POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Business Continuity in Kubernetes Multi-Cluster Environments

Supervisors	Candidate
Prof. Fulvio RISSO	Francesco TORTA

Academic Year 2022-2023

Summary

Over the past two decades, cloud computing has become a disruptive technology that has revolutionized the way businesses and individuals access and use computing resources. The growth of cloud computing has been driven by the increasing demand for low-cost, scalable, and easily accessible computing resources. Cloud-native applications are software applications that are specifically designed to run on a cloud computing infrastructure. They are built using cloud computing principles and technologies such as microservices architecture, containerization, and orchestration.

"Liquid computing" is a term used to describe the ability to dynamically allocate resources as needed, allowing for quick and easy scaling up or down. This "liquid" nature of cloud computing allows organizations to become more agile and respond quickly to changes in their business needs. Liqo is an open source project started at Politecnico di Torino that supports this concept and enables the creation of multi-cluster topologies within Kubernetes. Multiple independent clusters can be interconnected to share resources and workloads, while being managed as a single entity.

The goal of this thesis is to investigate how to ensure business continuity in a multi-cluster environment powered by Liqo, enabling an organization to maintain its critical business functions and processes in the event of a disruption. Given the increased complexity of a multi-cluster topology, various failure scenarios are analyzed, taking into account all key elements of the architecture of a Kubernetes cluster. The thesis presents the design and implementation of the ShadowEndpointSlice, an abstraction that allows to transparently guarantee service continuity even if some parts of the multi-cluster infrastructure are out of service. It also presents the implementation of a custom controller that ensures that the expected workload is running on the "big" cluster when some worker nodes are not functioning properly. Lastly, it is described a possible disaster recovery solution that leverages the potential of Liqo to easily use peered clusters as failover sites.

Part of the work of this thesis has been integrated into the Liqo project and is available on the official GitHub repository.

Table of Contents

List of Figures

Acronyms

K8s
: Kubernetes

CRD
: Custom Resource Definition

CR
: Custom Resource

API
: Application Programming Interface

REST
: Representational State Transfer

VK
: Virtual Kubelet

HA
: High Availability

Chapter 1

Introduction

In recent decades, cloud-native technologies have become increasingly popular to handle the high volume of requests and computation that large organizations handle on a daily basis to serve millions of users. This has led to a paradigm shift in the way enterprises and individuals approach IT infrastructure, making it more flexible, scalable and cost-effective. Cloud computing has become a ubiquitous technology, enabling a wide range of services and applications, from data storage and processing to machine learning and artificial intelligence. Its flexibility and scalability have made it a go-to solution for businesses of all sizes, from startups to large enterprises. Containerization techniques and container orchestrators have made it even easier to deploy these workloads on a cluster. As a result, clusters are being used in various areas of the software industry, and there is a growing need to connect them together to take full advantage of their capabilities.

Kubernetes, one of the most widely used container orchestration platforms, has played a significant role in the cloud computing industry. Its user-friendly features and powerful declarative API enable developers to handle the dynamic nature of modern workloads with ease. Its open source nature and developer-friendly tools have contributed to its popularity and role in the growth of the cloud community.

1.1 Introducing Liqo

The concept of "liquid computing" refers to the dynamic allocation of computing resources based on the needs of the user. This approach has become essential for companies that want to quickly adapt to changes in their operations. Liqo, an open source project launched at Politecnico di Torino, exploits this idea to enable the creation of dynamic multi-cluster topologies within Kubernetes.

Liqo's approach allows multiple independent clusters to be connected together so that they can share resources and workloads while being managed as a single

entity. By extending the Kubernetes API, Liqo allows different clusters to be joined together to form a multi-cluster network of computing nodes. This is achieved by automatically establishing a peer-to-peer relationship that enables the sharing of resources and services between independent and heterogeneous clusters.

One of the key benefits of Liqo is its ability to seamlessly offload workloads to remote peers without requiring any changes to Kubernetes or the applications. This makes multi-cluster computing native and transparent, as remote clusters are considered nodes added to the other available nodes in the local cluster. To facilitate communication between remote pods, Liqo provides a network fabric that enables multi-cluster pod-to-pod connectivity.

By providing a unified, scalable and flexible computing environment, Liqo has the potential to change the way enterprises approach their IT infrastructure.

1.2 Goal of the thesis

The goal of the thesis is to investigate how to ensure business continuity in a multi-cluster environment so that an enterprise can maintain its critical business functions and processes in the event of a disruption. While multi-cluster topologies bring many benefits and new capabilities, they also increase infrastructure complexity. This work analyzes how these limitations can be overcome using Liqo. The business continuity analysis is divided into two main categories.

- **Service Continuity**: several failure scenarios are analyzed, taking into account all key elements of a Kubernetes cluster's architecture. Different architectural changes to Liqo are proposed to make it more robust in all these scenarios.
- **Disaster Recovery**: it is presented a POC that exploits the potential of Liqo to easily use peered clusters as failover sites for application data backup and recovery.

1.3 Structure of the work

The thesis will unfold with the following structure:

1. **Chapter 2** provides an overview of Kubernetes, the technology that enables the orchestration and deployment of cloud-native applications.
2. **Chapter 3** provides an overview of Liqo and its key concepts and components.
3. **Chapter 4** introduces the concept of business continuity, its problems and possible solutions in a Kubernetes environment.

4. **Chapter 5** provides a description of the architectural changes to Liqo and its implementation details to provide service continuity in a Kubernetes multi-cluster environment.

5. **Chapter 6** provides a disaster recovery strategy that leverages a multi-cluster environment powered by Liqo.

6. **Chapter 7** concludes the presented work and summarizes the obtained results.

Chapter 2

Kubernetes

This chapter provides an overview of the Kubernetes architecture, including its history and evolution over time. This summary forms the basis for many of the concepts presented later. Kubernetes (often abbreviated to K8s) is a large framework, and a thorough examination would require much more time and discussion; therefore, only a description of the core concepts and components is provided here. For more information, see the official documentation [1].

2.1 Kubernetes: a bit of history

Around 2004, Google created the **Borg** [2] system, a small project that initially had fewer than 5 people working on it. The project was developed in collaboration with a new version of Google's search engine. Borg was a large-scale internal cluster management system that "ran hundreds of thousands of jobs, from many thousands of different applications, across many clusters, each with up to tens of thousands of machines" [2].

In 2013 Google announced **Omega** [3], a flexible and scalable scheduler for large compute clusters. Omega provided a "parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, in order to achieve both implementation extensibility and performance scalability".

In mid-2014, Google presented **Kubernetes** as an open source version of Borg. Kubernetes was developed by Joe Beda, Brendan Burns, and Craig McLuckie and other engineers at Google. The development and design of Kubernetes were heavily influenced by Borg, and many of the early contributors had previously worked on Borg. The original Borg project was written in C++, while the Go language was chosen for Kubernetes.

In 2015, Kubernetes v1.0 was released. At the same time as the release, Google partnered with the Linux Foundation to form the **Cloud Native Computing**

Foundation (CNCF) [4]. Since then, Kubernetes has grown significantly, achieving the CNCF graduated status and being adopted by almost all major enterprises. Today, it is the de facto standard for container orchestration [5, 6].

2.2 Evolution of workloads management

Traditional deployment era In the traditional deployment era, organizations ran applications on physical servers. There was no way to define application constraints to limit resource usage, and some applications would end up taking most of the resources available, making the remaining applications starve. This led system managers to deploy one server per application, increasing costs and maintenance work. At this point, the community rediscovered the abandoned concept of virtualization.

Virtualized deployment era In the virtualized deployment era, developers could run multiple virtual machines (VMs) on a single physical server and ensure that applications did not interfere with each other by running one VM per application. Virtualization allows defining resource usage constraints for each VM and ensures that software running on a VM is isolated from the rest of the system and other VMs, resulting in a much more stable and secure environment, as applications cannot interfere with each other and cannot freely access private application data. It also enables better scalability, as application instances can be easily scaled up or down by creating or deleting VMs as needed. Each VM contains a complete operating system and can be tweaked to include the properly versioned dependencies required by the running application: this creates sealed compartments that are easy to manage, maintain, and troubleshoot. Overall, fewer physical servers are used, costs are lower, and companies can get the most out of their available servers, preventing them from being underused.

Containerized deployment era The next step in the evolution of workloads deployment came with the advent of containerization. Containers work similarly to VMs, but with less strict isolation properties, allowing different applications to share the same operating system. For this reason, they are considered lightweight. Just like VMs, containers have their own file system, share of CPU, memory, process space and more. Containers are decoupled from the underlying infrastructure: this makes them portable across clouds and OS distributions. What makes them so popular is the set of extra benefits they offer, such as:

- Agile application creation and deployment, as container images are very easy to create compared to VM images.

- Continuous development, integration and deployment, thanks to reliable and frequent container image build and deployment.
- Application health checks and observability.
- Cloud and OS distribution portability.
- Application-centric management that raises the abstraction level to simply focus on application execution.
- Resource utilization that yields high efficiency and density.

In parallel with technological advancements, there has been an improvement in workload management methods: from handling VMs as single entities, we have moved to a "cattle" model, where VMs are treated in a more general way (although their management is still strongly coupled to their life), to then go further and achieve a decoupled approach, as used by Kubernetes: a declarative way that expresses general intentions that are taken by the system and applied to all interested resources, without having to deal with the individual instances. This leads to a more detached view where resources are seen as commodities that can be created, destroyed, and replaced as needed.

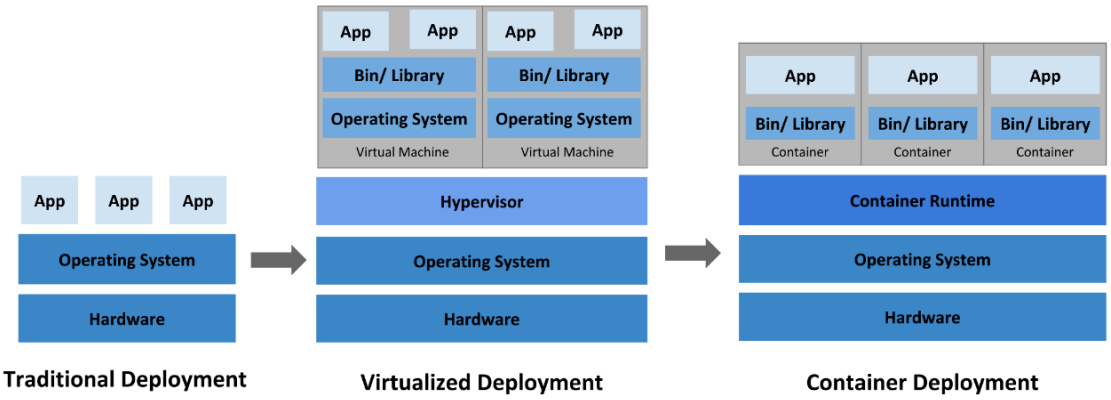

Figure 2.1: Evolution of applications deployments

2.3 Container orchestrators

When hundreds or thousands of containers are created, the need of a way to manage them becomes essential; container orchestrators serve this purpose. A container orchestrator is a system designed to easily manage complex containerized deployments across multiple machines from a central location. As shown in Figure

2.2, Kubernetes is by far the most commonly used container orchestrator. The following is a description of such system.

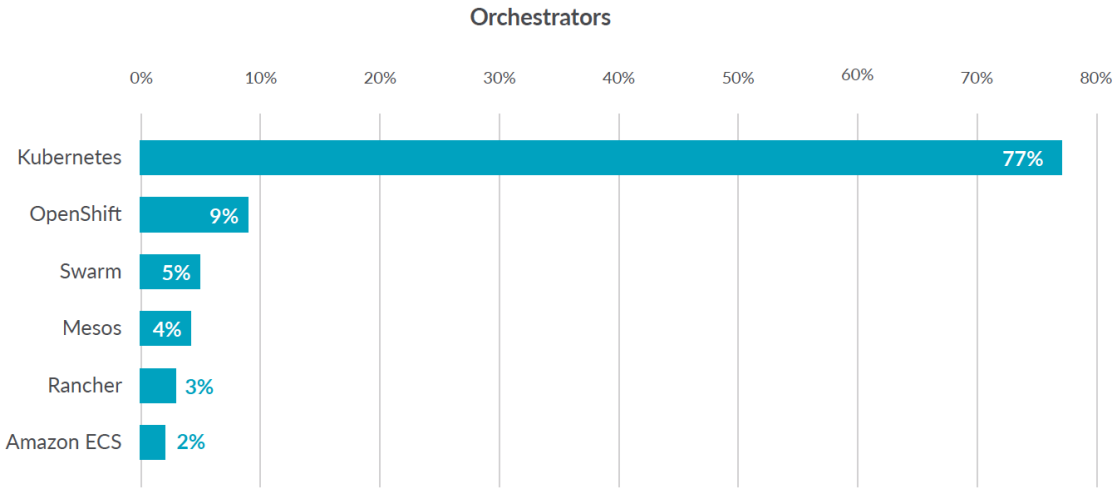

Figure 2.2: Container orchestrators market share [7]

Kubernetes provides many services, including:

- **Service discovery and load balancing** A container can be exposed using the DNS name or using its own IP address. If traffic to a container is high, a load balancer able to distribute the network traffic is provided.
- **Storage orchestration** A storage system can be automatically mounted, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** The desired state for the deployed containers can be described, and the actual state can be changed to the desired state at a controlled rate. For example, it is possible to automate the creation of new containers of a deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing** Kubernetes is provided with a cluster of nodes that can be used to run containerized tasks. It is possible to set how much CPU and memory (RAM) each container needs, and automatically the containers are sized to fit in the nodes to make the best use of the resources.
- **Secret and configuration management** It is possible to store and manage sensitive information in Kubernetes, such as passwords, OAuth tokens, and SSH keys. It is possible to deploy and update secrets and application configuration without rebuilding the container images, and without exposing secrets in the stack configuration.

2.4 Kubernetes architecture

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

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

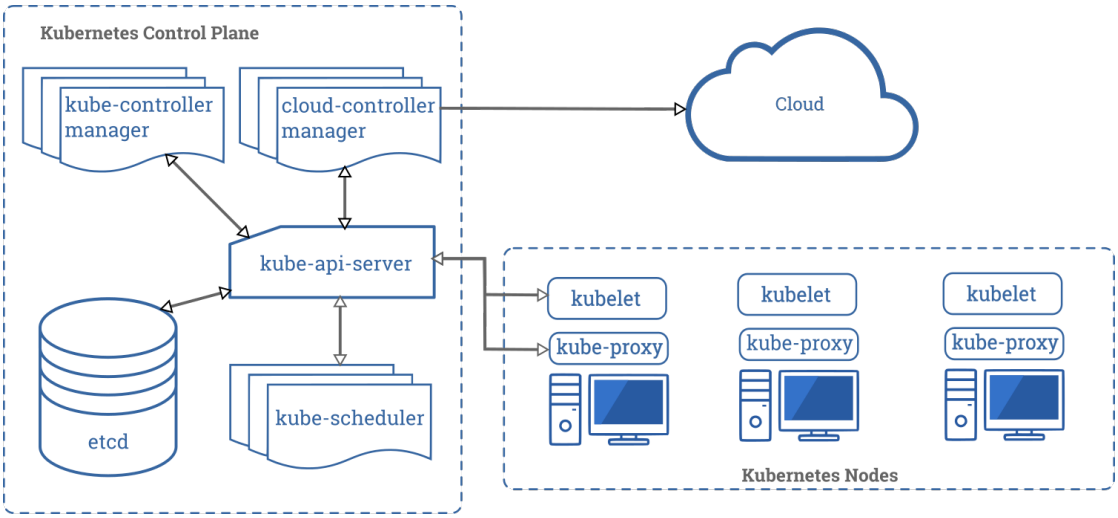

Figure 2.3: Kubernetes architecture

2.4.1 Control plane components

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

API server

The API server is the component of the Kubernetes control plane that exposes the Kubernetes REST API, and constitutes the front end for the Kubernetes control

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

etcd

`etcd` is a distributed, consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. It is based on the Raft consensus algorithm [8], which allows different machines to work as a coherent group and survive to the breakdown of one of its members. `etcd` can be stacked in the master node or external, installed on dedicated host. Only the API server can communicate with it.

Scheduler

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

kube-controller-manager

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

These controllers include:

- Node Controller: responsible for noticing and reacting when nodes go down.
- Replication Controller: in charge of maintaining the correct number of pods for every replica object in the system.
- Endpoint Controller: populates Endpoint objects (which link Services and Pods) [deprecated and substituted by the EndpointSlice API].
- EndpointSlice Controller: populates EndpointSlice objects (which link Services and Pods).

- Service Account & Token Controllers: create default accounts and API access tokens for new namespaces.

cloud-controller-manager

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

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

- Node Controller: checks the cloud provider to update or delete Kubernetes nodes using cloud APIs.
- Route Controller: responsible for setting up network routes in the cloud infrastructure.
- Service Controller: for creating, updating and deleting cloud provider load balancers.
- Volume Controller: creates, attaches, and mounts volumes, interacting with the cloud provider to orchestrate them.

2.4.2 Node components

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

Container runtime

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

kubelet

An agent that runs on each node in the cluster, making sure that containers are running in a pod. The `kubelet` receives from the API server the specifications of

the Pods and interacts with the `container` runtime to run them, monitoring their state and assuring that the containers are running and healthy. The connection with the `container` runtime is established through the Container Runtime Interface and is based on gRPC.

kube-proxy

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

Addons

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

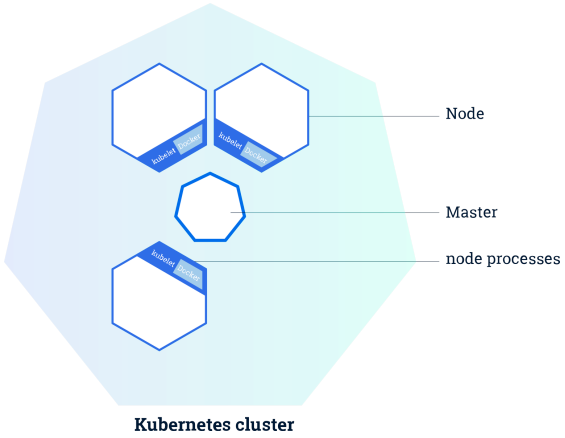

Figure 2.4: Kubernetes master and worker nodes [5]

2.5 Kubernetes objects

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

- `APIVersion` the versioned schema of this representation of the object
- `Kind`: a string value representing the REST resource this object represents

- `ObjectMeta`: metadata about the object, such as its name, annotations, labels etc.
- `ResourceSpec`: defined by the user, it describes the desired state of the object
- `ResourceStatus`: filled in by the server, it reports the current state of the resource

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

- **Create**: create the resource in the storage backend. Once a resource is created, the system applies the desired state
- **Read**: comes with 3 variants
  - **Get**: retrieve a specific resource object by name
  - **List**: retrieve all resource objects of a specific type within a namespace, and the results can be restricted to resources matching a selector query
  - **Watch**: stream results for an object(s) as it is updated
- **Update**: comes with 2 forms
  - **Replace**: replace the existing spec with the provided one
  - **Patch**: apply a change to a specific field
- **Delete**: delete a resource. Depending on the specific resource, child objects may or may not be garbage collected by the server

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

2.5.1 Labels and Selectors

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

2.5.2 Namespace

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

- `kube-system`: it contains objects created by K8s system, mainly control-plane agents

- `default`: it contains objects and resources created by users and it is the one used by default
- `kube-public`: readable by everyone (even not authenticated users), it is used for special purposes like exposing cluster public information
- `kube-node-lease`: it maintains objects for heartbeat data from nodes

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

2.5.3 Pod

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

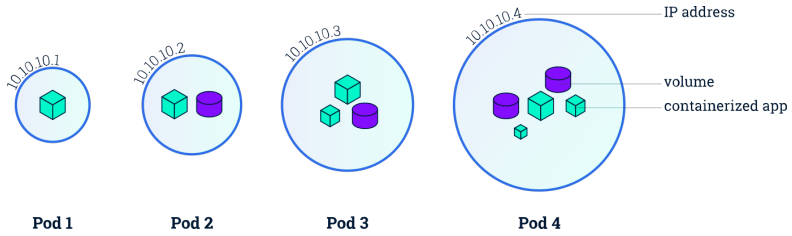

Figure 2.5: Kubernetes pods [1]

2.5.4 ReplicaSet

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

2.5.5 Deployment

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

Listing 2.1: Basic example of a Kubernetes Deployment [1]

```
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
```

The code above allows to create a Deployment with name `nginx-deployment` and a label `app`, with value `nginx`. It creates three replicated pods and, as defined in the selector field, manages all the pods labelled as `app:nginx`. The template field shows the information of the created pods: they are labelled `app:nginx` and launch one container which runs the nginx DockerHub image at version 1.14.2 on port 80.

2.5.6 DaemonSet

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

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

Listing 2.2: Basic example of a Kubernetes DaemonSet [1]

```
apiVersion: apps/v1
kind: DaemonSet
```

```
3  metadata:
4    name: fluentd-elasticsearch
5    namespace: kube-system
6    labels:
7      k8s-app: fluentd-logging
8  spec:
9    selector:
10     matchLabels:
11       name: fluentd-elasticsearch
12   template:
13     metadata:
14       labels:
15         name: fluentd-elasticsearch
16     spec:
17       tolerations:
18       # these tolerations are to have the daemonset runnable on
     control plane nodes
19       # remove them if your control plane nodes should not run pods
20       - key: node-role.kubernetes.io/control-plane
21         operator: Exists
22         effect: NoSchedule
23       - key: node-role.kubernetes.io/master
24         operator: Exists
25         effect: NoSchedule
26       containers:
27       - name: fluentd-elasticsearch
28         image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
29         resources:
30           limits:
31             memory: 200Mi
32           requests:
33             cpu: 100m
34             memory: 200Mi
35         volumeMounts:
36         - name: varlog
37           mountPath: /var/log
38       terminationGracePeriodSeconds: 30
39       volumes:
40       - name: varlog
41         hostPath:
42           path: /var/log
```

2.5.7 Service

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

- `ClusterIP`: service accessible only from within the cluster, it is the default

type

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

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

Listing 2.3: Basic example of a Kubernetes Service [1]

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

2.5.8 EndpointSlice

An EndpointSlice is an abstraction that contains references to a set of network endpoints of a service. It is created by the `kube-controller-manager` and contains a list of IP addresses and ports for each pod that backs the service. The EndpointSlice provides an alternative that is more scalable and extensible than the original and deprecated Endpoint resource. It tracks IP addresses, ports, readiness, and topology information for pods backing a service. Listing 2.4 shows an example of an EndpointSlice resource:

Listing 2.4: Basic example of a Kubernetes EndpointSlice [1]

```
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: example-eps
  labels:
    kubernetes.io/service-name: example
```

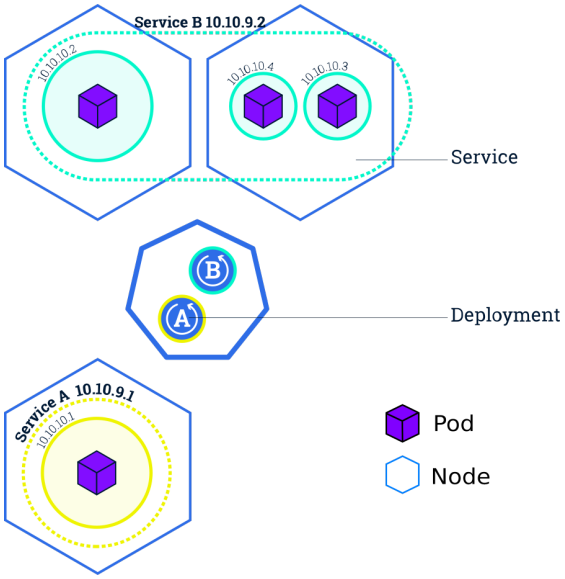

Figure 2.6: Kubernetes Services [1]

```
addressType: IPv4
ports:
  - name: http
    protocol: TCP
    port: 80
endpoints:
  - addresses:
      - "10.1.2.3"
    conditions:
      ready: true
    hostname: pod-1
    nodeName: node-1
    zone: us-west2-a
  - addresses:
      - "10.1.2.7"
    conditions:
      ready: false
    hostname: pod-2
    nodeName: node-1
    zone: us-west2-a
```

2.6 Kubernetes networking architecture

Kubernetes defines a network model that provides simplicity and consistency across a range of networking environments and network implementations. The Kubernetes

network model provides the foundation for understanding how containers, pods, and services communicate with each other within Kubernetes [9]. The Kubernetes network model specifies:

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

As a result, pods can be treated similarly to VMs or hosts (they all have unique IP addresses) and the containers within pods can be treated similarly to processes running within a VM or host (they run in the same network namespace and share an IP address). This model makes it easier to migrate applications from VMs and hosts to pods managed by Kubernetes. In addition, because isolation is defined by network policies rather than the structure of the network, the network remains simple to understand. This type of network is sometimes referred to as a "flat network".

2.6.1 Container communication within same pod

Containers in a pod are accessible via `localhost`, they use the same network namespace. For containers, the observable hostname is the name of the pod. Since containers share the same IP address and port space, containers must use different ports for incoming connections. For this reason, applications in a pod must coordinate the use of ports.

2.6.2 Pod communication within the same node

Before the infrastructure container is started, a virtual Ethernet interface pair (a `veth` pair) is created for the container. One interface of the `veth` pair remains in the host's namespace (it is tagged `vethxxx`), while the other interface is moved to the container's network namespace and renamed `eth0`. These two virtual interfaces

are like two ends of a pipe where everything that goes in one side comes out on the other. The interface in the host's network namespace is attached to a network bridge that the container runtime is configured to use. The `eth0` interface in the container is assigned an IP address from the bridge's address range. Everything that the application running in the container sends to the `eth0` network interface and comes out at the other `veth` interface in the host's namespace is sent to the bridge. Thus, any network connected to the bridge can receive it.

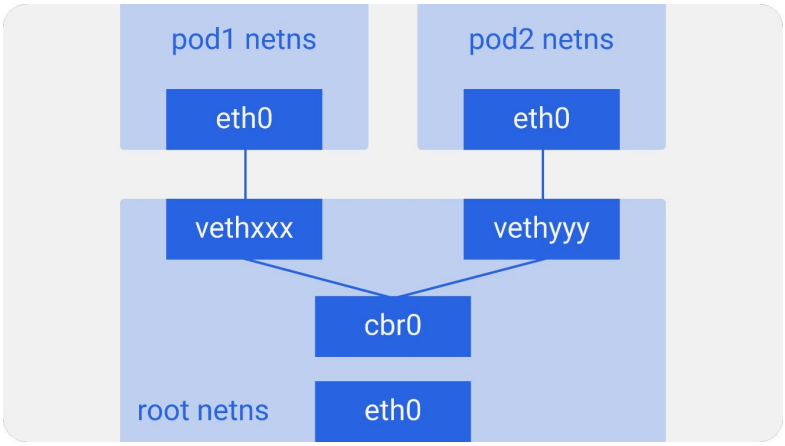

Figure 2.7: Pod to pod communication within same node

2.6.3 Pod communication on different nodes

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

2.6.4 CNI (Container Network Interface)

CNI (Container Network Interface) is a Cloud Native Computing Foundation project that consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers. CNI only takes care of container network connectivity and removing allocated resources when the container is deleted. Kubernetes uses the CNI specifications and plugins to orchestrate networking. It can also address the IP addresses of other containers without using the Network Address Translation (NAT). Each time a pod is initialized or removed, the default CNI plugin is invoked with the default configuration. This CNI plugin creates a pseudo-interface, connects

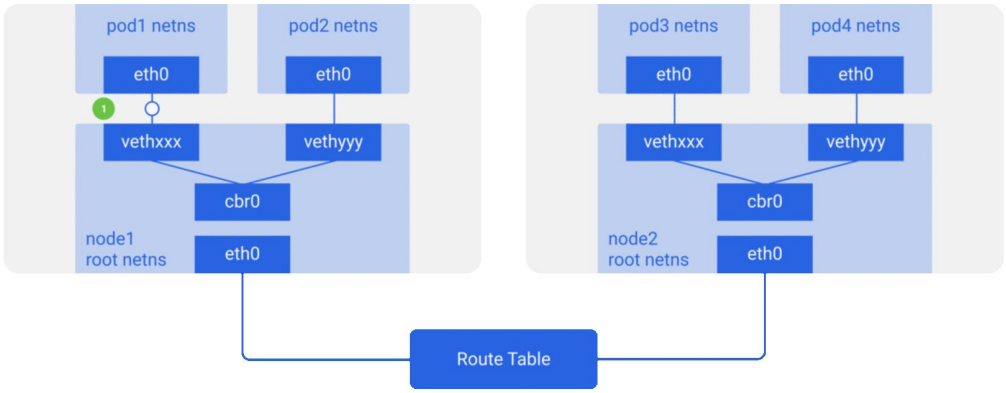

Figure 2.8: Pod to pod communication across different nodes.

it to the underlay network, sets the IP address and routes, and maps them to the pod's namespace.

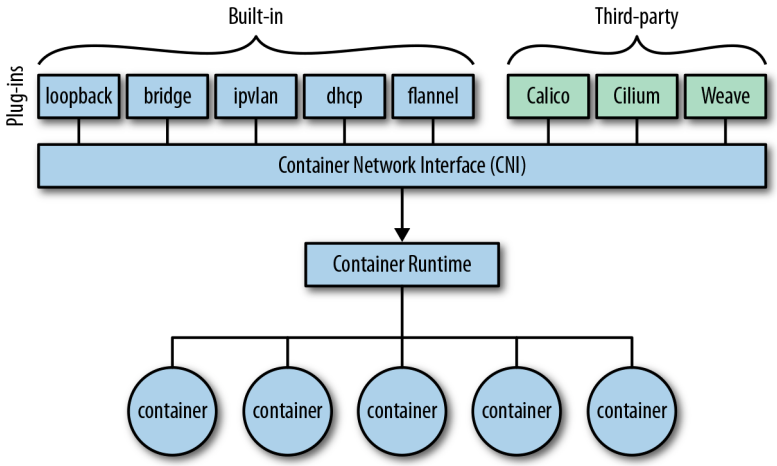

Figure 2.9: Container network interface [10]

2.6.5 Pod to service networking

Pod IP addresses are not permanent and will appear and disappear in response to replica scaling up or down, application crashes, or node restarts. Any of these events can cause the pod's IP address to change without warning. To address this issue, services have been integrated into Kubernetes. The Kubernetes service manages the state of pods and allows us to keep track of a set of pod IP addresses that change dynamically over time. Services act as an abstraction over pods and assign a single virtual IP address to a group of pod IP addresses. Any traffic

addressed to the service's virtual IP is forwarded to the group of pods associated with the virtual IP. In this way, the set of pods associated with a service can be changed at any time. Clients only need to know the virtual IP of the service, which does not change [11]

2.7 Kubebuilder

Kubebuilder [12] is a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs).

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

A **CustomResource** is an endpoint in the Kubernetes API that is not available in a default Kubernetes installation and frees users from writing their own API server to manage it [1]. With custom resources, you can easily store and retrieve structured data. In order to have a more powerful management, you also need to provide a custom controller which executes a control loop over the custom resource it watches: this behaviour is called Operator pattern [13].

Kubebuilder helps a developer define his Custom Resources, take automatically basic decisions and write a lot of boilerplate code. These are the main actions performed by Kubebuilder:

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

Chapter 3

Liqo

This chapter presents the conceptual foundations of Liqo [14] and the core elements that make up its architecture.

3.1 An overview of Liqo

Kubernetes technology is widely used to handle cloud tasks. Clusters are designed to provide more resources (in terms of sheer computing power, available memory, and storage capacity) than the ones normally required to handle temporary load spikes. This means that this excess capacity can be used by other clusters that are less heavily loaded for a period of time. Liqo aims to unleash this potential power by connecting clusters together and letting them work synergically to pursue their goals.

To accomplish this task, clusters establish a peering session that results in a larger virtual cluster that hosts the sum of the resources exposed by each cluster involved in the peering process.

The advantage of Liqo is that it takes the core concepts known in the Kubernetes environment and leverages them to achieve more capabilities. Indeed, a cluster simply sees its peers as (virtual) nodes that add up to its (physical) ones, and schedules tasks to its nodes regardless of their actual nature.

The next sections describe the concepts presented in more detail, starting with a core element and how to set it up: the Liqo peering.

3.2 Liqo Peering

Once two (or more) Kubernetes clusters are available to host workloads, they can become part of a multi-cluster topology by enabling a peering session between them. This is where the Liqo experience starts off. A Liqo peering connects separate

entities into a larger environment capable of handling larger workloads. As a result, each involved cluster becomes aware of the existence of other remote peers, modeled by the ForeignCluster Custom Resource (CR). This process involves exchanging network parameters and other cluster information to create a secure VPN that pods will leverage to communicate with each other as part of a large, distributed, cross-cluster application.

Cluster peerings are not required to be symmetric. Their flexibility allows a cluster to establish:

- **an outgoing peering**, so that the cluster can offload its workloads, but won't receive any by its peer.
- **an incoming peering**, so that the cluster hosts remote workloads, but won't offload any to its peer.
- **a bidirectional peering**, the union of the two above.

When an outgoing peering is active, it is of utmost importance to control what can and cannot be offloaded. This is done by leveraging some native Kubernetes concepts, namely Namespaces and label selectors, as well as some logic provided by Liqo to select which namespaces to offload, which pods within those namespaces to offload, and even which remote peers are the target of this offloading mechanism. The possibilities are endless.

The basic requirement for starting a peering session is to have access to the remote Kubernetes API server. This allows clusters to exchange information and create resources remotely. The result is a VPN that remote pods use to communicate as if they were all in the same Kubernetes cluster.

3.3 Liqo Reflection

Once a peering is established, the workload offloading is enabled by leveraging the virtual node abstraction and the namespace extension.

A virtual node represents a remote cluster and all of its shared resources (e.g., CPU and memory). This enables transparent extension of the local cluster's resources, as the virtual node added to the cluster is seamlessly taken into account by the vanilla Kubernetes scheduler when selecting the best place to execute workloads.

In addition, Liqo allows Kubernetes namespaces to be extended across cluster boundaries. Once a namespace is selected for offloading, Liqo automatically creates twin namespaces in the selected subset of remote peers. These remote twin namespaces host the remotely offloaded pods as well as other resources that live in the local namespace that has been extended remotely, such as those related to service

exposition (Ingress, Service, and Endpoints resources) or storing configuration data (ConfigMaps and Secrets), to name a few.

3.4 Network Fabric

The network fabric is the Liqo subsystem that transparently extends the Kubernetes network model across multiple independent clusters, allowing offloaded pods to communicate with each other as if they were all running locally.

Specifically, the network fabric ensures that all pods in a given cluster can communicate with all pods on all remote peered clusters, either with or without NAT translation. The support for arbitrary clusters, with different parameters and components (e.g., CNI plugins), makes it impossible to guarantee non-overlapping pod IP address ranges (i.e., PodCIDR). Therefore, address translation mechanisms may be required, provided that NAT-less communication is preferred whenever address ranges are disjointed.

Figure 3.1 represents at a high level the network fabric established between two clusters, with its main components detailed in the following.

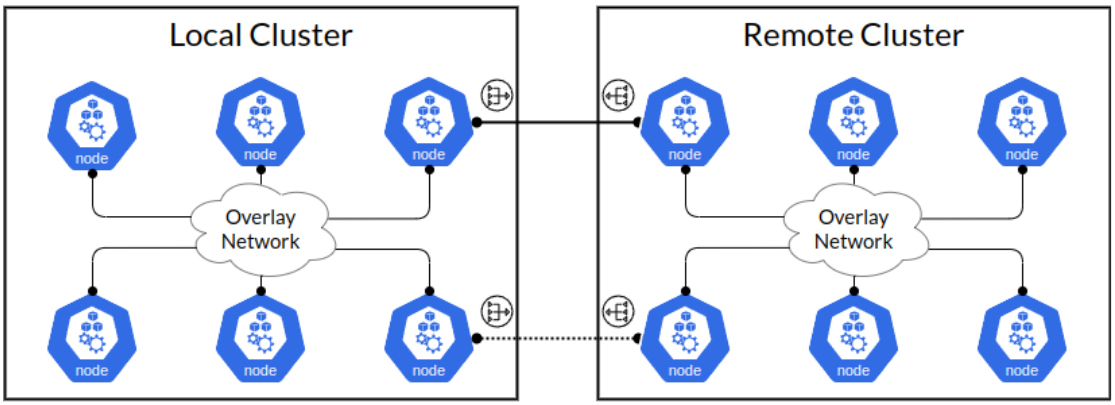

Figure 3.1: Network Fabric

3.4.1 Cross-cluster VPN tunnels

The interconnection between the peered clusters is implemented through secure VPN tunnels made with **WireGuard** and dynamically established at the end of the peering process, based on the negotiated parameters.

Tunnels are set up by the Liqo gateway, a component of the network fabric that runs as a privileged pod on one of the cluster nodes. It also populates the

routing table appropriately and uses **iptables** to configure the NAT rules required to comply with address conflicts.

Although this component is executed in the host network, it relies on a separate network namespace and policy routing to ensure isolation and prevent conflicts with the existing Kubernetes CNI plugin. Moreover, active/standby high-availability is supported, to ensure minimal downtime in case the main replica is restarted.

3.4.2 In-cluster overlay network

The overlay network is used to forward all traffic originating from local pods/nodes, and directed to a remote cluster, to the gateway, where it will enter the VPN tunnel. The same process takes place on the other side, where traffic leaving the VPN tunnel enters the overlay network to reach the node hosting the destination pod.

Liqo leverages a VXLAN-based setup, which is configured by a network fabric component executed on all physical nodes of the cluster (i.e., as a DaemonSet). Additionally, it is also responsible for the population of the appropriate routing entries to ensure correct traffic forwarding.

3.5 Liqo CRDs

The following subsections introduce some of the Custom Resources that Liqo uses to provide the peering and reflection features.

3.5.1 NetworkConfig CR

This CR represents a set of network parameters (mainly IP addresses) that clusters use to know how a remote peer has remapped the local PodCIDR, as well as the remote peer's PodCIDR. The "spec" part contains data related to the local cluster, while its "status" part reports the changes to the specifications. The idea is that a cluster creates this CR and sends it to the remote cluster it is going to establish a peering with. The remote cluster processes this CR and annotates in the "status" part everything it needed to change in terms of IP address ranges to avoid conflicts. These updates are reported back to the owning cluster.

At the same time, the same thing happens in the opposite direction, i.e. the remote cluster generates a NetworkConfig, writes its "spec" part, and sends it to the local cluster, which annotates any changes in the "status" part to make the remote cluster aware of any changes to the original specifications.

Once both CRs are processed, a Liqo control loop reconciles them to create the TunnelEndpoint CR.

3.5.2 TunnelEndpoint CR

This CR contains the relevant network configuration to establish a VPN tunnel with the remote cluster. This allows pods to reach other remote pods as if they were on the same network.

3.5.3 ForeignCluster CR

This CR models a remote cluster. It contains details about the peering session that is in place between two clusters, such as whether the peering was successfully established and in which direction it is going (outgoing, incoming, or both). A ForeignCluster is created starting from the NetworkConfig that the two parties have exchanged and processed.

3.5.4 ShadowPod CR

When a pod is scheduled on a virtual node, a pod is created in the remote cluster for the actual workload execution. In the remote cluster, a new object paired with the remote pod is created: the ShadowPod. This resource, combined with its controller, guarantees the presence of the pod in the remote cluster, even in case of connection failures.

3.6 Liqo Components

3.6.1 CRD Replicator

This component is dedicated to the reflection of some Liqo CRs just presented. To do so, it requires access to the remote API server. It is a core element as it implements the network parameters exchange between clusters to set up the TunnelEndpoint CRs, which are later used to keep track of the active peering sessions and ensure remote pod-to-pod communication. The replicated CRDs are:

- NetworkConfig
- ResourceRequest
- ResourceOffer
- NamespaceMapping

The CRD Replicator architecture is quite complex, but essentially it is implemented through a so-called reflector. This is a data structure that contains the objects and data necessary to detect changes in local and remote namespaces (using

local and remote informers), as well as to perform traditional CRUD operations in those namespaces (using local and remote clients). When an object, such as a NetworkConfig, is created in a namespace enabled for reflection and with the appropriate metadata labels, the local reflector (i.e., the one belonging to the cluster that created the object) performs the following steps:

- it detects a new object to be reflected.
- it creates a copy of that object in the remote namespace by using a pre-configured client to access the remote API server.
- it listens to any changes occurring in the reflected object, which usually boils down to a status update performed by the remote cluster controllers, as happens with NetworkConfig to let the sender cluster know about possible remappings.
- it listens to any changes occurring in the local original copy, such as a deletion that needs to propagate to the remote cluster's namespace so that the remote copy gets deleted as well.

3.6.2 Virtual Kubelet

This component is a custom version of the Virtual Kubelet project [15]. Whenever a peering session is established with a remote cluster, a dedicated instance of this component is created. Once created, it is used to offload pods to remote clusters, seen by the Kubernetes control plane as normal cluster nodes on which a normal task can be scheduled. In addition, it is used to reflect core Kubernetes resources, such as Services and Endpoints: once deployed in a Liqo-enabled namespace (i.e., a namespace extended remotely) they will always be reflected to the selected remote peers.

3.6.3 IPAM component

This component contains the logic that translates IP addresses back and forth and keeps track of all possible remappings between the local cluster and the remote peers. It is fundamental to Liqo because it knows all the NAT rules used to avoid address conflicts.

3.6.4 Network manager

The network manager represents the control plane of the Liqo network fabric. It is executed as a pod, and it is responsible for the negotiation of the connection parameters with each remote cluster during the peering process.

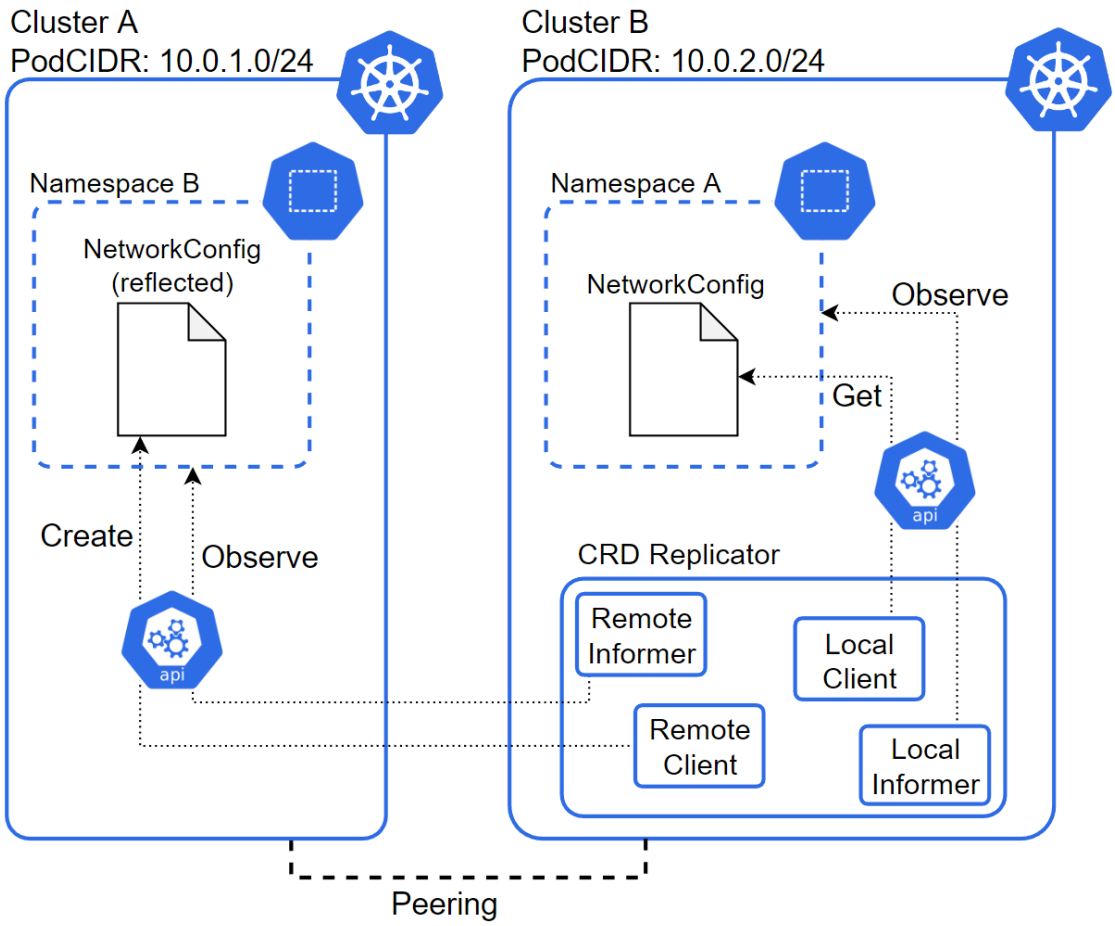

Figure 3.2: CRD Replicator

It features an IP Address Management (IPAM) plugin that handles potential network conflicts by defining high-level NAT rules (enforced by data plane components). It also expose an interface used by the reflection logic to handle IP addresses remappings. In particular, this is leveraged to handle the translation of pod IPs (i.e., during the synchronization process from the remote to the local cluster), as well as during EndpointSlices reflection (i.e., propagated from the local to the remote cluster).

3.6.5 Liqo Gateway

This component is responsible for managing connections with other clusters. All traffic between two peered clusters must pass through this component. It is possible to have more than just one Liqo Gateway, but only one can be active at a time and the others can be used in case of failures. The connection between the clusters is

managed with VPN tunnels and this component is responsible for managing these tunnels. Liqo supports more VPN drivers (e.g., WireGuard, OpenVPN, IPSec) and provides an interface to implement the logic. However, only the WireGuard driver is implemented at the moment.

Chapter 4

Business Continuity in Kubernetes

Business continuity in the context of cloud computing refers to the ability of an organization to maintain its critical business functions and processes in the event of a disruption. Cloud computing can provide the infrastructure, platforms, and services necessary for organizations to maintain their operations during and after an interruption. This includes providing a secure, resilient and scalable environment to store and process data, as well as the ability to recover quickly from unexpected events or outages. For example, if an organization's data center is affected by a power outage, the cloud computing provider can automatically reroute processing and storage to another location, reducing downtime and minimizing the impact on business operations. Cloud providers typically offer service-level agreements (SLAs) that guarantee uptime and availability. This can provide additional assurance to businesses that their critical systems will remain operational in the event of an unexpected event. This can include identifying critical applications and data that need to be replicated and backed up, and setting recovery time objectives (RTOs) and recovery point objectives (RPOs). Business continuity is achieved through a combination of technologies and best practices, such as disaster recovery planning, data backup and recovery, high availability (HA), redundancy, DevOps, and CI/CD techniques.

Business continuity can be divided into two main categories:

- **Service Continuity**: the ability to provide services or products 24/7, even in the face of unforeseen events such as natural disasters, cyberattacks, power outages, or spikes in demand.
- **Disaster Recovery**: the process of restoring critical systems and data after a disruptive event. Disaster recovery planning involves identifying critical

systems and data, and establishing backup and recovery procedures.

The goal The goal is to analyze how business continuity is achieved in a Kubernetes [1] single-cluster, so that we can better understand and extend the analysis in a multi-cluster topology in the next chapters. In this chapter, it is presented an overview of the current state of the art solutions for achieving service continuity and disaster recovery. Chapter 5 looks at service continuity in a multi-cluster environment powered by Liqo [16], while chapter 6 describes a possible disaster recovery solution that leverages the potential of Liqo to easily use peered clusters as failover sites.

4.1 Service Continuity

In Kubernetes, service continuity is achieved through a combination of high availability and fault tolerance features.

Node recovery Kubernetes clusters typically consist of multiple nodes, which are individual machines or virtual machines that host containers. Nodes are the building blocks of a Kubernetes cluster and can be added or removed from the cluster at any time. When a new node is added, Kubernetes can automatically schedule pods to run on that node. When a node becomes unavailable, Kubernetes detects the failure and reschedules the affected pods to other available nodes. This process is called node recovery and ensures that workloads continue to run even if a node fails.

Pod rescheduling Pods can be created, updated, or deleted independently of the underlying nodes, enabling seamless deployment and management of applications. Once created, pods can be updated, scaled, or deleted based on application needs. Kubernetes also monitors the health of pods and can automatically restart them if they fail or become unresponsive.

Control plane high availability In Kubernetes, the control plane refers to the set of components that manage and control the cluster's state, including the API server, etcd database, kube-controller-manager and kube-scheduler. The control plane is critical to the functioning of the cluster, and any failure can lead to application outages and service interruptions.

To ensure high availability of the control plane, Kubernetes provides several built-in mechanisms that allow control plane components to run on multiple nodes and avoid single points of failure. These mechanisms include:

- replication: Kubernetes uses multiple replicas of each control plane component to ensure high availability. Each replica is scheduled on a different node, and replicas communicate with each other to maintain a consistent cluster state.
- self-healing: Kubernetes automatically monitors the health of control plane components and can automatically detect and fix failures. If a component fails, Kubernetes can automatically restart it or create a new replica to replace it.
- backup and restore: Kubernetes provides backup and restore mechanisms for the etcd database, which stores the state of the cluster.

Figure 4.1 shows an example of a HA control plane with a stacked etcd configuration.

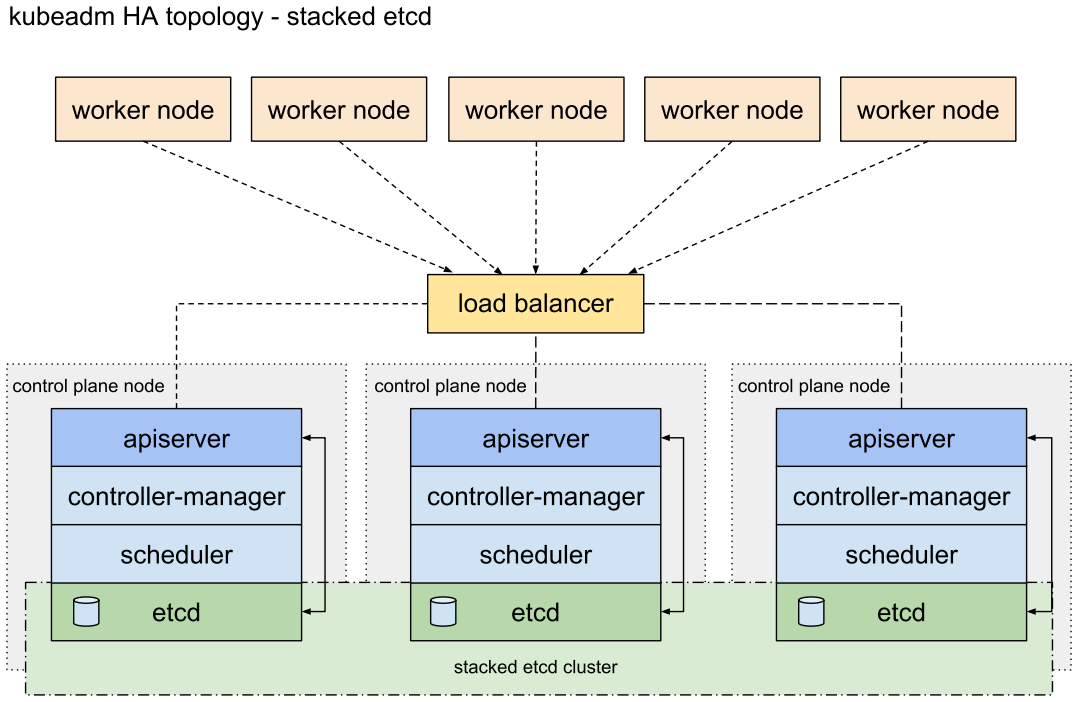

Figure 4.1: High availability control plane with stacked etcd [1]

4.1.1 Pods and Nodes lifecycles

This section describes the lifecycle of nodes and pods in Kubernetes in detail.

Kubernetes uses a heartbeat mechanism to monitor the health of nodes in the cluster. The node heartbeat mechanism is responsible for periodically sending a signal from the node to the Kubernetes control plane indicating that the node is still active and responding. This mechanism is implemented using a combination of the Kubernetes lease API and the Node Controller, a process running on the `kube-controller-manager` component that is responsible for managing the lifecycle of nodes in the cluster. The following describes how the node heartbeat mechanism works.

- When a node is added to the cluster, the Kubernetes Node Controller creates a lease in the API server. The lease has a specific duration and is associated with the name of the node.
- Each node's kubelet periodically updates the lease to indicate that it is still active and responding. It does this by updating the lease object with a new timestamp. The lease is updated every `node-status-update-frequency` seconds.
- In parallel, the kubelet also updates its status conditions [17], but at a lower frequency.
- The node controller monitors the status of the node and updates the lease every `node-monitor-period` seconds. It updates the *Ready* condition to:
  - *True* if the node is healthy and ready to accept pods
  - *False* if the node is not healthy and does not accept pods
  - *Unknown* if the node controller has not heard from the node in the last `node-monitor-grace-period` seconds
- If the status of the *Ready* condition remains *Unknown* or *False* for longer than `pod-eviction-timeout` seconds, then the node controller triggers the **API-initiated eviction** for all pods assigned to that node. In some cases where the node is unreachable, the API server is unable to communicate with the kubelet on the node. The decision to delete the pods cannot be communicated to the kubelet until communication with the API server is restored. For this reason, the pod is not deleted from the API server and it is left in a *Terminating* state. In the meantime, pods scheduled for deletion may continue to run on the partitioned node.

All of the above settings can be configured at cluster level and/or application level. Tuning these parameters can be useful to better manage certain use cases and guarantee certain SLAs. Table 4.1 shows an overview of these configuration parameters with their default value and how they can be configured. Figure 4.2 shows a diagram of the overall process.

Setting	Description	Default	Configurable
node-status-update-frequency	Frequency with which the kubelet updates its status	10 sec.	Yes, in the the kubelet of each node
node-monitor-period	Frequency with which the kube-controller-manager checks nodes status	5 sec.	Yes, in the control plane
node-monitor-grace-period	Maximum amount of time which the node is allowed to be unresponsive before it is marked as unhealthy	40 sec.	Yes, in the control plane
pod-eviction-timeout	Interval before the kube-controller-manager marks as *Terminating* the pods assigned to a *NotReady* node	5 min.	Yes, in the control plane and at application level using *tolerations*

Table 4.1: Main Kubernetes settings to manage pods and nodes lifecycles

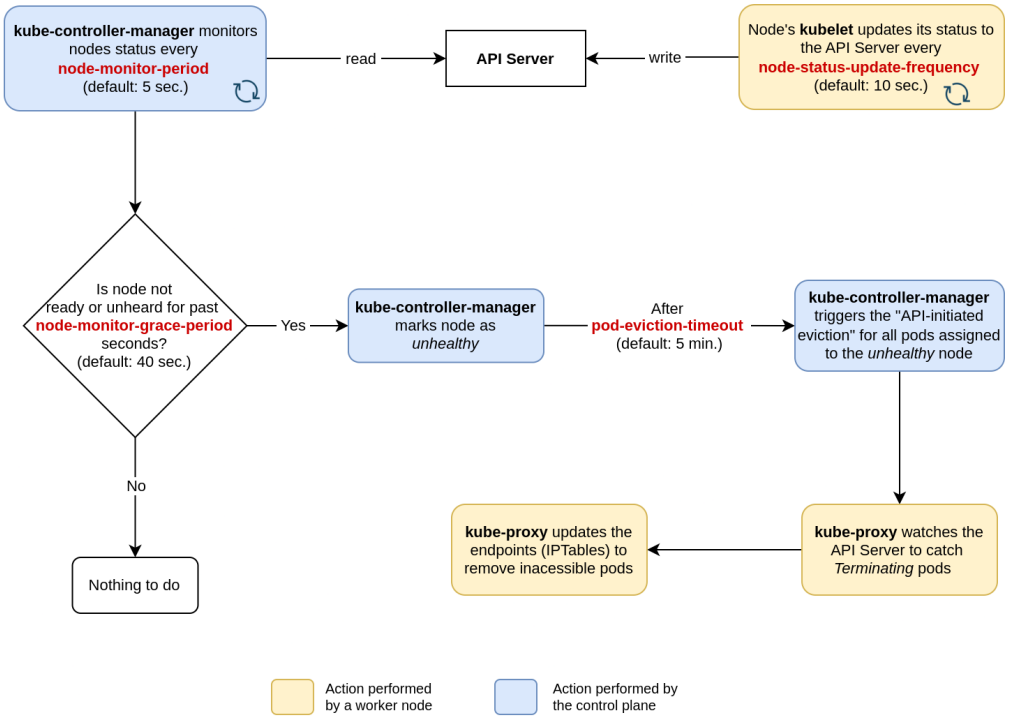

Figure 4.2: Node failure diagram

4.1.2 Service Mesh solutions

Unfortunately, Kubernetes alone is not always enough to ensure service continuity. K8s is a deployment platform that provides some basic service-to-service communication capabilities, such as DNS-based service discovery and L3/L4 load balancing. This might be insufficient for critical workloads.

A widely used solution to improve service communication in a microservices-based architecture is the **Service Mesh**. A service mesh is a dedicated infrastructure layer that abstracts away the underlying network topology and enables service-to-service communication within a microservices architecture. Service mesh solutions can improve service continuity in several ways:

- **Load balancing**: service mesh solutions can distribute traffic across multiple instances of a service, improving availability and reducing the risk of service disruptions due to a single point of failure.
- **Service discovery**: service mesh solutions can automatically detect and route traffic to available instances of a service. This ensures that requests are always routed to a healthy instance, even if others are experiencing issues.
- **Circuit breaking**: service mesh solutions can detect when a service is unresponsive or slow to respond and can quickly disconnect it from the network. This helps prevent cascading failures that can lead to service outages.
- **Automatic retries**: service mesh solutions can automatically retry failed requests to a service, which can help improve availability and reduce the risk of service disruptions due to temporary issues.
- **Canary deployments**: service mesh solutions can support canary deployments, where a new version of a service is rolled out incrementally to a small percentage of traffic before being deployed to the entire service. This can reduce the risk of service disruptions due to a faulty new version.

Depending on the technology, the service mesh infrastructure can be deployed either as a per-host proxy or as a sidecar proxy. Istio [18] and Linkerd [19] are some popular service mesh implementations for Kubernetes. Overall, service mesh solutions provide a centralized way to manage and control service-to-service communication, which can help improve service continuity and reduce the risk of downtime in a microservices architecture.

4.2 Disaster Recovery

Kubernetes is the de facto platform for stateless applications, but it also natively supports stateful workloads. In this context, it is necessary to include a disaster

recovery (DR) strategy to ensure business continuity in the event of a disruptive event. The challenge is to provide a flexible environment capable of sustaining the requirements of the business, while minimizing costs and complexity.

Metrics A good DR plan must define a set of business requirements that depend on the type of application. The two most important metrics to optimize are:

- **RTO**, or Recovery Time Objective, is the maximum amount of time an organization can tolerate for a system to be down before it negatively impacts the business. It is the amount of time it takes to recover a system after a disaster or disruption has occurred. RTO is typically measured in hours, minutes or even seconds and is an important metric for disaster recovery planning. RTO helps organizations understand how long they can be without a particular system and guides the development of a disaster recovery plan that includes strategies to reduce downtime and ensure systems can be restored as quickly as possible.

- **RPO**, or Recovery Point Objective, is the maximum amount of data an organization can afford to lose in the event of a disruption or disaster. It is the point in time to which data must be recovered in order to resume normal operations after a disruption. RPO is typically measured in hours, minutes or even seconds and helps organizations understand the amount of data loss they can tolerate before it starts to have a significant impact on their operations. RPO is also an important metric in disaster recovery planning. It is used to guide the selection of backup and recovery strategies that minimize data loss and ensure that critical data can be recovered quickly.

Together, RTO and RPO help businesses understand how quickly they need to recover from a disaster, and how much data loss they can tolerate.

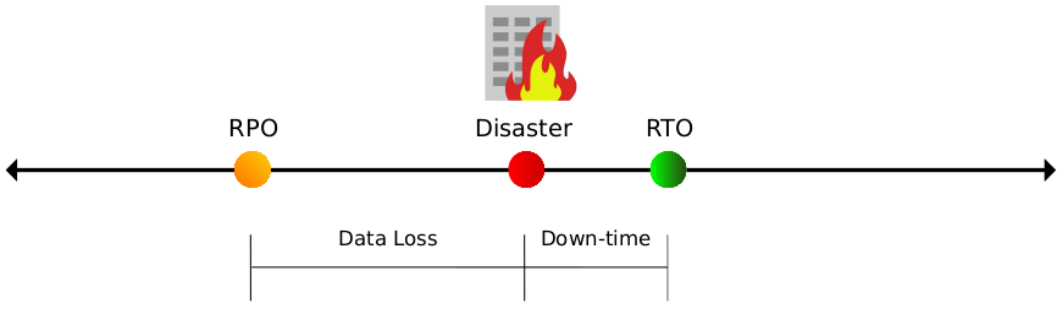

Figure 4.3: RTO and RPO metrics [20]

Traditional vs cloud-era approach The most common approach to disaster recovery is to use a failover site to restore the application state and data. In the pre-container era, the traditional approach was to perform regular backups on the running server and restore the backup on a new server. This led to a strong bond between applications and the specific servers on which they were deployed. The side effect is that additional technologies, configuration and coordination are required to maintain these complex setups. Also, the time required to fully transition the application to the recovery site is often not negligible, making it difficult to meet RTO requirements. Several techniques have been introduced to shorten RTOs and RPOs, such as:

- continuous replication instead of periodic backups
- recovery data center active and online (read-only mode) instead of idle and offline

The advent of cloud technologies, such as virtual machines and containers, allowed to decouple the applications deployment from the infrastructure on which they run. Deployment platforms such as **OpenStack** [21] (VM-based) and Kubernetes (container-based) greatly simplify the deployment and management of these infrastructures, overcoming the limitations of the traditional approach.

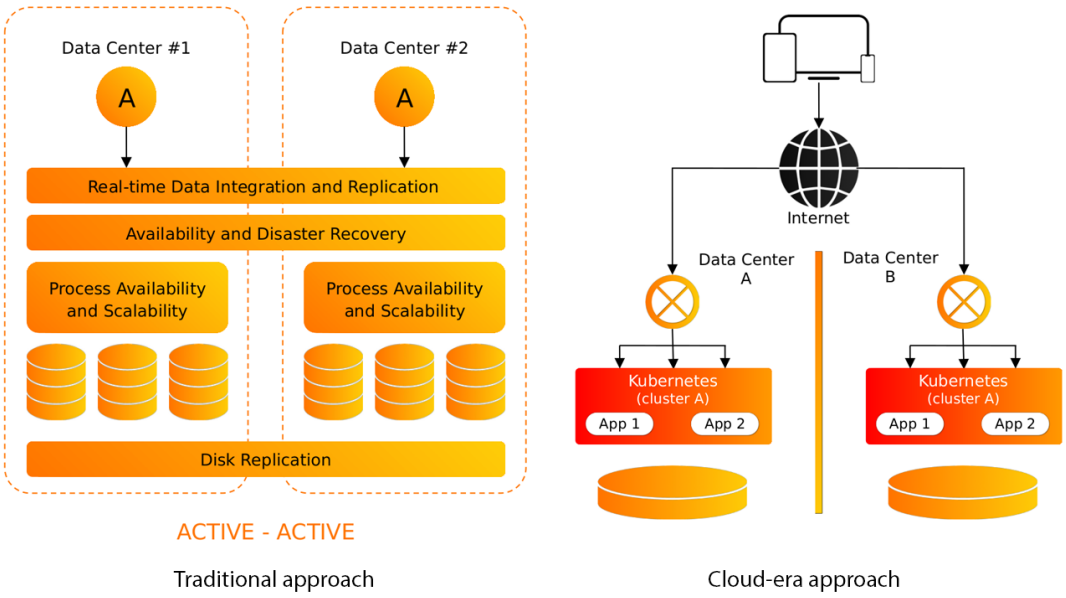

Figure 4.4: Disaster recovery: traditional vs cloud-era approach [20]

Kubernetes for DR Kubernetes as a platform can provide a flexible infrastructure to manage stateful deployments. The only stateful components are the etcd database and PVs (Persistent Volumes). The former contain cluster-level data (i.e., cluster state), while the latter are volumes that contain application-level persistent data. When talking about DR, it is important to distinguish between cluster-level and application-level backups.

- **Cluster-level backups**. Many existing tools provide this feature, such as **Velero**, an open source tool that can be used to backup and restore Kubernetes cluster resources, including persistent volumes, namespaces, deployments and more. Another approach is the one used by **Argo CD**, a GitOps continuous delivery tool for Kubernetes. It can be used for backup and recovery of the entire cluster.
- **Application-level backups**. Some CSI drivers implement the volume snapshots feature. This technique enables backup and restore of native stateful Kubernetes resources such as PVs. A non-CSI approach is to use the built-in redundancy replica sets offered by most modern databases.

This thesis focuses on analyzing disaster recovery techniques of application-level data leveraging a Kubernetes multi-cluster environment. Chapter 6 analyzes Percona [22], an open source operator for Kubernetes that can be used to easily manage the deployment of a MongoDB [23] server on Kubernetes. Percona allows to deploy multiple replicas of the database on multiple clusters, which are used as failover sites to perform disaster recovery. Given the multi-cluster approach used by Percona, the analysis is extended to leverage the potential of Liqo [14] to simplify and accelerate the entire process.

Chapter 5

Service Continuity with Liqo

This chapter analyzes how to achieve business continuity in a multi-cluster environment where clusters are peered and interconnected with Liqo. Clusters can offload resources to other clusters by delegating the actual execution of pods and/or the storage and management of data to the peered cluster. In this scenario, unexpected events or failures can occur at any time and must be accounted for.

5.1 Multi-cluster setup with Liqo

Thanks to Liqo, in a typical 2-cluster deployment, one cluster (the consumer) can offload tasks to a remote cluster (the provider), but not the other way around. In this case, we say that the consumer establishes an outgoing peering towards the provider, which in turn is subject to an incoming peering from the consumer. The peering is unidirectional, resulting in an asymmetric setup. However, bidirectional peering is transparently supported by combining outgoing peering with incoming peering. In addition, a cluster can offload to multiple clusters through multiple peerings. Since a single peering is unidirectional and involves only two clusters, it allows to consider only a simple consumer-provider setup, knowing that we can safely extend the considerations to more complex setups involving bidirectional peerings and/or multiple clusters. An example is shown in Figure 5.1.

As described in Section 3.3, the reflection logic replicates and synchronizes all necessary resources in the remote cluster. Liqo supports the reflection of the resources dealing with:

- **service exposition**: Ingresses, Services and EndpointSlices
- **persistent storage**: PersistentVolumeClaims and PersistentVolumes
- **storage of data**: ConfigMaps and Secrets

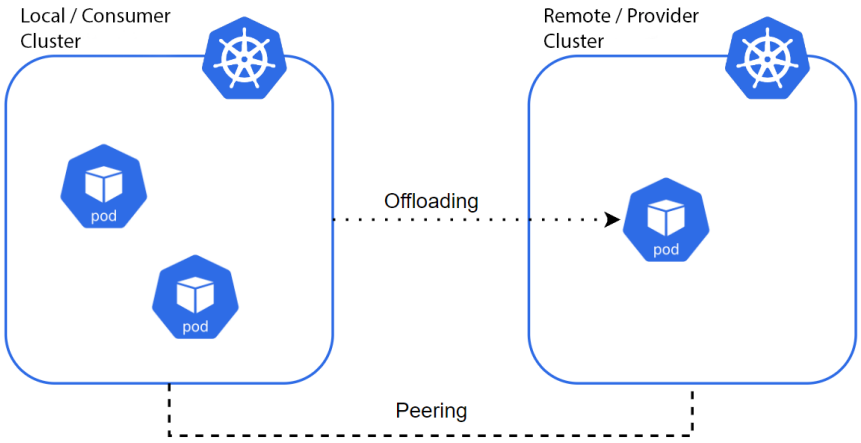

Figure 5.1: 2-cluster unidirectional deployment

Thus, when an application (or a client connecting to one of the clusters via an external endpoint) wants to access a microservice running in the cluster, it contacts the corresponding service, which has the IP endpoints of all the associated pods (whether they are local or remote). Figure 5.2 shows an overview of the reflection architecture for an application composed of three microservices (i.e., pods), namely P_1, P_2 and P_3, exposed through the respective service S_i, in turn associated with endpointslice E_i. P_1 and P_2 are executed on local workers, while $P_3 \equiv P'_3$ is offloaded to a remote cluster through a virtual node.

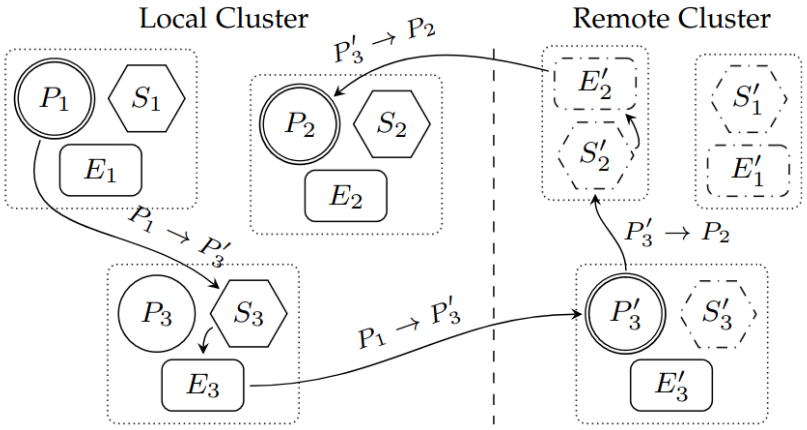

Figure 5.2: Communication patterns between three microservices spread across two different clusters through Liqo. Dashed polygons represent shadow resources, while double circles indicate that the pod is actually in execution

5.1.1 Application use cases and policies

The virtual node abstraction enables workload offloading. During the peering process, the virtual node is created in the consumer (i.e., local) cluster. It represents (and aggregates) the subset of resources shared by the provider (i.e., remote) cluster. This technique provides a transparent extension of the local cluster, allowing to interact with the node via the standard Kubernetes API, enabling the interaction and inspection of offloaded pods as if they were executed locally. The virtual node is often referred to as "liqo-node" or virtual kubelet (abbreviated VK). It is important to note that in the configuration described at the beginning, the virtual node is only present in the consumer cluster (i.e., the one who does the offloading). Only in the case of bidirectional peering do both clusters have the VK.

The first task that the VK performs concerns the creation and management of the virtual node that abstracts the resources shared by the remote cluster. In particular, it aligns the node status (i.e., whether it is ready and how many its available resources it has) with respect to the negotiated configuration. Periodic healthiness checks are performed to assess the reachability of the remote cluster, marking the virtual node as not ready in case of repeated failures. Upon this event, two possible policies are possible, depending on the application use case [24]:

1. **Elastic cluster**: standard Kubernetes logic is used to evict all pods hosted on the failing cluster and reschedule them in a different location to ensure service continuity. This can be used, for example, to balance and absorb load spikes (*cloud bursting*). In this use case, it is often assumed that every other available cluster potentially has the resources needed to run the application. Indeed, clusters are said to be homogeneous because they can potentially perform the same tasks.

2. **Super cluster**: disconnections are explicitly foreseen and shall be tolerated (e.g., to account for edge devices in harsh environments). Existing workloads can evolve independently through the remote orchestration logic, with the virtual node no longer considered a valid scheduling target only for new applications. This can be achieved by setting the appropriate taints and tolerations.

5.1.2 High Availability (HA) Liqo components

In Liqo, critical components can be replicated across different nodes to achieve HA. Two or more replicas (in an active/standby configuration) of the gateway ensure that if one of the replicas is restarted, there is no cross-cluster connectivity downtime. The controller manager, which embeds the Liqo control plane logic, can also be replicated.

5.2 Failure scenarios

In real-world production environments, unexpected events or failures in the infrastructure are very common and cannot be neglected. A typical class of problems concerns the health of a node in a cluster. A node might be not healthy in case of saturation of one of its resources (e.g., disk, memory, number of processes, etc.) or in case of physical hardware failure. Another class of problem is the loss of connectivity: in our setup, it could be intra-cluster or, most often, inter-cluster, especially in the case of high-latency WAN links that connect geographically distant clusters. In the following, different failure scenarios are presented, considering how they affect both the expected workload and the availability of the app (*service continuity*).

5.2.1 Local worker node failure

In this scenario, a worker node in the local (i.e., consumer) cluster becomes not ready or unreachable. Since it is a standard Kubernetes node, the pods scheduled on it are not managed by Liqo. Consequently, their lifecycle is like described in Section 4.1.1: if the node stay unresponsive for `node-monitor-grace-period` seconds (usually tens of seconds), the *kube-controller-manager* in the control plane marks the node as *NotReady*. Immediately, all endpoints served by the failed node are invalidated (by updating the *Ready* condition of all endpoints in their respective endpointslice). In this way, services do not redirect traffic to pods running on the failed node, and service continuity is ensured (assuming there are enough pods running on other healthy nodes to handle the load). After another grace period called `pod-eviction-timeout`, all pods in the failed node are marked for deletion. Since this timeout can be modified at the application level, it may be useful to tune it accordingly to the needs of the application. In cases where the node is unreachable, the API server cannot communicate with the kubelet on the node. The decision to delete the pods cannot be communicated to the kubelet until communication with the API server is re-established. The node controller does not force the deletion of such pods until it is confirmed that they are no longer running in the cluster. For this reason, pods with a *DeletionTimestamp* are not evicted by the API server, but remain in status *Terminating* or *Unknown* indefinitely. In cases where Kubernetes cannot determine if a node has permanently left a cluster, the cluster administrator may need to manually delete the node object. Kubernetes does not automatically force the deletion of pods, because in the eventuality that the node returns ready again and the OS has never been restarted, all processes in the pods become zombies. However, if pods remain *Terminating* indefinitely, new pods on other working nodes are scheduled as replacements so that the expected workload is guaranteed after the second grace period.

In summary, both service continuity and the expected workload are provided by standard Kubernetes in a relatively short time. The downtime is configurable and can be lowered if needed. However, reducing grace periods can lead to unintended side effects where pods are frequently terminated and restarted due to temporary and recoverable failures (e.g., intermittent connection losses).

5.2.2 Remote worker node failure

In this scenario, a worker node in the remote (i.e, provider) cluster becomes not ready or unreachable.

The remote pod enforcement logic

Offloaded pods are not deployed directly on the remote cluster, but an intermediate custom resource, the ShadowPod (Section 3.5.4), is created on the remote cluster by the virtual kubelet. If pods were simply deployed remotely, split-brain scenarios (e.g., a temporary loss of connectivity between clusters) could occur, causing service disruption if the remote pods were deleted after a node failure or eviction. For this reason, Liqo resorts to the remote creation of a ShadowPod, a CR wrapping the pod definition and triggering the remote enforcement logic. Then, the ShadowPod controller in the remote cluster creates the corresponding twin pod. In this way, it transparently ensures execution resilience regardless of connectivity to the originating cluster. To simplify, local pod operations (e.g., create, update, and delete) are translated to the corresponding operations on remote ShadowPods. In addition, the incoming reflection logic propagates pod status updates from the remote pod to the local pod in the main cluster when appropriate. The overall offloading process is summarized in Figure 5.3.

The problem

Given this architecture, the scenario where a remote worker node hosting offloaded pods becomes unhealthy is a bit problematic. In a vanilla Kubernetes deployment, the pod would be marked for deletion and a newly created pod (with a different name) would be spawned in a different node. However, this cannot be applied here as it would lead to race conditions in the reconciling logic between local and remote clusters. For example, if the remote pod is marked for deletion (i.e., it has the *DeletionTimestamp* label), the ShadowPod controller in the remote cluster would create a new pod (with a different name) and leave the other one in status *Terminating* indefinitely, leading to a situation where the ShadowPod would have two children. The reconciliation in the local cluster is problematic because it would be unclear which of the two child pods (having different names and statuses) needs to be reconciled in the local cluster. It's even more complex to handle in case of

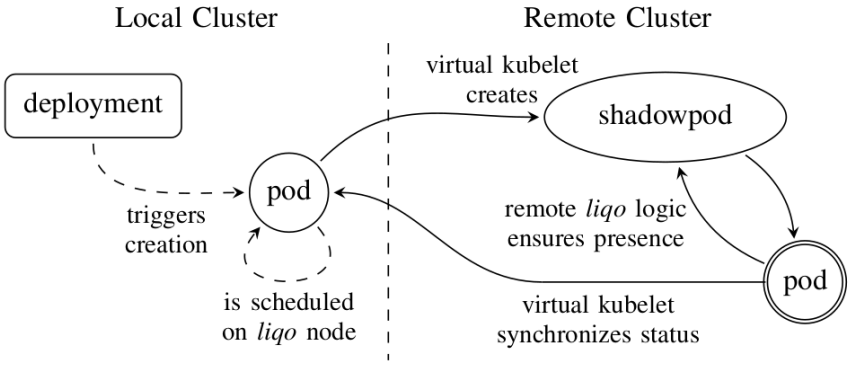

Figure 5.3: Schematic representation of the pod offloading workflow. Solid lines refer to liqo-related tasks, while dashed ones to standard Kubernetes logic. Double circles indicate the pod in execution (i.e., whose containers are running) [24]

a StatefulSet, because the pod names are fixed and cannot be changed. For the above reasons, Liqo does not create a new pod to replace the terminating pod, as is the case with vanilla Kubernetes. As a result, if a node fails in the remote cluster, the actual workload is less than expected and service continuity could be affected if the load is too heavy for the remaining pods in the entire multi-cluster. Figure 5.4 shows a representation of the lifecycle of both offloaded and normal pods in the event of a worker node failure.

An implementation of a possible solution to this problem is part of the work of this thesis. It is presented in detail in Section 5.3.

5.2.3 Local cluster failure

This scenario includes all cases where the local cluster can be considered unhealthy, i.e., the API server is unresponsive and/or unable to handle requests, or there is no connection to the local cluster. There are a variety of reasons for this: misconfigurations, power outages, saturation of resources in control plane nodes, possible network partitions within the cluster or between clusters, etc. However, it is possible that there is still an active network connection between the two peered clusters, e.g. via the liqo-gateway (the VPN tunnel that handles pod-to-pod traffic).

The EndpointSlices reflection logic

This section explains the details of the Liqo EndpointSlices reflection to better understand this scenario and its possible flaws.

When a new service is deployed in a namespace offloaded with Liqo, it is

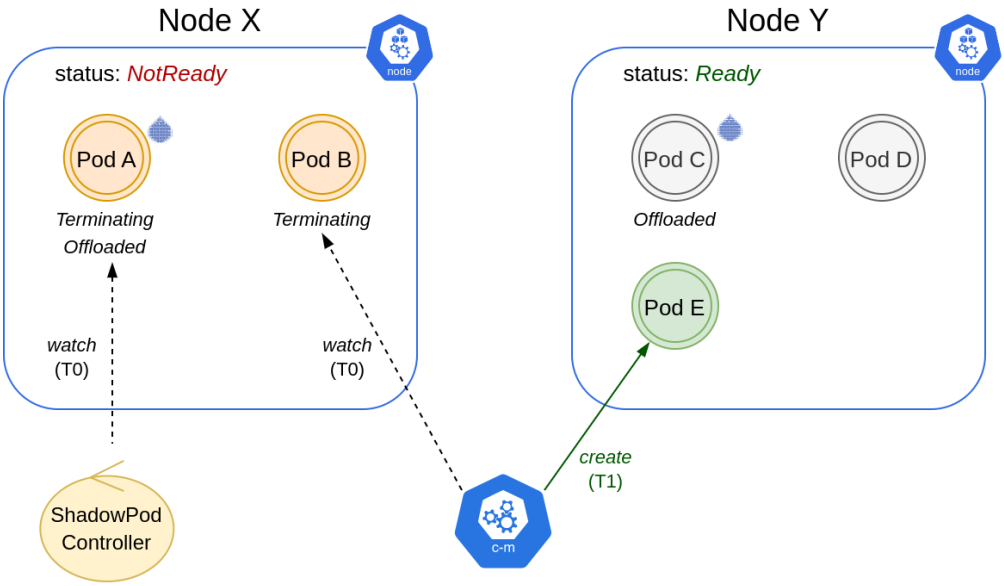

Figure 5.4: Graphical representation of the lifecycle of pods during a worker node failure

replicated verbatim to remote clusters, except for the ClusterIP, LoadBalancerIP, and NodePort fields (if applicable), which are left empty (and therefore assigned by the remote cluster) due to potential conflicts. In the local cluster, services are transparently handled by the vanilla Kubernetes control plane, as it has full visibility of all pods (even those offloaded), hence leading to the creation of the corresponding endpointslice entries. In contrast, the control plane of each remote cluster perceives only the pods running in that cluster. Therefore, the standard endpointslice creation logic alone is not sufficient (as it would not include the pods hosted by other clusters).

This gap is handled by the Liqo EndpointSlice reflection logic, which takes care of propagating all endpointslice entries that are not already present in the destination cluster. During the propagation process, the endpoint addresses are appropriately remapped according to the network fabric configuration, to ensure that the resulting IPs are reachable from the destination cluster.

With this approach, multiple replicas of the same microservice spread across different clusters and backed by the same service are handled transparently. Each pod, no matter where it is located, contributes with its own endpoint entry in the associated endpointslice, either through the standard control plane or through resource reflection, hence becoming eligible during the service load-balancing process.

Figure 5.5 shows an example of the endpointslices replication in a 4-replica

deployment offloaded with policy *LocalAndRemote*, which equally balances the spread of pods across clusters. As described above, the local cluster perceives all 4 pods, although only 2 are actually running (indicated by double circles). The remote cluster, on the other hand, perceives only the (two) running pods. The *kube-controller-manager* present in each cluster has automatically created the native K8s endpointslice(s) for all perceived pods (i.e., endpointslice A and C). The virtual kubelet fills the gap in the remote cluster by creating a new custom endpointslice with the endpoints of the pods hosted by the local cluster (i.e., P_1 and P_2).

To ensure that multiple entities can manage endpointslices without interfering with each other, Kubernetes defines the label *endpointslice.kubernetes.io/managed-by*, which specifies the entity that manages the endpointslice. The endpointslice controller sets `endpointslice-controller.k8s.io` as the value for this label on all endpointslices it manages. Liqo, on the other hand, sets the value to `endpointslice.reflection.liqo.io`.

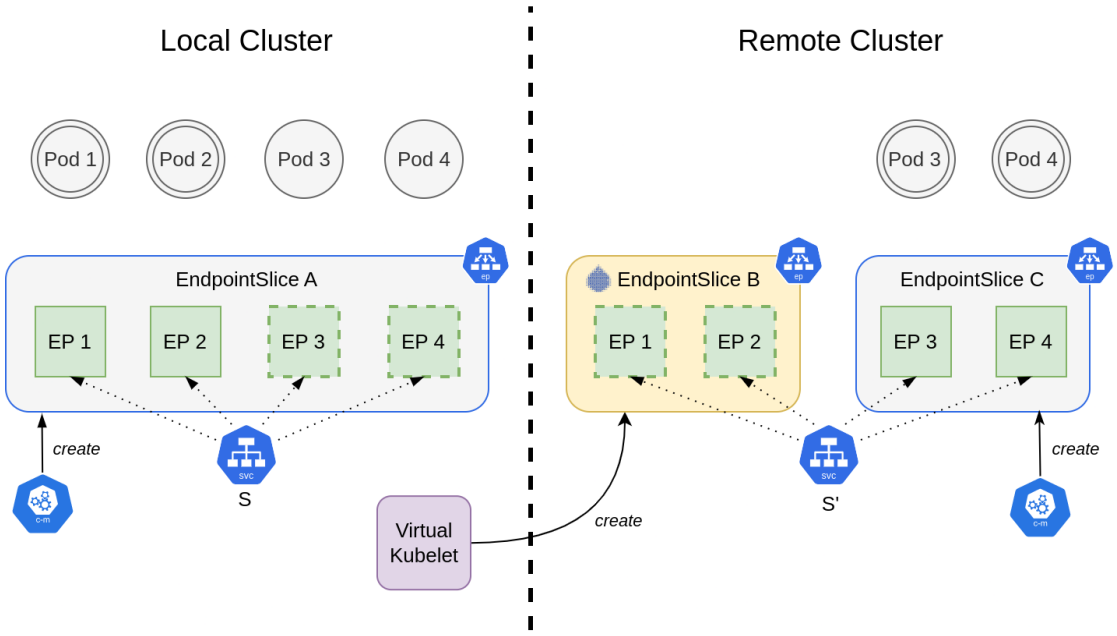

Figure 5.5: Graphical representation of the EndpointSlices reflection. Double circles indicate that the pod is in execution (i.e., its containers are running). Dashed squares indicate endpoints that point to pods in execution in a different cluster.

The problem

The described architecture works flawlessly when there is perfect communication and cooperation between the two clusters. Unfortunately, problems occur when the local cluster is unreachable or unhealthy.

The EndpointSlice abstraction, introduced in Section 2.5.8, allows to keep track not only of all the endpoints network specifications of the pods associated with a given service, but also of their current status (i.e., readiness). Therefore, each endpoint has a condition *ready* that maps to the *Ready* condition of the associated pods. This synchronization between the status of the endpoint and its associated pod allows services to avoid redirecting traffic to unhealthy or terminating pods by avoiding endpoints that are not ready.

This mechanism works well as long as the pod is present in the cluster and its status can be reconciled in the endpointslice. In the scenario described earlier in Figure 5.5, there is no associated pod in the cluster for the endpoint slice created by Liqo. For this reason, the endpoint is always marked as ready since it is not updated by any entity. The side effect is that in the event of a local cluster disruption, services will redirect the traffic in a round robin fashion, including also the endpoints that reference pods in the local cluster. Such requests will inevitably fail and lead to service disruption. The possible causes of disruption are as follows:

- **pod-to-pod connectivity**: if the VPN tunnel is down, all requests are lost.
- **local API server healthiness**: if the local API server is down, requests towards local pods can still reach the target if the VPN tunnel is still working and the pod is running on a healthy worker node. However, the API server of the local cluster being down means that the app cannot evolve and it is not possible to know the current status of the pods, so requests may fail in this case as well.

Figure 5.6 shows a graphical representation of this scenario. Ready endpoints are colored green. The S' service load balance requests to all ready endpoints with a round-robin policy. All requests to endpoints targeting pods in the local cluster are not served. Depending on the pod distribution between clusters, requests fail with a certain probability (50% in this particular example).

Section 5.4 presents a possible solution that ensures service continuity and resiliency in this scenario.

5.2.4 Remote cluster failure

This scenario includes all cases where the remote cluster can be considered unhealthy. Possible causes are the same ones described in Section 5.2.3, but applied to the remote cluster.

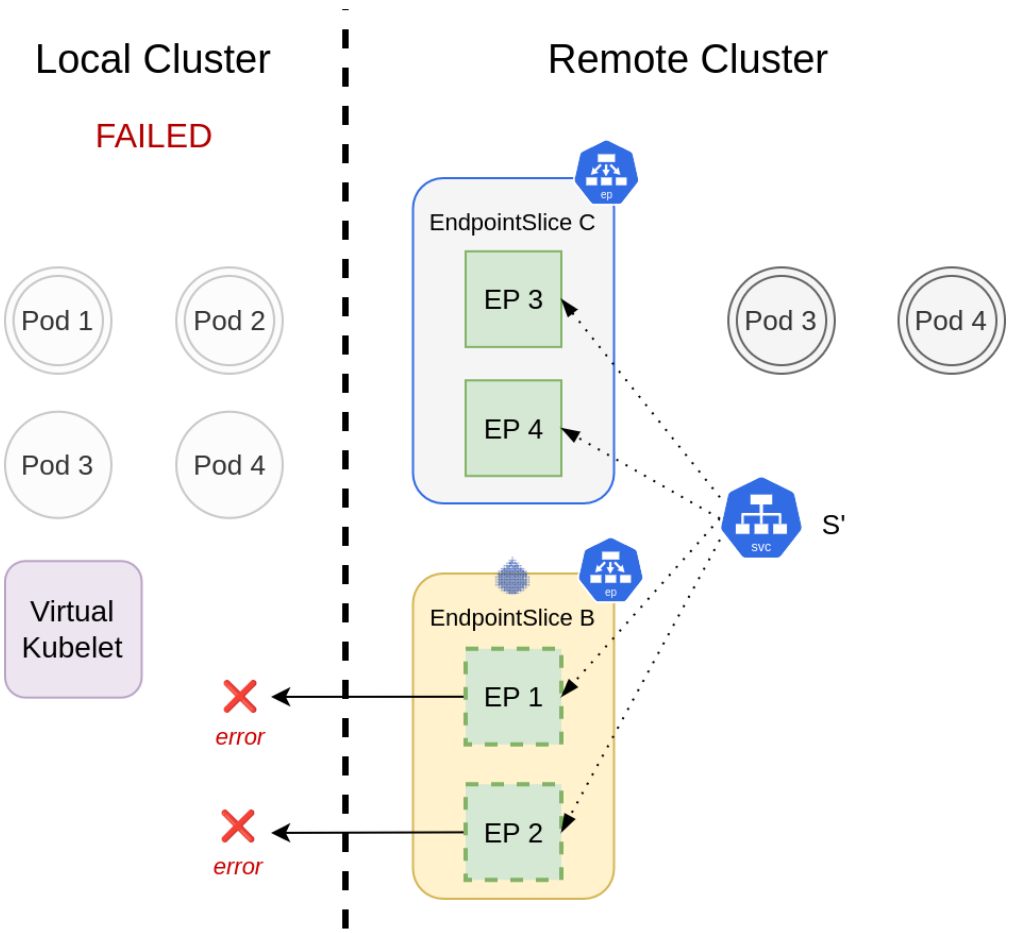

Figure 5.6: Graphical representation of failed requests in the event of a local cluster failure. Double circles indicate that the pod is in execution (i.e., its containers are running). Dashed squares indicate endpoints that point to pods in execution in a different cluster.

The remote cluster is represented in the local cluster by the virtual kubelet, which periodically checks the health status of the remote cluster and updates its status conditions [17] accordingly. The VK performs the following periodic checks:

1. **remote API server readiness**: it updates the current status with the results of the *liveness probe* on the remote API server.
2. **VPN tunnel network availability**: it updates the current status by monitoring the TunnelEndpoint *Connection* condition.
3. **resources availability**: it updates the current status by monitoring the ResourceOffer resource.

If the status is not ready or has not been updated in the last *node-monitor-grace-period* (default: 40 seconds), the virtual node is marked respectively *NotReady* or *Unknown*. Upon this event:

- all endpoints to pods in the remote clusters are invalidated immediately, so that services do not redirect the traffic to those (service continuity)
- the standard **API-initiated eviction** is triggered for all pods in the (virtual) node, which in this case represents the entire remote cluster: after a grace period (`pod-eviction-timeout`), all pods are marked for deletion with a *DeletionTimestamp*. New pods are spawned in the local cluster as replacements (expected workload guaranteed).

Figure 5.7 shows an example of a remote cluster failure, considering the same 4-replica deployment. The virtual kubelet is marked as not ready. Service S does not redirect the traffic to EP_3 and EP_4, because the associated pods P_3 and P_4 are not ready. When pods become *Terminating*, the *kube-controller-manager* reacts by creating the replacement pods P_5 and P_6 and thus the associated endpointslices EP_5 and EP_6.

In summary, the presence of the virtual node allows the remote cluster to be treated transparently as if it were a normal node, hence service continuity and the expected workload are guaranteed by standard Kubernetes. In contrast, for local cluster failures (Section 5.2.3), the virtual node is not present, so service continuity cannot be transparently supported. Unfortunately, the local cluster may not be able to handle all of the offloaded resources, resulting in excessive load and disruption. Therefore, depending on the type of application being deployed, it is necessary to carefully distribute the load between clusters by using the appropriate tolerations and taints, topology selectors, node affinity constraints, and Liqo offloading policies.

5.2.5 Local control plane failure

This section analyzes all scenarios in which a control plane node of the local cluster becomes unhealthy.

In Kubernetes, the control plane can have a single-node or multi-node topology. The latter is referred to as HA (High Availability), and the number of nodes is usually odd to facilitate leader selection in the event of a machine or zone failure.

- **Single-node control plane**: in this setting, the API server cannot function if the single node of the control plane is unhealthy, and therefore the entire cluster can be considered out of service. This is a special case of local cluster failure (5.2.3) and is subject to the same considerations.
- **Multi-node (HA) control plane**: the control plane can tolerate the loss of one or more nodes as long as at least one is still running. With this topology,

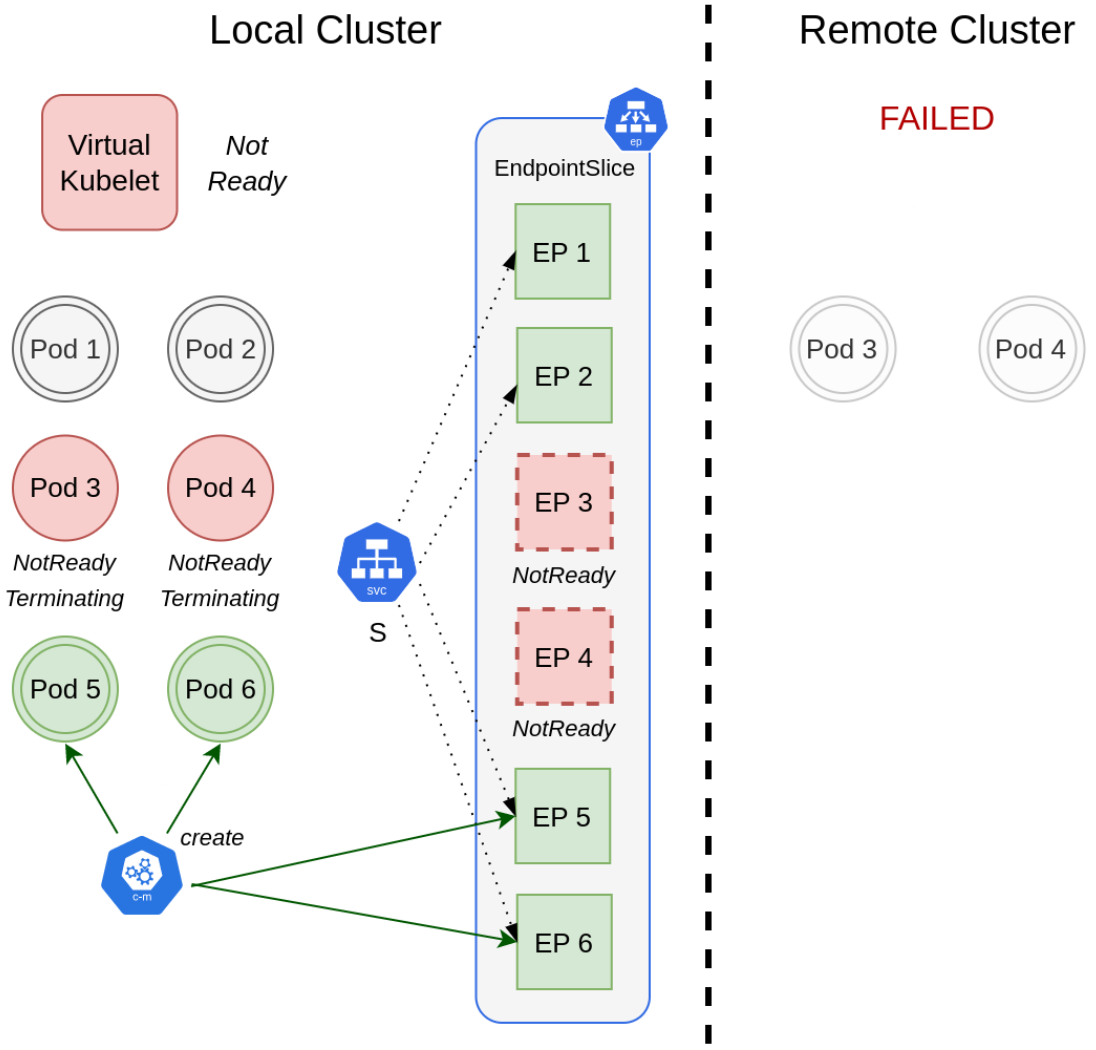

Figure 5.7: Graphical representation of a remote cluster failure. Double circles indicate that the pod is in execution (i.e., its containers are running). Dashed squares indicate endpoints that point to pods in execution in a different cluster.

the infrastructure is more robust and traffic is automatically redirected to healthy nodes. Failures are prevented in most cases, but at the expense of setup complexity and increased resource requirements.

5.2.6 Remote control plane failure

For this scenario, the same considerations as described in the previous section apply, but applied to the remote cluster. Indeed, a failure in a single-node control

plane can be considered as a remote cluster failure (Section 5.2.4), while a HA setup ensures service continuity in most cases.

5.2.7 Inter-cluster network failure

A split-brain scenario can occur when there is no direct connection between two peered clusters, resulting in a network partition. This is one of the most common failures, especially if the clusters are geographically distant from each other or communicate via a public WAN.

In this case, both clusters are still alive (all nodes are healthy and both API servers are responsive), but each believes it is the only one running. Due to the nature of split-brain problems, it is necessary to analyze the situation independently for both clusters:

- **Local cluster**: the remote cluster is unreachable. When the virtual kubelet checks the status of the remote cluster, it also accounts for possible network problems. If there are any, it marks the virtual node as *NotReady* or *Unknown*. This scenario is a special case of remote cluster failure (Section 5.2.4). As a result, service continuity is handled transparently by Kubernetes.
- **Remote cluster**: the local cluster is unreachable. All requests to services with endpoints that redirect traffic to the local cluster fail. This scenario is a special case of local cluster failure (Section 5.2.3). A possible solution to this problem is explained in Section 5.4.

In Liqo, one or more endpoints may be exposed depending on the type of peering. In a In-Band peering, all traffic flows in a single VPN tunnel: only one endpoint is exposed, as shown in Figure 5.8. In a Out-Of-Band peering, three different endpoints are exposed: the Liqo authentication service, the Liqo VPN tunnel, and the Kubernetes API server. A schematic representation of the setup is presented in Figure 5.9. Therefore, all scenarios where only a subset of the three endpoints are down are also included in this section. For example, both clusters can communicate with the foreign API server, but there is no pod-to-pod connectivity due to one of the VPN endpoints being down. The solution proposed in Section 5.4 checks both API server readiness and VPN tunnel connectivity, handling those particular cases as well.

5.3 NodeFailure controller: resiliency to remote worker nodes failures

Part of the work of this thesis was invested in implementing a solution to the remote worker node failure problem presented in Section 5.2.4.

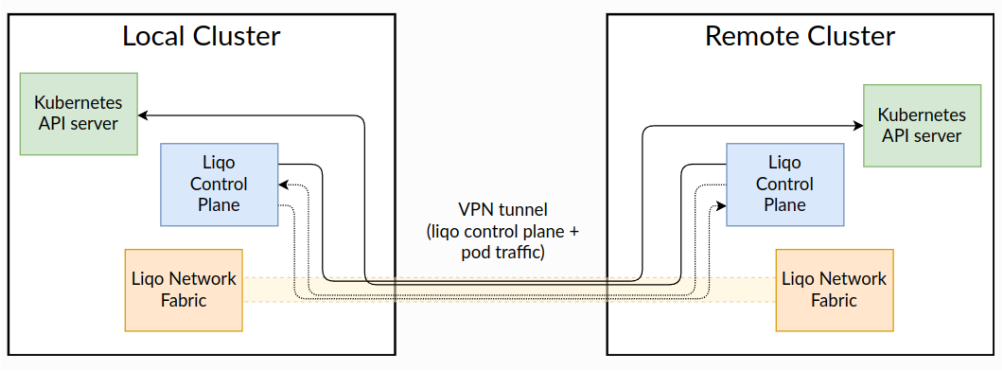

Figure 5.8: In-Band peering

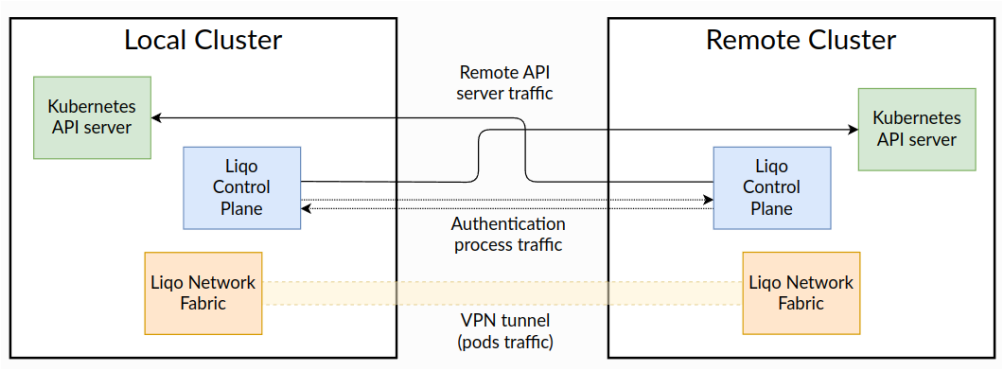

Figure 5.9: Out-Of-Band peering

As described earlier, an offloaded pod running on a *NotReady* node remains in a *Terminating* state indefinitely and is not automatically replaced by either Kubernetes or Liqo. The main problem is figuring out who should handle the offloaded pod and how. There are several possible solutions, each with their own advantages and disadvantages.

1. **Delegation to the remote cluster**: the local cluster creates the remote ShadowPod and delegates all responsibility for creating and running the actual pod to the remote cluster. The ShadowPod controller is the component that enforces the presence of the actual remote pod. Only the status is synchronized from the remote to the local pod. This is essentially the process described in Figure 5.3. When a remote node failure happens, a component in the cluster must react and force delete all pods that are (1) offloaded, (2) terminating, (3) scheduled on a failed node. In this way, the ShadowPod controller will react to this event and enforces the presence of its remote pod by creating a new one. This approach is easy to manage, very effective, and responds

quickly to failures. However, it is less compliant with Kubernetes standards, as terminating pods are force deleted.

2. **Delegation to the local cluster**: the local cluster must manage the entire lifecycle of the offloaded pod. Local and remote pods are synchronized directly and no ShadowPod resource is created. The local cluster detects if a pod is (1) offloaded, (2) terminating, (3) scheduled on a failed node. If all conditions are met, the local pod is also marked as terminating. Then, the Kubernetes ReplicaSet controller of the local cluster will leave the pod in a terminating state and enforces the presence of a new pod, which will be scheduled either on the local or on the remote cluster. This approach is the most compliant with Kubernetes, but it is less resilient to network failures because all enforcement logic is performed by the local cluster since the ShadowPod is not present.

3. **Hybrid Approach**: in this solution, the responsibility for managing the offloaded pod is delegated to the remote cluster via the ShadowPod, but the deletion of the pod in the event of a node failure is performed by the local cluster. In this way, resilience to network issues and compliance with Kubernetes standards are maintained. It is more complex to handle and the responsibility is distributed between the two clusters.

From a pure service availability perspective, the first approach is the most efficient, since the terminating pod is fully managed by the remote cluster and less synchronization is required between the two clusters. This results in a clear decoupling of responsibilities and a less complex setup. For these reasons, this solution was implemented and merged in the official Liqo GitHub repository [16]. The next section analyzes in details the implementation design.

5.3.1 NodeFailure controller implementation

The adopted solution is a custom controller running in the *liqo-controller-manager* pod and named *NodeFailure* controller. The main idea is to detect when a pod is (1) offloaded, (2) terminating, (3) scheduled on a failed node. When a pod satisfies all three conditions, it must be force deleted. Let's break down all the conditions:

- *offloaded*: the controller has to watch only for pods that are directly managed by Liqo (i.e., offloaded). The controller must not touch normal pods as their lifecycle is handled correctly by vanilla Kubernetes even in case of node failure, as described in Section 5.2.1.

- *terminating*: a pod needs to be evicted only if it has a *DeletionTimestamp*. This is only added when the grace periods (`node-monitor-grace-period` and `pod-eviction-timeout`) have expired. This is to prevent deleting pods

running on a temporarily failed node (e.g., intermittent loss of connection), as they would likely be false positives.

- *scheduled on a failed node*: the controller handles only node failures. Offloaded and terminating pods scheduled on a healthy node are deleted by the remote cluster API server.

If a pod that meets all three conditions is detected and deleted by the *NodeFailure* controller, its job is complete. It will be the ShadowPod controller's job to enforce the presence of the remote pod by creating a new one, which will be scheduled on a healthy node. A schematic representation of the workflow with the *NodeFailure* controller activated is shown in Figure 5.10. Note: the name of the new pod created by the ShadowPod controller is the same as that of the deleted pod (i.e. A), since it is extracted from the associated shadowpod. In contrast, the other (not offloaded) pod created by the *kube-controller-manager* has a newly generated name (i.e. E).

Throughout the whole process, the local pod remains in the *Running* state because it is synchronized with the ShadowPod. However, a feedback of the event is provided in the local cluster by inferring additional restarts for each newly created remote pod. Also, since the status is propagated from the remote to the local cluster, it is possible to observe during the failure that the pod's *Ready* condition is set to `False` (until the pod is evicted by the *NodeFailure* controller).

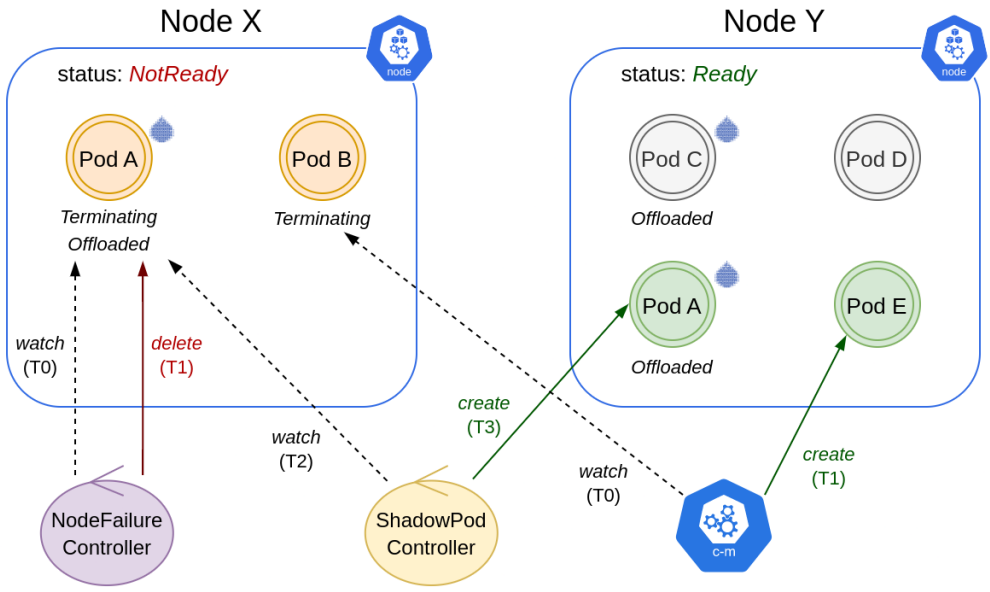

Figure 5.10: Graphical representation of the lifecycle of pods during a worker node failure, with the NodeFailure controller enabled

The *NodeFailure* controller implementation has been merged in the official Liqo

GitHub repository and can be enabled by passing to the *liqo-controller-manager* the `enable-nodefailure-controller` argument. The source code of the Pull Request can be found at `https://github.com/liqotech/liqo/pull/1633`

5.3.2 The algorithm

The controller reconciles objects of type *Node*, so it is configured to respond to node create/delete/update events. It also watches for events on pod resources. Since events on pods are very common, an event handler has been set up to filter only meaningful events. In particular, it only reacts to update events and checks if the pod is offloaded and terminating. If so, it adds the name of the node hosting the pod to the controller *workqueue* so that it can be reconciled. The event handler's check is fast and efficient because all the required information is available in the resource:

- a *DeletionTimestamp* as a Metadata field indicates that the pod is terminating
- the label "`liqo.io/managed-by: shadowpod`" (added by Liqo via the ShadowPod controller) indicates that the pod is offloaded
- the name of the node hosting the pod is set in the `spec.nodeName` field

The reconciliation logic is straightforward. The controller fetches the node resource with a `GET` request to the API server. If the node is *NotReady*, it lists all pods that are offloaded and scheduled on that node using the appropriate *LabelSelector* and *FieldSelector*. If there are any, all terminating pods (i.e., the ones with a *DeletionTimestamp*) will be force deleted. The `spec.nodeName` field is not cached by default. Its field is added to the cache indexer to improve performance and reliability.

5.3.3 Drawbacks

As mentioned earlier, this solution is a trade-off that prioritizes service availability over compliance. But it also comes with some drawbacks. When the controller force deletes the pod, the resource is removed from the K8s API server. This means that in the (rare) case that the failed node becomes ready again and without an OS restart, the containers in the pod will not be deleted by the API server because it has already deleted the resource in the past and removed its entry from the database. The side effect is that zombie processes remain in the node until the next OS restart or manual cleanup. This is the reason why vanilla Kubernetes leaves the pod pending in *Terminating* state, so that if the node becomes ready again, the API server will gracefully delete the pod and release its resources.

5.4 Shadow EndpointSlices: resiliency to local cluster failures

This section describes the design and implementation of a possible solution to the local cluster failure problem presented in Section 5.2.3.

As previously described, the EndpointSlice reflection logic of Liqo is not resilient to the following conditions:

1. **VPN tunnel down (no pod-to-pod connectivity)**: requests to pods running in the local cluster will not be served

2. **local API server broken or unreachable**: the status of the app can't evolve or even be tracked. Endpoints could point to pods in the local cluster who could have been deleted or modified, causing requests to fail again.

The basic idea is to delegate to some entity the task of dynamically updating the status of all endpoints towards the local cluster, depending on the conditions described above. Kubernetes itself does this job by mapping an endpoint status to the pod status, but this is not possible in this case because there is no associated pod in the remote cluster, as shown in Figure 5.5. As a result, Liqo itself must be modified to handle this scenario.

A naive solution might be to implement a custom controller in the remote cluster that periodically checks the status of the VPN tunnel and the local API server and updates the endpoints accordingly. This solution can't work because it would lead to race conditions between the two clusters. If we consider a scenario where only the tunnel is down (i.e., both API servers are live and can communicate), the controller sets the endpoints to *NotReady*. This triggers the local API server (actually the virtual kubelet) to reinforce the status to *Ready*, and the cycle repeats infinitely since there is no single source of truth between the two clusters. Figure 5.11 shows a diagram of the race condition.

The solution

The proposed solution is to decouple the link between the local and the remote endpointslices by introducing an intermediate resource, the ShadowEndpointSlice CR. The idea is similar to the one adopted for the pods (i.e., ShadowPod). The shadow object in this case is an abstraction that serves as a template for the desired configuration of the remote endpointslice. The virtual kubelet task is to forge the remote shadow resource of the reflected endpointslice and create it on the remote cluster. If the local endpointslice changes (e.g., when some endpoints are added), the remote shadow endpointslice is also updated. The responsibility of the local cluster is limited to creating and updating the required ShadowEndpointSlices.

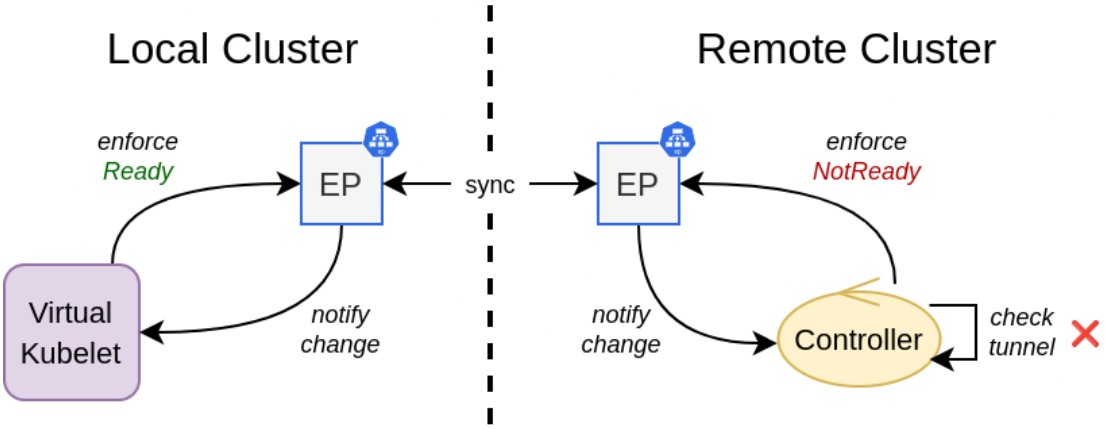

Figure 5.11: EndpointSlice enforcement race condition between two clusters

Note that the new ShadowEndpointSlice resource contains only the endpoints of the pods running in the local cluster, just like the endpointslices managed-by Liqo described earlier.

At the same time, a custom controller in Liqo must run in the remote cluster and enforce the presence of the actual endpointslice, using the shadow resource as a **source of truth**. The controller must also periodically check the status of the local cluster, taking into account both the API server and the VPN tunnel.If the local cluster is unhealthy, the controller updates all endpoints in the endpointslice by setting them to *NotReady*. This way, services do not redirect traffic to these endpoints, but only to the remaining ones served by the pods in the remote cluster. This ensures service continuity (assuming there are enough pods scheduled on the remote cluster to handle the load). When the local cluster eventually becomes available again, the controller resets the endpoints to their default settings according to the shadow template. Note: if the default status for an endpoint (specified in the shadow resource) is *NotReady*, the remote endpoint is also set to *NotReady*, regardless of the condition of the local cluster.

The following sections explain in more detail the design and implementation of the new shadow CR and custom controller, as well as the potential limitations of this solution in some application use cases.

5.4.1 The ShadowEndpointSlice CR

The ShadowEndpointSlice CRD has been defined in the `virtualkubelet.liqo.io` API. It has the following fields:

- `TypeMeta`: specifies the name of the resource and the custom-defined API.

- `ObjectMeta`: must have the following set of labels:
  - the name of the associated service
  - the `endpointslice.kubernetes.io/managed-by` label key has `endpointslice.reflection.liqo.io` as value, to prevent K8s from interfering
  - id of the origin cluster (i.e., local/consumer)
  - id of the destination cluster (i.e., remote/provider)
- `Spec.Template`: same fields present on native endpointslices that describe endpoints network parameters:
  - `AddressType`, IPv4 or IPv6
  - `Endpoints[]`, array of endpoints, each containing their IPs, status, a reference to the associated pod and its node)
  - `Ports[]`, array of ports, each containing port number and protocol

Listing 5.1 shows an example of a ShadowEndpointSlice resource:

Listing 5.1: Basic example of a ShadowEndpointSlice

```
apiVersion: virtualkubelet.liqo.io/v1alpha1
kind: ShadowEndpointSlice
metadata:
  creationTimestamp: "2023-02-22T17:49:26Z"
  generation: 1
  labels:
    endpointslice.kubernetes.io/managed-by: endpointslice.reflection.
    liqo.io
    kubernetes.io/service-name: hello-world
    virtualkubelet.liqo.io/destination: <DEST_ID>
    virtualkubelet.liqo.io/origin: <ORIG_ID>
  name: hello-world-dqbm7
  namespace: hello-world
  uid: a067f678-8851-4fc3-b686-2ca7fb36c61d
spec:
  template:
    addressType: IPv4
    endpoints:
    - addresses:
      - 10.40.2.7
      conditions:
        ready: true
      nodeName: node-1
      targetRef:
        kind: RemotePod
```

```
25          name: hello-world-744db8bd9-kdhjd
26          namespace: hello-world
27          uid: 03285c05-e2b5-4b1b-81bb-e31b1d9bb8cd
28      ports:
29      - name: http
30        port: 80
31        protocol: TCP
```

5.4.2 ShadowEndpointSlice controller implementation

The controller runs in the *liqo-controller-manager* pod. For simplicity, it is called the "ShadowEps" controller here. It reconciles objects of type ShadowEndpointSlice, so it responds to all events on these resources. For example, when the virtual kubelet creates or updates a ShadowEndpointSlice, it creates or updates the associated endpointslice and sets the shadow object as its *OwnerReference*. This way, when the shadow object is deleted, the owned endpointslice is also automatically evicted.

To check the current status of the local cluster, the ShadowEps controller watches for ForeignCluster (abbreviated FC) objects (see Section 3.5.3). This Liqo CR is used to gather information about a peered cluster (doesn't matter if outgoing, incoming, or bidirectional peering) and contains in its status the updated conditions that track the health of the local cluster. Specifically, the controller looks for the following conditions:

1. *TunnelEndpoint*: status of the VPN tunnel for pod-to-pod connectivity.
2. *APIServer*: readiness of the local API server

Unfortunately, the API server condition was not originally present in the FC CRD. The API server check was originally performed by the virtual kubelet, but is not present in the remote cluster. For this reason, this check is now periodically performed by a Go routine that is started by the ForeignCluster controller when a new peering is established. The routine itself takes care of updating the status conditions accordingly.

In the reconcile logic, the controller enforces the presence of the endpointslice by using the shadow resource as a template. The only fields that are changed are *Ready* conditions of the endpoints, which are updated with the current status of the local cluster (again, by looking at the *TunnelEndpoint* and *APIServer* conditions in the ForeignCluster resource). As mentioned earlier, an endpoint whose *Ready* condition is set to `False` is kept unchanged regardless of the status of the local cluster. To optimize and reduce the number of reconciliations, an event handler filters the events on the watched ForeignCluster objects: only update events where the *TunnelEndpoint* and/or *APIServer* conditions change are added to the controller's *workqueue*.

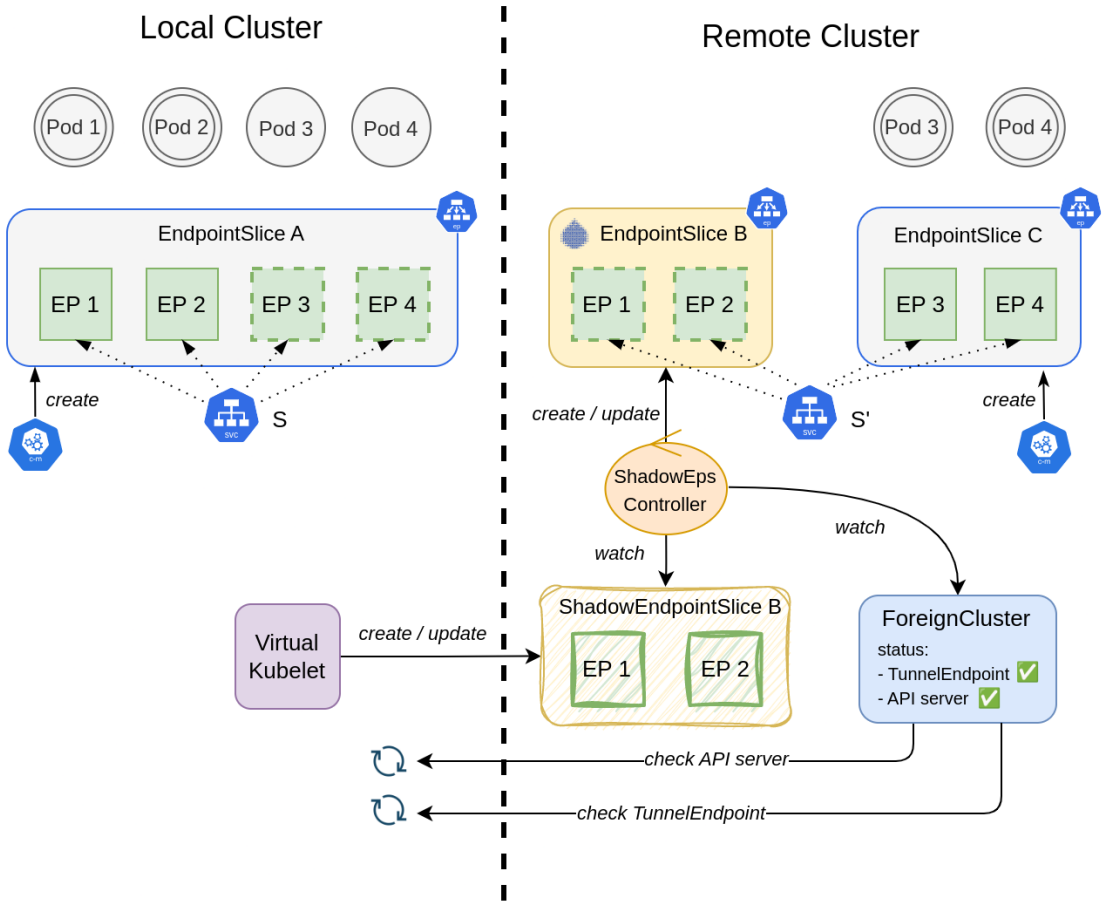

Figure 5.12: Graphical representation of the EndpointSlices reflection with the added ShadowEndpointSlice controller. Dashed squares indicate endpoints pointing to pods in execution in a different cluster

.

Figure 5.12 shows the workflow of the controller in the typical 4-replica deployment offloaded with a *LocalAndRemote* policy. The virtual kubelet creates/updates the ShadowEndpointSlice. The ForeignCluster status conditions are updated periodically. In the diagram, both checks are successful, so the ShadowEps controller creates the endpointslice with the two endpoints EP_1 and EP_2 ready. Service S' load balances the traffic towards all four endpoints.

Figure 5.13 shows how the ShadowEps controller reacts in the event of a local cluster failure. In the diagram, both the API server and TunnelEndpoint checks have failed. As soon as the ForeignCluster updates its status conditions, the ShadowEps controller reconciliation is triggered. It proceeds to invalidate all endpoints of the endpointslices associated with the given ForeignCluster.

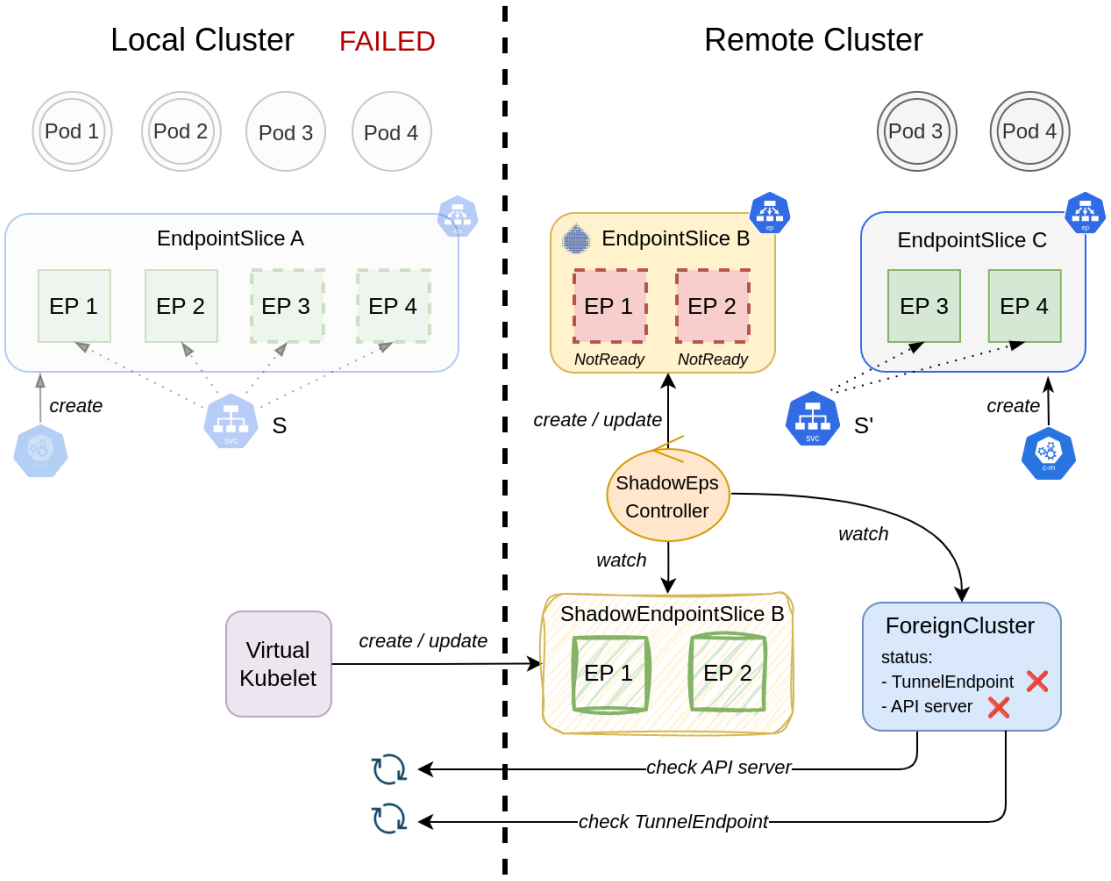

Figure 5.13: Graphical representation of the EndpointSlices reflection with the added ShadowEndpointSlice controller, during a local cluster fail. Dashed squares indicate endpoints pointing to pods in execution in a different cluster. Red endpoints are *NotReady*

.

5.4.3 Limitations

The ShadowEndpointSlice implementation provides solid resilience against cluster failures, but is not suitable for all use cases. The ShadowEndpointSlice controller allows traffic not to be redirected to pods in the failed cluster, thus avoiding requests from inevitably failing.However, this means that the cluster is temporarly disconnected from other peered clusters and must handle the load by itself, and all application logic (i.e., all the necessary pods) must be replicated in the cluster to keep the app working. This is suitable for *elastic cluster* deployments (Section 5.1.1), where the entire application is replicated among peered clusters so that it can run independently even during temporary failures. But for use cases where the

application logic is distributed across the peered clusters, the remaining healthy clusters may not have the necessary logic (i.e., pods) to keep the application running correctly. An example would be a deployment where one large cluster (the one who does the offloading) has all the logic and computational power to store and process huge amounts of data coming from multiple small edge clusters (acting as sensors, for example). In this case, the edge clusters cannot function properly because they require services that are only available in the local source cluster. This problem cannot be fixed at the infrastructure level, nor by Liqo or a service mesh. The solution is to implement a resilience mechanism in the application logic to deal with temporary failures.

5.4.4 Performance evaluation

In this section, we evaluate the performance of the Shadow EndpointSlices implementation. Compared to vanilla Liqo, it introduces some overhead in the exposition reflection: the remote endpointslices are not directly created by the virtual kubelet, only the shadow resources. The actual endpointslices are created by successively reconciliations of the ShadowEndpointSlice controller. The goal is to analyze how much overhead this intermediate step entails and whether it can affect performance in real-world use cases.

To perform the benchmark, we leverage an existing tool [25] already used by the Liqo team to compare the performance of Liqo with vanilla Kubernetes and other similar tools for multi-cluster deployments. For our scope, it is used to compare vanilla Liqo to a custom version built with the Shadow EndpointSlices implementation enabled. This customised version was built on top of Liqo v0.7.2.

The service exposition test The setup for the benchmark consists of two clusters peered with Liqo, one playing the role of the resource provider and the other of the resource consumer. A varying number of pods is started locally (i.e., on the provider) and once they are ready, they are exposed through a single Kubernetes service, making them accessible from the consumer thanks to the Liqo reflection logic. The tool measures the time elapsed between the creation of a service on the local cluster and the effective creation of all associated endpointslices on the remote cluster. The experiment is run twice (with and without the Shadow EndpointSlice implementation). Figure 5.14 shows a schematic representation of the setup.

The testbed The testbed consists of two Kubernetes clusters (k3s v1.21), one provider and one consumer. The resource provider is implemented as a **Kubemark** cluster [26] for scalability reasons. At a very high level, it consists of two parts: a real master and a set of *hollow* nodes. Each of them is is backed by a component, the *HollowKubelet*, which pretends to be an ordinary kubelet, but it does not start

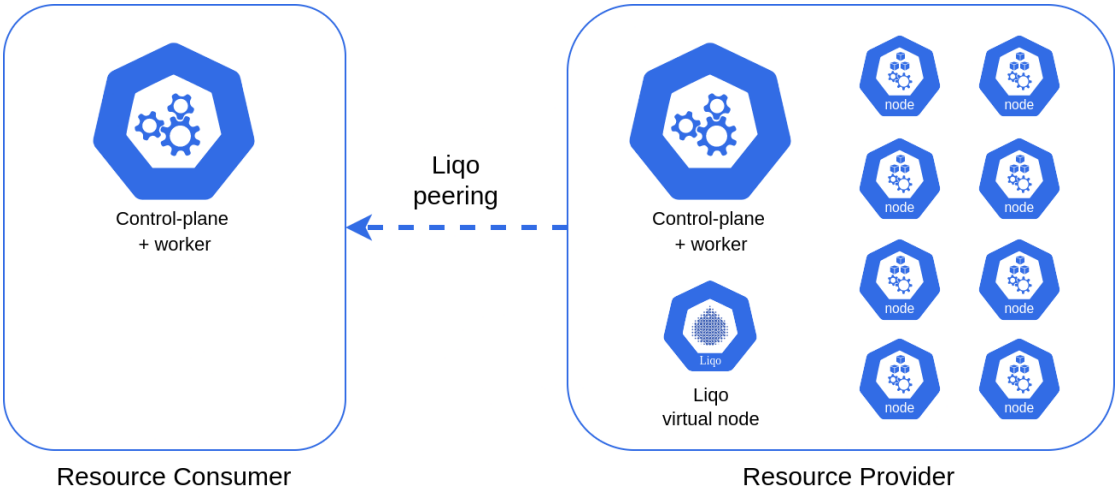

Figure 5.14: Exposition benchmark setup

any container it is assigned to, it just lies it does. In this way, a huge number of (fake) containers can be started, even tens of thousands, consuming few resources in the cluster because the containers are not actually running.

The provider cluster consists of a large number of *hollow* nodes (≈ 50) to host a large number of (fake) pods. Overall, the entire testbed is deployed on a real cluster with 7 worker nodes and the given configuration:

- 1 worker node to host the master node of the consumer cluster
- 1 worker node to host the master node of the provider cluster
- 5 worker nodes to host ≈ 50 *hollow* nodes of the provider cluster

Each node has 8 GB of memory (RAM), 4 vCPU, and 50 GB of hard disk space. In total, the cluster has 56 GB of memory, 28 vCPU, and 350 GB of disk space. The control plane is not configured for high availability.

The benchmark The benchmark runs different deployments, varying the number of replicas, so that we can evaluate the scalability of the solution. Considering that a node cannot host more than 110 pods [27], we can spawn more than 5000 pods using about 50 hollow nodes. The benchmark runs various tests, with the number of pods ranging from 10 to 5000. To make it more robust, each test is repeated 10 times and the results are averaged. The bar plot in Figure 5.15 shows the results of each test. The x-axis is the average time elapsed to expose all endpoints on the remote cluster. The error bars represent the resulting standard deviation.

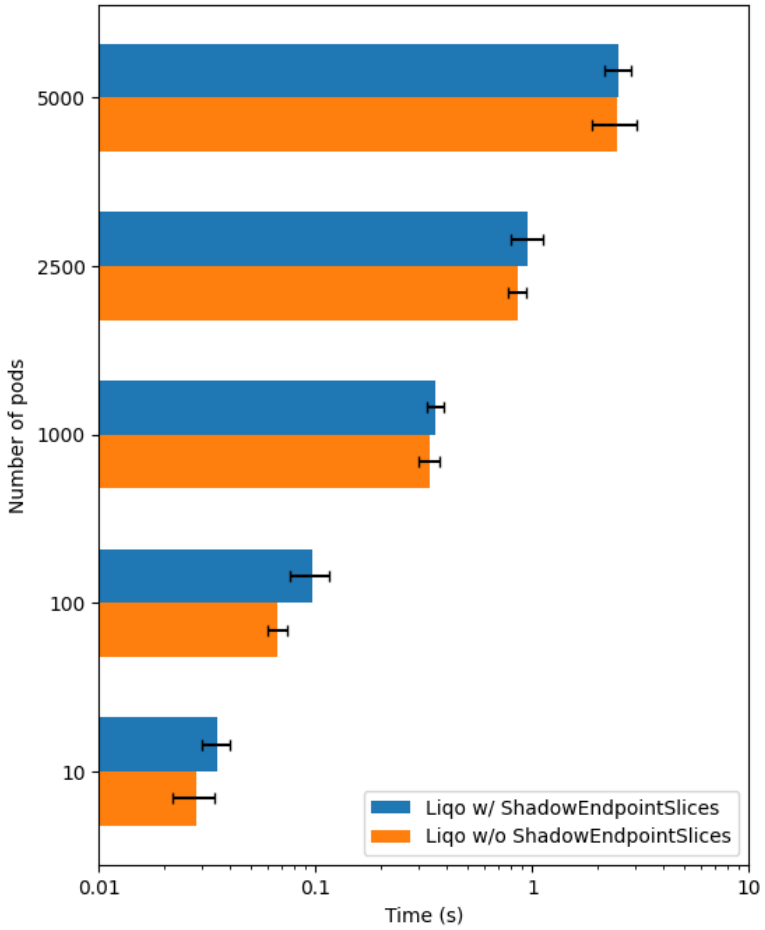

Figure 5.15: Liqo exposition benchmark: performance comparison with and without ShadowEndpointSlices

As can be seen, there is little to no overhead between the two experiments, even when deploying massive amount of pods. Increasing the number of pods has no effect on performance. The standard deviation also remains fairly constant for all experiments. With 5000 pods, a rather extreme and rare use case, all endpoints are ready after ≈ 2.5 seconds, and the overhead caused by the Shadow EndpointSlice logic is only ≈ 50 ms. In summary, the benchmark demonstrates the effectiveness of the implemented solution even under the most challenging conditions.

The ShadowEndpointSlice abstraction and its controller have been merged in the official Liqo GitHub repository and are enabled by default. The source code can be found at the following GitHub issue: `https://github.com/liqotech/liqo/issues/1705`.

Chapter 6

Disaster Recovery with Liqo

Chapter 4 introduced the idea of using a multi-cluster environment to perform disaster recovery of application-level data. The idea is to leverage a Kubernetes multi-cluster environment, with one cluster as the main site and the rest as failover sites. As an example of this concept, this chapter presents the Percona Operator for MongoDB [22]. Unfortunately, the implementation of this solution is tied to the specific MongoDB [23] database and cannot be generalized to all DBs, but the main architectural concepts are applicable to all modern DBs that support data redundancy (e.g., replica sets in MongoDB).

6.1 Disaster Recovery with Percona Operator

Percona Operator for MongoDB supports multi-cluster or cross-site replication deployments. This feature is extremely useful if you want to perform a disaster recovery deployment or migration to or from a MongoDB cluster running in Kubernetes. In a nutshell, it allows you to deploy Percona operators across different Kubernetes clusters to manage and expand members of a **replica set**. In this section, we present a POC of the disaster recovery strategy using the Percona operator [28].

MongoDB Replica Sets A replica set in MongoDB is a group of *mongod* processes that provide redundancy and high availability [29]. Members of a replica set can be:

- **primary**: receives all write operations. There can be only one primary. The primary records all changes to its data sets in its operation log (i.e. oplog).
- **secondary**: replicates operations from the primary to maintain an identical data set. Secondaries replicate the primary's oplog and apply the operations to

their data sets asynchronously. By having the secondaries' data sets reflect the primary's data set, the replica set can continue to function despite the failure of one or more members. Clients can only read from secondary members.

A member of a replica set is often referred to as a node (not to be confused with Kubernetes nodes).

The setup For this POC, we consider a 2-cluster setup where one cluster is the main site and the other is the failover site. However, it can also be extended to an N-cluster configuration where only one cluster is the main site. Percona Operator for MongoDB must be deployed on both clusters. The failover site runs a MongoDB cluster in *unmanaged mode*. In this mode, nodes are not assigned to any replica set and the operator does not control TLS certificates for authentication and encryption. Figure 6.1 shows an overview of the setup.

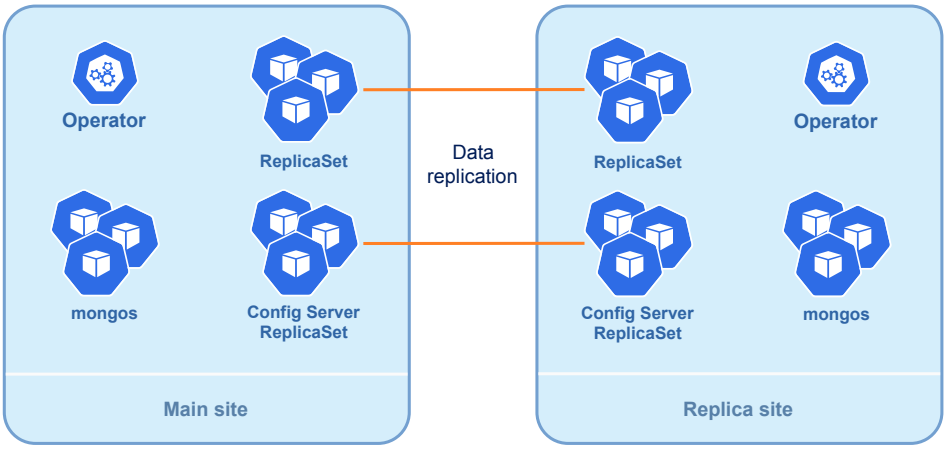

Figure 6.1: Cross-site Replication with Percona Operator [28]

Exposing Replica Set members between clusters A necessary requirement is that each replica set member (node) is exposed via a dedicated K8s service. This is required to ensure that replica set nodes (including *Config Servers*) on Main and DR can reach each other, as in a full mesh (Figure 6.2).

ClusterIP services and Kubernetes DNS addressing cannot be used in this context because members should be reachable from members outside the cluster. Using an external LoadBalancer provider is the ideal solution, although it requires allocating additional IPs for each member, which can be expensive, especially without a private network.

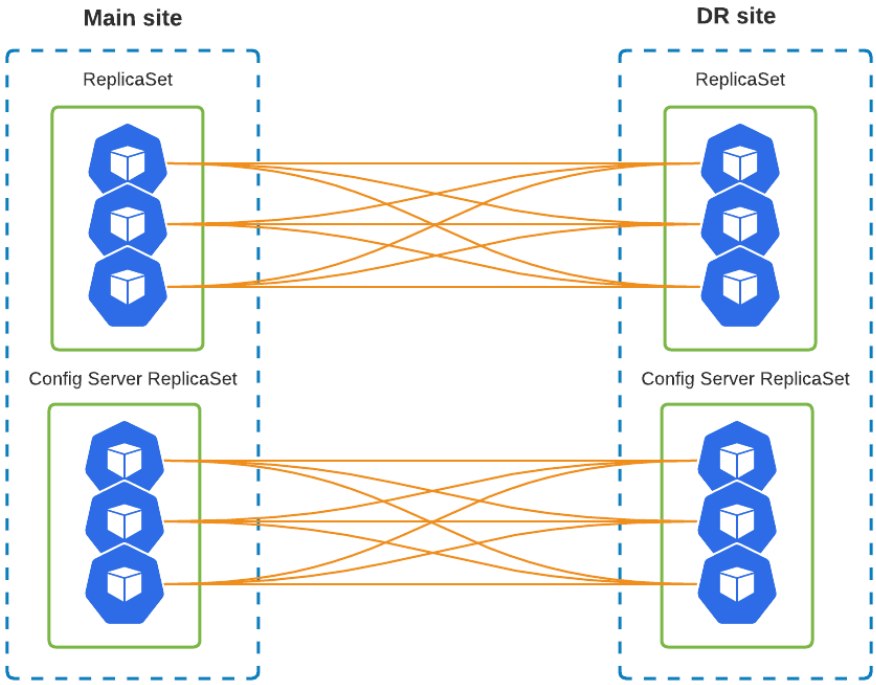

Figure 6.2: Cross-site Replication Mesh [28]

Sharing secrets between clusters The next step is to copy all the secrets required to manage the MongoDB deployment to the replica site cluster. This includes the secrets containing the authentication credentials of DB users, the TLS certificates used to encrypt communications between members, and the data encryption key. Copying the secrets manually can be a tedious process from a security perspective. Also, pre-sharing of the secrets must be properly managed and requires additional infrastructure to securely distribute the keys out-of-band (OOB). This problem is addressed with Liqo in Section 6.2.

Configuration of quorum votes for primary election In a MongoDB replica set, the primary member is the node that receives all write operations and forwards them to secondary nodes for replication. The election of a primary member is automatic and handled by the replica set's internal election protocol. When a replica set is initialized for the first time, or when a primary member steps down or cannot be reached by the other members, a new primary member must be elected. Nodes that are eligible for election compare their priorities to determine

which node has the highest priority. By default, all nodes have a priority of 1, but administrators can configure each node's priority to reflect the relative importance of that node in the replica set. Nodes with higher priorities have a greater chance of being elected as the primary member. Each node in the replica set can be assigned a certain number of votes, which are used to calculate the total number of votes in the replica set. During an election, a node must receive a majority of the total votes in order to be elected as the primary member.

An important step is to correctly configure the priority and the number of votes of each member. Let's consider a scenario with 3 members for each cluster (6 in total). In the failover site there are 3 nodes, but only 2 are voters. This is done to avoid **split-brain** [30] situations and not to start the primary election when the DR site is down or there is a network disruption between the Main and DR sites. Therefore, there will be three voters in the main cluster and two voters in the replica cluster, as shown in Figure 6.3. This means that if the main cluster fails, the replica nodes will not have a majority and will not be able to elect a new primary. This allows to step in and perform a **manual failover** as part of the disaster recovery plan.

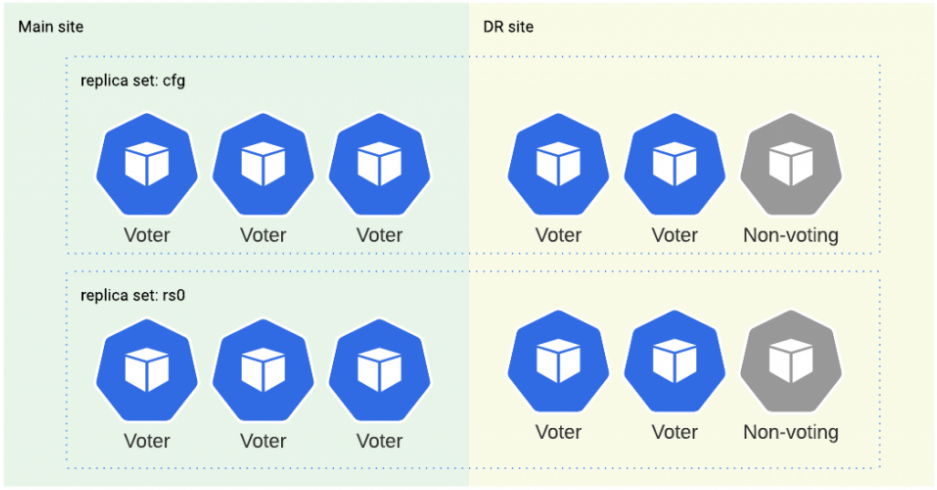

Figure 6.3: Configuration of quorum votes for each replica set member [28]

Manual Failover In case of disruption of the main site, the replica site is the next candidate to become the new main site. To account for split-brain scenarios, members of the replica site do not have the quorum to elect a new primary. To configure the new main site, manual intervention is required. Normally, you can change the replica set configuration only from the primary node, but in such a situation where you have no primary node and only a few surviving members,

MongoDB allows to force the reconfiguration from any alive member. After the reconfiguration, the replica set will consist of only three members, two of which will have votes and a majority. Thus, they will be able to elect a new primary. Once the replica cluster becomes the main cluster, all clients connected to the old main cluster should be reconfigured to point to the new main site.

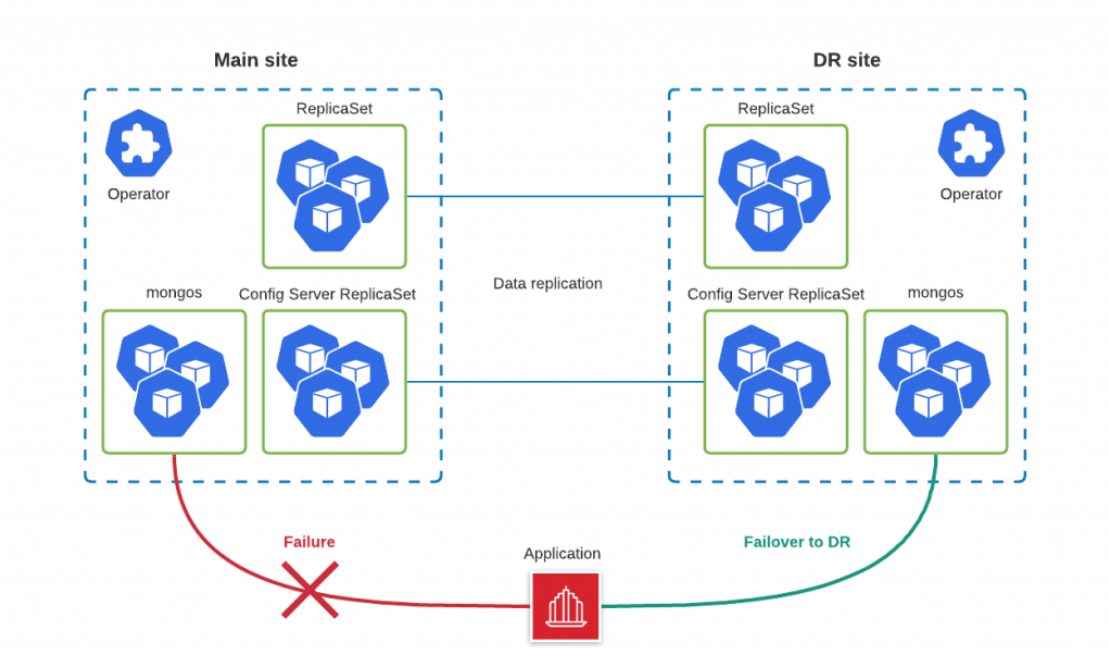

Figure 6.4: Failover to DR site [28]

6.2 Using Liqo to automate and simplify the creation of failover sites

The cross-site replication can be a valid DR solution, but is complex to manage and set up. The clusters are separate entities that need to share information (e.g., secrets) and communicate seamlessly. Vanilla Kubernetes does not provide enough flexibility and features to manage a multi-cluster setup. This is where Liqo can provide that flexibility and features to better automate and simplify the entire process.

The idea is to use Liqo to connect main and failover clusters. The offloading functionality allows to share secrets and expose all necessary endpoints for communication between members of the replica set.

Exposing replica set members between clusters As explained in the previous section, all members of the replica set must communicate with each other, forming a full mesh. To do this, each member must be exposed with a cross-cluster service such as NodePort or an external LoadBalancer. The former is built into Kubernetes, but it lacks flexibility. The latter requires allocating the necessary IPs for all members. This is not an easy solution for bare-metal deployments, especially when clusters cannot communicate over a private network. Cloud provider solutions, on the other hand, offer easy configuration but can be economically expensive. If the cloud-provided LoadBalancer service does not provide name addressing (e.g., DNS), all IP endpoints must be hard-coded. With Liqo, the members of a replica set can be exposed across clusters using Liqo service reflection. All you need to do is peer the clusters and offload the required namespaces. This way, replica set members can reach members in another cluster by contacting the offloaded services, which will be simple ClusterIP services. Consequently, the built-in Kubernetes DNS can be used, avoiding hard-coding the IPs in the resources: a member of a replica set can be easily reached through a standard and fixed endpoint name (e.g., `<RS-SERVICE>.<NAMESPACE>.svc.cluster.local`). The following example shows a possible CR definition for deploying 3 nodes on the main site + 3 external nodes on the replica site.

```
- name: rs0
  size: 3
  externalNodes:
  - host: replica-cluster-rs0-0.percona-repl.svc.cluster.local
    priority: 1
    votes: 1
  - host: replica-cluster-rs0-1.percona-repl.svc.cluster.local
    priority: 1
    votes: 1
  - host: replica-cluster-rs0-2.percona-repl.svc.cluster.local
    priority: 0
    votes: 0
```

Liqo essentially provides a built-in and secure private network (+ DNS addressing) to connect replicas across different clusters, avoiding the hassle of setting up complex and/or expensive external solutions.

Sharing secrets between clusters Another useful benefit is the ability to share secrets easily and securely thanks to the Liqo reflection. By default, Liqo automatically reflects verbatim any secrets that exist in an offloaded namespace. This allows one cluster to seamlessly access the TLS and encryption keys secrets of the other cluster without the need for complex and risky OOB keys distribution.

The setup Let's consider a 2-cluster setup with one main site and one DR site. The Percona operator is deployed on both clusters. Both clusters deploy a StatefulSet containing a number of replica set members, each backed by the corresponding service. Then, both clusters offload the respective namespaces via Liqo. It is convenient to use the `Local` policy as the pod offloading strategy, since only the services and secrets need to be offloaded, not the pods. Now the pods in the main site can reach the pods in the DR site (and vice versa) by contacting the correct (offloaded) services. Figure 6.5 shows a graphical representation of the setup. As evident, all members can reach each other, forming a full mesh as required by the database. Note that a bidirectional peering is required, as each cluster must offload its services to the other. This strategy can also be extended to an N-cluster setup by establishing a bidirectional peering between each cluster.

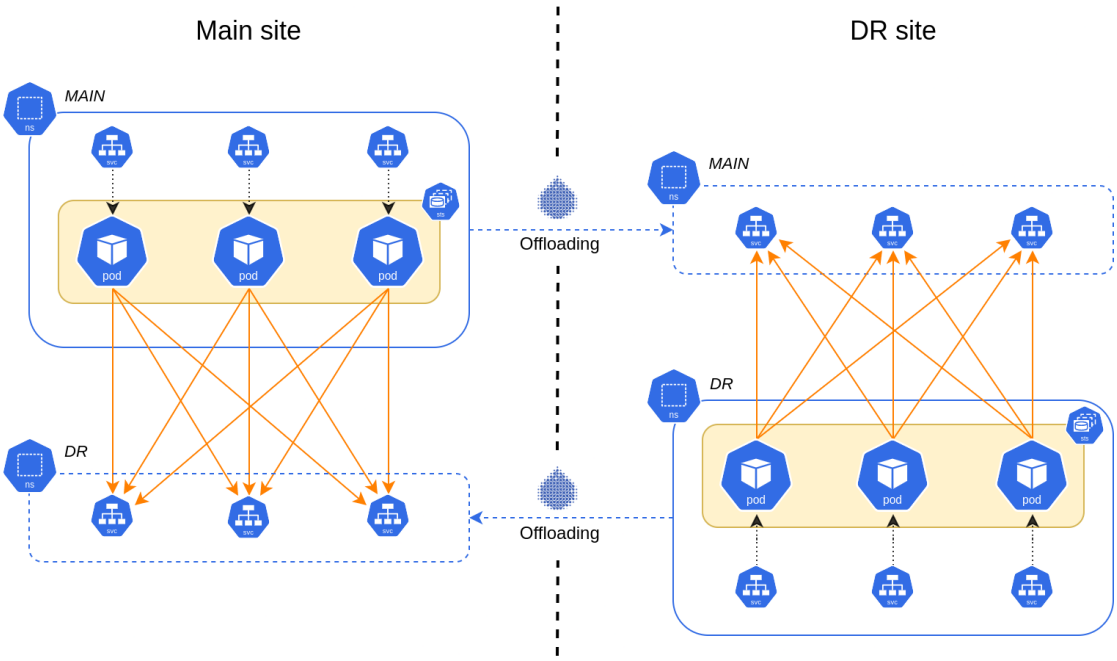

Figure 6.5: Cross-site Replication with Percona Operator and Liqo

Chapter 7

Conclusions

The need for multi-cluster environments has increased dramatically in recent years to meet the demands and to provide resources for modern businesses. The ability to create a federation of clusters that share resources and computation is the way to achieve this goal. However, multi-cluster topologies lead to increased infrastructure complexity. The presented thesis examined the potential problems that can arise in such setups.

The first part focused on analyzing service continuity in a multi-cluster environment powered by Liqo. Different scenarios were analyzed considering the failure of key elements of the infrastructure such as worker nodes, control planes, the network, etc. A new controller was introduced to better handle the failure of a worker node. Then, a new abstraction, the Shadow EndpointSlice, was developed to provide resilience in case of failures of the (local) cluster and/or the network connectivity between peers. The feature was tested with several performance benchmarks to evaluate its impact and overhead. Both works were implemented and integrated into the official Liqo codebase.

Finally, a proof of concept is proposed for a disaster recovery strategy that leverages a federation of Kubernetes clusters peered with Liqo. The idea is to interconnect multiple clusters, one of which is used as the main site and the others as failover sites for application data backup and recovery. In this context, Liqo's flexibility and functionality in multi-cluster environments can greatly simplify and automate the entire process.

Bibliography

[1] *Kubernetes official documentation*. URL: https://kubernetes.io/docs/home (cit. on pp. 4, 13, 14, 16, 17, 21, 31, 32).

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

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

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

[5] Kalyan Ramanathan. *5 business reasons why every CIO should consider Kubernetes*. Oct. 2019. URL: https://www.sumologic.com/blog/why-use-kubernetes (cit. on pp. 5, 11).

[6] Steven J. Vaughan-Nichols. *The five reasons Kubernetes won the container orchestration wars*. URL: https://blogs.dxc.technology/2019/01/28/the-five-reasons-kubernetes-won-the-container-orchestration-wars (cit. on p. 5).

[7] Eric Carter. *Sysdig 2019 Container Usage Report: New Kubernetes and security insights*. Oct. 2019. URL: https://sysdig.com/blog/sysdig-2019-container-usage-report (cit. on p. 7).

[8] *Raft Consensus Algorithm*. URL: https://raft.github.io (cit. on p. 9).

[9] *k8s Network Model*. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-model (cit. on p. 18).

[10] *k8s CNI*. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins (cit. on p. 20).

[11] *k8s Services*. URL: https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model (cit. on p. 21).

[12] *Kubebuilder git repository*. URL: https://github.com/kubernetes-sigs/kubebuilder (cit. on p. 21).

[13] *Kubernetes Operator pattern*. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator (cit. on p. 21).

[14] *Liqo documentation*. URL: https://docs.liqo.io (cit. on pp. 22, 38).

[15] *Virtual Kubelet GitHub repository*. URL: https://github.com/virtual-kubelet/virtual-kubelet (cit. on p. 27).

[16] *Liqo GitHub repository*. URL: https://github.com/liqotech/liqo (cit. on pp. 31, 53).

[17] *Kubernetes Nodes Conditions*. URL: https://kubernetes.io/docs/concepts/architecture/nodes/#condition (cit. on pp. 33, 48).

[18] *Istio Service Mesh*. URL: https://istio.io (cit. on p. 35).

[19] *Linkerd Service Mesh*. URL: https://linkerd.io (cit. on p. 35).

[20] Stakater AB. *Effective disaster recovery strategies for Kubernetes*. URL: https://www.cncf.io/online-programs/effective-disaster-recovery-strategies-for-kubernetes (cit. on pp. 36, 37).

[21] *OpenStack*. URL: https://www.openstack.org (cit. on p. 37).

[22] *Percona Operator for MongoDB*. URL: https://docs.percona.com/percona-operator-for-mongodb/index.html (cit. on pp. 38, 65).

[23] *MongoDB*. URL: https://www.mongodb.com (cit. on pp. 38, 65).

[24] Marco Iorio, Fulvio Risso, Alex Palesandro, Leonardo Camiciotti, and Antonio Manzalini. «Computing Without Borders: The Way Towards Liquid Computing». In: *IEEE Transactions on Cloud Computing* (2022), pp. 1–18. DOI: 10.1109/tcc.2022.3229163. URL: https://doi.org/10.1109%2Ftcc.2022.3229163 (cit. on pp. 41, 44).

[25] *Liqo Benchmarks repository*. URL: https://github.com/liqotech/liqo-benchmarks (cit. on p. 62).

[26] *Kubemark hollow nodes*. URL: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scalability/kubemark-guide.md (cit. on p. 62).

[27] *Kubernetes: considerations for large clusters*. URL: https://kubernetes.io/docs/setup/best-practices/cluster-large (cit. on p. 63).

[28] *Percona Operator: disaster recovery for mongodb on Kubernetes*. URL: https://www.percona.com/blog/disaster-recovery-for-mongodb-on-kubernetes (cit. on pp. 65–69).

[29] *MongoDB Replication*. URL: https://www.mongodb.com/docs/manual/replication (cit. on p. 65).

[30] *Split-brain (computing)*. URL: https://en.wikipedia.org/wiki/Split-brain_(computing) (cit. on p. 68).