessary K8s resources, such as Deployments, Services, ConfigMaps and Secrets. Templates can be customized to suit the specific requirements of each service and plan. They allow for the dynamic configuration of resource properties, such as the number of replicas, resource limits and volume mounts. By utilizing templates, the Catalog Server ensures consistent and reproducible deployment of services within the K8s cluster.

```
templates:
  - name: database-deployment
    singleton: true
    template:
      apiVersion: apps/v1
      kind: Deployment
      metadata:
        name: '{{ registry "instance-id" }}'
      spec:
        replicas: 1
        selector:
          matchLabels:
            app: '{{ registry "instance-id" }}'
        template:
          metadata:
            labels:
              app: '{{ registry "instance-id" }}'
          spec:
```

```
19              containers:
20                - name: '{{ registry "instance-id" }}'
21                  image: 'mysql:5.6'
22                  env:
23                    - name: MYSQL_ROOT_PASSWORD
24                      valueFrom:
25                        secretKeyRef:
26                          name: '{{ registry "secret-name" }}'
27                          key: MYSQL_ROOT_PASSWORD
28                    - name: MYSQL_USER
29                      valueFrom:
30                        secretKeyRef:
31                          name: '{{ registry "secret-name" }}'
32                          key: MYSQL_USER
33                    - name: MYSQL_PASSWORD
34                      valueFrom:
35                        secretKeyRef:
36                          name: '{{ registry "secret-name" }}'
37                          key: MYSQL_PASSWORD
38                    - name: MYSQL_DATABASE
39                      valueFrom:
40                        secretKeyRef:
41                          name: '{{ registry "secret-name" }}'
42                          key: MYSQL_DATABASE
43                  ports:
44                    - containerPort: 3306
45                      name: mysql
```

Listing 6.2: Example template for MySQL deployment

The template specification within the ServiceBrokerConfig resource includes YAML representations of the resources to be created. It defines the structure, properties and relationships of these resources. For example, a template for a database service may include a Deployment resource, a Service resource and a ConfigMap resource. The template specifies the desired configuration for each resource, including labels, selectors, ports and environment variables. By using templates, the Catalog Server simplifies the provisioning process and ensures consistent deployment across different services and plans.

Bindings and Registry

The ServiceBrokerConfig resource also includes the bindings configuration. Bindings define the actions and operations that are performed when creating or updating a service instance or service binding for each service plan. They specify the relationship between the service instance and its associated resources, such as secrets,

configuration files, or other dependencies. Bindings ensure that the necessary resources are provisioned and properly connected to the service instance.

The bindings configuration within the ServiceBrokerConfig resource includes details about the specific resources that are bound to each service instance or service binding, as well as the parameters and values that are required for proper configuration. For example, a database service may require a secret for storing database credentials. The bindings configuration specifies the secret that should be created and associated with the service instance or service binding.

```
bindings:
- name: SuperDB-Demo-binding
  service: MySQLDatabase
  plan: Basic
  registryScope: InstanceLocal
  serviceInstance:
    registry:
    - name: secret-name
      value: '{{ parameter "/Database credentials secret name " }}'
    steps:
    - name: database-cluster
      templates:
      - database-deployment
  serviceBinding:
    registry:
    - name: port
      value: '{{ parameter "/Port" | default 3306 }}'
    - name: host
      value: '{{ printf "%s.%s" (registry "instance-id") (registry "namespace") }}'
    - name: binding-namespace
      value: '{{ parameter "/Destination namespace" }}'
    - name: secret-name
      value: '{{ parameter "/Database credentials secret name " }}'
    templates:
    - database-service
    - database-secret-binding
```

Listing 6.3: Example template for MySQL deployment

Additionally, the bindings configuration also involves the use of a registry. The registry serves as a storage mechanism for holding information related to the service instances and service bindings. It acts as a centralized repository for storing critical data, such as credentials, connection details, or any other information required

by the services or applications. The registry is implemented as a Kubernetes secret, ensuring the secure storage of sensitive information. The Catalog Server populates the registry with the necessary data during the creation or update of service instances and service bindings. This allows the services and applications to access the required information at runtime, enabling seamless integration and functionality.

Integration of the Cloning Operator: Synator

The catalogue server, equipped with the knowledge to initiate the service, utilizes templates to describe the necessary resources. The primary function of these templates is to facilitate the sharing and cloning of the secret, which contains the access credentials, into the customer's preferred namespace. This is the namespace where the customer's application will be executed and which needs access to the service. To perform this task, a cloning operator, specifically the *Synator*, is employed.

Operator Functionality and Resource Structuring The Synator operator exclusively runs on the customer's cluster. However, it's crucial for the service provider to understand its functionality to properly structure its resources. These resources will be later utilized by the customer's application. The process specifically involves labelling the resources that the service provider anticipates to be useful for the customer's application. Subsequently, these resources will appear in the Kubernetes (K8s) namespace of the customer's application, thereby enabling immediate access.

- The operator on the customer side operates by identifying resources labelled appropriately.
- Upon completion of the identification, it determines the namespaces into which the resources should be cloned.

```
1  - name: database-secret-binding
2      template:
3        apiVersion: v1
4        kind: Secret
5        metadata:
6          name: '{{ registry "secret-name" }}'
7          annotations:
8            synator/sync: 'yes'
9            synator/include-namespaces: '{{ printf "%s,%s" (
       registry "namespace") (registry "binding-namespace") }}'
10       stringData:
```

```
11          MYSQL_ROOT_PASSWORD: '{{ parameter "/root-password"
    | generatePassword 32 }}'
12          MYSQL_USER: '{{ parameter "/user-name" | default "
    myuser" }}'
13          MYSQL_PASSWORD: '{{ parameter "/user-password" |
    generatePassword 32 }}'
14          MYSQL_DATABASE: '{{ parameter "/database-name" |
    default "mydb" }}'
15          MYSQL_HOST: '{{ registry "host" }}'
16          MYSQL_PORT: '{{ printf "%s" (registry "port") }}'
17          MYSQL_HOST_PORT: '{{ printf "%s:%s" (registry "host
    ") (registry "port") }}'
```

Listing 6.4: Example of catalogue resource template as secret implementing the synator label

Resource Cloning and Namespace Identification These namespaces are detailed within the label itself, allowing for pre-entry by the service provider at the time of resource creation. As this information can vary, it is stored as a register value within the namespace. This value is populated priorly by a value derived from the parameters of the API endpoint request.

6.2.3 Namespace Configuration

The Catalog Server relies on the concept of namespaces within the K8s cluster. A namespace is a virtual environment that provides a level of isolation and resource management within the cluster. When a PUT request is received by the Catalog Server via the REST API, it automatically creates the corresponding namespace based on the provided context. The context includes information about the desired namespace name, allowing for the proper organization and segregation of resources within the cluster.

The namespace configuration is crucial for ensuring the proper deployment and management of resources. Each service instance and service binding is associated with a specific namespace, ensuring that the resources are deployed and managed within the designated environment. The Catalog Server validates the namespace specified in the PUT request and ensures its existence within the K8s cluster. This ensures that resources are deployed in the correct namespace and are accessible to the corresponding service instances and service bindings.

```
1 {
2   "service_id": "superdb01",
3   "plan_id": "superdb01-demo",
4   "context": {
```

```
5       "namespace": "customer-3038488c-3d4d-43c2-9800"
6     },
7     "parameters": {
8       "secret-name": "db-secret"
9     }
10  }
```

Listing 6.5: Example of the body of a PUT request to */v1/service_instances*

In the provided example, the PUT request body includes the service ID, plan ID and the desired namespace within the context. Additionally, it specifies parameters relevant to the request, such as the name of the secret binding.

The Catalog Server's ability to seamlessly integrate with K8s and leverage the OSBAPI protocol empowers organizations to effectively manage services within their K8s clusters. By utilizing the ServiceBrokerConfig resource and the associated CRD, administrators can configure the catalog, templates and bindings to suit their specific needs. This flexibility enables the seamless deployment and management of services, ensuring optimal utilization of resources and efficient provisioning.

In summary, the Catalog Server acts as the central hub for managing services within a K8s cluster, leveraging the OSBAPI protocol to deliver comprehensive service management capabilities. Its integration with K8s, combined with the configurability offered by the ServiceBrokerConfig resource, provides organizations with the necessary tools to orchestrate services effectively. By facilitating the creation, configuration and management of services, the Catalog Server plays a critical role in streamlining operations and enabling organizations to make the most of their K8s environments.

6.3 Limitation of the original specifications

The implementation of our catalogue server is based on the 'Couchbase Service Broker', which adheres to the original OSBAPI (Open Service Broker API) specifications. While these specifications served their purpose in a local environment, they exhibit certain limitations when it comes to multi-user support and robust security mechanisms. This section aims to highlight these limitations and explore potential solutions.

6.3.1 Lack of Multi-User Support

The original OSBAPI specifications were primarily designed for environments where an OSBAPI service broker operates within its own controlled setting. However, in practical scenarios involving multiple users, these specifications fall short. Currently, there is no mechanism to differentiate between individual users and access control

is limited to distinguishing between authorized and unauthorized users. This lack of multi-user support poses challenges in environments where fine-grained access control and user-specific privileges are required.

6.3.2 Static Tokens and Limited Authorization

In the existing implementation, the API tokens used for authentication and authorization are static. Once a user possesses a valid token, they gain access to all protected APIs, without any further differentiation or granular permissions. This approach lacks the ability to recognize individual users and their associated roles, resulting in a binary authorized/unauthorized distinction. Furthermore, if an API token is intercepted, it can lead to unauthorized access and potential financial damage.

6.3.3 Security Challenges in a Distributed Environment

The original specifications suggest two methods for secure access to protected APIs: HTTP basic authentication or HTTP bearer token. However, HTTP basic authentication is not recommended in distributed environments due to the potential interception of requests. As a result, the preferred method is to use bearer tokens. While this offers a more secure option, it introduces additional challenges, such as token management, token issuance and maintaining associations between tokens and users.

6.3.4 Need for a Centralized Authentication and Authorization Server

To address the limitations mentioned above, our research has shifted towards a solution involving a centralized authentication and authorization server. By implementing this server, we can introduce user recognition, role-based access control and fine-grained permissions. This section aims to provide an overview of the proposed solution, highlighting its benefits and its adherence to industry standards.

6.3.5 Proposed Solution: Centralized Authentication and Authorization Server

In our pursuit of an improved security mechanism, we have turned to the OIDC (OpenID Connect) standard. The OIDC standard offers a robust framework for implementing authentication and authorization in distributed systems. By

leveraging OIDC, we can establish a centralized authentication and authorization server that provides enhanced security features and user-centric access control.

By adopting a centralized authentication and authorization server, we can overcome the limitations of the original specifications. This approach offers several benefits, including:

- **User Identity Management**: The centralized server enables the management of user identities, including authentication and user-specific information.
- **Role-Based Access Control**: With the introduction of the centralized server, we can implement role-based access control, allowing for fine-grained permissions based on user roles and responsibilities.
- **Secure Token Issuance**: The centralized server can handle token issuance, ensuring that each user is assigned a unique and secure token.
- **Reduced Token Management Overhead**: With a centralized server, the client no longer needs to store and manage thousands of individual tokens associated with multiple catalogue servers. Instead, a single token from the centralized server provides access to the necessary resources.
- **Enhanced Security**: The adoption of OIDC and the centralized server approach improves the overall security of the system, reducing the risk of unauthorized access and potential financial losses.

6.3.6 Integration with the Catalogue Server

The proposed solution involving a centralized authentication and authorization server would seamlessly integrate with the existing catalogue server. The catalogue server would communicate with the centralized server during authentication and authorization processes, verifying user identities, roles and permissions before granting access to protected APIs.

The original OSBAPI specifications, while suitable for local environments, lack support for multi-user scenarios and robust security mechanisms. By exploring alternative solutions, such as a centralized authentication and authorization server based on the OIDC standard, we can address these limitations and provide a more secure and scalable system. The proposed solution offers benefits such as user recognition, role-based access control, reduced token management overhead and enhanced security. The integration of the centralized server with the catalogue server ensures a seamless authentication and authorization process. Through these enhancements, we aim to create a more comprehensive and secure environment for multi-user access to the catalogue server.

6.4 Liqo Technology Integration

The inherent capacity of the proposed model, which has been elucidated upon in our previous discussions, is majorly anchored upon the integration of Liqo technology. This amalgamation is a strategic maneuver, deliberately orchestrated with a view to bridging the latent disconnect between the provider's cluster and that of the client. This effectively enhances the operational capabilities of the catalog server, as well as broadens the scope of service delivery.

6.4.1 Unraveling the Operational Framework

The operational blueprint upon which Liqo runs is a captivating one that merits due attention. Let's picture a service that at first glance seems to be local to the catalog server. However, upon more careful scrutiny, one realizes that this service has been remotely fashioned within the client's cluster. This smartly executed process leverages the power of Liqo technology, demanding the implementation of some very specific measures. The outcome is the virtual creation of a cluster, defined by a singular node that distinctly portrays the client's cluster with which peering was initiated.

On the surface, this newly-conceived virtual cluster bears no apparent deviation from a typical cluster. However, it is this uniformity that plays a pivotal role in ensuring the smooth operation of the "Couchbase Service Broker" even under these conditions. It is noteworthy that the original service broker continues its operation within its cluster, but with a key twist: it is explicitly instructed about the specific K8s namespace where the requisite resources are meant to be instantiated.

6.4.2 Unpacking the OSBAPI Protocol and Context Specification

Before we proceed, it's pertinent to revisit the specifics of the OSBAPI protocol we examined earlier. You may recall that amongst the two potential PUT requests, there exists a JSON field tagged 'context' within the body of the request. This ostensibly trivial parameter plays a critical role within the service broker developed by Couchbase. Encapsulated within this 'context', we discover another parameter known as the "namespace". This parameter essentially guides the server in deciding the suitable namespace within which resources are to be created.

Here's a quick run-through of how it works: the server merely requires the referenced namespace to pre-exist and be available within the cluster. Once this is confirmed, the server is at liberty to create resources within this namespace. It is at this juncture that Liqo technology swings into action. With the namespace in

place, the catalog server is enabled to create a Liqo resource within it, referred to as the Namespace Offloading.

6.4.3 Understanding Namespace Offloading and Resource Direction

The creation of a Namespace Offloading is essentially a command to Liqo, instructing it to direct or offload resources deployed within it to a remote cluster that has previously been peered with. It is important to underscore that offloading must be restricted to a particular remote cluster. Absence of this restriction could potentially lead to each cluster that has been peered with the cluster provider offloading resources. This could result in a service meant for a specific client ending up in a different user's cluster, an eventuality that we are determined to evade.

To counteract this possibility, Liqo provides a feature known as "Cluster selector" label for Namespace Offloading. This allows us to specifically designate the cluster ID to which offloading is to be performed. As such, when a request is made to create resources for a certain service, the namespace name is specified within the request. This ensures the precise identification of the client cluster to which the resources are to be deployed.

6.4.4 Significant Modification in the Original Design

The described operational behavior necessitates a marked modification in the original design of the Couchbase service broker. It calls for the inclusion of a logic that informs the catalog server (which now assumes a new role) about the correlation between the clusterID and the user. This starts with the introduction of a feature into the catalog server, which is not catered for in the standard OSBAPI. This feature is the creation of a Liqo peering. To accomplish this, we introduce the following protected API:

```
1 POST "/peering"
```

The body of this API is structured to encapsulate all the necessary data for Liqo peering: cluster id, cluster name, authentication URL and the authentication token. In the grand scheme of things, some fields have a critical role: offloading policy and prefix namespace.

To gain a deep understanding of their function, we need to delve into the behavior of the catalog server when this specific endpoint is invoked. On its call, the server initially verifies the identity of the user attempting to initiate the peering. By inspecting the security token sent, the server infers the user ID and proceeds with the initiation of the peering.

6.4.5 Introduction of Database: A Major Paradigm Shift

The introduction of a database marks a significant departure from the Couchbase Service Broker. This database stores vital information such as the association between cluster ID and user ID, effectively linking a specific peering to a particular user ID. While the Couchbase's service broker also retains static information, it is reliant on K8s structures and the secrets it directly creates. This method, albeit straightforward and effective, necessitates constant calls to the cluster to retrieve resources. In the absence of knowledge of the specific namespace to search, this process could become notably more complex and potentially resource-intensive.

By bringing a database into the picture, the process of searching and integrating with existing code is greatly simplified. The catalog server logs the user ID, peering information and the association with a set of namespaces in the database.

Sequence of Creation for Namespace and Peering

However, the creation of the namespace isn't instantaneous as the process mandates that several steps be carried out in a particular sequence:

1. Creation of peering
2. Creation of the namespace
3. Creation of the offloading namespace linked with the cluster ID of the previously established peering

Despite these operations being sequential and dependent, they are time-consuming. As such, the catalog server delegates the entire series of operations to a separate thread, which records the status in the database. This strategy leverages an already present and inherently thread-safe component. The presence of 'ready' fields for peering and namespace, coupled with the 'error' field, provides a detailed status of the overall progress of the peering operation.

Given the asynchronous nature of this operation, the user receives the ID of the peering request made, linked to the specific namespace to be created, upon calling this specific API. To monitor its progress, a simple yet effective solution was implemented in the form of a REST API endpoint on the catalog server which is useful for polling the status. The protected API endpoint is as follows:

```
1 GET "/peering/:id"
```

The server first verifies if it can provide the information to the requesting user. It then fetches the overall status of the request associated with the ID from the database. For it to be fully ready, both the peering and namespace creation must

be completed. If an error occurs at either phase, the API response will reflect the error detail and 'not ready' status. If the entire operation is successful, the real name of the namespace created by the catalog server is retrieved.

In the peering request, an optional field, known as "prefix namespace", can be defined. This field acts as a prefix for the namespace name that will be returned. However, the actual name is generated by the catalog server based on the cluster ID and the user ID. This random composition is necessary to create a namespace that is unique within the provider's cluster and potentially unique within the client's cluster as well.

6.4.6 Liqo's Role in Namespace Creation

Liqo plays a substantial role in namespace creation. With namespace offloading, the server can abstract the remote namespaces, making them appear as if they are locally available. This unique capability allows us to implement Liqo's innovative namespace offloading strategy. In this scenario, namespaces created are actually intended for remote clusters and the peering process is implemented via the use of the Catalog API.

6.4.7 Namespace and Resource Deployment

Once created, the namespace is prepared for resource deployment. The Couchbase Service Broker requires certain information about the client cluster. To achieve this, the OSBAPI protocol is slightly extended to provide all the necessary information. Specifically, an API endpoint is introduced to notify the catalog server when a service instance is being created in the cluster. This service instance is created with a PUT request to the broker, using the following endpoint:

```
1 PUT "/v2/service_instances/:instance_id"
```

6.4.8 Service Instance Creation

In the request body, the field 'context' is used to specify the namespace in which the service instance creation occurs. The catalog server checks whether the namespace is associated with a specific user in the database and then verifies if the namespace is ready. If it is, the namespace name is replaced with the real name in the provider's cluster. This way, the broker continues with the resource creation as if they were to be deployed within the provider's cluster, oblivious to the fact that they are being offloaded to the client's cluster. The resources created in this manner can be viewed through Liqo's K8s APIs as if they were present in the provider's cluster.

In summary, the integration of Liqo technology with the Couchbase Service Broker in the context of OSBAPI provides a compelling example of a broader theme: how technology can transcend the boundaries of traditional operational constraints, connecting distinct, distributed systems into a seamless, virtual construct. Liqo offers an exceptional platform for facilitating this process, bringing together various clusters and broadening the scope of capabilities in a distributed and scalable manner.

6.5 Elaboration on the Security Configuration

The structure of the implementation was thoughtfully devised to ensure optimal security. Initially, the implementation was built around a standard protocol that inherently provided a level of security. However, this was only the first layer of the security strategy. The mechanisms involved simple bearer tokens or username and password, both transmitted via HTTP requests to the API.

Although these approaches are suitable in a closed and isolated environment, they can show significant weaknesses in a distributed model, as the one our implementation is based on. A simple token could be intercepted and reused unauthorizedly and the use of username and password could potentially lead to security risks.

6.5.1 Adapting the Bearer Token Mechanism

The implementation in question extends the bearer token mechanism. Instead of a permanent bearer token, as per the original standard, the revised mechanism employs a temporary token. This approach significantly mitigates the risk of leakage or misuse of a 'passkey' to the catalogue server, ensuring optimal security.

6.5.2 OpenID Connect (OIDC) Protocol

The OpenID Connect protocol was employed to enhance the security model. This modern, industry-standard protocol allows clients to verify the identity of the end-user based on the authentication performed by an authorization server.

OIDC is essentially a simple identity layer built on top of the OAuth 2.0 protocol. It enables clients to verify the identity of an end-user based on the authentication performed by an authorization server and to obtain basic profile information about the end-user in a secure and REST-like manner.

Key Features of the OIDC

OIDC boasts the following key features that make it an ideal choice for our security implementation:

- Standardized: It uses web-friendly JSON-based identity tokens (JWTs) with a broad industry support.
- Secure: Identity data is carried in the token, reducing the need for multiple calls to the server.
- Flexible: It supports a range of applications, including mobile apps, browser-based Single Page Applications (SPAs) and traditional web apps.
- Interoperable: It can smoothly operate with any system that supports HTTP and JSON.

6.5.3 Keycloak: The OIDC Implementation

Keycloak is an open-source Identity and Access Management solution aimed at modern applications and services. It primarily provides mechanisms for authentication, single sign-on and authorization.

Keycloak stands out for its ability to secure modern applications and services with little to no code. Some of its main features include:

- User Authentication: Keycloak can handle user authentication, including single sign-on (SSO), two-factor authentication (2FA) and password policies. It can also integrate with third-party identity providers such as Google and Facebook.
- User Federation: Keycloak can federate external user databases. It supports a variety of providers like LDAP and Active Directory and can also work with custom user storage.
- Authorization Services: Keycloak handles centralized authorization policies and decision making, which applies even in federated environments.
- Security Token Service: Keycloak can issue tokens for applications to use. It supports standard protocols such as OpenID Connect and SAML.

In our security model, Keycloak plays an integral role as the server that truly implements the security configuration. Acting as the OIDC authorization server, it is contacted by the customer to obtain the token. Keycloak ensures that the login process is secure and that the token provided is temporary, thereby fulfilling the desired extension of the bearer token mechanism we previously discussed.

Keycloak in the Marketplace and Dashboard Implementation

Keycloak, along with the OIDC protocol, has been significantly useful in implementing the security mechanism for the dashboard through the Marketplace. The ability to login through any registered client allows third-party entities to interact on behalf of the logged-in user. This is precisely how the Marketplace operates, promoting seamless interaction and enhanced security.

Keycloak serves the dual purpose of not only providing the necessary security layer but also streamlining the user interaction process. Its ability to work with OIDC and facilitate secure token-based user authentication makes it a critical component of our security framework.

6.6 The Marketplace

The final component of this thesis implementation is the 'Marketplace' – a central platform bridging the gap between demand and supply. It is a stage where service providers can showcase and promote their services, while potential customers can discover the most suitable solutions for their specific needs. These needs can be both technical and commercial in nature.

6.6.1 Platform Development

The Marketplace platform is developed as a Single Page Application (SPA) using ReactJS. Alongside, a backend was developed using GoLang, serving as a REST API server. These two components together form the Marketplace. As previously stated, this platform contacts Keycloak for user authentication.

6.6.2 User Interface and Experience

Upon accessing the Marketplace, the user is prompted to choose the action they wish to perform: either search for a service or offer one. Selecting the first option allows the user to view all registered catalogues, while the second guides the user, post-login, through the process of adding their catalogue server to the Marketplace.

Login and Catalogue Server Registration

The login phase redirects the user to the authentication server (Keycloak in our case), where an authentication token is obtained. This token is then used to contact the Marketplace backend server, which further verifies the token at the Keycloak server.

Subsequently, information about the catalogue server to be registered is obtained. It should be noted that users have the option to register their own custom catalogue

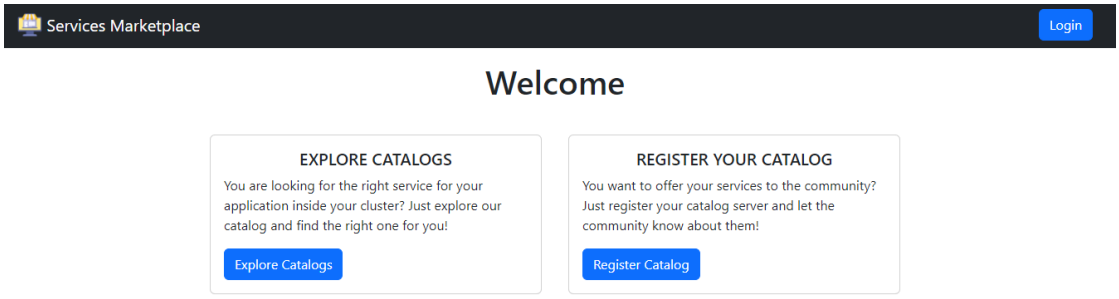

Figure 6.2: marketplace dashboard home page

server (assuming it adheres to the expanded version of the OSBAPI specifications) or use the one created for this thesis work.

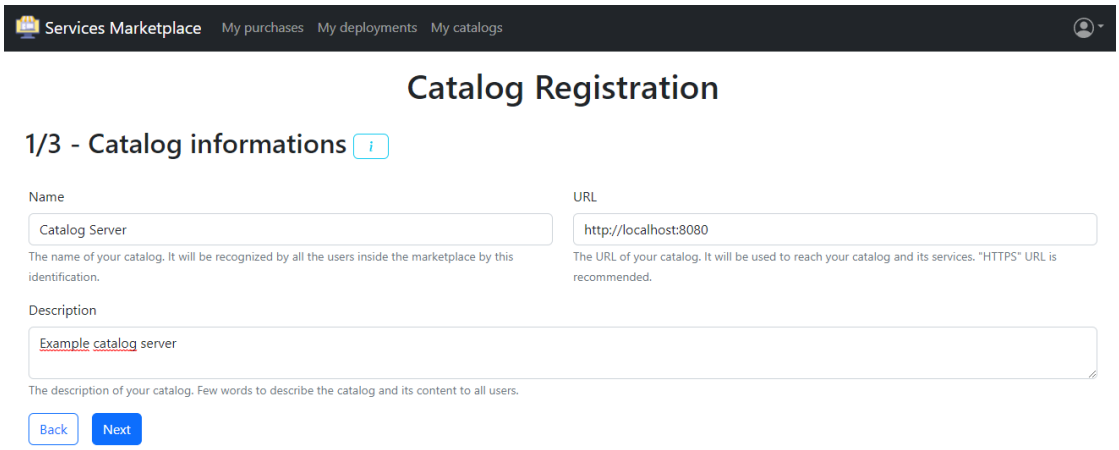

Figure 6.3: Marketplace dashboard catalog registration form

Following the collection of data about the catalogue server, security configuration information is provided and registration is carried out on the OIDC authentication server, i.e., Keycloak. Reachability and protected APIs functionality is confirmed via a token issued by the agreed OIDC server.

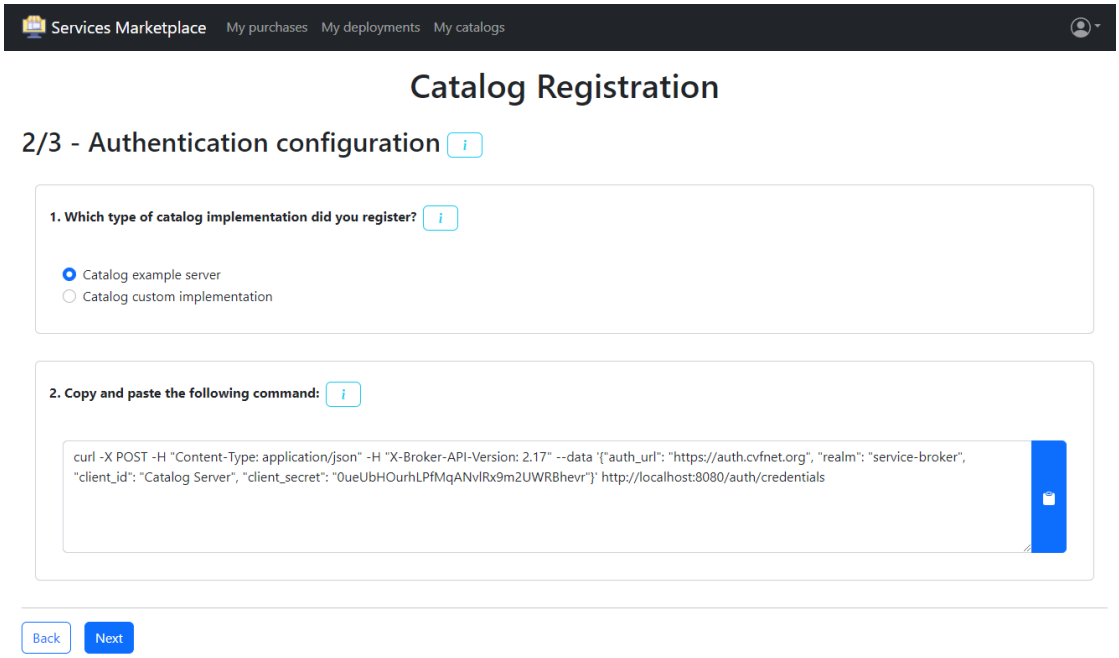

Figure 6.4: Marketplace dashboard catalog registration, security configuration

Exploring the Catalogue

Once registered, other users can explore the catalogue to find the service that best suits their needs. Detailed information, including available plans, hosting options and relevant details, are displayed. These are directly related to the policy that will be adopted for offloading the service with Liqo. Accordingly, the service can be:

- Maintained locally in its own cluster (where Liqo will offload all resources to the customer's cluster)
- Maintained remotely (where Liqo will adopt a 'Local' offloading policy to keep all resources on the origin cluster, i.e., the provider's cluster)
- Given a hybrid function, where only certain resources, as determined by the service provider, will be hosted by the customer, while others will remain on the provider's cluster

6.6.3 Service Deployment

After the purchase, the actual deployment of the service can begin. The process emulates a 'wizard', guiding the user through the deployment and creation of

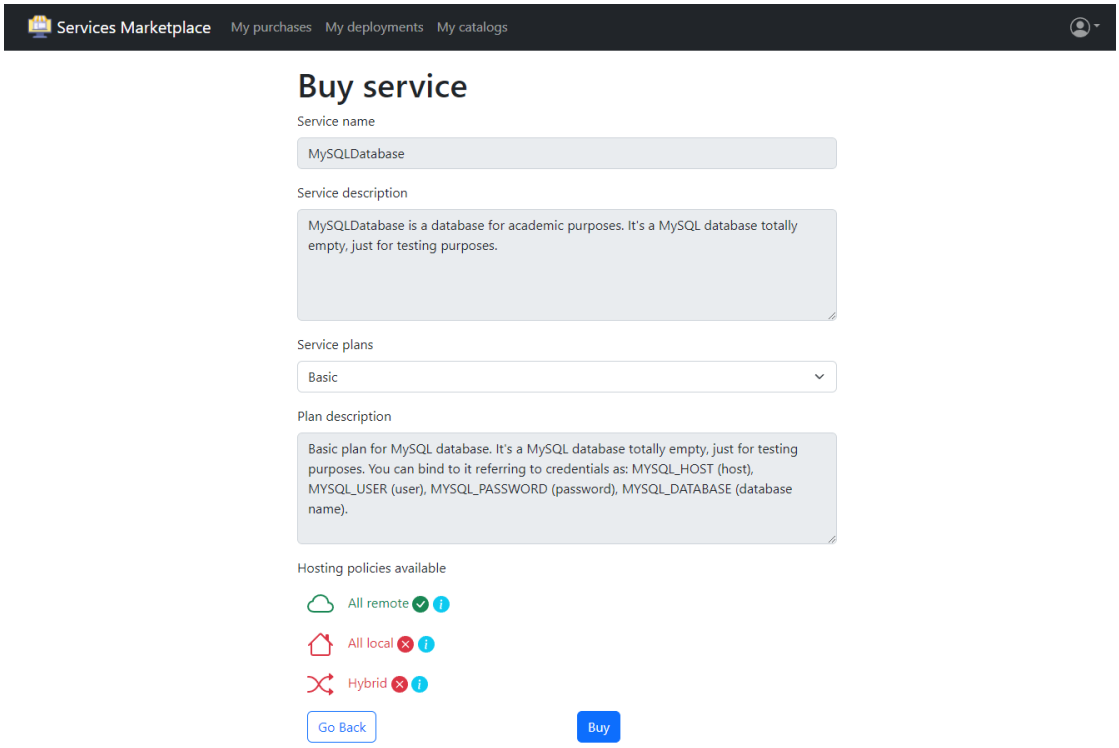

Figure 6.5: Marketplace dashboard service details

resources in a step-by-step manner.

Prerequisites

Firstly, the user is prompted to verify the prerequisites, including a K8s cluster, Liqo and the cloning operator: Synator. The latter was specifically implemented to allow the automatic copying of the secret containing service access credentials into the client application's namespace.

Initiating Liqo Peering

Next, information for starting Liqo peering between the service provider and the customer's cluster is requested. The Marketplace then begins pulling on the provider's catalogue server endpoint about the peering. Once the namespace related to the deployment being done is obtained, the next operation is unlocked.

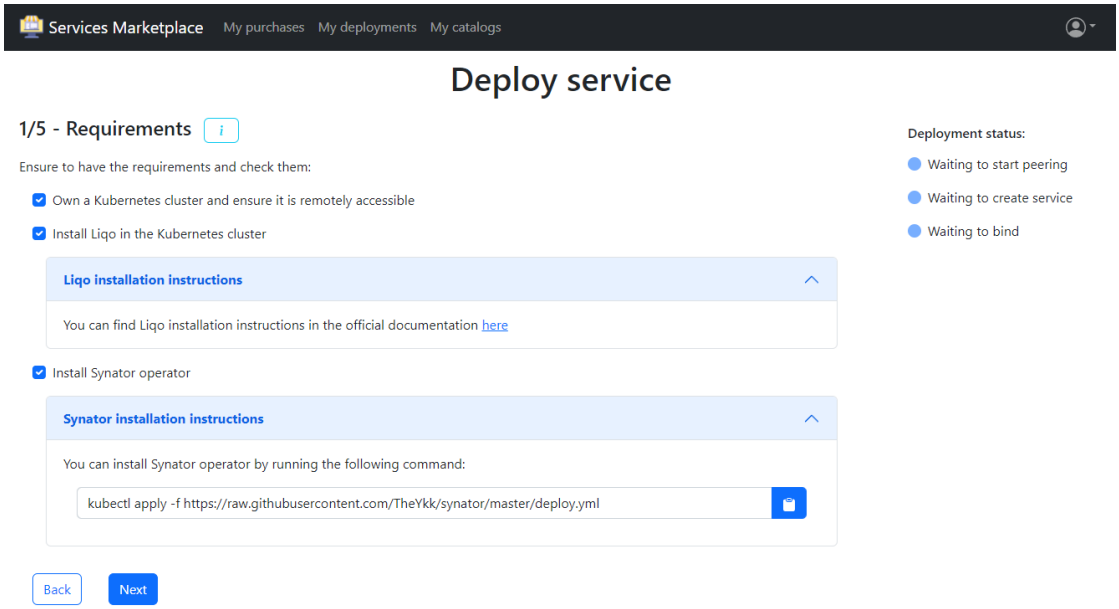

Figure 6.6: Marketplace dashboard deployment requirements

Service Creation

Then, information for creating the service and, if necessary, the name of the secret to which the client application will refer to access the service credentials is requested. Similar to the previous step, the Marketplace sends a request to the endpoint of the catalogue server and, given the operation ID, starts pulling until the operation is successful.

Binding and Finalization

The last step involves binding and entering all necessary information required by the provider. Subsequently, the Marketplace replicates the same requesting and polling behaviour as before, then directs the user to the final page, indicating the name of the namespace they will find in their cluster with any resources related to the deployed service. Note that the resources found depend on the chosen hosting policy and thus, in the case of opting for completely remote hosting, no resources will be found in that namespace.

At the end of this procedure, the customer will have a fully functional application that can connect to the purchased and deployed service through the Marketplace platform.

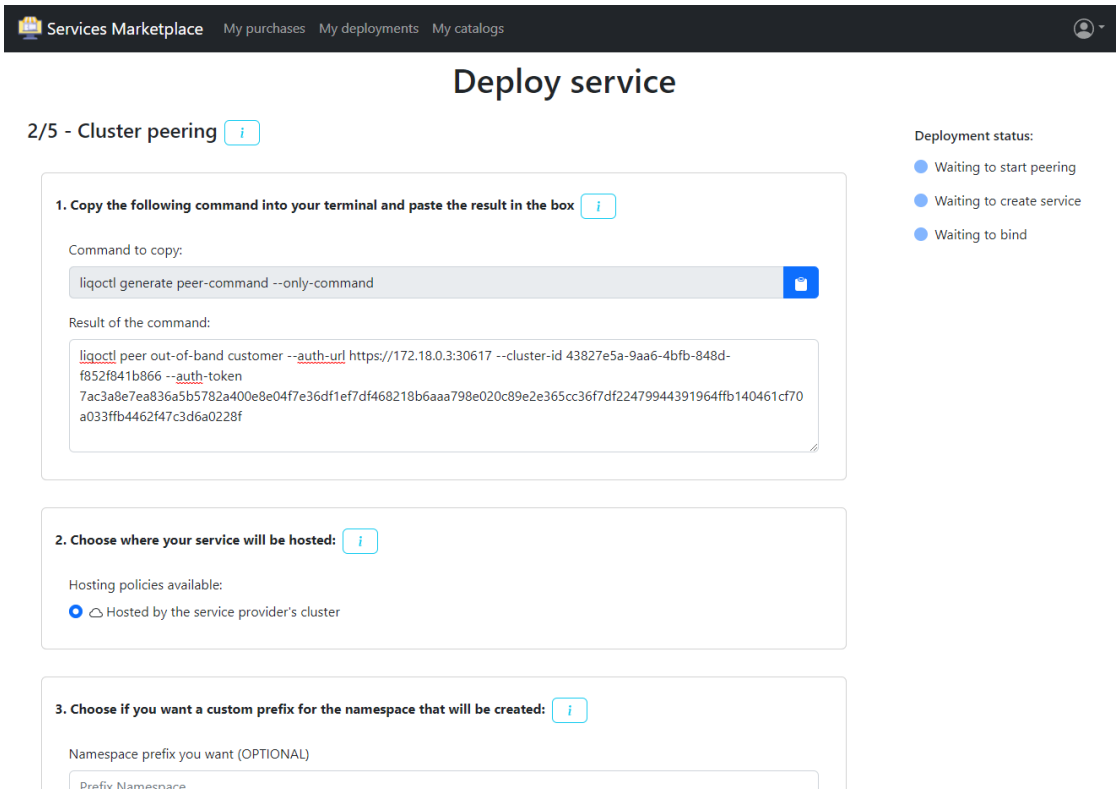

Figure 6.7: Marketplace dashboard Liqo peering information form

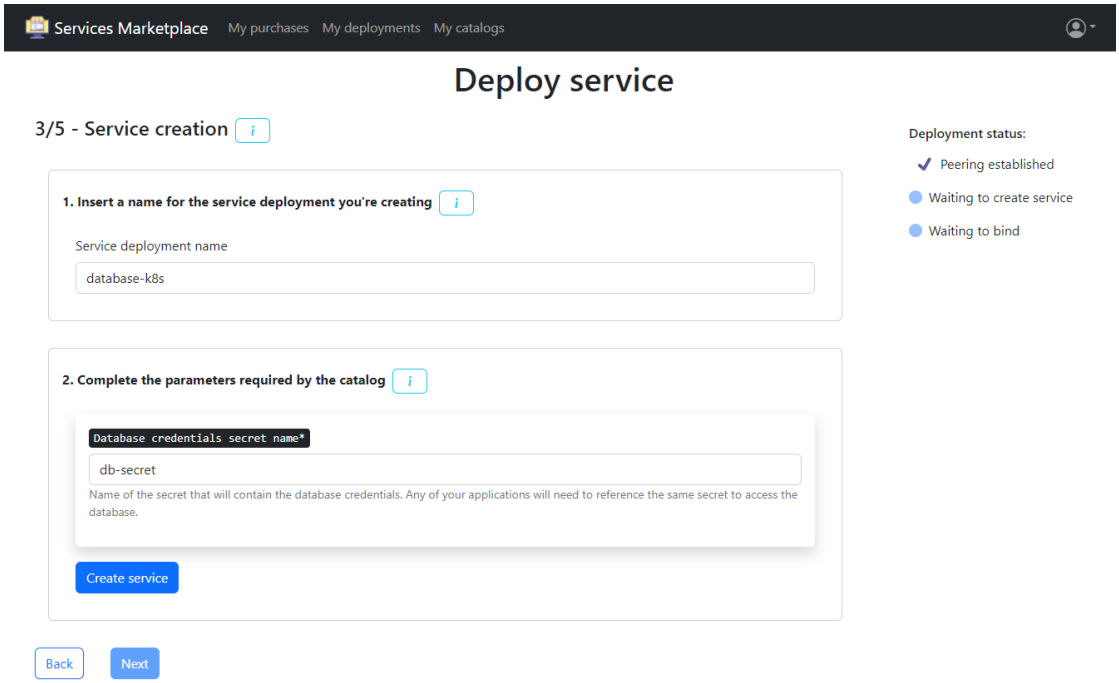

Figure 6.8: Marketplace dashboard service creation information form

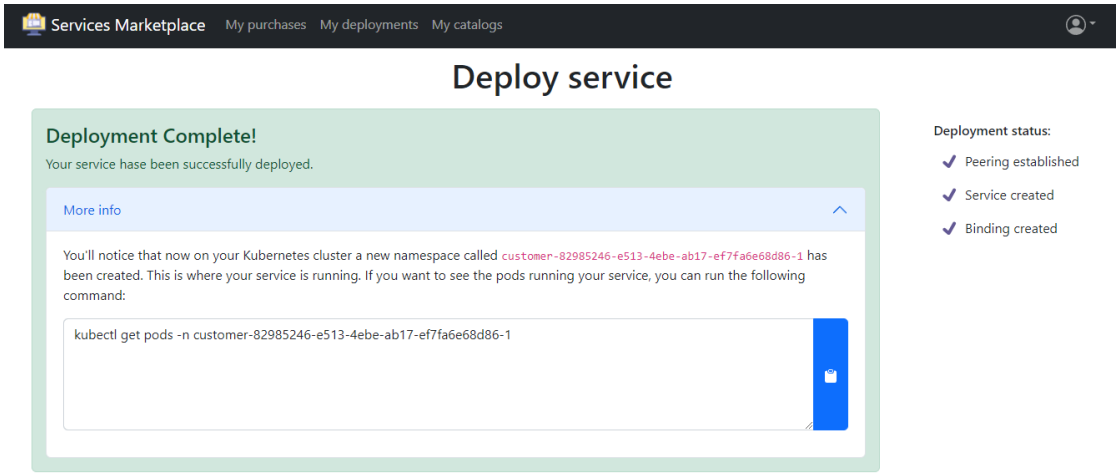

Figure 6.9: Marketplace dashboard deployment completed

Chapter 7

Measurements

In this chapter, we examine the performance of our model implementation, developed in the previous chapter, through a series of tests. These tests were performed in a local environment, implying that the service-provider cluster and the customer cluster were both implemented as single node clusters using KIND, which runs on a local machine in Docker. It is worth mentioning that in a real-world environment, the data may be considerably affected by higher latencies.

7.1 Benchmarking

To gauge the efficiency of our catalogue server implementation, we simulated the entire procedure from purchase to deployment. Specifically, we measured the time taken by our catalogue server implementation for the peering, service creation and binding phases. In order to ascertain any significant variations, the operations leading to a complete deployment were also performed on a customer cluster with which peering had been carried out previously. Two such tests were conducted.

7.1.1 Benchmark Results

The results, as shown in the table 7.1, indicate that the peering phase is faster when the cluster has previously been peered. This is because the same foreign cluster is reused, leading to only the creation of the namespace and namespace offloading within this time frame.

Regarding the service creation phase, it is important to note that the duration is highly dependent on the nature of the resources to be created. In this particular test case, a deployment was created with just a single replica of the MySQL Docker image. There was no significant variation in this phase between a previously peered cluster and one that was not.

	Peering [s]	Service Creation [s]	Binding Phase [s]
Initial Peering	10.538720	4.422730	0.000160
Already Peered Test #1	6.367760	4.385800	0.000120
Already Peered Test #2	4.148930	4.391170	0.000110

Table 7.1: Duration of various asynchronous operations by the catalogue server.

In contrast, the binding phase was considerably faster than the other two operations. This can be attributed to the specific design of our catalogue server implementation. Even though the specifications allow for the binding operation to be asynchronous, our implementation does not implement this feature. Instead, all resources necessary for this phase are created during the API call, since they are supposed to be fast to deploy resources. This includes the K8s service for displaying the database service pod and the secret containing credentials and data for access.

7.1.2 Resource Consumption

In terms of resource consumption, the memory usage gradually increased over time but ultimately stabilized at a very low and constant value, as depicted in Figure 7.1. CPU consumption, on the other hand, exhibited occasional peaks, primarily during the peering phase. Despite these peaks, the overall CPU consumption remained remarkably low.

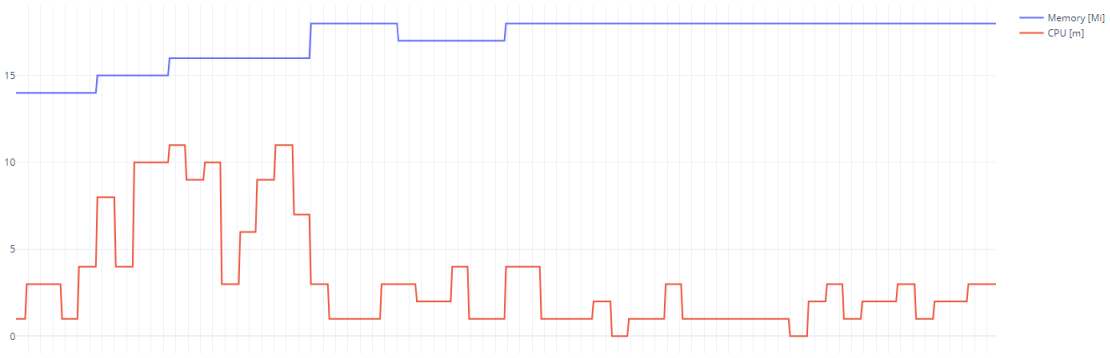

Figure 7.1: Temporal trends in CPU and memory consumption by the service catalog pod.

The relatively low resource consumption by the service catalogue can be explained by its role as an orchestrator in resource creation. The heavy lifting in terms of resource consumption is delegated to the newly created resource pods, since K8s is responsible for actual resource creation.

In conclusion, the implemented model has produced satisfactory results, both in terms of request processing times and resource consumption. These results validate our implementation approach and suggest that the model is ready for deployment in larger-scale real-world scenarios.

Chapter 8

Conclusions

This thesis project has indeed accomplished its initial objective, which was to establish a model tailored to specified requirements. Despite its raw state, with multiple areas for potential improvement, the model fosters a viable and beneficial basis for collaboration among numerous entities. The ultimate goal is to acquire beneficial services within their cloud environments.

8.1 What's next?

The model, even if functional, still has a considerable scope for refinements and enhancements, which can greatly add to its value and utility.

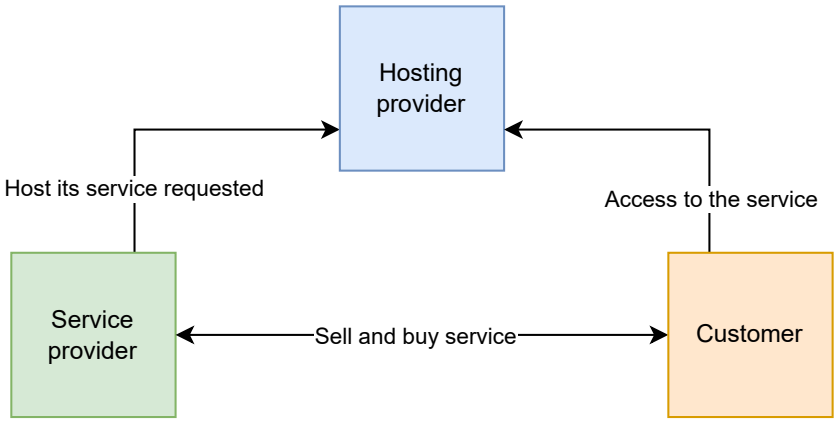

Figure 8.1: Proposed operational scheme with the inclusion of the hosting provider

As a preliminary step, it is suggested to introduce a new actor into the framework. To elaborate, the service provider so far is considered as the entity owning the source code of the software service intending to commercialize it and the customer

as the one desiring the service for internal purposes within his cloud environment. The new actor introduced can be labeled as the 'hosting provider'.

This entity is envisaged to offer hosting space for the deployed purchased service. The hosting provider can either be chosen by the customer or the service provider. The addition of this third party could bring forth several benefits, especially in terms of resource utilization. For instance:

- The service provider, by delegating this potentially burdensome task to the hosting provider, can solely focus on the commercialization of the service without the concern of providing and managing the required space in its cluster.
- The customer, on the other hand, could potentially host part of his application needing the service on this hosting provider's infrastructure.

Certainly, these propositions should be analyzed in-depth, yet they could be presented as alternative options within the model. This addition not only enriches the model but also offers more flexibility and options to the parties involved.

Bibliography

[1] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. «Borg, Omega, and Kubernetes». In: *ACM Queue* 14 (2016), pp. 70–93. URL: http://queue.acm.org/detail.cfm?id=2898444 (cit. on p. 4).

[2] *Spotify Case Study | Kubernetes*. URL: %5Curl%7Bhttps://kubernetes.io/case-studies/spotify/%7D (cit. on p. 5).

[3] CNCF. *The New York Times*. URL: https://www.cncf.io/case-studies/newyorktimes/ (cit. on p. 5).

[4] *Kubernetes: Production-Grade Container Orchestration*. URL: https://kubernetes.io/ (cit. on p. 5).

[5] Project Calico. *Calico - Networking and Network Security for Containers*. URL: https://www.projectcalico.org/ (cit. on p. 11).

[6] CoreOS. *Flannel - Network Fabric for Containers*. URL: https://github.com/coreos/flannel (cit. on p. 11).

[7] Weaveworks. *Weave - Simple, resilient networking and monitoring for containers and microservices*. URL: https://www.weave.works/ (cit. on p. 11).

[8] Cilium. *Cilium - Network and API-Aware Security for Containers*. URL: https://cilium.io/ (cit. on p. 11).

[9] Kubernetes. *DNS for Services and Pods*. URL: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/ (cit. on p. 11).

[10] *Kubernetes Documentation*. 2023. URL: https://kubernetes.io/docs/ (cit. on p. 12).

[11] *Kubernetes Blog*. 2023. URL: https://kubernetes.io/blog/ (cit. on p. 13).

[12] Liz Rice and Michael Hausenblas. *Kubernetes Security*. O'Reilly Media, 2021 (cit. on p. 14).

[13] *Using RBAC Authorization*. URL: https://kubernetes.io/docs/reference/access-authn-authz/rbac/ (visited on 06/14/2023) (cit. on p. 14).

[14] *Kubernetes Scaling: The Comprehensive Guide to Scaling Apps*. https://bluexp.netapp.com/blog/cvo-blg-kubernetes-scaling-the-comprehensive-guide-to-scaling-apps. (Accessed on 06/16/2023) (cit. on p. 14).

[15] *Deploying Microservices on Kubernetes | by Mehmet Ozkaya | aspnetrun | Medium*. https://medium.com/aspnetrun/deploying-microservices-on-kubernetes-35296d369fdb. (Accessed on 06/16/2023) (cit. on p. 16).

[16] *Liqo Documentation*. Liqo Documentation. 2020. URL: https://doc.liqo.io/ (cit. on p. 18).

[17] *Open Service Broker API*. URL: https://www.openservicebrokerapi.org/ (cit. on pp. 23, 24).

[18] *Open Service Broker API Specification*. 2023. URL: https://github.com/openservicebrokerapi/servicebroker/blob/v2.16/spec.md (visited on 06/29/2023) (cit. on pp. 24, 26, 27).

[19] *Open Service Broker API Use Cases*. 2023. URL: https://www.openservicebrokerapi.org/use-cases.html (visited on 06/29/2023) (cit. on p. 26).