POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Enabling Multi-Tenancy and Fine-grained Security in a Multi-Cluster Architecture

Supervisors

Prof. Fulvio RISSO

Dott. Alex PALESANDRO

Candidate

Andrea TERZOLO

October 2021

Summary

In the last two decades, the cloud has gained growing relevance. The current trend is to engineer the new web applications to be cloud-native, thus to be split up into loosely coupled micro-services, each one containerized and deployed as a part of a bigger application. The use of containers allows to cut oneself off the hosting physical hardware and operating system, letting to focus on the main purposes of a web application: to be widespread and high-available. The cloud allows to achieve this goal by gathering the infrastructure control under the cloud provider tenants and implementing the IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) paradigms: the computational, networking and storage resources are provided on demand to the cloud provider's customers as if they were services. A technology that broke through the cloud market is Kubernetes. This project, kicked off by Google, allows to automate deployment, scaling, and management of containerized applications. In recent years also the edge computing has gained growing importance. This is a distributed computing paradigm that brings the computational and storage resources close to the final user. The idea is to improve the QoS standards in terms of latency and bandwidth.

The goal of the project behind this thesis is to create a Kubernetes clusters federation. Many different tenants are connected to cooperate in creating a federation of clusters with computational, storage, and networking resources shared between them. In this scenario, every tenant can make its cluster resources available to others by sharing or leasing them out in a common environment.

This project needs a standard solution to take advantage of the resources offered by the federated clusters. The current implementation allows to create multi-cluster topologies, but without giving the tenant strict control in the use of shared resources. Furthermore, sharing resources requires full privileges on all federated clusters. This requirement in terms of privileges can undermine the support of a multi-ownership model where different companies are involved. This thesis has two core purposes: to provide tenants with fine-grained controls over the shared resources and to minimize the privileges required to allow solutions based on the multi-ownership model.

Table of Contents

List of Tables

List of Figures

Acronyms

K8s
: Kubernetes

CNCF
: Cloud Native Computing Foundation

CRD
: Custom Resource Definition

CR
: Custom Resource

CIDR
: Classless Inter-Domain Routing

API
: Application Programming Interface

REST
: Representational State Transfer

RPC
: Remote Procedure Call

KIND
: Kubernetes IN Docker

Chapter 1

Introduction

In the last years, container orchestration is becoming more and more relevant. Kubernetes, one of the main actors in this field, is used by big companies to orchestrate their jobs in their data centers. This trend is also becoming popular in small and medium companies that need to execute their jobs in smaller clusters or even in tiny ones at the edge or in IoT devices.

Technologies like the 5G or edge computing are leading to a multitude of small clusters geographically distributed that can serve the applications near the end-user. On the one hand, these clusters allow the companies to use the same APIs and the same applications both in the core of the cloud and at the edge. On the other, it is complicated to have enough resources to deal with traffic peaks. Here is where multi-cloud solutions like **Liqo** come.

Liqo, the project behind this thesis, enables the creation of a multi-cluster architecture with liquid resources sharing between different clusters. This is what is required in a large ecosystem of small clusters, with a relatively low medium load but with a lot of load peaks, where each one can use, for a short amount of time, the resources available in other places, both in the core of the cloud or in other edge/IoT sides.

1.1 Different multi-cluster environments

Organizations may need a multi-cluster environment for many different reasons. It is possible to distinguish between two main categories of environments:

1. **Cloud Environment**: companies can have large data centers, both on-premise (in private infrastructure and on proprietary hardware) and on managed solutions (in a public cloud provider, like Amazon Web Services, Google Cloud Platform, Microsoft Azure, and many others).

An organization may need multiple clusters in a cloud environment to have a high resource availability. These clusters can be hosted by different cloud providers to contain costs and to avoid binding the environment to a specific provider. The usage of multiple clusters can also reduce some scalability problems in huge environments.

2. **Edge or IoT Environment**: companies can have many small clusters, even single-node ones. They can be geographically distributed to be closer to the end-user.

   In this scenario, the core needs are the availability of the same API and the re-use of the same skills, already achieved for the cloud world, to manage small devices. With the interoperability of the API, a new and closer integration becomes possible: moving applications between different devices.

1.2 Goal of the thesis

Both in the Cloud Environment, with a static and well-known infrastructure, both in the Edge one, with available clusters and devices that can change rapidly over time, there is the need for an automatic discovery mechanism. Liqo already provides this feature allowing Kubernetes clusters to discover and join them, keeping the neighborhood knowledge updated over time. What is missing is the possibility for every tenant to freely use the available clusters resources, requiring specific security and business policies.

So far, Liqo offers the opportunity to offload applications using all available clusters. It is not yet possible to specify any constraints during the offloading phase. Besides, full privileged access to remote clusters is required to use these shared resources. Unfortunately, this requirement undermines the support of a multi-ownership model where different companies are involved.

This thesis aims, therefore, not only to extend the multi-cluster framework allowing the definition of fine-grained offloading policies but also to reduce the required privileges on remote clusters. The solution must be smooth and seamless. Liqo is just a *cluster accessory* it can be enabled and disabled freely without affecting the performance of the clusters involved in the resource sharing. The objectives of this work are mainly three:

1. Find a solution that allows every tenant to define custom constraints on the shared resources use. For example, the tenant must be able to offload his applications only on some available clusters.

2. Figure out how to create the multi-cluster topology based on the tenant constraints. These topologies span across multiple clusters requiring some

privileges on them. The idea is to reduce as much as possible the requirements in terms of privileges allowing the support of a multi-ownership model.

3. Allow the tenant to take advantage of the *liquid resource sharing* to deploy his applications in a safe and controlled manner. The idea is to implement some abstractions allowing users to straightforwardly and automatically exploit the multi-cluster environment.

The analysis proceeds following this structure:

- **Chapter 2** provides an extensive presentation of Kubernetes concepts necessary to understand the implemented solutions.
- **Chapter 3** presents the Liqo architecture and some of its core concepts. This chapter figures out also the already existing multi-tenancy framework, analyzing its pros and cons.
- **Chapter 4** introduces the offloading constraints concept showing how it can be applied in the Liqo project.
- **Chapter 5** considers the privileges problem, analyzing some open-source multi-tenancy solutions. More precisely, it analyses with particular attention the pros and cons of the main approaches.
- **Chapter 6** presents the solution chosen to realize multi-cluster topologies with reduced privileges, paying particular attention to the namespace replication process.
- **Chapter 7** analyzes the abstraction implemented to automatically create deployment replicas on multiple remote clusters.
- **Chapter 8** analyses the achieved results in terms of reliability and scalability.

Chapter 2

Kubernetes

This chapter provides an overview of the Kubernetes architecture showing its history and evolution through time. This summary lays the foundations for all the concepts which will be exposed later on. Kubernetes (often shortened as K8s) is a huge framework, and a deep examination of it would require much more time and discussion, hence we only provide here a description of its core concepts and components. Further details can be found in the official documentation [1].

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

2.1 Kubernetes: a bit of history

Around 2004, Google created the **Borg** [4] system, a small project with less than 5 people initially working on it. The project was developed in collaboration with a new version of Google's search engine. Borg was a large-scale internal cluster management system, which "ran hundreds of thousands of jobs, from many thousands of different applications, across many clusters, each with up to tens of thousands of machines" [4].

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

In the middle of 2014, Google presented **Kubernetes** as an open-source version of Borg. Kubernetes was created by Joe Beda, Brendan Burns, Craig McLuckie, and other engineers at Google. Its development and design were heavily influenced by Borg, and many of its initial contributors used to work on it. The original Borg

project was written in C++, whereas for Kubernetes, the Go language was chosen.

In 2015 Kubernetes v1.0 was released. Along with the release, Google set up a partnership with the Linux Foundation to form the **Cloud Native Computing Foundation** (CNCF) [6]. Since then, Kubernetes has significantly grown, achieving the CNCF graduated status and being adopted by nearly every big company. Nowadays, it has become the de-facto standard for container orchestration [7, 8].

2.2 Applications deployment evolution

Kubernetes is a portable, extensible, open-source platform for running and coordinating containerized applications across a cluster of machines. It manages the life cycle of applications and services using methods that provide consistency, scalability, and high availability.

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

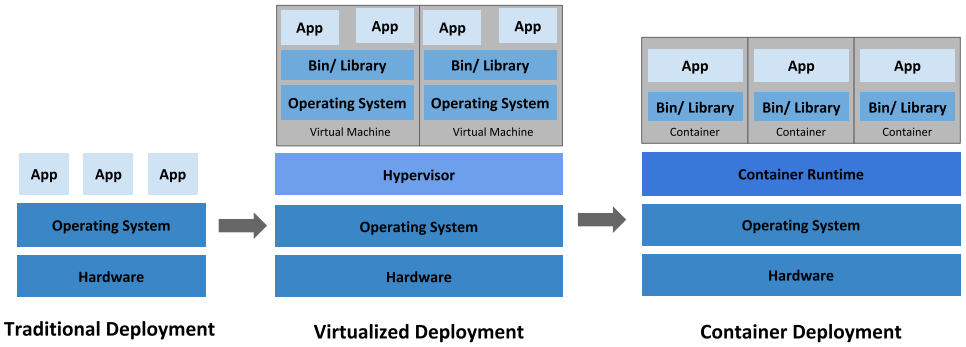

Figure 2.1: Evolution in applications deployment.

Traditionally, organizations used to run their applications on physical servers. One of the problems of this approach was that resource boundaries between applications could not be applied in a physical server, leading to resource allocation issues. For example, if multiple applications run on a physical server, one of them could take up most of the resources, and as a result, the other applications would starve. A possibility to solve this problem would be to run each application on a different physical server, but clearly, it is not feasible. This solution could not scale, would lead to resources under-utilization, and would be very expensive for organizations to maintain many physical servers.

The first real solution has been **virtualization**. Virtualization allows multiple Virtual Machines to run on a single physical server. This technique grants

isolation of the applications between VMs, providing a high level of security, as the information of one application cannot be freely accessed by other applications. Virtualization enables better utilization of resources in a physical server, improves scalability because an application can be added or updated very easily, reduces hardware costs, and much more. With virtualization, it is possible to group a set of physical resources and expose it as a cluster of disposable virtual machines. Isolation certainly brings many advantages, but it requires a quite 'heavy' overhead: each VM is a full machine running all the components, including its operating system, on top of the virtualized hardware.

A second solution has been proposed recently: **contai nerization**. Containers are similar to VMs, but they share the operating system with the host machine, relaxing isolation properties. Therefore, containers are considered a lightweight form of virtualization. Similarly to a VM, a container has its filesystem, CPU, memory, process space, etc... One of the key features of containers is that they are portable. They are decoupled from the underlying infrastructure and are totally portable across clouds and OS distributions. This property is particularly relevant nowadays with cloud computing: a container can be easily moved across different machines. Moreover, being "lightweight", containers are much faster than virtual machines: they can be booted, started, run, and stopped with little effort and in a short time.

2.3 Container orchestrators

When hundreds or thousands of containers are created, the need for a way to manage them becomes essential; container orchestrators serve this purpose. A container orchestrator is a system designed to easily manage complex containerization deployments across multiple machines from one central location. As depicted in figure 2.2, Kubernetes is by far the most used container orchestrator. A description of this system is provided in the following.

Kubernetes provides many services, including:

- **Service discovery and load balancing** A container can be exposed using the DNS name or using its IP address. If traffic to a container is high, a load balancer able to distribute the network traffic is provided.
- **Storage orchestration** A storage system can be automatically mounted, such as local storage, or dynamic storage supplied by public cloud providers, and more.
- **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

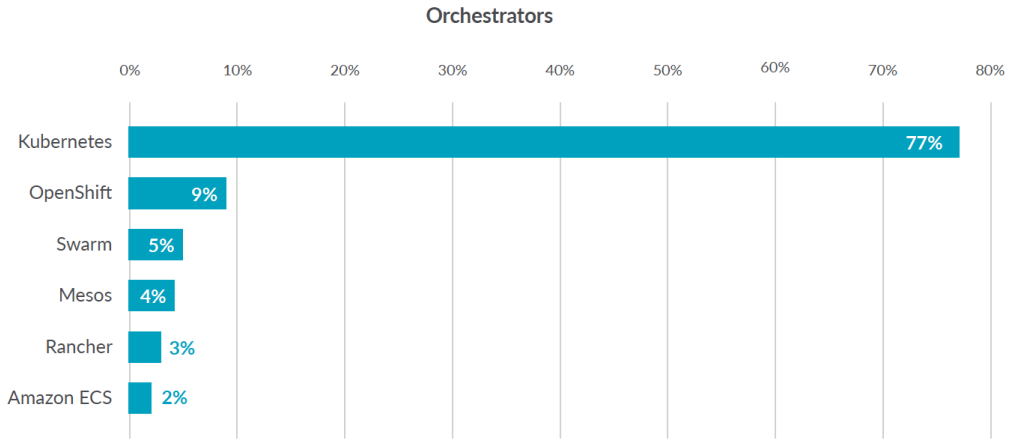

Figure 2.2: Container orchestrators use. [9]

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

- **Automatic bin packing** Kubernetes is provided with a cluster of nodes that can be used to run containerized tasks. It is possible to set how much CPU and memory (RAM) each container needs, and automatically the containers are sized to fit in the nodes to make the best use of the resources.
- **Secret and configuration management** It is possible to store and manage sensitive information in Kubernetes, such as passwords, OAuth tokens, and SSH keys. It is possible to deploy and update secrets and application configuration without rebuilding the container images and exposing secrets in the stack configuration.

2.4 Kubernetes architecture

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

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

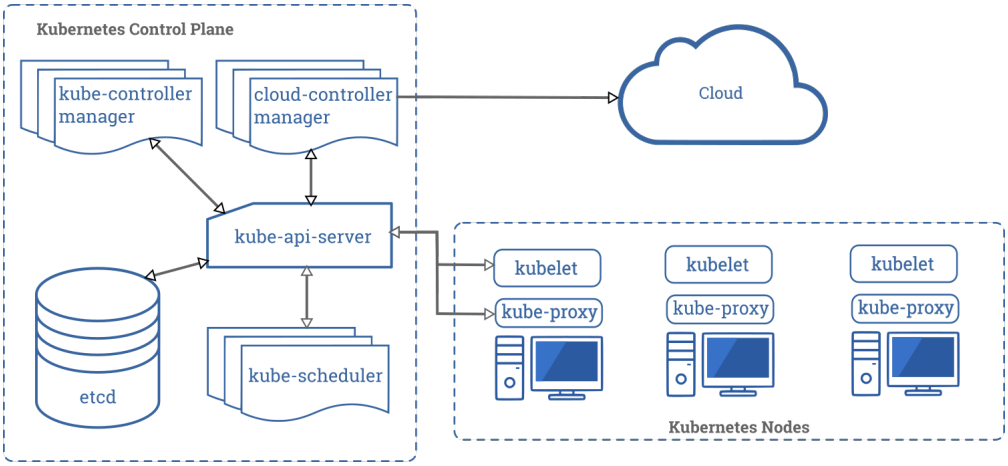

Figure 2.3: Kubernetes architecture.

2.4.1 Control plane components

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

API server

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

Etcd

The etcd is a distributed, consistent, and highly available key-value store used as Kubernetes backing store for all cluster data. It is based on the Raft consensus algorithm [10], which allows different machines to work as a coherent group and survive the breakdown of one of its members. The etcd can be stacked in the master node or external, installed on a dedicated host. Only the API server can communicate with it.

Scheduler

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

Kube-controller-manager

The kube-controller-manager is a component that runs controller processes. It continuously compares the desired state of the cluster (given by the objects' specifications) with the current one (read from etcd). From a logical point of view, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process. These controllers include:

- **Node Controller**: responsible for noticing and reacting when nodes go down.
- **Replication Controller**: in charge of maintaining the correct number of pods for every replica object in the system.
- **Endpoints Controller**: populates the Endpoint objects (which link Services and Pods).
- **Service Account & Token Controllers**: create default accounts and API access tokens for new namespaces.

Cloud-controller-manager

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

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

- **Node Controller**: checks the cloud provider to update or delete Kubernetes nodes using cloud APIs.

- **Route Controller**: responsible for setting up network routes in the cloud infrastructure.
- **Service Controller**: responsible for creating, updating and deleting cloud provider load balancers.
- **Volume Controller**: creates, attaches, and mounts volumes, interacting with the cloud provider to orchestrate them.

2.4.2 Node components

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

Container Runtime

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

Kubelet

The kubelet is an agent that runs on each node of the cluster, making sure that containers are running in the node's pods. This agent receives from the API server the specifications of the Pods and interacts with the container runtime to run them, monitoring their state and assuring that the containers are running and healthy. The connection with the container runtime is established through the Container Runtime Interface and is based on gRPC.

Kube-proxy

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

Addons

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

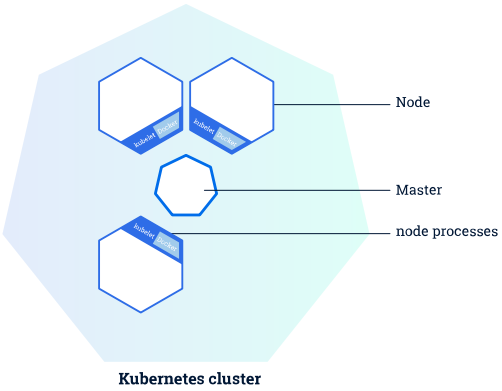

Figure 2.4: Kubernetes master and worker nodes. [1].

2.5 Kubernetes objects

Kubernetes defines several types of objects, which constitutes its building blocks. Usually, a K8s resource object contains the following fields [11]:

- `apiVersion`: the versioned schema of this representation of the object;
- `kind`: a string value representing the REST resource this object represents;
- `ObjectMeta`: metadata about the object, such as its name, annotations, labels etc.;
- `ResourceSpec`: defined by the user, it describes the desired state of the object;
- `ResourceStatus`: filled in by the server, it reports the current state of the resource.

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

- **Create**: create the resource in the storage backend; once a resource is created, the system applies the desired state.
- **Read**: comes with 3 variants:
  - **Get**: retrieve a specific resource object by name;
  - **List**: retrieve all resource objects of a specific type within a namespace, and the results can be restricted to resources matching a selector query;
  - **Watch**: stream results for an object(s) as it is updated.

- **Update**: comes with 2 forms:
  - **Replace**: replace the existing spec with the provided one;
  - **Patch**: apply a change to a specific field.
- **Delete**: delete a resource. Depending on the specific resource, child objects may or may not be garbage collected by the server.

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

2.5.1 Namespace

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

- **kube-system**: it contains objects created by 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 workload into many Namespaces to better virtualize the cluster.

2.5.2 Pod

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

An important feature widely used in this thesis is the possibility to constrain a Pod so that it could only run on a particular set of Nodes. There are several ways to do this, and the recommended approaches all use label selectors to facilitate the selection. In particular, the Node affinity approach is one of the more expressive.

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

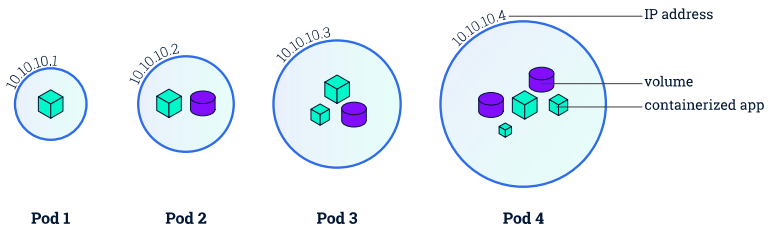

Figure 2.5: Kubernetes pods. [1]

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-node-affinity
spec:
  affinity:
    nodeAffinity:

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

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

2.5.3 ReplicaSet

ReplicaSets control a set of pods allowing to scale the number of pods currently in execution. If a pod in the set is deleted, the ReplicaSet notices that the current number of replicas (read from the `Status`) is different from the desired one (specified in the `Spec`) and creates a new pod. ReplicaSets are usually not used directly: a

higher-level concept, called **Deployment**, is provided by Kubernetes.

2.5.4 Deployment

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

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

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

2.5.5 Service

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

on its `ServiceType`:

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

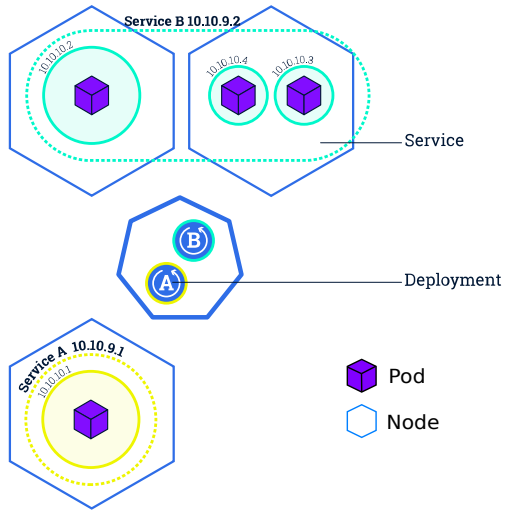

Figure 2.6: Kubernetes Services. [1]

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

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
```

```
spec:
  selector:
    app: myApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

2.6 RBAC

Kubernetes defines several APIs for the management of accesses. The Role-based access control (RBAC) is a method of regulating access to computers or network resources based on the users' roles.

The API group `rbac.authorization.k8s.io` defines four object types to define these permissions:

- **Role**: define rules valid for a specific namespace
- **ClusterRole**: define rules valid for all namespaces
- **RoleBinding**: link an identity to a set of rules in a specific namespace
- **ClusterRoleBinding**: link an identity to a set of roles in all namespaces

2.6.1 ServiceAccount

The ServiceAccount is a Kubernetes object in the `core/v1` API group that provides an identity for processes. When a new object of this kind is created, the API Server provides to it a new client certificate that will be used in all future authentications.

2.6.2 Role and ClusterRole

The *Role* and the *ClusterRole* contains rules that represent a set of permissions. In these permissions, there cannot be "deny" rules.

The only difference between them is that the first sets the permissions within a particular namespace (the one which contains the resource), while the second is a cluster-wide resource and can be used in all the namespaces.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
```

```
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

In this example [1] we are creating a set of permissions in the *default* namespace that will grant access to get, watch, and list pod resources. We can have a similar example, but cluster-wide scoped, with the following ClusterRole.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

2.6.3 RoleBinding and ClusterRoleBinding

The *RoleBinding* and the *ClusterRoleBinding* resources [1] grant the permissions defined in a *Role* or a *ClusterRole* to a given user, set of users or to a ServiceAccount. A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide.

```
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" namespace.
# You need to already have a Role named "pod-reader" in that namespace.
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
```

```
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

2.7 Virtual-Kubelet

Two Kubernetes-based tools which have been used during the development of this project are Virtual-Kubelet and Kubebuilder. Virtual Kubelet is an open source Kubernetes kubelet implementation that masquerades a cluster as a kubelet for connecting Kubernetes to other APIs [2]. Virtual Kubelet is a Cloud Native Computing Foundation sandbox project.

The project offers a provider interface that developers need to implement to use it. The official documentation [2] says that "providers must provide the following functionality to be considered a supported integration with Virtual Kubelet:

1. Provides the back-end plumbing necessary to support the lifecycle management of pods, containers, and supporting resources in the context of Kubernetes.

2. Conforms to the current API provided by Virtual Kubelet.

3. Does not have access to the Kubernetes API Server and has a well-defined callback mechanism for getting data like secrets or configmaps".

2.8 Kubebuilder

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

CustomResourceDefinition is an API resource offered by Kubernetes, which allows to define Custom Resources (CRs) with a name and schema specified by the user. When a new CustomResourceDefinition is generated, the Kubernetes API server instantiates a new RESTful resource path. The CRD can be either namespaced or cluster-scoped, and its name must be a valid DNS subdomain name.

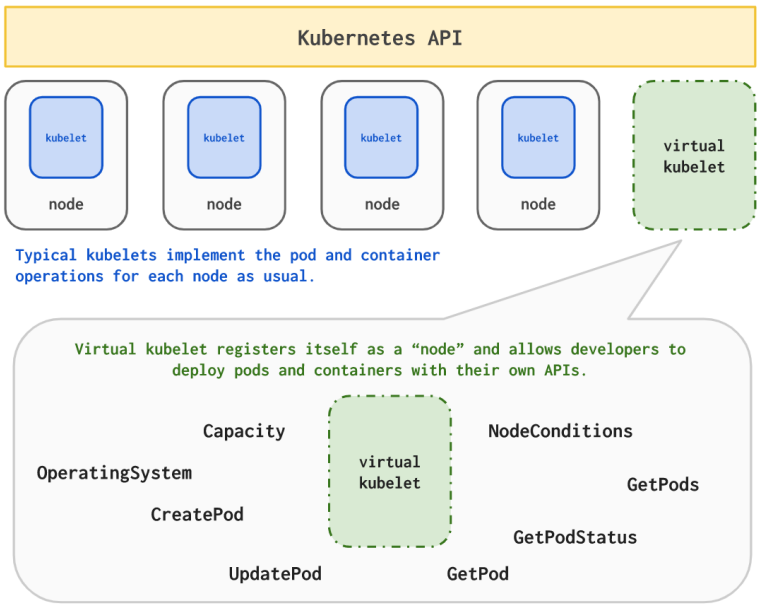

Figure 2.7: Virtual-Kubelet concept. [2]

A **Custom Resource** is an endpoint in the Kubernetes API that is not available in a default Kubernetes installation and which frees users from writing their own API server to handle them [1]. To have more powerful management, you also need to provide a custom controller which executes a control loop over the custom resource: this behavior is called Operator pattern [12].

Kubebuilder helps a developer in defining his Custom Resources taking basic decisions, and writing a lot of scaffolded code. These are the main actions operated by Kubebuilder [3]:

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

Chapter 3

Liqo project

This chapter analyzes the Liqo architecture, showing the idea behind it. It describes the overall picture of the open-source project where this work is involved, how it is integrated, and its importance.

3.1 Liqo philosophy

Liqo aims to create an opportunistic interconnection of multiple Kubernetes clusters allowing seamless resource and service sharing among them. The idea is to create an *endless Kubernetes ocean* where the user applications can be deployed.

Clusters usually have underutilized resources because they deal with peaks of load on their own, but during the day, they also have moments of low load. In these moments, they are wasting part of their resources that could be available to be shared.

Liqo aims to extend the resources present in the local cluster using, in an opportunistic way, the ones currently not busy in the neighbor clusters. The philosophy is that no peering and no sharing are definitive or not reversible it is always possible to disable the peering between clusters, coming back to the original state. When a cluster is extended with Liqo, there is no change in the standard Kubernetes APIs. The resources described in Chapter 2 are still valid in the new environment, and the user applications have not to be changed to work with Liqo.

3.2 Liqo five pillars

Liqo manages multiple Kubernetes clusters, allowing the user to take advantage of external resources transparently. The cluster management functionality introduced by Liqo can be described with five pillars:

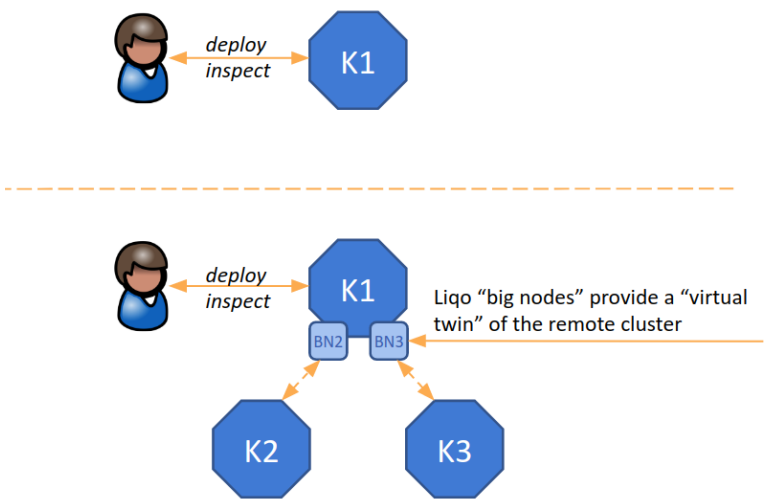

Figure 3.1: No Change in Kubernetes API.

1. **Discovery**: Discover available clusters.

2. **Peering**: Establish an administrative interconnection between the clusters and negotiate the parameters.

3. **Network Interconnection**: Establish a network infrastructure between the clusters.

4. **Resource Management**: Create an abstraction to make the external resources available.

5. **Usage**: Offload your pods.

3.2.1 Discovery

Liqo can dynamically discover and add new clusters to the *Big Cluster* abstraction. The figure 3.2 shows how clusters can be discovered in many different ways, such as manually (for testing or not-yet-configured domains), or by an automatic configuration with DNS (on selected domains), or with mDNS (only on local area network).
The discovery process takes information from different data sources in the remote clusters and generates a new ForeignCluster CR in the local cluster.

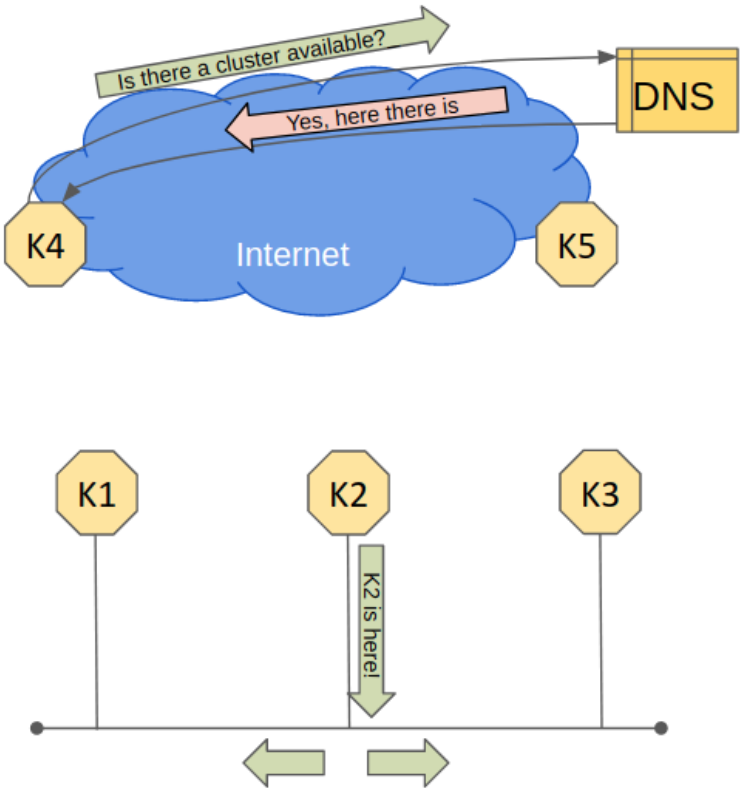

Figure 3.2: Discovery.

3.2.2 Peering

Liqo can dynamically peer different and administratively separate clusters with a policy-driven and direct relationship. This connection has to be established before sharing any resources. Liqo creates a peer-to-peer architecture, so no *master* cluster is involved.
The peering phase uses information collected during the discovery phase to contact the remote cluster and checks that both clusters that will be part of the peering are available and have accepted the interconnection.

3.2.3 Network Interconnection

Liqo can extend the cluster network to the remote clusters, basing on the peering information. The network parameters required to establish the VPN tunnel are dynamically negotiated with dedicated CR. Liqo supports overlapping pod CIDR in the different clusters. It does not make any assumption on the IP address space

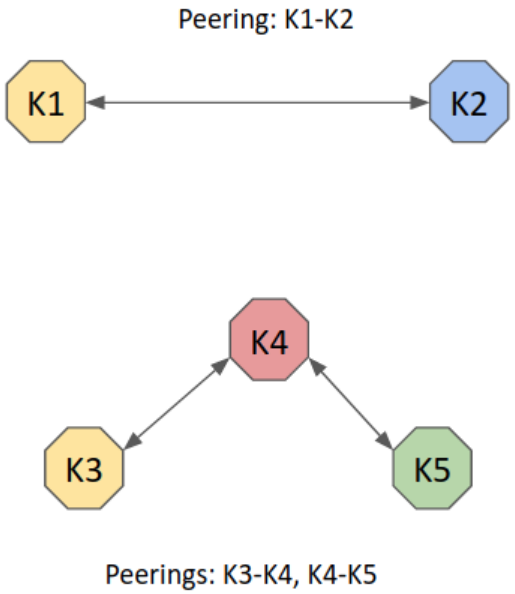

Figure 3.3: Peering.

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

3.2.4 Resource Management

When a cluster accepts the peering with another one, it can advertise the resources' amount it can share. If the remote cluster accepts the offer, it instantiates a new *virtual node* with these resources (i.e. CPU and memory). The virtual nodes are equivalent to physical nodes, hence they can be controlled by the vanilla Kubernetes scheduler and controller-manager.
When the peering and the network interconnection are completed, the *virtual kubelet* enables the new node, setting it to ready. The figure 3.5 shows the two virtual nodes created in the two different clusters after the peering phase.

3.2.5 Usage

When a new virtual node is set up and marked as ready, the vanilla Kubernetes scheduler can schedule new pods on it. If some pods are scheduled on the virtual node, the virtual kubelet is in charge of offloading these pods, reflecting them into

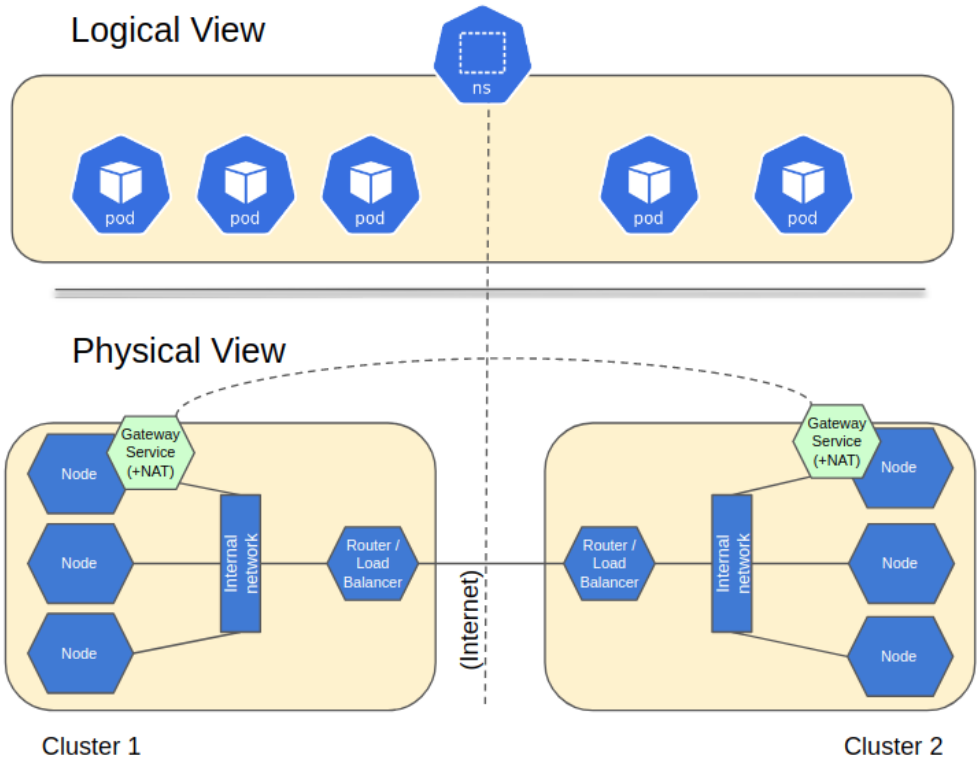

Figure 3.4: Network Interconnection.

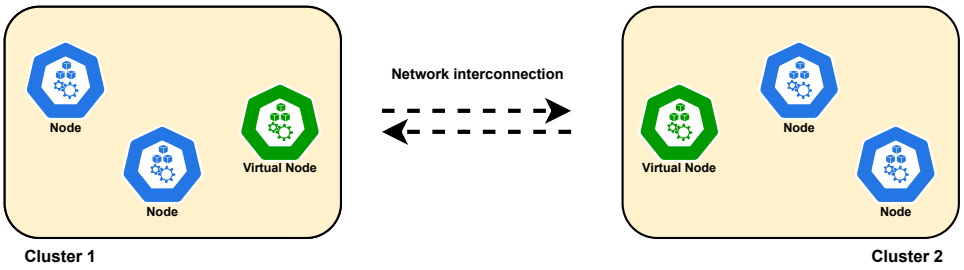

Figure 3.5: Resource Management.

the corresponding remote cluster. The virtual kubelet must also keep the local shadow pods aligned with real remote pods, as seen in the figure 3.6.

The remote pods are reachable from the local cluster and they can also connect themselves to the local services. Services and Endpoints are consistent on both the clusters because of the virtual kubelet reflection of EndpointSlices and IP translation.

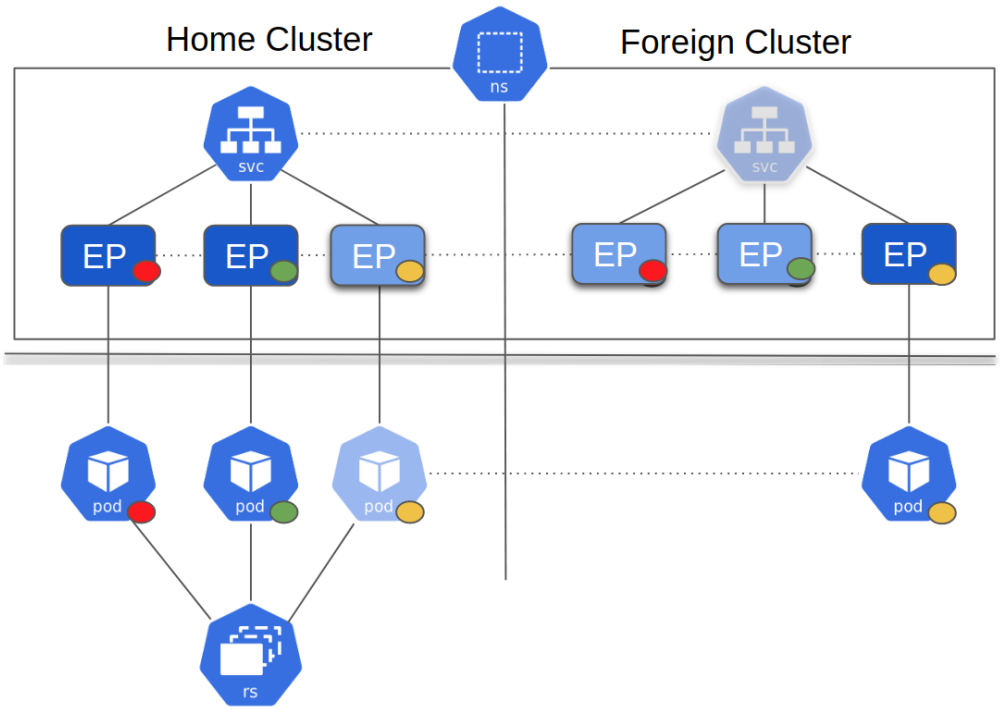

Figure 3.6: Usage.

3.3 Current multi-tenancy support

The actual Liqo version provides support for multi-tenancy in a multi-cluster environment. More precisely, a tenant can replicate his local namespace on all available clusters. The following sections figure out how this feature works and the possible limits that it carries with it.

3.3.1 The Liqo webhook role

When the peering phase ends, the virtual kubelet instantiates a new virtual node, and pods could be potentially scheduled on it. This could be a problem. For example, some tenants maybe do not want to offload their workload on remote clusters, so the scheduling on virtual nodes cannot be enabled by default. To solve this problem, the virtual kubelet creates the virtual node with a particular k8s taint. This taint denies the scheduler to chose this node for pods that do not support toleration to the taint. Tenants, who do not care about Liqo features, can simply deploy their pods as in a standard k8s cluster. If a tenant wants to take advantage of Liqo's capabilities, he must create a particular namespace and deploy the pods within it. The Liqo webhook component recognizes that these pods belong

to the particular namespace and applies toleration for the virtual nodes. Now the scheduler may also choose the virtual nodes as a target for these pod scheduling. More precisely, the Liqo webhook is a standard Kubernetes mutating webhook that has the task of applying tolerations to the correct pods. During the thesis, the webhook logic will be expanded, allowing additional features and more selective scheduling.

3.3.2 The offloading process

This section shows the complete offloading process flow. The figure 3.7 is used to better understand the situation.

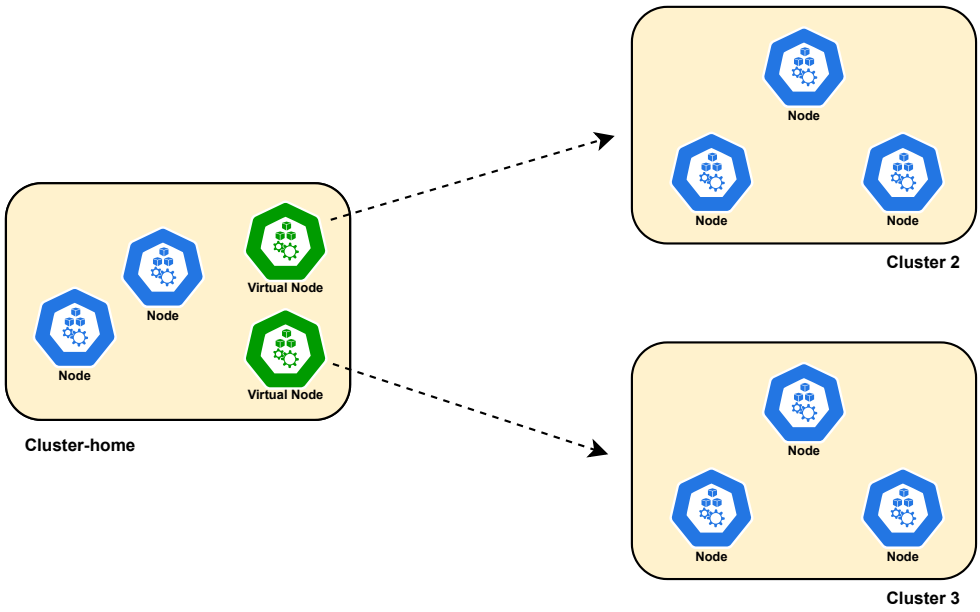

Figure 3.7: Offloading Process.

The *cluster-home* has a unidirectional peering with the *cluster-2* and *cluster-3*, so only the local cluster can use the resources of the other two. As shown in the figure 3.7, the *cluster-home* has two virtual nodes to exploit these foreign resources.

Proceeding in order, these are the main steps that lead to the pods deployment on remote clusters:

1. After the peering phase, each virtual kubelet creates a virtual node with the Liqo taint (the virtual nodes have all the same taint when a pod tolerates a virtual node, it tolerates all).
2. When a tenant wants to exploit the Liqo offloading, he has to label a standard namespace with the label `liqo.io/enabled=true`.

3. Once the namespace is correctly labeled, the tenant can simply deploy his pods inside it. As seen previously, the Liqo webhook will recognize the enabling label and will mutate pods with the right toleration.

4. Now, the scheduler can freely choose where to schedule pods, according to the nodes' available resources. It is worth noting that the scheduler is not obliged to schedule pods on virtual nodes, so it is possible that they will all be deployed locally.

5. If a pod is scheduled on a virtual node, the virtual kubelet creates a remote replica of the local namespace, if it is not already there, and reflects the pod inside it. The remote namespace will always have a name composed by the local namespace name plus the cluster-id of the local cluster. In this way, there will never be name conflicts in the remote clusters.

If the tenant deploys, in a local labeled namespace, three pods and a Service object to expose them, a possible outcome could be the one expressed in figure 3.8. In this case, the scheduler has chosen to schedule one pod locally and the other two remotely, one for each remote cluster. This is just an assumption, the scheduler could also choose to deploy all pods locally. Unlike pods, services will always be replicated inside all remote namespaces to guarantee complete access from all clusters.

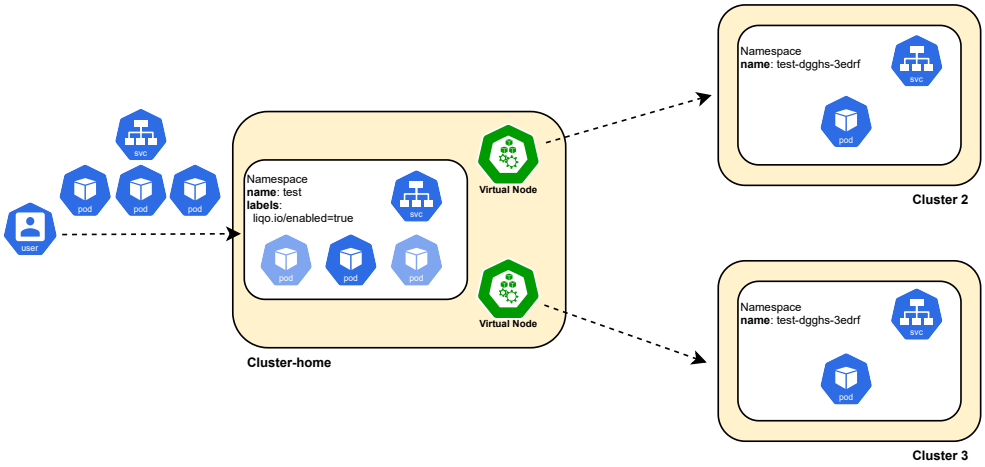

Figure 3.8: Offloading Example.

3.3.3 Solution evaluation

The main advantages of this approach are the simplicity and transparency with which a tenant can create a multi-cluster topology on all available clusters. He can simply label a namespace to enable the remote namespace replication process and deploy his applications inside it. Tenants who do not want to take advantage of these features can simply use the cluster as if Liqo was not there. However, this solution, although simple, has some not negligible disadvantages. These are the most relevant ones:

- Each virtual kubelet has the privileges to create as many remote namespaces as it wants on all remote clusters. With such an approach, a cluster could easily use all the resources of another cluster without any constraints. In a multi-ownership scenario with clusters belonging to different companies, there could be catastrophic consequences.
- When a tenant enables Liqo features, the pods scheduled within a labeled namespace could end up on any available clusters. This may be an unwanted behavior. For example, some sensitive applications may require a specific security policy, which only some remote clusters can offer.
- As already mentioned, even if pods have the right toleration, they don't necessarily have to be remotely scheduled. In some cases, the user may want to schedule some pods on a specific cluster, or maybe schedule them only locally, allowing the remote access thanks to services replicated in remote namespaces.
- The remote namespaces always have a name consisting of: the local namespace name as a prefix and the local cluster-id as a suffix. This could be a problem if users want to offer cross-namespace services that require particular or identical names for the various namespaces of the topology.
- The virtual kubelet creates the remote namespace when the first pod is remotely scheduled. This would not be a problem but, the virtual kubelet has to cover too many functions, and such an approach may not be clear and thoughtful in terms of performance.

These problems are solved in the next chapters of this work. More precisely, chapter 4 introduces a new Liqo CR that allows users to specify offloading constraints in a simple and intuitive way.

Chapter 4

Offloading constraints

This chapter aims to present a possible solution for some problems of the previous section. The main goal is to find an approach that allows admins to impose specific offloading constraints on users' deployments. More precisely, among these constraints, it is necessary to introduce the possibility of restricting offloading only to a limited number of clusters.

The current problem is that all remote clusters are seen as indistinguishable resource containers. The admin cannot select a topology only with clusters that provide certain characteristics because virtual nodes do not expose any properties of the corresponding remote cluster. The idea is that every virtual node can expose a set of labels with the most relevant features of the remote cluster.

4.1 Cluster labels

When an admin installs Liqo on his cluster, he may decide to expose some of its most significant characteristics in the form of labels. It is worth noting that there is no restriction on the labels to choose, they can characterize the clusters showing their geographical location, the underlying provider, or the presence of specific hardware devices. During the installation, a Liqo CR, called **ClusterConfig**, collects these labels. This resource will be read during the peering process so that these characteristics can be exported to other clusters. Therefore, in addition to exchanging the amount of the available resources, this information will also be sent. After the peering phase, the virtual node will expose those labels, enabling the possibility to select them during the offloading configuration.

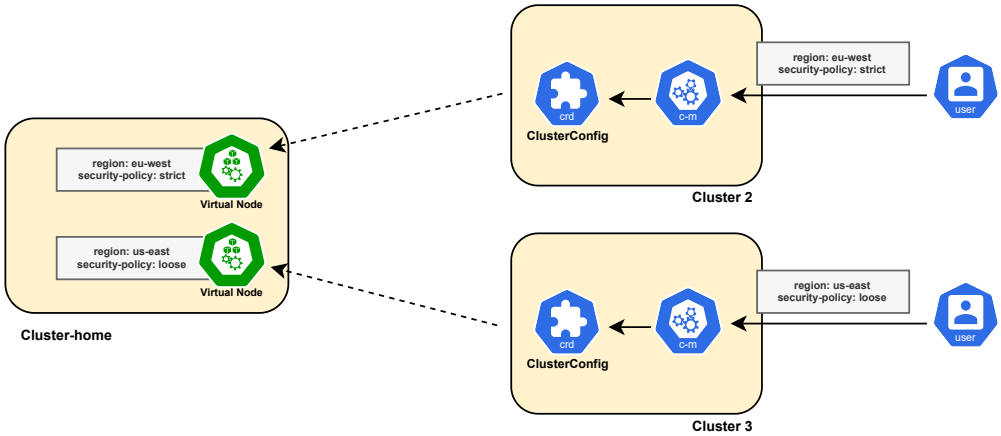

Figure 4.1: Cluster labels addition.

4.2 NamespaceOffloading resource

What this solution lacks is the possibility to select clusters using the cluster labels just presented. The admin can observe the remote clusters' characteristics, but with the current implementation, he has no way to inform the Liqo control plane of his selection. To solve this problem, Liqo provides a new CR, called **NamespaceOffloading**. More precisely, this resource not only allows users to specify the clusters involved in the offloading but also to enforce additional constraints such as the pod offloading strategy and the remote namespaces name. The resource offers users three spec fields to specify these constraints:

1. The `namespaceMappingStrategy` defines the naming strategy used to create the remote namespaces.

2. The `podOffloadingStrategy` defines constraints about pod scheduling.

3. The `clusterSelector` specifies filters to target specific remote clusters.

The following sections figure out the role that these fields play during the offloading process.

4.2.1 NamespaceMappingStrategy

The namespaceMappingStrategy defines the naming strategy used to create the remote namespaces. The accepted values are the ones expressed in table 4.1.

Value	Description
DefaultName (Default)	Remote namespaces have the name of the local namespace followed by the local cluster-id to guarantee the absence of conflicts.
EnforceSameName	Remote namespaces have the same name as the namespace in the local cluster (this approach can lead to conflicts if a namespace with the same name already exists inside the selected remote clusters).

Table 4.1: NamespaceMappingStrategy values.

The DefaultName value is recommended if the user does not have particular constraints related to the remote namespaces name. However, using the DefaultName policy, the namespace name cannot be longer than 63 characters, according to RFC 1123 [13]. Since the cluster-id is 37 characters long, the local namespace name can have at most 26 characters.

4.2.2 PodOffloadingStrategy

The podOffloadingStrategy defines constraints about pod scheduling. The table 4.2 figures out the possible values.

Value	Description
LocalAndRemote (Default)	Pods deployed in the local namespace can be scheduled both locally and remotely.
Local	Pods deployed in the local namespace are always scheduled inside the local cluster, never remotely.
Remote	Pods deployed in the local namespace are always scheduled inside the remote clusters, never locally.

Table 4.2: PodOffloadingStrategy values.

The LocalAndRemote strategy does not impose any constraints: it leaves the scheduler the choice to select both local and remote nodes. The Remote and Local strategies force the pods to be scheduled respectively only remotely and only locally. Unlike pods, standard Kubernetes Services are always replicated inside all the selected clusters.

4.2.3 ClusterSelector

The clusterSelector specifies nodeSelectorTerms to target specific clusters of the topology. Such nodeSelectorTerms can be defined by using the Kubernetes NodeAffinity syntax. Here the previously installed cluster labels are used to choose a cluster rather than another.

If not specified at creation time, the clusterSelector will target all virtual nodes available, enabling the offloading on all peered clusters. More precisely, the default value corresponds to:

```
clusterSelector:
  nodeSelectorTerms:
  - matchExpressions:
    - key: liqo.io/type
      operator: In
      values:
      - virtual-node
```

The clusterSelector will be applied on every pod created inside the namespace to enforce these constraints. In that way, users can deploy pods only inside the remote clusters chosen by the admin.

4.2.4 NamespaceOffloading status

The NamespaceOffloading status collects information about the actual conditions of remote namespaces, e.g., if the replication succeeded or not. More in detail, the information provided are the following:

- The `remoteNamespacesConditions` allows users to verify remote namespaces' presence and their status inside all remote clusters. This field is a map that has the remote cluster-id as key and as value, a vector of conditions for the namespace created inside that remote cluster. There are two types of conditions:

  1. The **Ready** condition indicates whether the remote namespace is correctly created.
  2. The **OffloadingRequired** condition specifies if a namespace replica is required inside that remote cluster.

- The `offloadingPhase` field informs users about the namespaces offloading status. It can assume different values:

Value	Description
Ready	Remote Namespaces have been correctly created inside previously selected clusters.
NoClusterSelected	No cluster matches user constraints or constraints are not specified with the right syntax.
SomeFailed	There was an error during some remote namespaces creation.
AllFailed	There was an error during all remote namespaces creation.
Terminating	Remote namespaces are undergoing graceful termination.

Table 4.3: OffloadingPhase values.

4.3 Constraints enforcement

This section presents a possible use case for the NamespaceOffloading resource. Assuming an initial situation like the one explained in the Figure 4.2, the admin wants to impose the following constraints:

- Pods can only be scheduled remotely.
- Remote namespaces must have the same name as the local one.
- Only clusters with *strict* security policies can be selected (in this case, only *cluster-2*).

The necessary NamespaceOffloading resource would be the following:

```
apiVersion: offloading.liqo.io/v1alpha1
kind: NamespaceOffloading
metadata:
  name: offloading
  namespace: ns-test
spec:
  namespaceMappingStrategy: EnforceSameName
  podOffloadingStrategy: Remote
```

```
9   clusterSelector:
10    nodeSelectorTerms:
11    - matchExpressions:
12      - key: security-policy
13        operator: In
14        values:
15        - strict
```

Once this resource has been created, the Liqo controller manager will replicate the local namespace on the selected clusters following the specified constraints. The figure 4.3 shows the *cluster-2* as the only selected cluster. The NamespaceOffloading resource is not just a way to specify the offloading constraints, but it is the starting point for the creation of the multi-cluster topology. Chapter 6 will explain how this resource is processed and which solution was chosen to allow the namespace replication with minimum privileges.

The solution presented in chapter 3 has a great advantage: if the admin does not want to specify any offloading constraints for his solution, he can simply label the namespace. With this new approach, it seems necessary to define cluster labels at installation time and to define a specific NamespaceOffloading CR. Indeed, this is not the case: if an admin wants to create deployment topologies that include all available clusters without additional constraints, he can just set the enabling label on the namespace as in the previous solution. How is this possible? All virtual nodes expose the label `liqo.io/type = virtual-node` by default. What Liqo does under the hood when users label the namespace is to create a standard

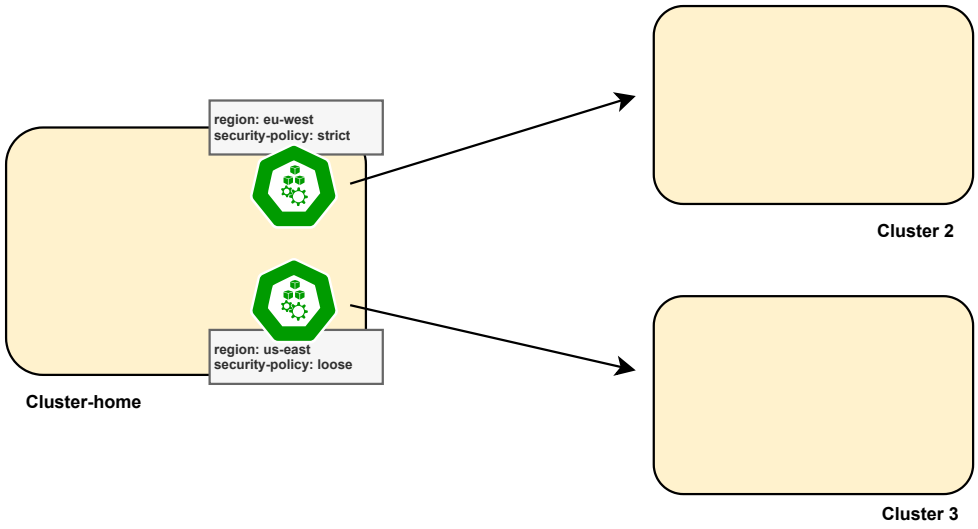

Figure 4.2: Use case: starting scenario.

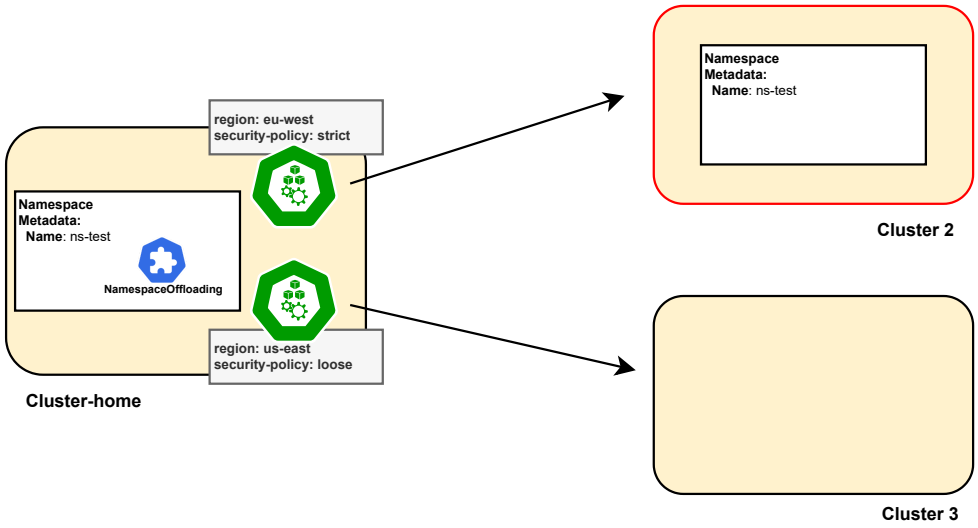

Figure 4.3: Use case: remote namespace creation.

NamespaceOffloading resource with all the fields set to the default values seen before:

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

Then both approaches create a NamespaceOffloading to specify constraints and start the replication. This is the first major difference from the existing implementation, where this resource did not exist.

Chapter 5

The privileges problem

One of the core problems of the already implemented solution was the full privileges requirement on remote clusters. Each virtual kubelet must have privileges to create remote namespaces and replicate local resources. Another problem previously highlighted is that the virtual kubelet has to perform too many tasks, risking having a poorly performing and not so intuitive approach.

The idea is to create a solution detached from the virtual kubelet that allows creating remote namespaces with a limited number of privileges, leaving the virtual kubelet only the reflection task. Some opensource solutions face the problem of privileges reduction but with a different perspective. These approaches address the problem of multi-tenancy in a single cluster. Since the topic of multi-cluster sharing is still quite innovative, we have decided to revisit these single-cluster approaches in a multi-cluster environment. The common philosophy behind the considered projects is the following:

Kubernetes is designed as a single-tenant platform, which makes it hard for cluster admins to host multiple tenants in a single Kubernetes cluster. However, sharing a cluster has many advantages, e.g. more efficient resource utilization, less admin/-configuration effort, or easier sharing of cluster-internal resources among different tenants. [14]

Liqo's purpose is exactly the same, but instead of having multi-tenancy in a single cluster, it wants to provide multi-tenancy in a multi-cluster environment. The solutions considered are mainly two and follow quite different approaches. The following sections will show the two solutions with their pros and cons.

5.1 Capsule solution

5.1.1 Basic idea

Capsule helps to design a multi-tenancy and policy-based environment in a Kubernetes cluster. The project has been realized as a micro-services-based ecosystem with the minimalist approach, leveraging only upstream Kubernetes. Kubernetes introduces the namespace object to create logical partitions of the cluster as isolated slices. The namespace abstraction does not allow sharing resources among namespaces belonging to the same tenant. This is a pretty strong limitation. Cluster admins tend to provision a dedicated cluster for each team or department to overcome this problem. As an organization grows, the number of clusters to keep aligned becomes an operational nightmare, described as the well-known phenomenon of the *clusters sprawl*. Capsule chooses a different approach. In a single cluster, the Capsule Controller aggregates multiple namespaces in a lightweight abstraction called **Tenant**.

5.1.2 Architecture

A Tenant, in the Capsule idea, is basically a grouping of Kubernetes Namespaces. Users are free to instantiate their namespaces and share all the resources inside each Tenant, while the Capsule Policy Engine keeps the different Tenants isolated from each other. Objects like ResourceQuota, LimitRanges, RBAC, network and security policies are defined at the Tenant level and are automatically inherited by all the namespaces in the Tenant. Then users are free to manage their Tenants autonomously, without the intervention of the cluster administrator.

Each Tenant comes with a delegated user or group of users acting as the Tenant admin. The Tenant admin is also called *Tenant owner* in Capsule jargon. The Tenant owner can assign different levels of permission and authorizations to other users inside a Tenant. The diagram 5.1 shows how the Tenant architecture works: The Capsule controller creates and manages the various Tenants. When a new Tenant is instantiated, all the policies defined with it are propagated to all its namespace. Capsule Policy Engine keeps the different Tenants isolated from each other. The cluster admin, can enforce network traffic isolation between different Tenants while leaving the Tenant owner the freedom to set isolation between namespaces in the same Tenant or even between pods in the same namespace. The admin can also dedicate a pool of worker nodes to a specific tenant to isolate its applications from other noisy neighbors.

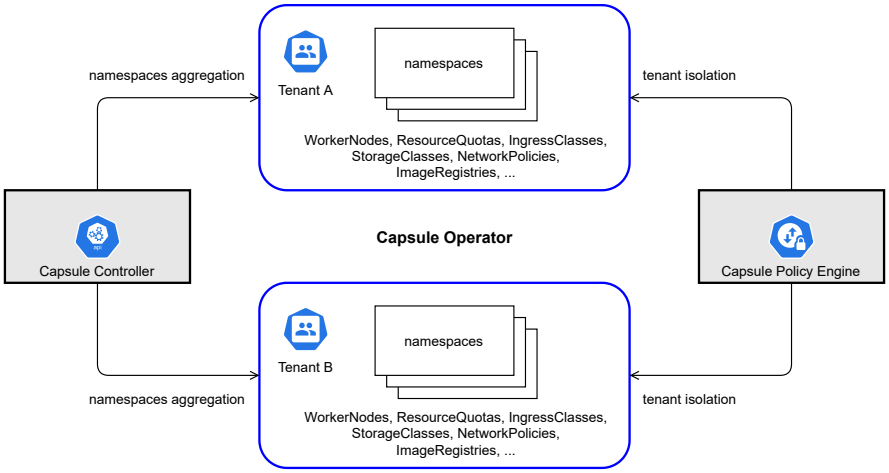

Figure 5.1: Capsule architecture. [15]

5.1.3 Pros and Cons

The Capsule solution is very lightweight and has many advantages:

- **Semplicty**: it provides a native Kubernetes experience without introducing additional management layers or plugins. Apart from the Tenant resource, all others are plain k8s resources.
- **Self-Service**: it leaves developers the freedom to self-provision their cluster resources according to the assigned boundaries.
- **Authentication agnostic**: it does not care about the authentication strategy used in the cluster. All the Kubernetes authentication methods are supported.
- **Governance**: it exploits k8s Admission Controllers to enforce the industry security best practices and meet legal requirements.

Although the limits imposed by Capsule are very effective, there may be some problems depending on the type of solution you are looking for:

- The Capsule framework does not support limits on the custom resources generation.
- This approach misses some powerful abstraction that other solutions provide.
- Tenant owner could be only a User or a Group, not a ServiceAccount.

5.2 Kiosk solution

5.2.1 Basic idea

Many Kubernetes distributions provide their own multi-tenancy solution, but there is no lightweight, pluggable and customizable solution that allows admins to easily add multi-tenancy capabilities to any standard Kubernetes cluster. This is where Kiosk comes in. The main idea of the Kiosk project is to use Kubernetes namespaces as isolated workspaces. Tenant applications inside this workspace can run isolated from each other. To minimize his intervention, the cluster admin is supposed to configure Kiosk, which then becomes a self-service system for provisioning Kubernetes namespaces for tenants.

5.2.2 Architecture

Kiosk architecture is more complex than the Capsule one. Many more custom resources are introduced to allow very powerful abstractions. The diagram 5.2 figures out the main actors involved as well as the most relevant Kubernetes resources and their relationships.

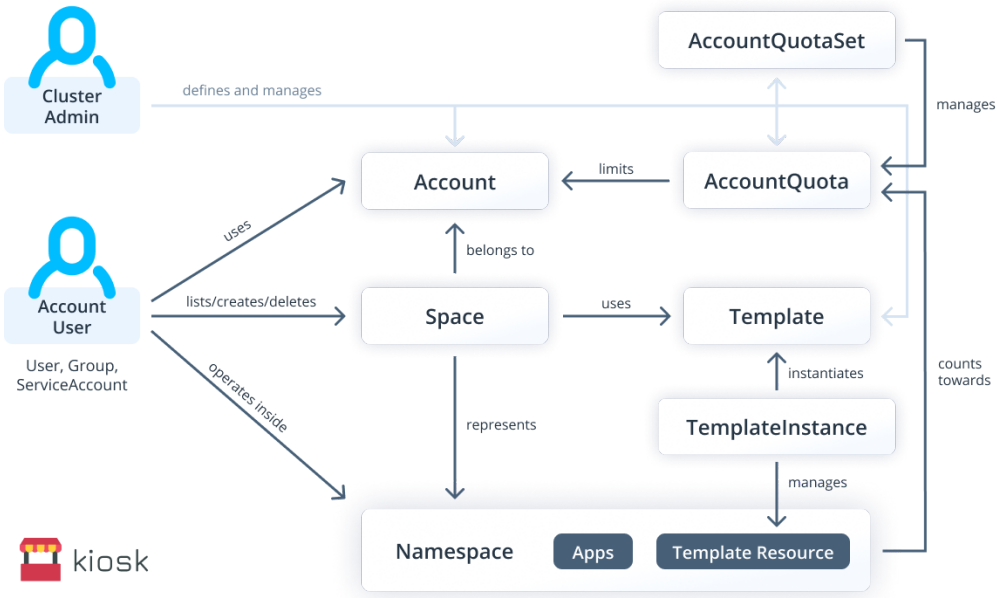

Figure 5.2: Kiosk architecture. [14]

A brief analysis of each component allow us to better understand the abstractions defined by this solution:

- **Cluster Admin**: The cluster admin configures Kiosk by creating and managing all `Accounts`, `AccountQuotas`, `AccountQuotaSets`, and `Templates` resources. He is also able to configure all `Spaces` owned by Accounts.
- **Account**: Every tenant is represented by an Account. Cluster admin declares and defines Accounts and assigns them to Account Users (Users, Groups, ServiceAccounts).
- **Account User**: An Account User performs API server requests while using a certain Account. Cluster admins can assign the same Account User to multiple Accounts. Account Users access Spaces that belong to their Accounts.
- **Space**: A Space is a non-persistent, virtual resource that represents exactly one Kubernetes namespace. These are the core Spaces characteristics:
  - Every Space can belong up to one Account which is the owner of this Space.
  - If a user can access the underlying Namespace, he can also access the associated Space.
  - Every User only watches the Spaces to which he has access. With regular namespaces, users can only list all namespaces or none.
  - Kiosk can deploy a set of resources inside the Space before the user accesses it. Configuring default Templates in the Account is possible to obtain this feature.
- **Namespace**: A Namespace is a standard k8s resource having a 1-to-1 correspondence to the resource Space, which is a Kiosk custom resource.
- **Template**: Templates are declared and defined by the cluster admin. Templates can initialize Spaces with a set of Kubernetes resources (defined as manifests or as part of a Helm chart). Templates can be defined using a different ClusterRole from the one used by the Account User, so they can be used to create resources that are not allowed to be created by actors of the Space, like isolation resources. Cluster admins can define default Templates within the Account. Kiosk automatically applies these templates to each Space that is created using the respective Account. An Account User can also generate other Templates that must be applied when creating a Space.
- **TemplateInstance**: When a Template is attached to a Space, Kiosk creates a TemplateInstance to keep track of it. A TemplateInstance exists for every Template associated with Space. A TemplateInstance contains information about the Template and about the parameters used to instantiate it. Additionally, TemplateInstances can be used to sync the created resources when the original Template is updated.

- **AccountQuota**: AccountQuotas define cluster-wide aggregated limits for Accounts. The resources of all Spaces that belong to an Account count towards the defined aggregated limits. Namespaces can be limited by multiple ResourceQuotas. In the same way, an Account can be limited by multiple AccountQuotas.

5.2.3 Pros and Cons

Here are some of the advantages offered by the Kiosk approach:

- **Pluggable**: easy to install into existing infrastructures and suitable for different use cases.
- **Self-Service**: guarantees automation and self-service for tenants.
- **Hierarchical security**: default configurations are available for different levels of tenant isolation.

Following the analysis of components seen in the previous section, the solution complexity becomes clear. This is the main drawback of the Kiosk approach. All these additional resources guarantee features that may not be necessary depending on the solution you are looking for. More precisely, the disadvantages can be summarized as follows:

- The Account User, having the admin scope on its spaces only, does not have permission to create Custom Resources Definitions (CRDs) as already seen in Capsule solution.
- Some roles must be configured manually by the cluster admin, there are no controllers like in the Capsule approach.
- There are a lot of roles involved and new resources, but plain Kubernetes already provides similar features with vanilla resources.

5.3 Chosen approach

After considering advantages and disadvantages of both solutions, we have chosen the Capsule approach. In addition to looking more production-ready, Capsule is an easy solution to integrate. It presents few additional custom resources and minimal overhead on existing clusters. Capsule uses Tenant abstraction allowing users to create only a certain number of namespaces in a single cluster. Liqo aims to extend this approach to a multi-cluster environment. The figure 5.3 sums up the Capsule logic. Any user who wants to create a certain number of namespaces will be the

owner of a Tenant. Once the limit imposed by the Tenant has been exceeded, it will no longer be possible to create remote namespaces. This mechanism allows the admin to partition the resources of a cluster well among the various users.

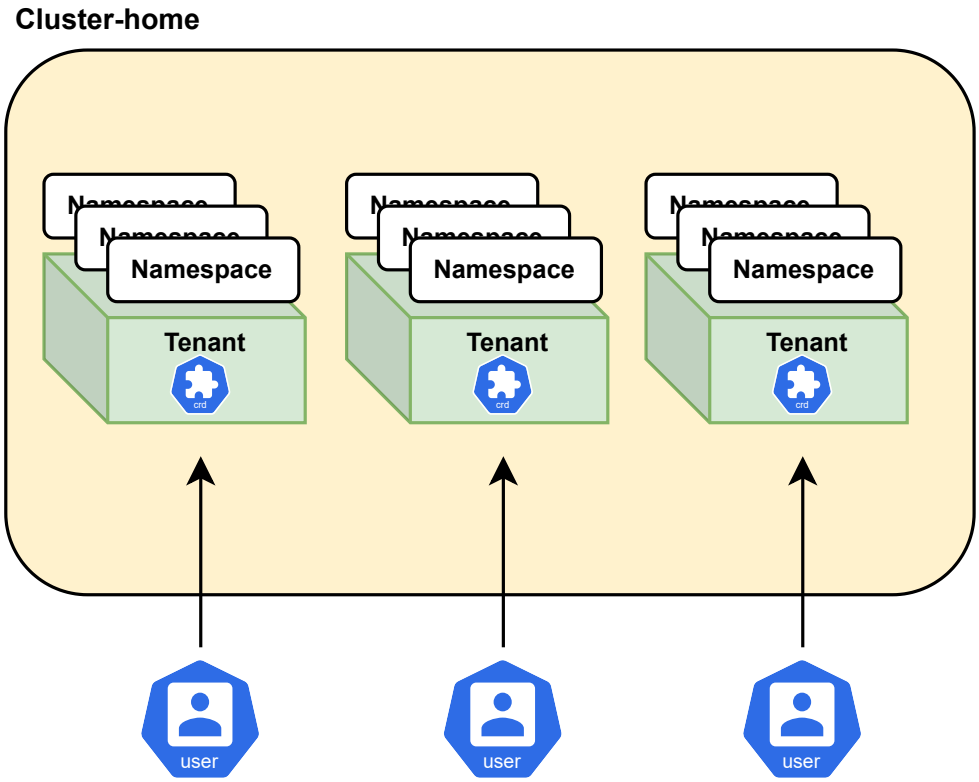

Figure 5.3: Capsule core logic.

Similarly, Liqo tries to exploit this mechanism, but with a slightly different approach. The figure 5.4 shows how each cluster can host namespace replicas from different clusters. A tenant is assigned to each cluster that wants to replicate at least one namespace remotely. The idea is to provide a set of remote namespaces in a controlled and secure manner. The figure 5.4 shows a remote cluster that has as many Tenant resources as there are remote clusters that wish to replicate a namespace.

The figure 5.5 represents a simple multi-cluster scenario where the *cluster-home* has two associated Tenants, one on each remote cluster. In this way, the *cluster-home* will be free to replicate namespaces on both remote clusters (*cluster-2* and *cluster-3*) with Tenant owner privileges.

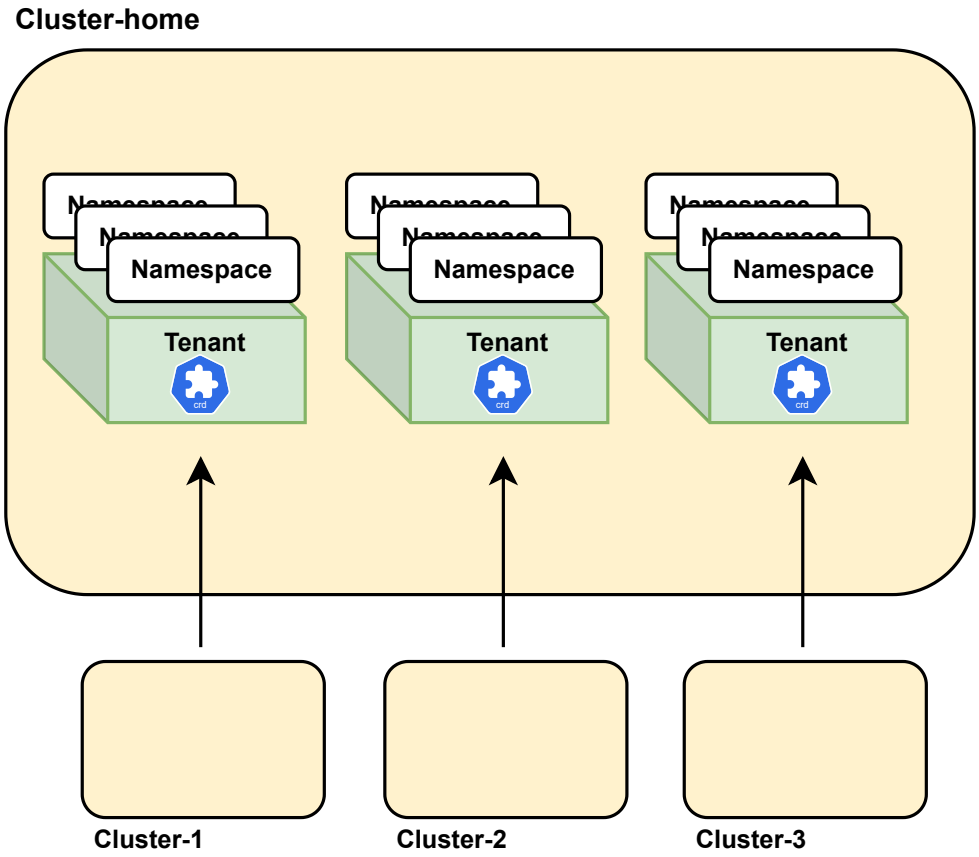

Figure 5.4: Liqo logic with the Capsule approach.

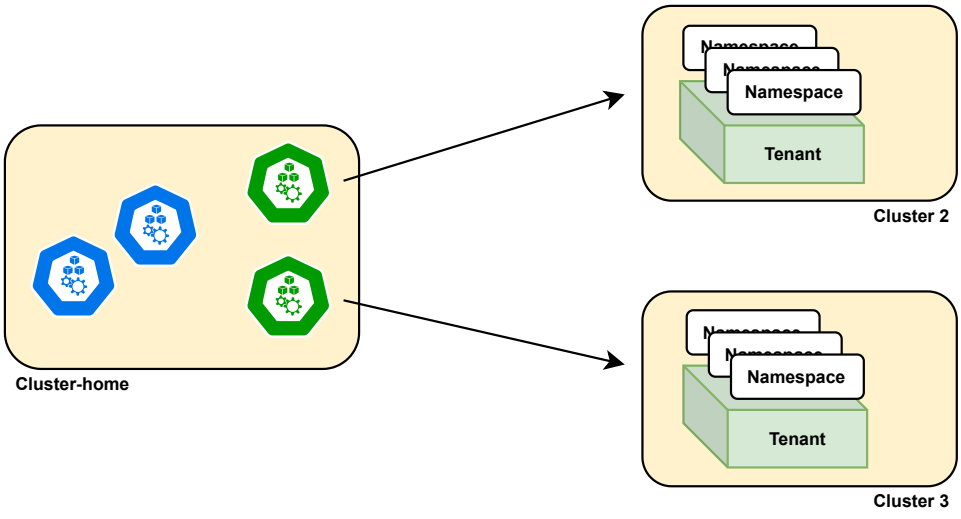

Figure 5.5: Simple multi-cluster scenario.

Chapter 6

Namespace replication model

The Liqo resource replication model includes a set of components that replicate each resource with an object-specific logic. The focus is on two core resources to be replicated. The first one discussed in this chapter is the namespace resource. This is the core element whose replication leads to the multi-cluster topology creation, allowing the cascade replication of all other resources. The second resource discussed in the next chapter is the deployment object. This k8s abstraction allows users to create different instances of an application while keeping them available and healthy. Liqo implements the replication of these two resources, exploiting the concepts of offloading constraints and privileges reduction seen previously.

6.1 Namespace replication details

The core replication feature is namespace replication. More precisely, namespace replication across multiple clusters implements the key mechanism to seamlessly extend a local cluster to a federated environment. This is the Liqo core idea: *create an extremely dynamic multi-cluster topology that evolves over time, always respecting the offloading constraints specified by the admin.*

In the Liqo model, the namespace replication operates on a standard namespace resource associated with a NamespaceOffloading CR. As already seen, this resource allows admins to specify remote clusters on which the namespace must be replicated and other offloading constraints, like the replicas name. The following sections show how the Liqo namespace model works under the hood, understanding which resources and controllers are involved.

6.1.1 Resources involved

The Liqo namespace replication involves instances of three kinds of resources:

1. **NamespaceOffloading**: this CR represents the initial trigger of the namespace replication process. On the one hand, the NamespaceOffloading spec describes the replication properties, such as the name of the replicated namespaces. On the other hand, the status collects information about the actual conditions of remote namespaces, e.g., if the replication succeeded or not.

2. **Tenant**: this resource allows external entities to create namespaces on demand in the local cluster. Tenant owner privileges allow users to create only a limited number of namespaces, safeguarding the local cluster security.

3. **NamespaceMap**: this CR contains the list of namespaces that must be replicated inside the remote cluster associated with the NamespaceMap. Every NamespaceMap is associated with a virtual node and so with a specific remote cluster. The resource spec stores the list of desired replications while the status keeps updated information about the remote namespace conditions. Each NamespaceMap is filled by several NamespaceOffloading resources targeting the same cluster. For example, if three different NamespaceOffloading require a replica on the same cluster, the corresponding NamespaceMap is filled with three new creation requests. It is worth noting that the NamespaceMap status represents the source of truth for information about the replicated namespaces. Every resource in the local cluster that wants to know something about the namespace replicas must consult the NamespaceMap status.

The figure 6.1 tries to explain more in detail how the NamespaceMap resource is manipulated. Every virtual node has an associated NamespaceMap. When a user requires the replication of the local namespace on that cluster, a Liqo controller fills the corresponding NamespaceMap. The example shows a local namespace called *ns-test* that is required to be replicated inside *cluster-2*. The user can express this constraint through the NamespaceOffloading resource created inside the namespace, as shown in the figure. The NamespaceOffloading object is like the one presented in the chapter 4 example, with the same replication constraints:

- Namespace replicas must have the same name as the local namespace.
- Pods deployed inside the local namespace can be scheduled only inside remote namespaces.
- Only clusters with a security policy of type *strict* must own a namespace replica.

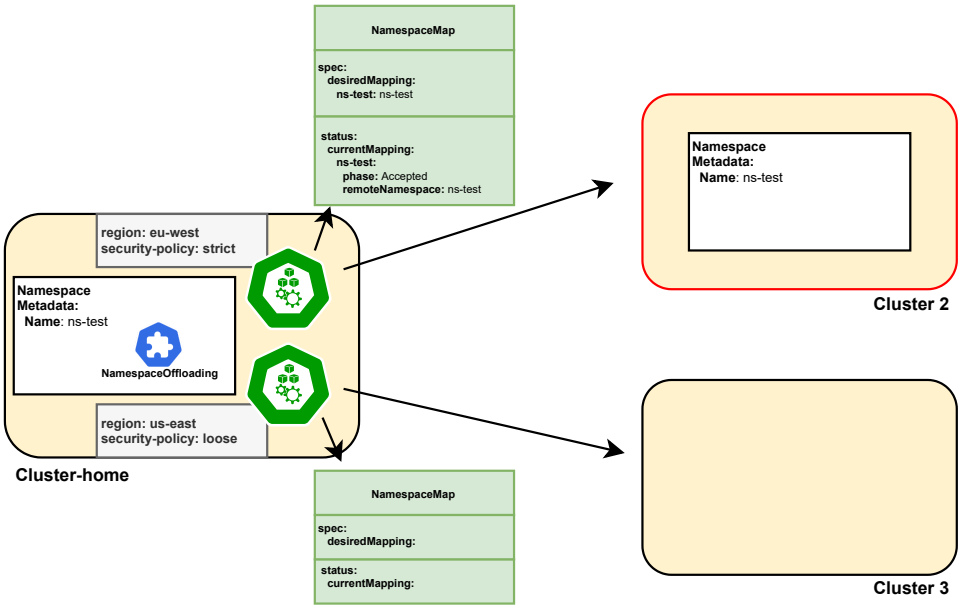

Figure 6.1: NamespaceMap resource manipulation.

When a new replication request is generated, it is inserted in the `desiredMapping` field of the corresponding NamespaceMap. According to the naming constraints previously quoted, the entry in the desiredMapping field should be like this: [`ns-test: ns-test`]. When the remote namespace is created, the corresponding NamespaceMap must be updated with the outcome. More precisely, a new entry will be created inside the `currentMapping` field. The following listing tries to clarify how the NamespaceMap resource is filled during the replication:

```
apiVersion: virtualkubelet.liqo.io/v1alpha1
kind: NamespaceMap
metadata:
  name: "defined at creation time"
  namespace: "defined at creation time"
spec:
  desiredMapping:
    ns-test: ns-test
status:
  currentMapping:
    ns-test:
      phase: Accepted
      remoteNamespace: ns-test
```

It is worth noting that the namespace replication is required only inside *cluster-2*, so the NamespaceMap associated with *cluster-1* is not involved, the resource remains unchanged as shown in the figure 6.1.

6.1.2 Replication workflow

Once we have seen which resources are involved, it is necessary to see which controllers are in charge of managing and manipulating them. The following controllers cover a key role in the replication process:

- The **NamespaceOffloading Controller** processes the NamespaceOffloading spec and fills the proper replication requests into NamespaceMap resources.
- The **NamespaceMap Controller** takes care of the remote namespaces' creation and updates their status in the NamespaceMap resources.
- The **OffloadingStatus Controller** updates the NamespaceOffloading status gathering information about namespace replicas from the NamespaceMap status.

More precisely, the figure 6.2 divides the replication process in five simple steps, highlighting the role of each controller:

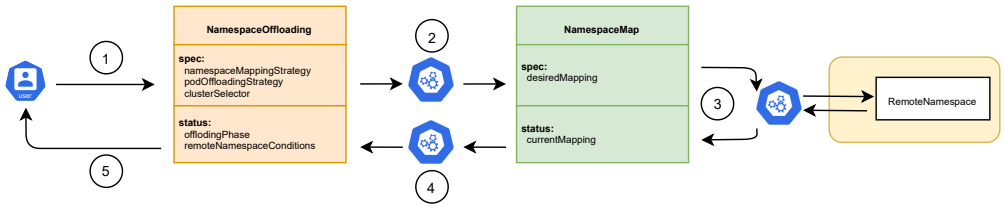

Figure 6.2: Namespace replication workflow. [16]

1. When the user creates a NamespaceOffloading object in a local namespace, the NamespaceOffloading controller processes the resource spec.
2. After having detected the virtual nodes compliant with the NamespaceOffloading selector, the NamespaceOffloading controller fills the NamespaceMap resources of the selected nodes. More precisely, the controller sets the replication requests in the desiredMapping field. This logic is recalled every time a new virtual node joins the topology to check if it is compliant with the requirements.

3. Once the NamespaceOffloading controller has filled NamespaceMaps with requests, the NamespaceMap controller should enforce the namespaces replication. This controller has the privileges of Tenant owner inside remote clusters, so it can create a certain number of namespace replicas. The operation outcome is saved in the currentMapping field of each NamespaceMap involved. The NamespaceMap Controller periodically checks if each entry in the desiredMapping field has an associated remote namespace; in case of absence, it immediately enforces a new namespace replica. Furthermore, the controller performs health probes on these namespaces. Whenever it detects a change in the namespaces state, it immediately updates the NamespaceMaps status.

4. The OffloadingStatus Controller is responsible for the NamespaceOffloading status reconciliation. It periodically checks the status of all NamespaceMaps in the clusters, and for each NamespaceOffloading object, it updates its status fields: remoteNamespaceConditions and offloadingPhase.

5. The NamespaceOffloading status provides the user with all the information about the replication process.

The figure 6.3 provides a concrete example of the replication mechanism. The steps represented are the same as described above. More precisely, the figure highlights which fields are manipulated by the different controllers. The constraints specified in the NamespaceOffloading require a namespace replica only inside the cluster with a strict security policy. Consequently, only the corresponding NamespaceMap is filled.

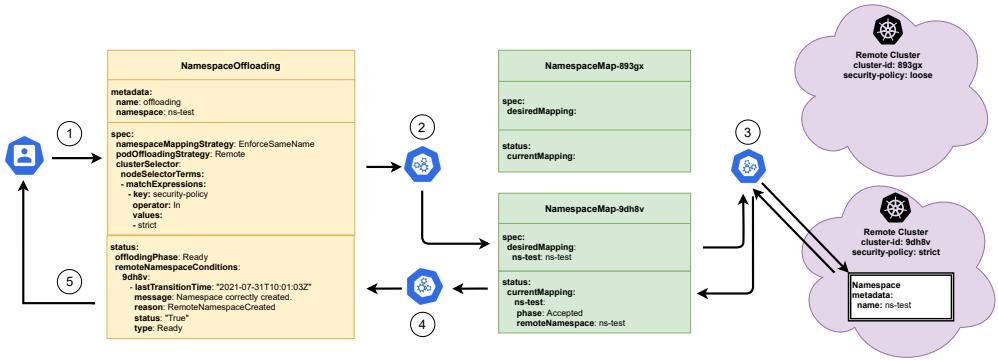

Figure 6.3: Namespace replication example. [16]

6.1.3 Deletion workflow

The NamespaceOfflaoding creation starts the replication logic. In the same way, the resource deletion terminates the replication, triggering the namespace replicas deletion. The following steps briefly summarize the mechanism:

1. When the user decides to delete the NamespaceOffloading resource, the termination of the replication process starts, and the OffloadingStatus controller updates the offloadingPhase field to "Terminating".

2. The NamespaceOffloading controller removes all the entries associated with that resource from NamespaceMap objects.

3. The NamespaceMap controller reacts to this event enforcing the deletion of all namespace replicas no longer required. In particular, when a remote namespace is deleted, the controller updates the NamespaceMaps status.

4. When the NamespaceMap controller removes an entry from the currentMapping field of one NamespaceMap resource, the OffloadingStatus controller deletes the remote conditions associated with that namespace in the NamespaceOffloading resource. Once all remote namespaces have been removed and, therefore, all associated remote namespace conditions, then the NamespaceOffloading resource is finally removed, and the deletion process is complete.

6.2 Multi-cluster deployments

We have seen how namespace replicas are generated, and we have seen how the naming constraints and the clusters selection are forced during replication. It remains to be seen how applications can be deployed within the new topology, what are the strengths and weaknesses of this approach and why a deployment replication mechanism was introduced.

The NamespaceOffloading gives the admin the possibility to choose where to schedule pods through the podOffloadingStrategy field. But how is this constraint imposed? When a namespace enables the Liqo features through the NamespaceOffloading resource, all pods scheduled within it are mutated by the Liqo webhook. As already seen in chapter 3, the Liqo webhook applies tolerations to pods so that they can be scheduled on virtual nodes and then remotely. However, this is no longer sufficient and, above all, not always necessary. The webhook must be able to mutate pods differently based on the policy specified by the admin.

6.2.1 The new Liqo webhook role

When the user schedules pods inside a Liqo enabled namespace, the webhook gets the associated NamespaceOffloading resource and processes it. More precisely, the webhook considers just two fields of the resource spec: the podOffloadingStrategy and the clusterSelector. According to these fields, the webhook forces some nodeSelectorTerms inside pods to make sure that the constraints specified are respected by the scheduler.

Virtual nodes expose a taint so that no pod can be scheduled on them without the proper toleration. All pods that are not involved in the offloading process offered by Liqo do not have to be scheduled on virtual nodes, so the webhook must also manage the addition of an appropriate toleration.

Considering a NamespaceOffloading with a fixed clusterSelector, the webhook forces on pods different nodeSelectorTerms and tolerations for the three possible podOffloadingStategy values. A simple clusterSelector field could be:

```
clusterSelector:
  nodeSelectorTerms:
  - matchExpressions:
    - key: liqo.io/region
      operator: In
      values:
      - us-west-1
```

All virtual nodes that expose the label `liqo.io/region=us-west-1` could be selected as a target to run pods. Considering now the three different strategies:

1. **LocalAndRemote**

   Pods could be scheduled both on virtual nodes that expose that region label and on all local nodes. So there is the necessity of two nodeSelectorTerms:

   - The first nodeSelectorTerm selects the virtual nodes that expose the right region label.
   - The second one selects all the local nodes.

```
nodeSelectorTerms:
  - matchExpressions:
    - key: liqo.io/region
      operator: In
```

```
        values:
        - us-west-1
    - matchExpressions:
      - key: liqo.io/type
        operator: NotIn
        values:
        - virtual-node
```

Pods could be scheduled only on nodes that match one of these two nodeSelectorTerms. With this strategy, the approach used is to increment the nodeSelectorTerms provided by the clusterSelector with an additional nodeSelectorTerm that allows pods to be also scheduled locally. Since pods are enabled to be scheduled even remotely, the webhook must add on them the toleration to the Liqo taint.

2. **Local**

   This is the simplest case: pods can be scheduled only locally as if Liqo was not present. Consequently, the webhook does not have to apply the virtual node toleration and to enforce the clusterSelector.

3. **Remote**

   Pods can be scheduled only on virtual nodes. In this case, a single nodeSelectorTerm with two matchExpressions is sufficient:

   - The first matchExpression selects the remote cluster with that region label.
   - The second one makes sure that a local node is not selected, in the unusual case that a local node exposes the same labels of a virtual one.

```
nodeSelectorTerms:
  - matchExpressions:
    - key: liqo.io/region
      operator: In
      values:
      - us-west-1
    - key: liqo.io/type
      operator: In
      values:
      - virtual-node
```

With this strategy, the approach used is to provide every nodeSelectorTerm of the ClusterSelector with an additional matchExpression which prevents pods from being scheduled on local nodes. Since pods must be scheduled remotely, the webhook must add toleration to the Liqo taint.

The webhook also takes into account each nodeSelectorTerm added by the user at creation time. However, these user constraints must always be more restrictive than those specified by the admin, otherwise, in case of conflicts, pods would remain pending.

6.2.2 Different deployment scenarios

The positive aspects of this solution are its simplicity and transparency. Once an offloading policy has been chosen, pods scheduled in that namespace will only go to selected clusters. This solution can be very convenient when the user is interested in exploiting available resources without having a particular replication pattern. To better understand, we can think of an example. The figure 6.4 shows a deployment that requires 20 replicas of the same pod. These replicas can be scattered across clusters, in this example 5 on *cluster-2*, 10 on *cluster-3*, and 5 locally. *Cluster-4* cannot host this type of application, so the NamespaceOffloading resource has not created a replica namespace inside it. Within the local namespace, there is also an horizontal pod autoscaler (HPA) that allows the deployment to scale effectively in case of load peaks. The new pods can be created within the various clusters, taking advantage of all the resources made available by peerings.

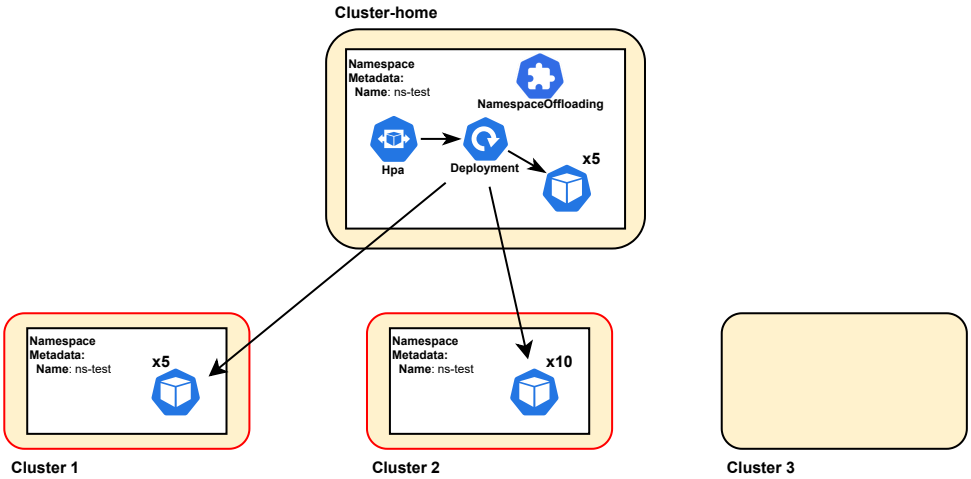

Figure 6.4: Single deployment scattered across multiple clusters.

In other cases, it may be necessary to create replicas of the same deployment on

more than one cluster. Figure 6.5 schematically shows 50 clusters each one with an identical deployment inside. The current solution allows users to create this type of pattern, however they should manually create 50 deployments with the right affinity to select the single clusters. Creating hundreds of deployments manually and managing them in scenarios with many clusters does not seem a viable solution. For this reason, Liqo introduces a new replication logic for deployment, making the management of multi-cluster applications straightforward and automatic.

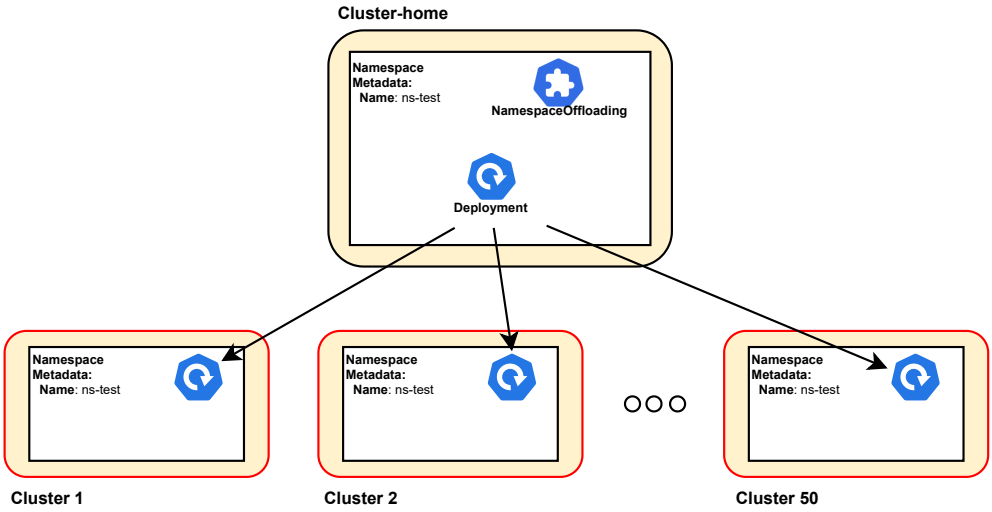

Figure 6.5: Introduction to the deployment replication concept.

Chapter 7

Deployment replication model

The previous chapter introduced several ways of deploying applications in a multi-cluster environment. A controlled and automatic replication is not straightforward to obtain without an appropriate abstraction. The user should manually configure offloading and monitor applications in case of any failures. Liqo offers users a feature able to replicating and monitoring deployments on a clusters subset without worrying about their management but considering them as a single resource.

7.1 Deployment replication details

In the Liqo model, the deployment replication takes place through a new custom resource, called textbfLiqoDeployment. Before creating a LiqoDeployment resource, it is necessary to have replicated the local namespace through an appropriate NamespaceOffloading object. Once the multi-cluster topology has been created, it can therefore be enriched with automatically replicated applications. Furthermore, Liqo already offers k8s service replication mechanisms; by creating services within a local enabled namespace, they will be replicated on all available namespace replicas, allowing total access to the applications from all clusters. So by adding this last feature, it will be possible to deploy an entire application running and performing in a multi-cluster environment.

It is worth noting that Liqo will not exactly replicate a deployment to remote clusters, but it will simply duplicate the underlying ReplicaSet. A ReplicaSet is sufficient for remote clusters to manage the offloaded pods autonomously and reliably. For the sake of clarity, the following sections refer to the generic replication of a deployment, neglecting this implementation detail. Even the figures will omit this aspect to make the reasoning more linear and straightforward.

7.1.1 Resources involved

The Liqo deployment replication involves an instance of just one resource: the LiqoDeployment. This resource allows users to specify in its spec the replication constraints to be imposed, while its status provides a summary of the deployment replicas conditions. Let's see in detail the various fields by looking at a resource example:

```
apiVersion: offloading.liqo.io/v1alpha1
kind: LiqoDeployment
metadata:
  name: nginx-replicator
spec:
  template:
    metadata:
      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
  generationLabels:
    - liqo.io/provider
    - liqo.io/region
  selectedClusters:
    nodeSelectorTerms:
      - matchExpressions:
          - key: liqo.io/region
            operator: NotIn
            values:
```

```
                - B
status:
  currentDeployment:
    nginx-replicator-4dsfs:
      generationLabelsValues:
        liqo.io/provider: A
        liqo.io/region: A
      deploymentLastCondition:
        lastTransitionTime: ...
        lastUpdateTime: ...
        message: ...
        reason: MinimumReplicasAvailable
        status: "True"
        type: Available
      generationLabelsValues:
        liqo.io/provider: B
        liqo.io/region: A
      deploymentLastCondition:
        lastTransitionTime: ...
        lastUpdateTime: ...
        message: ...
        reason: MinimumReplicasAvailable
        status: "True"
        type: Available
```

The resource offers three core fields in its spec:

1. `template`: It allows users to specify the deployment template to replicate, with all the attached information such as the configuration of the pod to replicate, the number of pod replicas, and the deployment strategy to replace existing pods with new ones.

2. `generationLabels`: They allow users to specify the granularity to replicate deployments. In this case, we want to create a deployment replica for each combination of values that the two labels `liqo.io/region` and `liqo.io/provider` assume. Users can specify any number of labels for generating deployments. In particular, if no label is specified, the replication has a node granularity. This labels vector provides an extremely dynamic feature: users can update the labels at will by changing the replication granularity. It is important to note that this change does not lead to an interruption of the service offered by the replicated application.

3. `selectedCluster`: This field allows users to exclude clusters from the replication process. It is an additional filter with respect to the one specified in the NamespaceOffloading resource. The selection takes place at two different levels: the NamespaceOffloading imposes constraints on all applications that will be deployed in the local namespace, while the LiqoDeployment imposes replication constraints only on the deployment to which it refers. Another LiqoDeployment resource generated in the same namespace allows the user to specify completely different constraints for the new application. Even the filtered clusters can be updated during replication, ensuring a great dynamism of the solution.

The resource status is basically a list of deployment replicas. For each replica, the name and two main information are available:

- `generationLabelsValues`: Indicates the combination of labels to which the deployment corresponds. This field allows users to immediately understand which deployment they are referring to. When the number of replicas becomes large, this information becomes essential to keep track of which deployments already exist in the cluster.
- `deploymentLastCondition`: Reports the last condition that the replicated deployment provides. This condition is kept up to date with deployment changes. In case the user wants to access more information about the deployment, he can directly retrieve them by getting the deployment thanks to the name provided in the status.

The resource status can be seen as a summary of each deployment replica, with a direct pointer (the name) for immediate access to more detailed information. The name of the deployment replicas is not straightforward: it is composed of the LiqoDeployment resource name plus a unique suffix in the cluster. The figure 7.1 shows an example of deployment replication; the LiqoDeployment resource considered above derives directly from this situation. Let's consider a scenario with four remote clusters: each one exposes a label of type `liqo.io/region` and `liqo.io/provider`. The generationLabels specified in the LiqoDeployment resource denote a replication with region and provider granularity. Consequently, the four available combinations of values would be:

- `liqo.io/region=A`, `liqo.io/provider=A`
- `liqo.io/region=A`, `liqo.io/provider=B`
- `liqo.io/region=B`, `liqo.io/provider=A`
- `liqo.io/region=B`, `liqo.io/provider=B`

However, the selectedCluster field excludes remote clusters with `liqo.io/region=B`, allowing replication only on clusters that expose `liqo.io/region=A`. Before starting the deployment replication, it is necessary to define a NamespaceOffloading resource. In this case, the NamespaceOffloading resource allows the creation of namespaces on all available remote clusters.

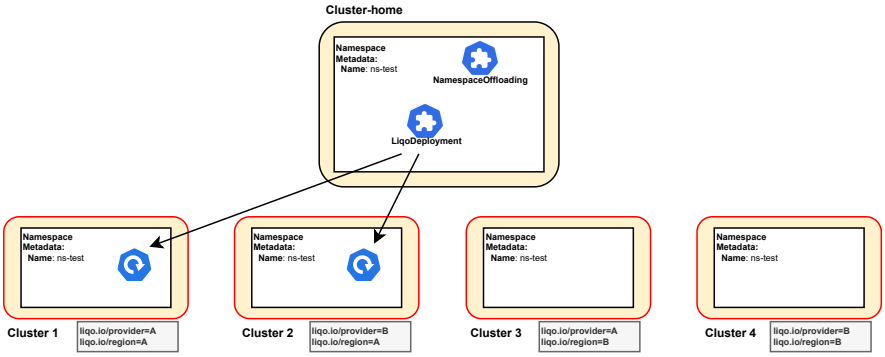

Figure 7.1: Replication with region and provider granularity.

It is worth noting that there is no 1:1 correspondence between a deployment replica and a remote cluster. If multiple remote clusters expose the same label combination, a replica will be created for that combination, not for individual clusters. Pods underlying the replicated deployment could therefore be scheduled on one cluster rather than another. Figure 7.2 represents *cluster-3* and *cluster-4* with the same labels, region and provider, exposed. Replicating deployments with that granularity and assuming that each deployment generates 5 pods, we could have a situation like the one shown in the figure, where 3 pods are scheduled on *cluster-3* and 2 on *cluster-4*. The deployment replica is therefore associated with 2 clusters that expose the same properties.

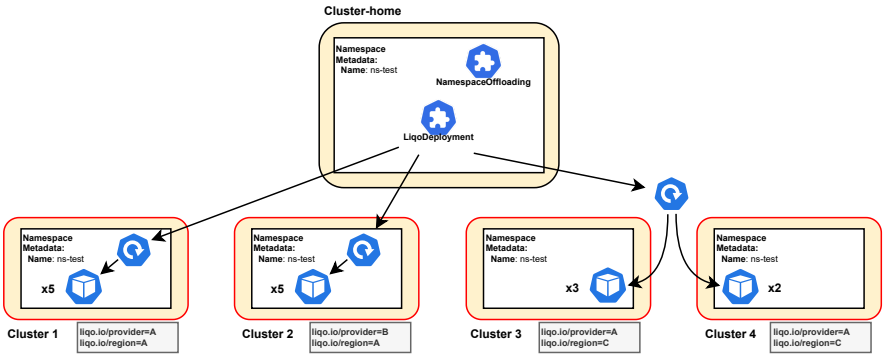

Figure 7.2: Deployment replica associated with two clusters.

7.1.2 Replication workflow

The replication logic begins when a LiqoDeployment object is created inside an already replicated local namespace. Liqo has a dedicated controller to manage this resource called LiqoDeployment controller. The controller performs five simple steps that the figure 7.3 briefly summarizes:

1. It reads the cluster filters imposed by the NamespaceOffloading and those possibly present in the LiqoDeployment resource.
2. It excludes from the analysis the virtual nodes that must be filtered.
3. It considers the labels exposed by remaining virtual nodes and calculates the possible combinations derived from the generationLabels field.
4. It creates deployment replicas corresponding to computed combinations.
5. Once the deployment replicas have been created, it updates the LiqoDeployment status to provide the user with necessary information.

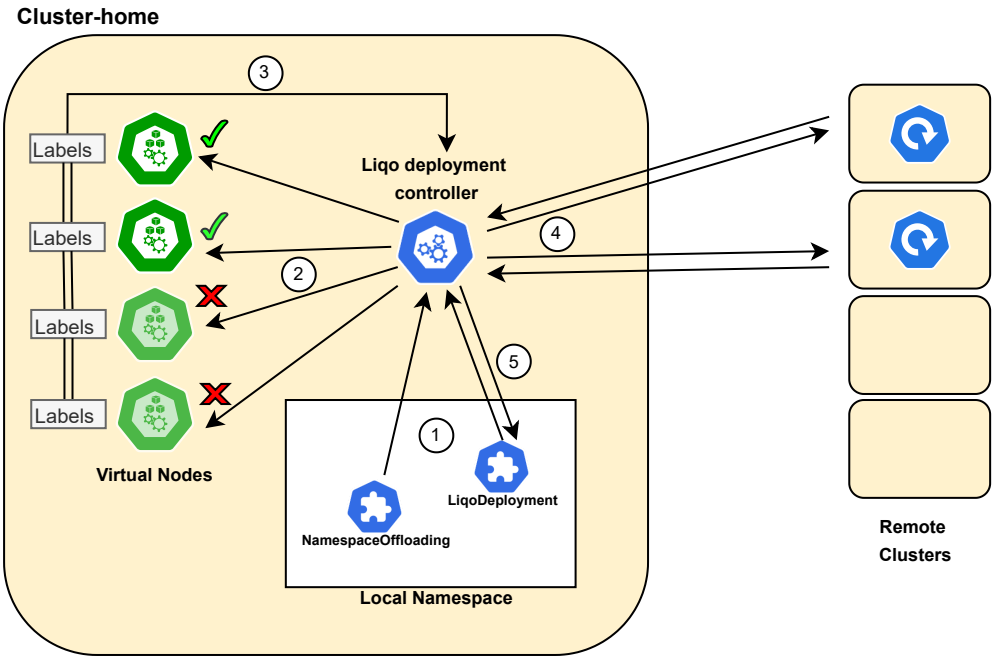

Figure 7.3: Deployment replication workflow.

The deployment replication logic is extremely dynamic, like the namespace one. When a new virtual node joins the architecture, the NamespaceOffloading

controller checks if it is necessary to create a new namespace replica according to the constraints specified by the admin. Similarly, the LiqoDeployment controller checks whether a new combination of values is available and creates a new deployment replica if necessary. In case of a virtual node unpeering, the logic also checks that it is not necessary to delete any deployment replicas.

7.1.3 Deletion workflow

The LiqoDeployment resource has the ownership of all deployment replicas created. Consequently, it will be sufficient to delete the LiqoDeployment resource to drop all replicas in cascade. If a deployment belonging to a LiqoDeployment is deleted by the user, the controller will take care of promptly regenerating the replica. The only way to terminate the replication process is to notify the controller by deleting the LiqoDeployment resource.

7.2 Full multi-cluster application deployment

This section presents a final example that brings together all the concepts seen above and shows the steps required to deploy a micro-services based application on multiple clusters. The first step to create the multi-cluster topology is to replicate a local namespace through the NamespaceOffloading resource. This resource allows to specify which clusters to select and some constraints such as the name of the namespace replicas and the pods scheduling policy. The example 7.4 suppose to replicate a namespace on all remote clusters keeping the same name and allowing the pods scheduling both locally and remotely.

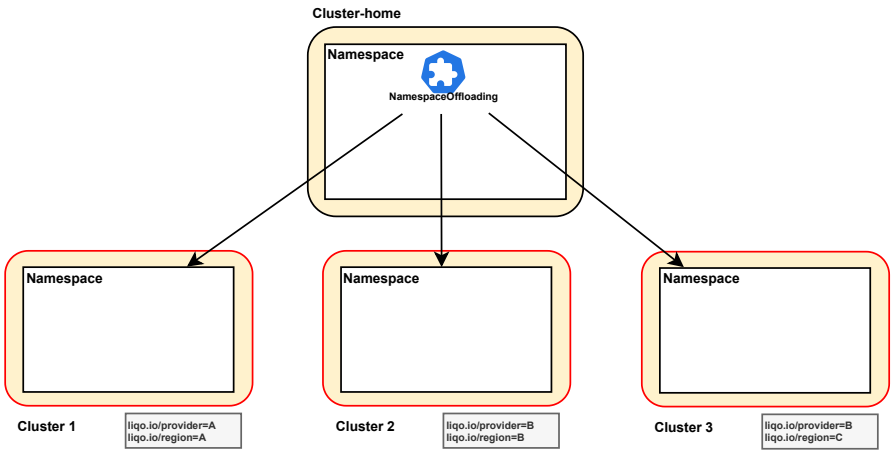

Figure 7.4: 1° Step: Namespace replication phase.

Remote clusters expose the labels `liqo.io/provider` and `liqo.io/region`. The application in this example consists of 2 deployments:

- The first must be replicated with region granularity, so in this case for each cluster, since they all belong to different regions. Each deployment replica generates 3 pods (green LiqoDeployment in the figure 7.5).

- The second must be replicated with provider granularity, so in this case, the *cluster-2* and the *cluster-3* will have only one shared replica, as they expose the same label provider. Each deployment replica generates 4 pods (orange LiqoDeployment in the figure 7.5).

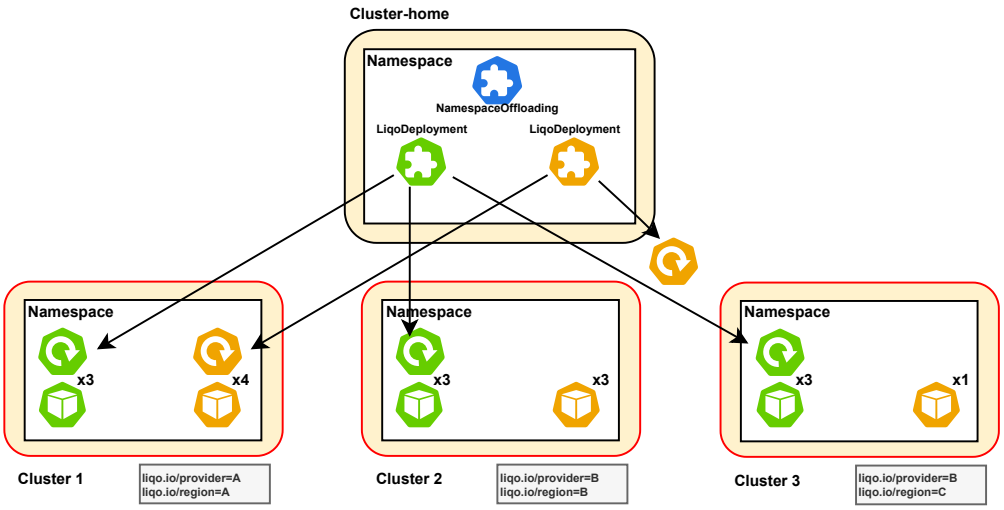

Figure 7.5: 2° Step: Deployment replication phase.

Finally, to make the application accessible from each cluster, it will be sufficient to create a k8s service in the local namespace to automatically replicate it within all the namespace replicas (figure 7.6).

Thanks to the use of these three simple resources (NamespaceOffloading, LiqoDeployment, and plain Kubernetes Service), it was possible to obtain a performing and reliable application able to withstand failures that also involve an entire cluster. This is the power of Liqo, a great ability to scale and withstand major failures using shared resources that would otherwise be wasted.

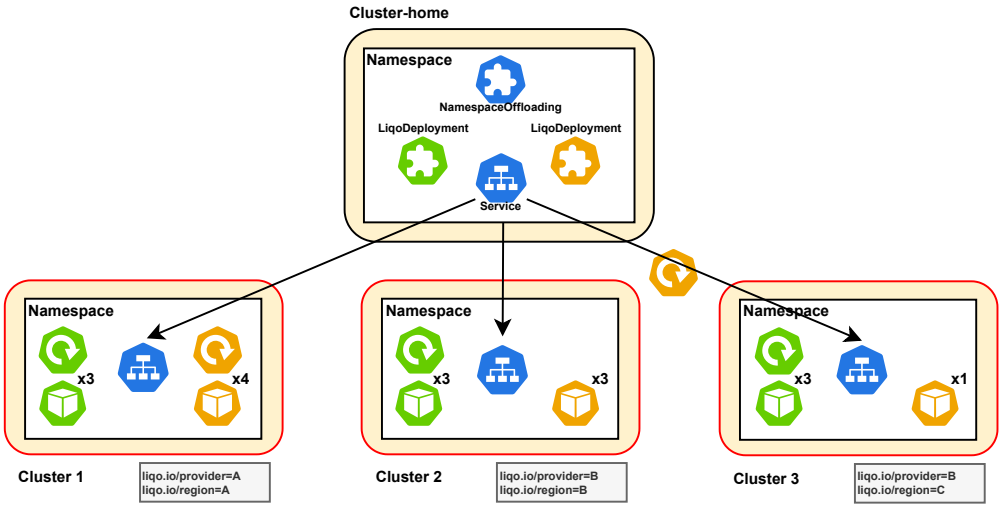

Figure 7.6: 3° Step: Service replication phase.

Chapter 8

Evaluation

The proposed work requires validation in terms of solution stability and scalability in high-load scenarios. The analysis separately considers the two replication approaches seen: Namespace replication and Deployment replication. Both replication processes are implemented by seeking a stable and reliable approach. Liqo controllers carry out periodic reconciliations to check that the resource replicas are healthy and, in case of problems, recreate them promptly. However, this type of choice based on stability can lead to a loss in terms of performance. In this regard, we have implemented some micro-benchmarks to validate the most critical logic flows from the performance point of view and to identify possible latencies and any improvements. All micro-benchmarks are performed on a local machine equipped with the following resources: 16 GB of RAM, 256 GB of PCI-E SSD, 1 TB of disk storage, and 6 CPU cores.

8.1 Namespace replication benchmarking

We have chosen the Kubebuilder test environment to carry out analyzes concerning the namespace replication. This environment allows to easily obtain large amounts of clusters, supporting the creation of a multi-cluster topology in few simple steps. Taking up the schema concerning the namespace replication workflow of chapter 6, we can search for possible performance issues. More precisely, referring to the figure 8.1 we can identify 3 possible aspects to test:

1. The time taken to create remote namespaces and update the NamespaceMaps status as the number of remote clusters involved increases. The logic to be tested concerns step 3 of the figure and involves only the NamespaceMap controller.

2. The scalability of namespace replication on a single remote cluster as the

number of replicas requested increases. The logic to be tested is the same as in the previous case, however the focus is shifted to replication within a single remote cluster.

3. The time taken to fill NamespaceMap resources with replication requests as the number of NamespaceMaps involved increases. The logic considered concerns step 2 of the figure and involves only the NamespaceOffloading controller.

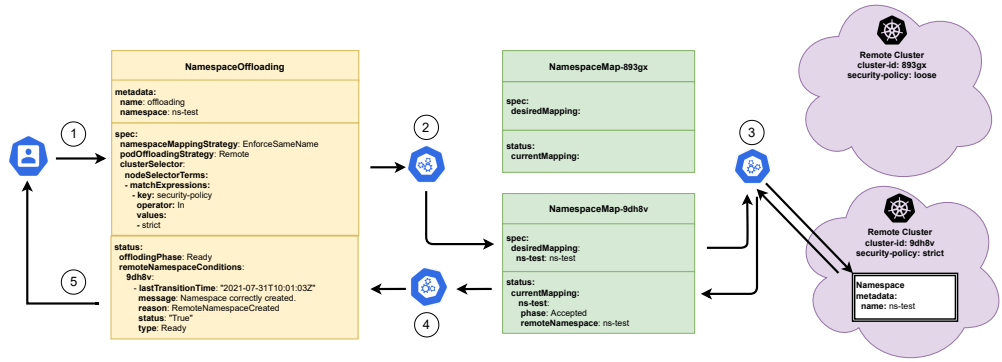

Figure 8.1: Namespace replication example.

The logic of updating the NamespaceOffloading resources starting from the NamespaceMaps (described in step 4) is not considered a critical feature from the point of view of scalability and consequently is not taken into consideration in this analysis. Points 1 and 5 instead, as already mentioned in chapter 6 are related to manual actions carried out by users and therefore are not subject to testing. The following sections will analyze in detail each of the three cases highlighted above.

8.1.1 Replication scalability on multiple clusters

This benchmark is intended to analyze the performance of the NamespaceMap controller under high-load situations. More precisely, referring to the figure 8.1, we want to analyze how long the controller takes to fulfill creation requests and to update NamespaceMap resources with the correct status. To carry out the test, we created several remote clusters: as we can see in the figure 8.2 we started with 5 clusters up to a maximum of 50. Each remote cluster corresponds to a NamespaceMap resource within which only one replication request has been inserted. The goal is, therefore, to create a namespace replica on each remote cluster and see how the time required changes as the number of remote clusters involved increases.

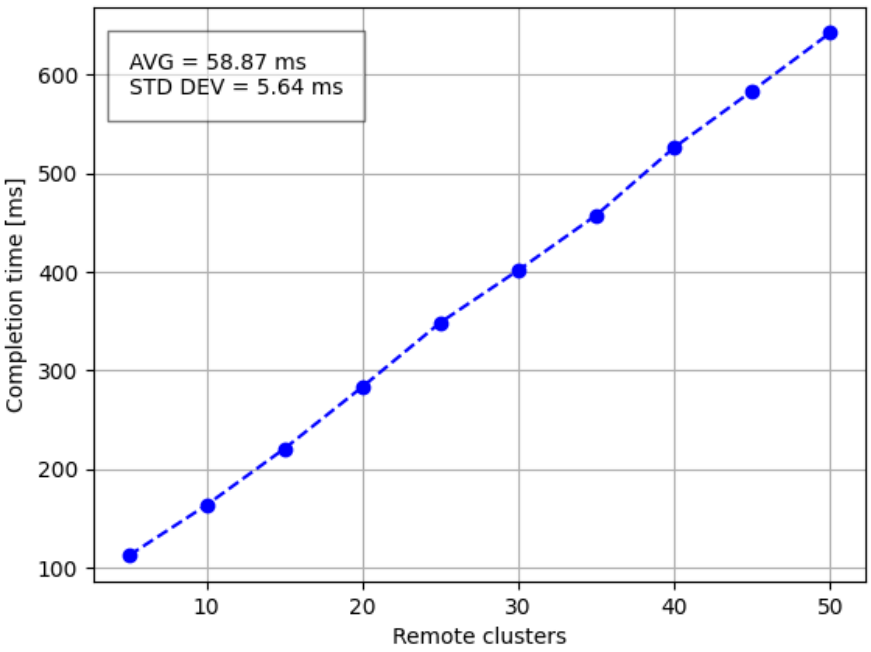

Figure 8.2: The namespace replicas creation time plus the NamespaceMaps update time scale linearly as the number of remote clusters increases.

For every 5 clusters added, there is an almost constant increase in time; this denotes a linear trend due to the fact that each NamespaceMap triggers a different controller reconciliation, and each reconciliation is completely disconnected from all others. Consequently, adding a constant clusters number will always introduce the same temporal delay.

It is worth noting that the time represented on the ordinate axis is the namespace replicas creation time plus the update time of NamespaceMaps status. The *AVG* symbol stands for the average of time differences between one measurement and another, while *STD DEV* stands for standard deviation.

8.1.2 Replication scalability on a single cluster

The previous benchmark examines the replication process with multiple remote clusters involved. In this scenario, however, we consider a single remote cluster. The NamespaceMap associated with the cluster is filled with an increasing number of replication requests, starting from 10 up to 100. As in figure 8.1 only one NamespaceMap is filled, but instead of having a single entry in the desiredMapping

field, we have inserted up to 100 requests. The time under analysis is always the replicas creation time plus the NamespaceMap status update time.

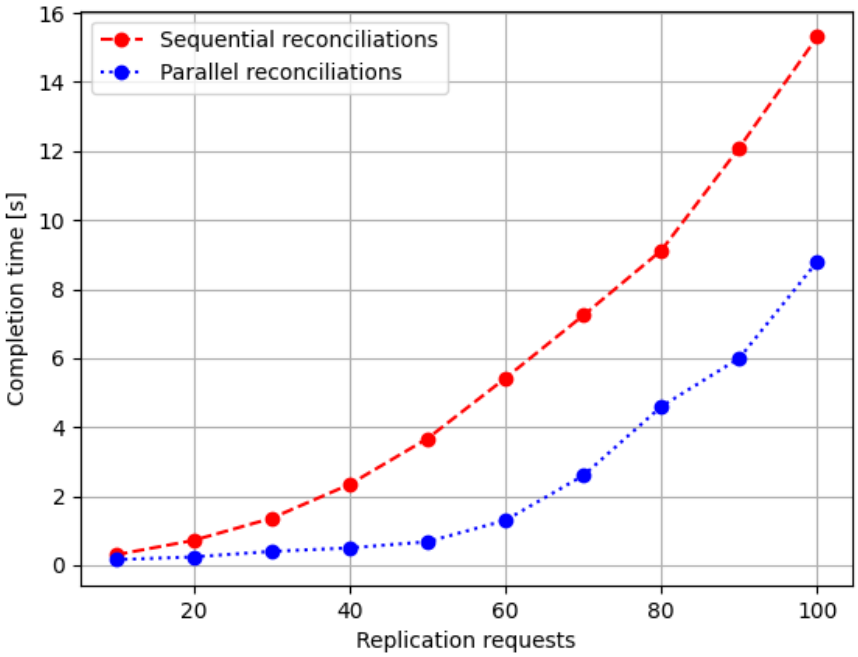

Figure 8.3: The namespace replicas creation time plus the NamespaceMaps update time scale exponentially as the replication requests number increases.

The dashed line in figure 8.3 shows how increasing the replication requests number does not result in linear growth over time. In this scenario, in fact, with each new request added, the NamespaceMap controller must check that all namespace replicas already created exist before creating the new one. This check consists of an increasing number of API server calls proportional to the number of replicas already present. In the previous analysis, the reconciliations concerned different resources, and the time required by each was about the same. In this case, however, each subsequent reconciliation on the same resource takes longer than the previous one, leading to an exponential trend over time. The chosen implementation aims to achieve good stability and reliability of the solution at the expense of performance. We have tried to use a multi-thread approach to improve this latency. More precisely, the dotted line in figure 8.3 represents the time taken by 5 workers to perform necessary reconciliations. As we can see, the time taken is much shorter than before, despite the fact that the exponential trend persists. A multi-thread approach may be one of the possible reliefs but not the only one: another solution

could be to detach the enforcement logic from the creation one, enforcing the replicas periodically and not at every reconciliation.

8.1.3 Scalability of requests addition process

This benchmark aims to analyze the performance of the NamespaceOfflaoding controller under high-load situations. More precisely, referring to the figure 8.1, we want to analyze how long it takes the controller to fill the NamespaceMaps with replication requests. Replication requests are generated by the controller from a NamespaceOfflaoding resource. Inserting a replication request means creating an entry within the specs of a NamespaceMap resource as shown in step 2 of figure 8.1. Creating a NamespaceOfflaoding resource that imposes a namespace replica on each available remote cluster, we want to see how long it takes before requests are correctly entered and how this time changes as the number of NamespaceMaps to fill increases. During this benchmark, the number of virtual nodes and consequently the number of NamespaceMap resources has been increased from 10 to 100. When a NamespaceOfflaoding is created, the NamespaceOffloading controller identifies all virtual nodes compatible with replication constraints and fills the corresponding NamespaceMap resources. Increasing the number of virtual nodes by a fixed value introduces a constant delay because the compatibility analysis of new nodes and the insertion in the corresponding maps are processes independent of the number of resources already processed. The trend recorded will therefore be linear, as shown in the figure 8.4.

8.2 Deployment replication benchmarking

The analysis concerning the deployment replication is carried out in a slightly different test environment than before. In this case, the Kubebuilder environment is integrated with a KinD cluster to have better visibility of the whole process. Taking up the schema 8.5, concerning the namespace replication workflow of chapter 7, we can identify two possible aspects to test:

1. A first benchmark tests the entire flow represented in the figure 8.5 starting from the LiqoDeployment creation up to the resource update, then from step 1 to step 5. More precisely, we want to check the LiqoDepoyment controller scalability as the number of clusters requiring a replica increases.

2. A second benchmark analyzes how the LiqoDeployment controller scales as the number of LiqoDeployment resources to be reconciled increases. So instead of having a single resource as in the figure 8.5, we will have several LiqoDeployment objects reconciled separately by the controller.

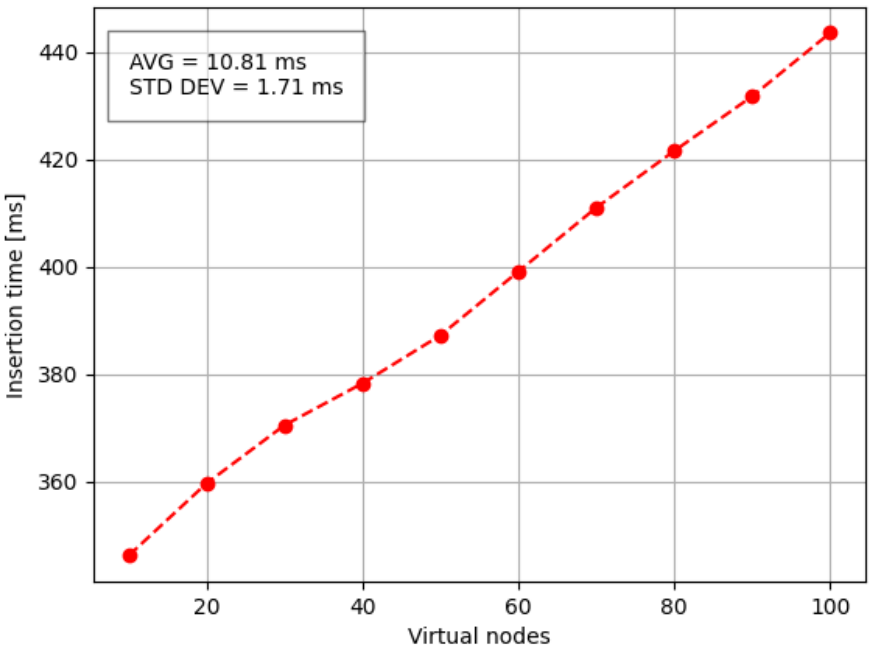

Figure 8.4: The replication requests insertion time scale linearly as the number of virtual nodes increases.

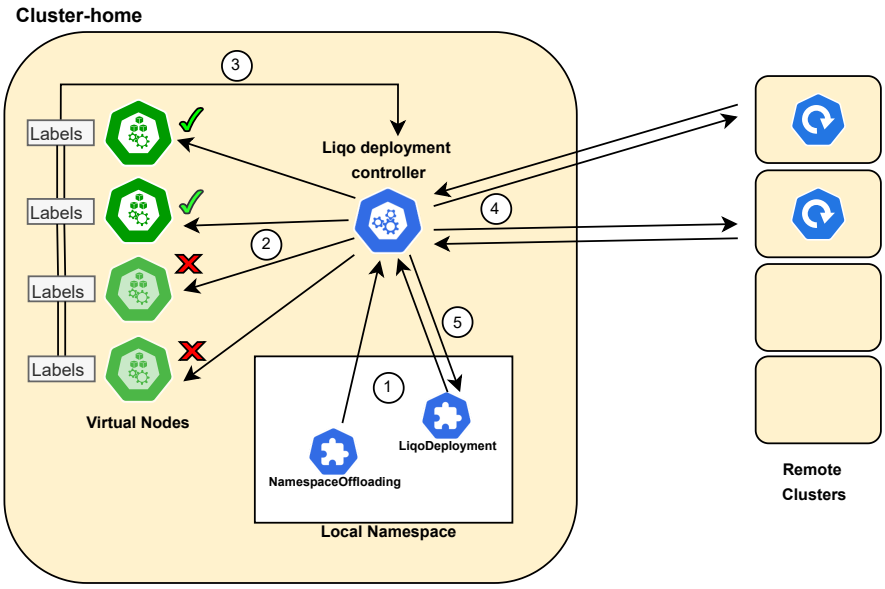

Figure 8.5: Deployment replication workflow.

8.2.1 Replication scalability with a single resource

This benchmark aims to create an increasing number of deployment replicas starting from a single LiqoDeployment resource and see how the creation time grows as the number of requested replicas increases. We have used a LiqoDeployment resource with a *cluster replication granularity*, so for every remote cluster, the controller has to create a deployment replica, and we have considered an increasing number of virtual nodes starting from 30 up to 100.

The creations of deployment replicas are unrelated, consequently increasing the number of the required replicas by a fixed value (in this case 10) will lead to a constant delta and, therefore, to a linear trend as the dashed line in figure 8.6 shows.

In addition to measuring the process scalability, we have evaluated how much the deployment replication times impact the overall start-up time of applications. Considering the generation of 3 pods starting from each replicated deployment, we have obtained the results shown by the dotted line in figure 8.6. As we can see, the replication time does not have a strong impact on the pods creation time consequently, the implemented approach seems sustainable even in large-scale scenarios.

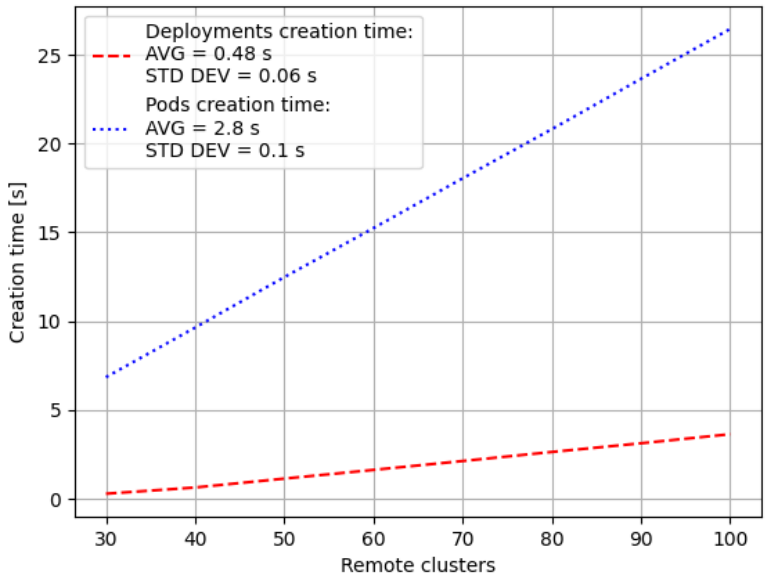

Figure 8.6: Deployment replicas creation time compared with the pods creation time.

8.2.2 Replication scalability with multiple resources

Keeping the virtual nodes number fixed at 10, we have increased the number of LiqoDeployment resources: each resource, therefore, requires the creation of 10 deployment replicas, one for each virtual node. The number of deployments to be created is the same as in the previous test, but in this case, the LiqoDeployment controller must perform a different reconciliation for each resource. Consequently, the dashed line in figure 8.7 shows slightly longer times than those of the previous benchmark. The creation time trend as the number of LiqoDeployment resources increases is still linear as each reconciliation is independent of the others.

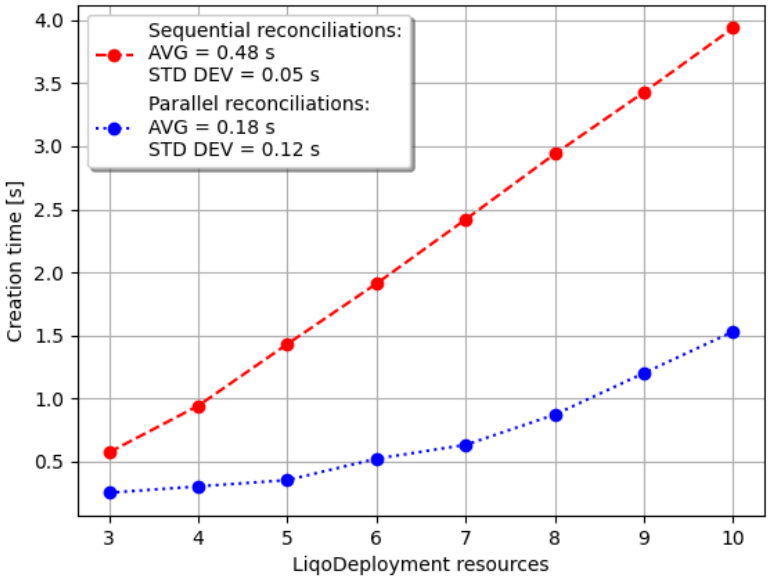

Figure 8.7: The creation time scales linearly as the LiqoDeployment resources number increases.

To reduce the cost due to the different reconciliations, we have decided to use a multi-thread approach: 5 workers were instantiated to perform the necessary reconciliations in parallel. The dotted line in figure 8.7 shows significantly reduced times: the trend is always linear since there is an overhead due to increasing interaction with the server API. Even by setting the threads number equal to reconciliations, the times could not be constant, as the load on the API server continues to increase with the number of replication requests.

8.3 Conclusions

This work proposes a solution to take advantage of the Liqo project in a straightforward and intuitive way, creating a multi-cluster topology and simply deploying applications within it. Thanks to new features introduced, it is possible to create multi-cluster architectures with fine-grained requirements, selecting only some clusters of the federation or choosing specific offloading policies for applications. Furthermore, the number of privileges required for the topology creation has been considerably reduced, allowing multi-ownership approaches where different companies are involved.

8.3.1 Future works

The deployment replication mechanism introduced in this thesis is not yet fully mature, so it does not provide complete support to important features such as the possibility to specify a replacement policy for deployment replicas or the ability to instantiate a different number of replicas for each remote cluster. Another community-driven feature plans to strategically distribute pods belonging to the same deployment replica among the various nodes of the remote cluster, ensuring better fault tolerance.

Liqo replication works at namespace granularity, but there are no major limitations to introduce a finer approach. So far, all the resources deployed in the enabled namespace are reflected inside remote clusters. The idea is to allow users to filter out which resources should be offloaded and which should not. The offloading constraints introduced in this thesis are the first step in this direction, but they work only for application offloading. As an example, the community requires a mechanism to selectively reflect service across multiple clusters, while right now, services deployed inside the local namespace are replicated on all available clusters.

Bibliography

[1] *Kubernetes official documentation*. URL: https://kubernetes.io/docs/home/ (cit. on pp. 4, 11, 13, 15, 17, 19).

[2] *Virtual-kubelet git repository*. URL: https://github.com/virtual-kubelet/virtual-kubelet (cit. on pp. 4, 18, 19).

[3] *Kubebuilder git repository*. URL: https://github.com/kubernetes-sigs/kubebuilder (cit. on pp. 4, 18, 19).

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

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

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

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

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

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

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

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

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

[13] *RFC 1123*. URL: `https://datatracker.ietf.org/doc/html/rfc1123` (cit. on p. 31).

[14] *Kiosk git repository*. URL: `https://github.com/loft-sh/kiosk` (cit. on pp. 36, 39).

[15] *Capsule git repository*. URL: `https://github.com/clastix/capsule` (cit. on p. 38).

[16] *Liqo official documentation*. URL: `https://doc.liqo.io/` (cit. on pp. 47, 48).