POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Integrating IoT Devices in a Geographical Computing Infrastructure Orchestrated with Kubernetes

Supervisors

Prof. Fulvio RISSO

Dott. Alex PALESANDRO

Candidate

Alessandro OLIVERO

April 2021

Summary

In the last two decades the cloud has gained a lot of importance, indeed the current trend is to engineer the new web applications to be cloud native, thus to be split up in 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, a project kicked off by Google in 2014 that allows to automate deployment, scaling, and management of containerized applications. Beside the cloud, in recent years the edge computing has gained a lot of importance: it is a distributed computing paradigm that brings the computational and storage resources close to the final user, in order to improve the QoS standards in terms of latency and bandwidth.

The goal of the project behind this thesis is to create a federation of Kubernetes clusters that cooperate at the edge of the network: many different tenants are connected together 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 own resource cluster available to the federation by sharing or leasing them out in a federated environment.

This solution needs a way to make possible that the cluster that will form this federation can discover, authenticate and join them each other in a liquid and dynamic way. This work proposes a solution to allow clusters, both in Local Area Networks (as a set of IoT or edge devices) and in Wide Area Networks (as between data-centers or between IoT devices and the central cloud), to have an automatic discover and to peer each other in a transparent and secure way in a multi-tenant scenario.

Table of Contents

List of Tables

List of Figures

Acronyms

DNS
Domain Name System

mDNS
Multicast DNS

HTTP
HyperText Transfer Protocol

LAN
Local Area Network

WAN
Wide Area Network

K8s
Kubernetes

CRD
Custom Resource Definition

CR
Custom Resource

Chapter 1

Introduction

In the last years, container orchestration is becoming more and more important and Kubernetes, one of the main actors in this field, is used by most of the big companies to orchestrate their jobs in their data-centres in a clustered environment. Now, 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 the 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 is very difficult to have enough resources to deal with peaks of traffic. Here is where multi-cloud solutions like Liqo come.

Liqo, the project behind this thesis, enables the creation of a multi-cluster environment with liquid resource 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 The need of Multiple Clusters

Organizations may need a multi-cluster environment for many different reasons. We can distinguish between two main categories of environments:

- In a **Cloud Environment**: where the company can have many large data-centres, 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, distributed in multiple zones, and may need that these clusters are hosted by different cloud providers to contain costs or to not be strictly linked to a specific one of them.

The usage of multiple clusters can also reduce some scalability problems on very big clusters.

- In an **Edge or IoT Environment**: where the company can have a lot of small clusters, even single-node ones. They can be geographically distributed to be closer to the end-user.

  In this scenario, the main needs are the availability of the same API (for the software) and the re-utilization of the same skills (for the humans) already achieved for the cloud world to manage small devices too. With the inter-operability of the API, a new and closer integration becomes possible with the movement of the applications between different devices.

1.2 Service Discovery

According to the previous scenario, when a new IoT/edge device is turned on, it has to discover the environment around it, to know who its neighbors are, and it can need to connect to the cloud infrastructure of the producer and/or of the owner company in the simplest and transparent way possible. This new scenario is very different from the core of the cloud, built of private and/or public data-centres with a static and well-known infrastructure, this can be very dynamic, and the available clusters and devices can change rapidly over time.

The only way to achieve this flexibility is to implement an automatic service discovery.

1.3 The Goal of the thesis

The goal of this thesis is to design and implement a strategy and a mechanism to allow Kubernetes clusters to discover and to join them together, keeping the knowledge of the neighborhood updated over time.

Many federation solutions like Kubefed (the multi-cloud solution by the Kubernetes community) do not have the required elasticity, do not provide a cluster discovery solution, require a master-slave architecture in clusters (no peer-to-peer connections), and only support manual join and unjoin from the federation. Kubefed can be a nice solution when applied in a static cloud infrastructure (even if it is not a perfect solution for many reasons), but it is totally not a viable solution in a dynamic environment like the IoT world.

The first step is to achieve the possibility to dynamically update the knowledge of the clusters, that in this scenario can be intended as composed by a single edge device, in our proximity and to be able to know *where* (the address and the port) and *how* (with which identity) to connect with them.

This work answers to the *where* question using already existent technologies like the DNS/mDNS Service Discovery, which allows generic devices (a printer for example) to be discovered in a Local Area Network (LAN), and bring them in a cloud-based ecosystem. This technology is enough flexible to be used for Wide Area Networks (WAN) too, using a DNS server instead of mDNS one, allowing us not to have to duplicate our discovery stack when we have a new different scenario where the IoT/edge devices need to connect to the cloud, or even in a cloud-to-cloud scenario.

In a second step a new service, the Authentication Service, has been created to answer to the *how* question and handle the cluster authentication and to provide a new identity with the least required permissions to the remote clients that want to establish a new peering.

With this new identity the two clusters are able to share and to negotiate the parameters required to establish both the peer-to-peer network connection between them and to create the Liqo "virtual big node" where the jobs are effectively scheduled, and the liquid resource sharing happens.

We will analyse this work with the following structure:

- **Chapter 2** provides an extensive presentation of Kubernetes, its architecture and concepts.
- **Chapter 3** describes the Service Discovery technologies, with particular attention to the pros and cons of the main ones.
- **Chapter 4** provides an extensive presentation of Liqo, its architecture and concepts.
- **Chapter 5** provides a general overview on the Discovery and Peering process.
- **Chapter 6** presents in depth the functionality of the Discovery Component, like the different discovery methods.
- **Chapter 7** presents in depth the cluster authentication mechanism and the component that handle it.
- **Chapter 8** analyses the achieved results in terms of platform supports and in terms of time needed for the discovery and the peering of two clusters.

Chapter 2

Kubernetes

In this chapter we analyse Kubernetes architecture, showing also its history and evolution through time, in order to lay the foundations for all the work 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 main 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, in particular **Virtual-Kubelet** [2], a project which allows to create virtual nodes with a particular behaviour, and **Kubebuilder** [3], a tool 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 as a 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".

In the middle of 2014, Google presented **Kubernetes** as on open-source version of Borg. Kubernetes was created by Joe Beda, Brendan Burns, and Craig McLuckie, and other engineers at Google. Its development and design were heavily influenced by Borg and many of its initial contributors previously 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 is designed to completely manage the life cycle of applications and services using methods that provide consistency, scalability, and high availability.

What does "containerized applications" means? In the last decades, the deployment of applications has seen significant changes, which are illustrated in figure 2.1.

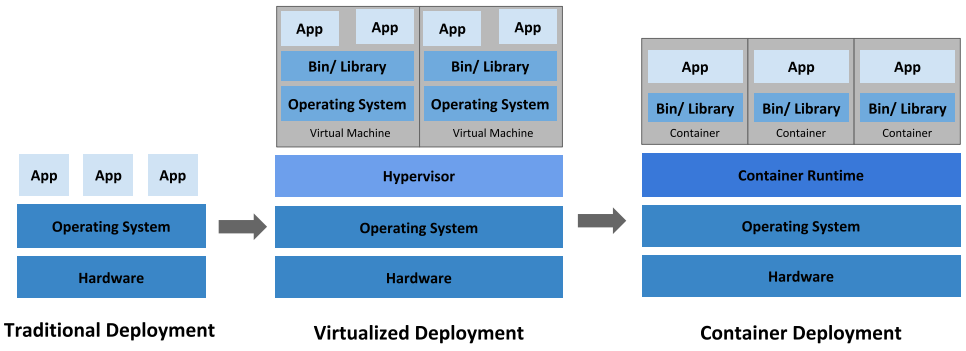


Figure 2.1: Evolution in applications deployment.

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: the 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 to run

multiple Virtual Machines on a single physical server. It grants isolation of the applications between VMs providing a high level of security, as the information of one application cannot be freely accessed by another application. 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 together 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 own operating system, on top of the virtualized hardware.

A second solution which has been proposed recently is **containerization**. Containers are similar to VMs, but they share the operating system with the host machine, relaxing isolation properties. Therefore, containers are considered a lightweight form of virtualization. Similarly to a VM, a container has its own filesystem, CPU, memory, process space etc. One of the key features of containers is that they are portable: as they are decoupled from the underlying infrastructure, they 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 of 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. We provide a description of such system 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 own IP address. If traffic to a container is high, a load balancer able to distribute the network traffic is provided.
- **Storage orchestration** A storage system can be automatically mounted, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** The desired state for the deployed containers can be described, and the actual state can be changed to the desired state at a controlled rate. For example, it is possible to automate the

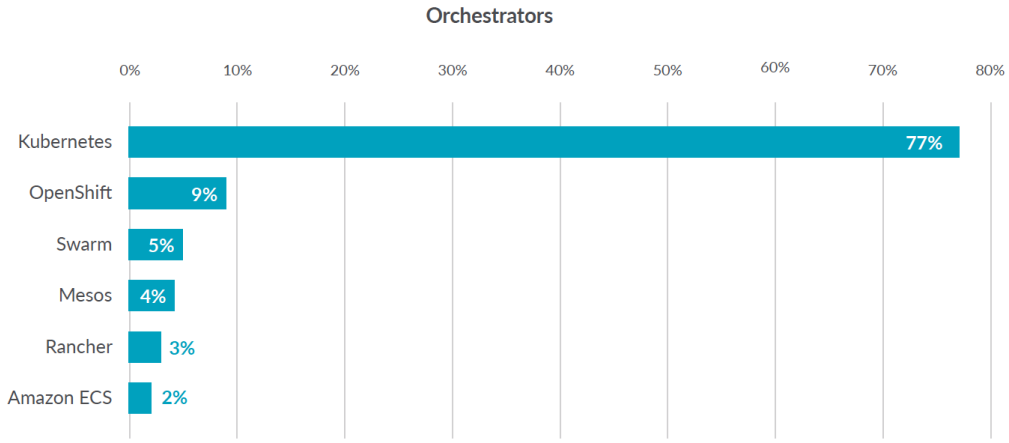


Figure 2.2: Container orchestrators use [9].

creation of new containers of a deployment, remove existing containers and adopt all their resources to the new container.

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

2.4 Kubernetes architecture

When Kubernetes is deployed, a cluster is created. A Kubernetes cluster consists of a set of machines, called **nodes**, that run containerized applications. At least one of the nodes hosts the control plane and is called **master**. Its role is to manage the cluster and expose an interface to the user. The **worker** node(s) host the **pods** that are the components of the application. 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.

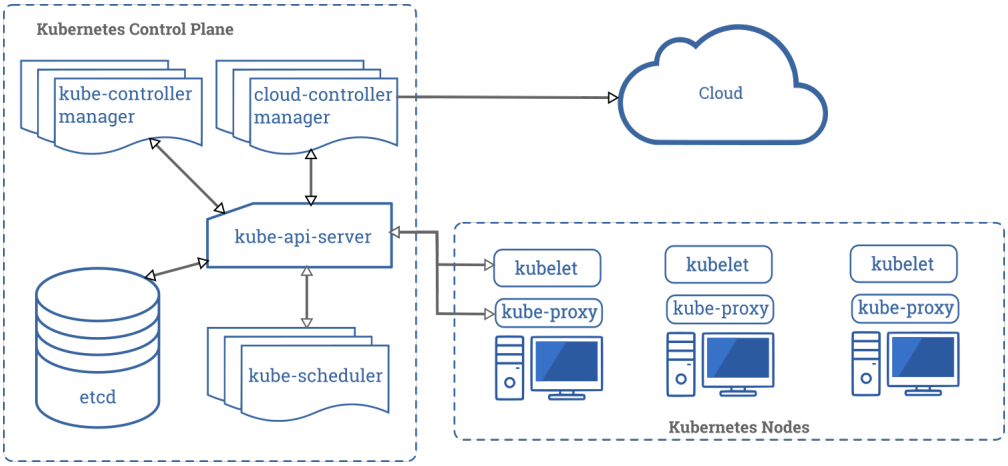


Figure 2.3: Kubernetes architecture

2.4.1 Control plane components

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

API server

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

etcd

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

Scheduler

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

kube-controller-manager

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

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

cloud-controller-manager

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

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

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

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

2.4.2 Node components

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

Container Runtime

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

kubelet

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

kube-proxy

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

Addons

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

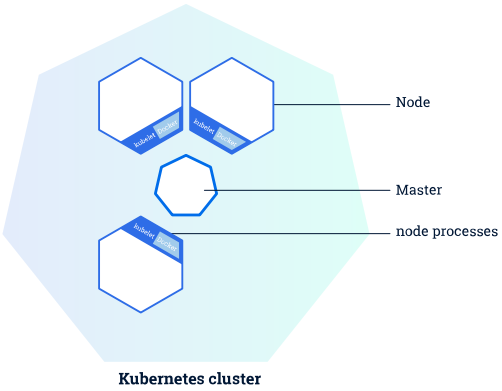


Figure 2.4: Kubernetes master and worker nodes [11].

2.5 Kubernetes objects

Kubernetes defines several types of objects, which constitutes its building blocks. Usually, a K8s resource object contains the following fields [12]:

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

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

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

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

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

2.5.1 Label & Selector

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

2.5.2 Namespace

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

- **kube-system**: it contains objects created by K8s system, mainly control-plane agents;
- **default**: it contains objects and resources created by users and it is the one used by default;
- **kube-public**: readable by everyone (even not authenticated users), it is used for special purposes like exposing cluster public information;
- **kube-node-lease**: it maintains objects for heartbeat data from nodes.

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

2.5.3 Pod

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

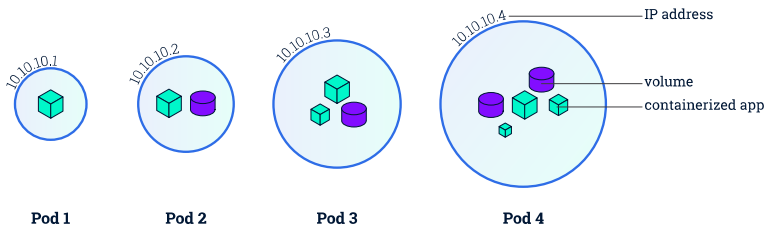


Figure 2.5: Kubernetes pods [11]

2.5.4 ReplicaSet

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

2.5.5 Deployment

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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
```

```
15          app: nginx
16      spec:
17        containers:
18        - name: nginx
19          image: nginx:1.7.9
20          ports:
21          - containerPort: 80
```

The code above allows to create a Deployment with name `nginx-deployment` and a label `app`, with value `nginx`. It creates three replicated pods and, as defined in the `selector` field, manages all the pods labelled as `app:nginx`. The template field shows the information of the created pods: they are labelled `app:nginx` and launch one container which runs the nginx DockerHub image at version 1.7.9 on port 80.

2.5.6 Service

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

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

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

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: my-service
5 spec:
6   selector:
```

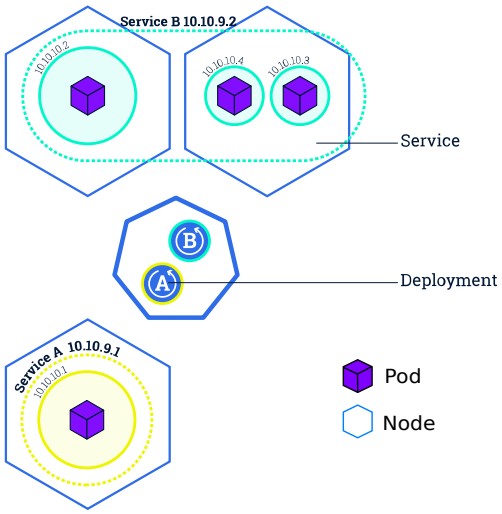


Figure 2.6: Kubernetes Services [11]

```
7       app: myApp
8     ports:
9       - protocol: TCP
10        port: 80
11        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 compute or network resources based on the roles of individual users.

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 provide to it a new client certificate that will be used in all the 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 of them is that the first sets the permissions within a particular namespace (the one which contains the resource), while the second is a non-namespaced resource and can be used in all the namespaces.

```yaml
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 [11] 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.

```yaml
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 [11] 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 the purposes of 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 in order to use it. The official documentation [2] says that “providers must provide the following functionality to be considered a supported integration with Virtual Kubelet:

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

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

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

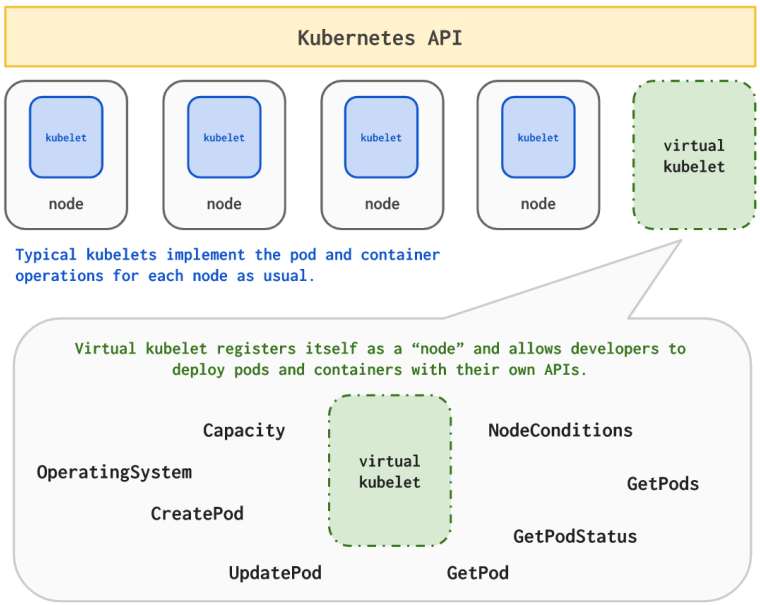


Figure 2.7: Virtual-Kubelet concept [2]

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 created, the Kubernetes API server creates a new RESTful resource path; the CRD can be either namespaced or cluster-scoped. The name of a CRD object must be a valid DNS subdomain name.

A **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 [11]. On their own, custom resources simply let you store and retrieve structured data. In order to have a more powerful management,

you also need to provide a custom controller which executes a control loop over the custom resource it watches: this behaviour is called Operator pattern [13].

Kubebuilder helps a developer in defining his Custom Resource, taking automatically basic decisions and writing a lot of boilerplate 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 to test new fields and business logic.
6. Build and publish a container from the provided Dockerfile.

Chapter 3

Service Discovery

In this chapter we analyze how service discovery can be implemented over an IP network, analyzing the chosen DNS-SD (DNS service discovery) protocol that we aim to integrate with our solution.

3.1 DNS Service Discovery

DNS Service Discovery is a way of using standard DNS programming interfaces, servers, and packet formats to browse the network for services. [14]

The DNS-SD protocol has been created to operate in a zero-configuration environment in substitution of old non IP-based discovery protocols like AppleTalk Name Binding Protocol. [15] [16]

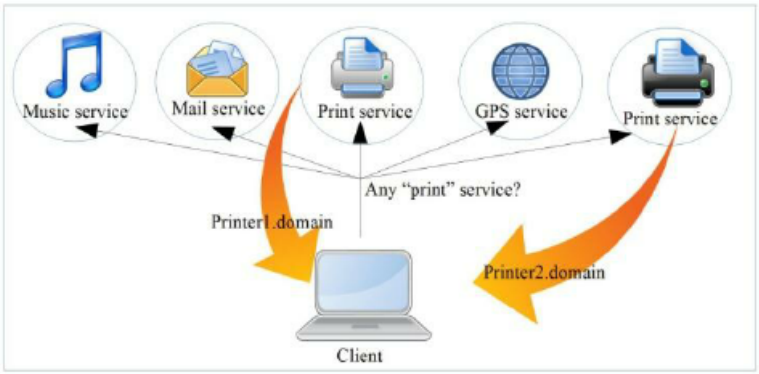


Figure 3.1: Service Discovery [17]

3.1.1 How does it work

Service Instance Enumeration (Browsing)

Traditional DNS SRV records are useful for locating instances of a particular type of service when all the instances are effectively indistinguishable and provide the same service to the client.

This specification adds a level of indirection over the DNS SRV records. Instead of requesting records of type "SRV" with the name "_ipp._tcp.example.com.", the client requests records of type "PTR".

The result of this PTR lookup for the name "<Service>.<Domain>" is a set of zero or more PTR records giving Service Instance Names of the form: `Service Instance Name = <Instance> . <Service> . <Domain>` [16]

Service Instance Resolution

When a client needs to contact a particular service, identified by a Service Instance Name, previously discovered via Service Instance Enumeration (browsing), it queries for the SRV and TXT records of that name. The SRV record for a service gives the port number and target hostname where the service may be found. The TXT record gives additional information about the service. [16]

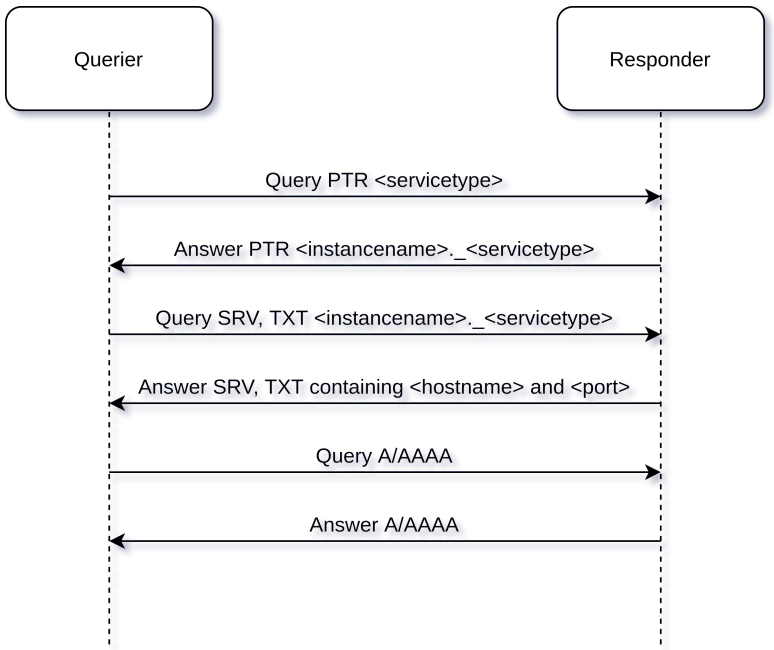


Figure 3.2: DNS Service Discovery

3.1.2 DNS for WANs

In a Wide Area Network, the DNS-SD protocol is able to successfully handle a Service Discovery. The only required configuration is the address of a DNS server serving the appropriate DNS records. On some networks, the administrator might manually enter these records into the name server's configuration file.

Another option, that can automatically manage the configuration, could be a network monitoring tool that could output a standard zone file to be read into a conventional DNS server. Alternatively, IP devices could use Dynamic DNS Update to automatically register their own PTR, SRV, and TXT records with the DNS server. [16]

3.1.3 mDNS for LANs

Multicast DNS is a way of using familiar DNS programming interfaces, packet formats, and operating semantics, in a small network where no conventional DNS server has been installed. [18]

In a Local Area Network, a Multicast DNS client may simply send standard DNS queries blindly to 224.0.0.251:5353, without necessarily even being aware of what a multicast address is. If a name being queried falls within one of the reserved Multicast DNS domains, then, rather than using the configured Unicast DNS server address, the query is instead sent to 224.0.0.251:5353. [19] If a host can offer the service that the remote host is looking for, it will answer with its local DNS records.

The mDNS service discovery is one of the pillars of Zero Configuration Networking, "that means making it possible to take two laptop computers, and connect them with a crossover Ethernet cable, and have them communicate usefully using IP, without needing a man in a white lab coat to set it all up for you". [20]

Chapter 4

Liqo

In this chapter, we analyze Liqo architecture, showing the idea behind it. We will describe the overall picture of the open-source project where this work is involved in, how it is integrated, and its importance.

4.1 Liqo Idea

Liqo aims to create an opportunistic interconnection of multiple Kubernetes clusters allowing seamless resource and service sharing among them, creating an "endless Kubernetes ocean" where the user applications can be scheduled.

We can have a multiple cluster environment in a lot of different scenarios, both owned by the same entity or owned by different entities, These cluster may have underutilized resources because all these clusters have to have enough resources to deal with a peak of load by their own, but during the day they have moments of low load. In these moments they are wasting a part of their resources that can be available to be shared.

Liqo aims to extend the resources present in an already existent cluster using the ones currently non-occupied in neighbor clusters in an opportunistic way, so no peering and no sharing are definitive or not reversible, and it's always possible unpeer the two clusters in a simple way and return to the original state. When we extend a cluster with Liqo there is no change in the standard Kubernetes APIs, the ones described in Chapter 2 are still valid in the new environment, and the user applications have not to be changed in order to wirk with Liqo.

Liqo extends the cluster by adding a new virtual node for each remote peered cluster, creating in that way a "virtual big node" where the pods can be scheduled by the default Kubernetes scheduler with no change. The Kubernetes Pods that will be scheduled on this virtual node will be took by the Virtual Kubelet and offloaded to the remote cluster.

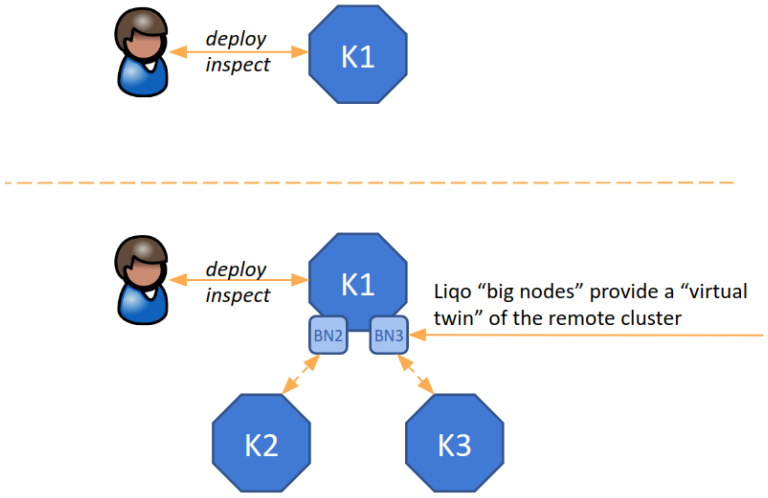


Figure 4.1: No Change in Kubernetes API

4.2 Cluster Management

Liqo is able to manage multiple Kubernetes clusters, allowing the user to use external resources, in a transparent way. We can describe the Liqo cluster management functionality with five pillars:

1. **Discovery**: Discover available clusters.
2. **Peering**: Establish an administrative interconnection between the clusters and negotiate the parameters.
3. **Network Interconnection**: Establish a network interconnection between the clusters.
4. **Resource Management**: Create the virtual node and make the external resources available.
5. **Usage**: Offload your pods.

4.2.1 Discovery

Liqo can dynamically discover and add new clusters to the "Big Cluster" abstraction. These clusters can be discovered in a lot of 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 will take the information from different data sources and will create a new ForeignCluster CR in the local cluster.

The discovery functionality is the main argument that we will deal with in this work, we will analyze it deeper in the next chapters.

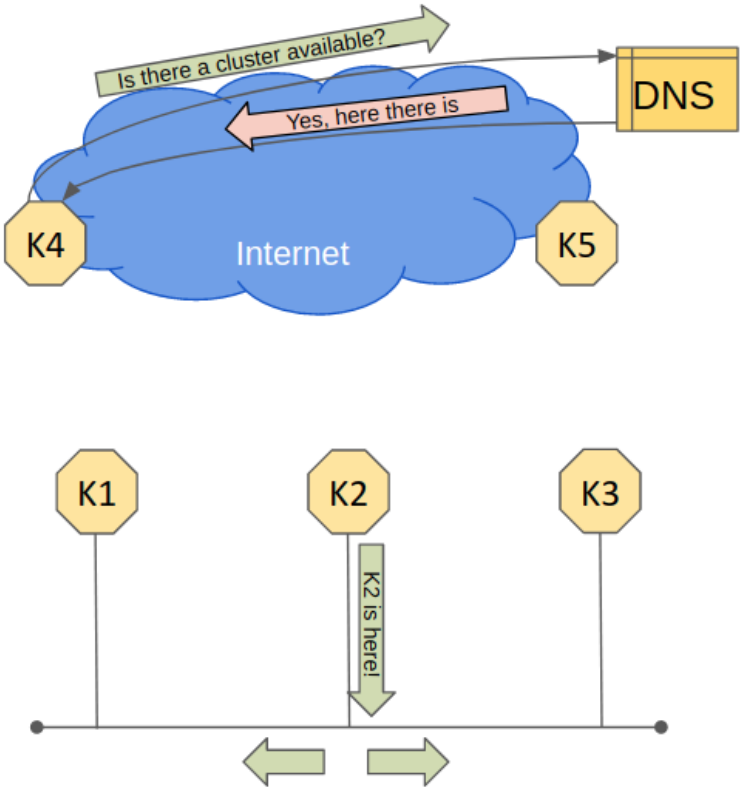


Figure 4.2: Discovery

4.2.2 Peering

Liqo can dynamically peer different and administratively separate clusters with a policy-driven, voluntary, and direct relationship. This connection has to be established before sharing any resources. It has a peer-to-peer architecture, so no *master* cluster is involved.

The Liqo peering uses the 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.

The peering process will be deeper analyzed in Chapter 5.

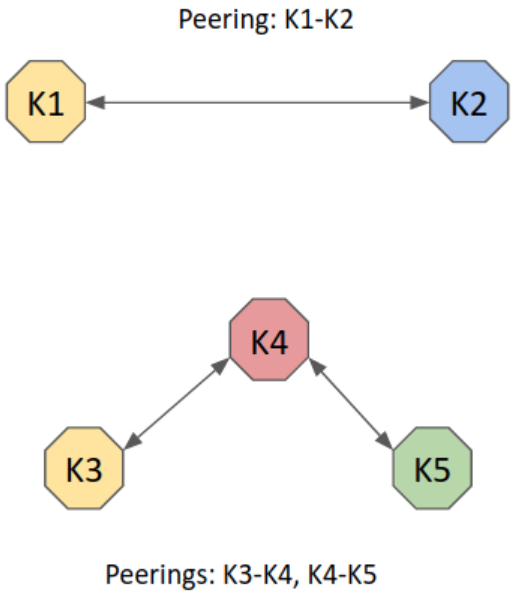


Figure 4.3: Peering

4.2.3 Network Interconnection

Liqo can extend the cluster network to the remote cluster, basing on the peering information. The network parameters, required to establish the VPN tunnel, are dynamically negotiated with dedicated CRD to allow Liqo to support overlapping pod CIDR in the two clusters. This let Liqo not take any assumption on the IP address space and on the networking in the peered clusters.

Liqo defines a *gateway* pod (possibly replicated) that works as a VPN terminator and allow 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 sapaces.

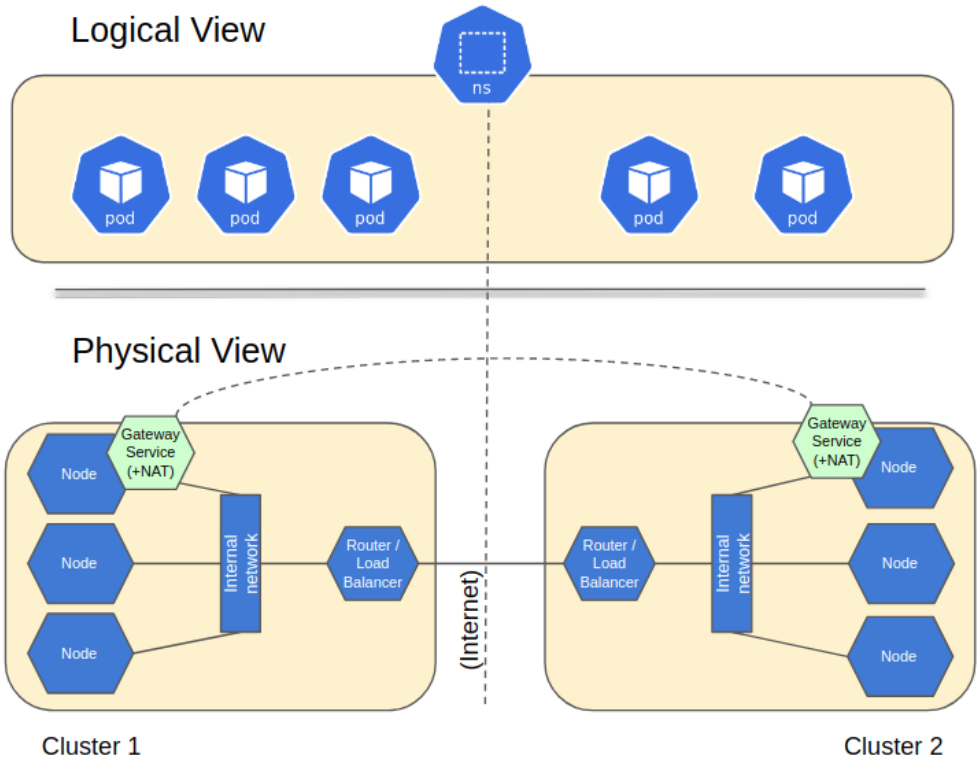


Figure 4.4: Network Interconnection

4.2.4 Resource Management

When a cluster accepted the peering with another one, it can advertise to the second the amount of resources that it can share. If the remote cluster accepts the offer, it will create a new "Big Node" with these resources (i.e. CPU and memory). The "Big Nodes" are equivalent to physical nodes, hence can be controlled by the vanilla Kubernetes scheduler and controller-manager.

When the peering and the network interconnection is completed, the Virtual Kubelet will enable the new node setting it to ready.

4.2.5 Usage

When in the local cluster the new virtual node is set up and marked as ready, and the vanilla Kubernetes scheduler can schedule new pods on this node, these new pods will no see differences when accessing a service being deployed locally or remotely.

The VirtualKubelet is in charge to take the Pods scheduled on the virtual node, to offload them reflecting them in the remote cluster, and to keep the local shadow

Pod Status aligned to the remote one. These Pods will be reachable from the local cluster and they can reach the services in the local cluster. Services and Endpoints are consistent on both the clusters, because of the VirtualKubelet reflection of EndpointSlices and them IP translation.

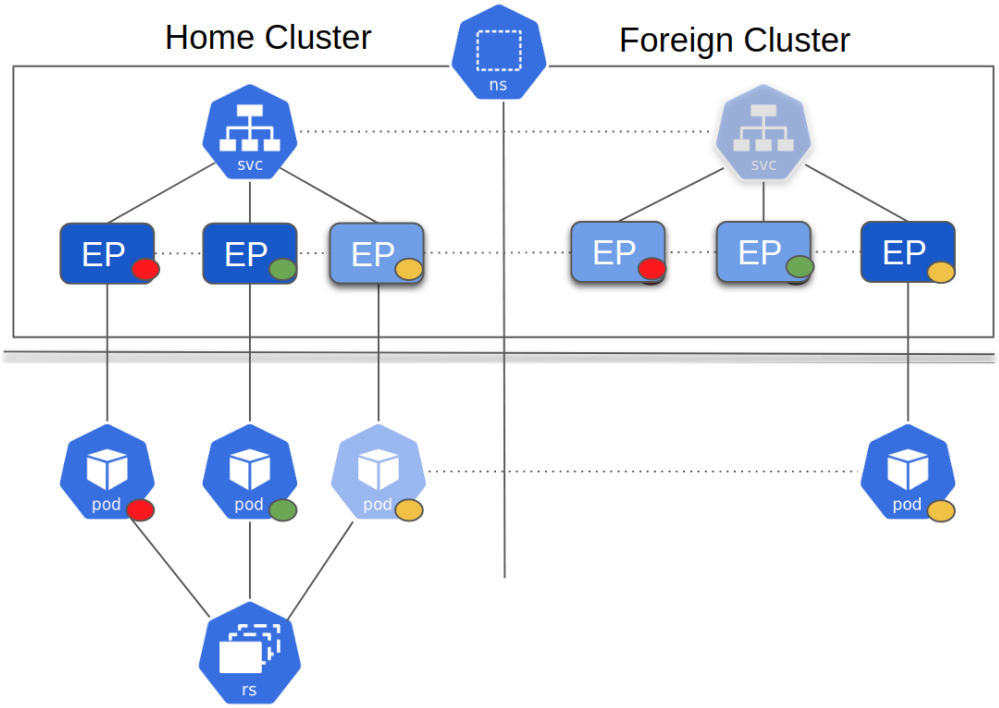


Figure 4.5: Use

Chapter 5

Discovery and Peering Process

In an environment where we can have multiple clusters, the first problem that we can find is: how can we make sure they know each other of their existence? And then, how can we share resources between them and make it possible to communicate? In this scenario, the Discovery and Peering Process comes in our help.

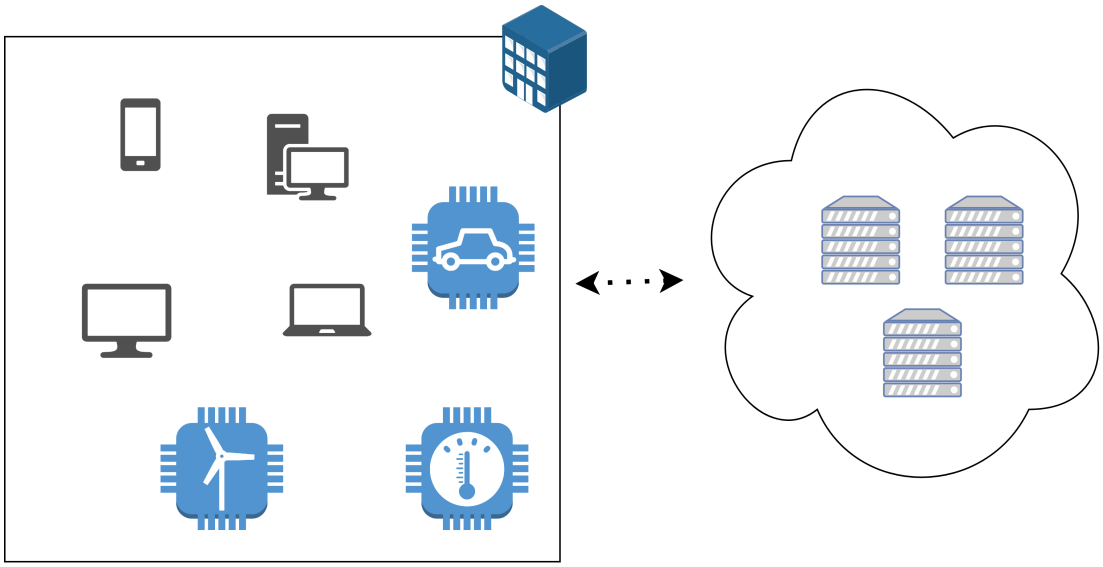


Figure 5.1: Multi-device environment example

Let us imagine having an environment with multiple servers and devices (both traditional devices as PCs or Laptops, and IoT devices) in our building that have

to be joined dynamically when the devices are turned on and to start sharing resources. These devices can need, not only need to discover among them, but also discover external enterprise services that can be outside of the building, either in another building or in the cloud.

5.1 Involved Components

We have to have some components to manage the discovery and peering process:

- **Discovery**: it performs the discovery process, both in LANs and in WANs, it has a mDNS server to make the cluster visible in the local area networks, a DNS and mDNS client to discover other clusters respectively in wide area networks and in local area networks. It also provides a garbage collection mechanism to delete those clusters no more available or reachable.

- **Authentication Service**: it provides an identity to the remote clusters that require it, and it links the required roles, basing on the peering status. This is the only component that is exposed to the external of the cluster.

- **ForeignCluster Operator**: it manages the reconciliation of the ForeignCluster CRs. It takes care of monitoring the peering status, making effective the user specification. It also checks the status of the other Liqo resources and takes the ForeignCluster CR status aligned with the system status.

- **PeeringRequest Operator**: it manages the reconciliation of the PeeringRequest CRs. If a request is accepted it creates the resources required to establish the peering (i.e. the Broadcaster).

- **Broadcaster**[21]: it collects the local cluster resource availability and sends the information to the remote cluster required for the virtual node creation. It is in charge to refresh periodically this information.

- **Advertisement Operator**[21]: it manages the reconciliation of the Advertisement CRs. When an advertisement is accepted it creates a VirtualKubelet[1] deployment that will manage the virtual node pointing to the foreign cluster.

[1]The Liqo VirtualKubelet is an implementation of the VirtualKubelet project, see it on GitHub at `https://github.com/virtual-kubelet/virtual-kubelet`

5.2 Communications

The Liqo components communicate each other in different ways. In Figure 5.2 we can see how the components interact. We can see in blue the components and the resources that are executed or stored in the remote cluster, and in green the ones that are in the home cluster. The boxes with squared border are logical Liqo components while the rounded ones are the requests between of them or the resources created.

In Liqo there are three ways that the components use to share information:

- **DNS** or **mDNS**
- **HTTP**(s)
- **Kubernetes CRDs**

DNS or mDNS This method is mainly used in the very early discovery stages when the two clusters are not aware reciprocally of their existence. The steps that are using these protocols implement the service discovery for the HTTP endpoint of the Authentication Service.

The DNS protocol has the key feature to be external to the Liqo installation and to be a well-known and always-available solution. While the mDNS protocol let us be discovered and to discover on LANs with zero configuration.

The information shared with these protocols can change over time and it needs to be periodically refreshed.

HTTP The HTTP protocol is used in the authentication step when the cluster is aware of the existence of another one, it is able to reach it, but it needs more information (for example a name to display to a human user), or it needs an identity with the required permissions to contact the API Server to complete the peering.

It is used when is required to have a custom response to a specific request (for example the request of a new identity), or when it is not possible to be authenticated from the remote API Server.

Kubernetes CRDs This is the most widely used method to share and store information in Liqo, in this way we can store some information that we collected in the previous steps and let that a Kubernetes Operator will reconcile the spec inserted in these resources with the observed status.

It allows us to persist some information and make all the components stateless, an important condition to run them in a distributed environment (as described in

the "from pet to cattle" paradigm [22]), and to promptly react to system status changes.

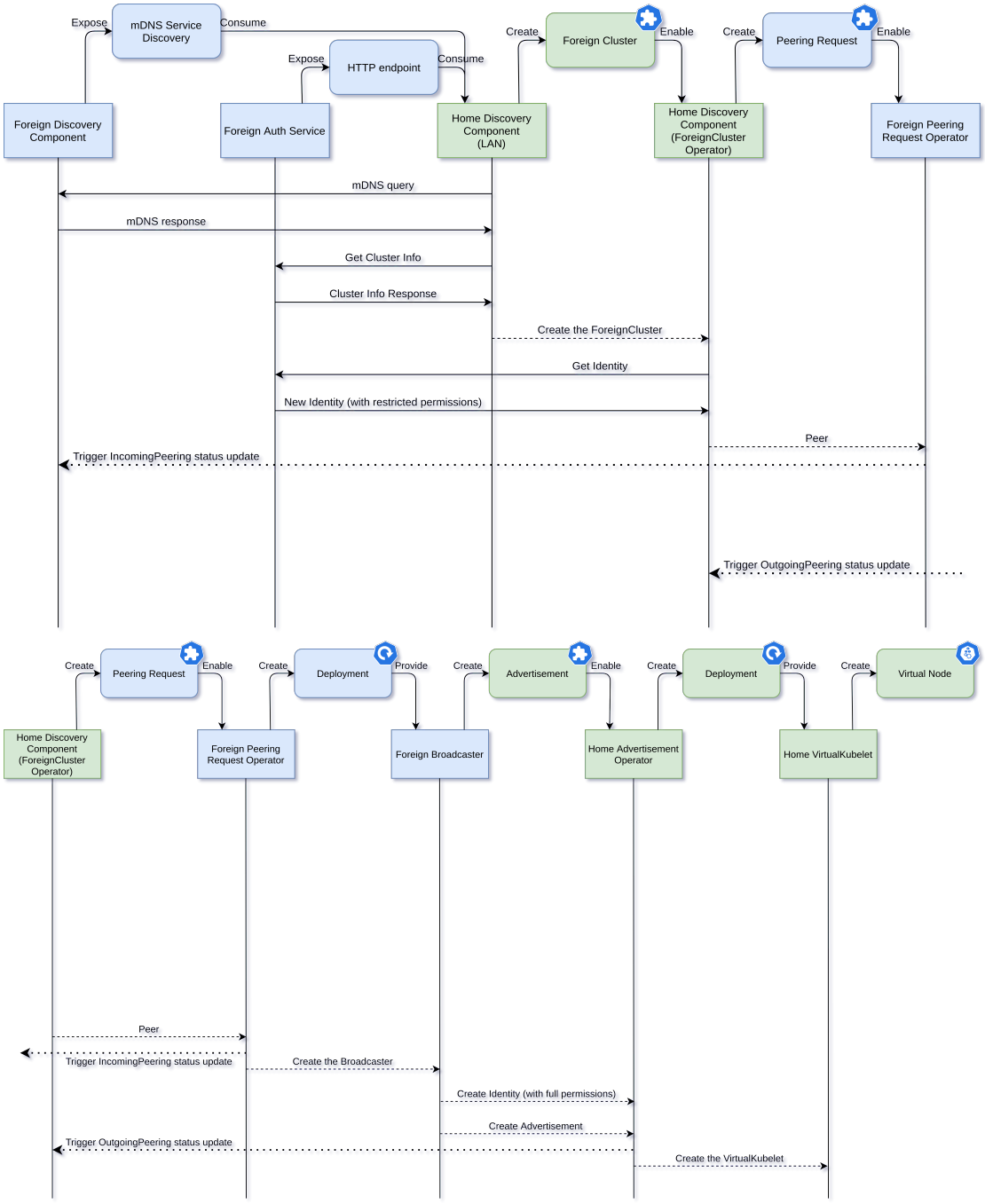


Figure 5.2: Liqo components inter-communication schema

5.3 Discovery and Peering Main Steps

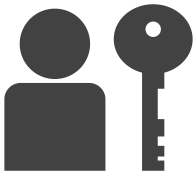
We can summarize the entire workflow described in the Figure 5.2 in 4 main steps: [2]

.

Discovery The goal of this step is to allow a cluster to be aware of the existence of another reachable cluster that is available to start a peering. This can be done in several different ways (mDNS, DNS, Manually, ...) and ends with the creation of a new ForeingCluster CR.

.

Authentication When a cluster is aware of the existence of another one, it needs to get its information (for example the ClusterName and the ClusterId), and it needs an identity to use to authenticate to its API Server. In this step, all this information is shared and the foreign cluster will create an identity uniquely associated with the home cluster.

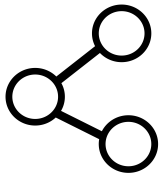


.

Peering This step can be enabled either automatically or manually by the user. The home cluster asks the foreign cluster to start a peering, the remote cluster will reply with an offer that the home cluster can accept or deny. If the request is accepted, the connection will be established and the peering completed.

.

Resource Sharing This is the last step when the peering was established and Liqo is ready to share the computing resources between the clusters.



[2] Illustrations by Vecteezy.com

5.4 CRDs involved in the Discovery and Peering Process

The CRD (*Custom Resource Definition*) abstraction allow us to map the observed state of some external[3] concept or knowledge to a real resource that can be managed natively by a Kubernetes Operator with an interaction with the API Server. This is exactly what we want to do with information retrieved during the discovery process.

In this section we will analyze in detail all the CRDs defined to make the process work.

5.4.1 ForeignCluster

A ForeignCluster CR maps the knowledge of the existence of another cluster to a Kubernetes resource.

Integrating Service Discovery with CRDs

With a DNS/mDNS service discovery, we can find the information required to contact a remote service. In particular, given service to discover we can find all the clusters that subscribed themselves as a provider for that service. This subscription can be to a DNS domain or to an mDNS group, but in both cases the retrieved information is very similar:

"DNS example"

```
1 example.com.                    PTR     liqo-cluster.example.com.
2
3 liqo-cluster.example.com.       SRV     0 0 443 auth.server.example.com.
4
5 auth.server.example.com.        A       1.2.3.4
```

"mDNS example"

```
1 _liqo_auth._tcp.local                          PTR   ClusterID._liqo_auth.
      _tcp.local
2
3 ClusterID._liqo_auth._tcp.local.local          SRV   0 0 31200 nodename.
      local
4
5 nodename.local                                 A     172.18.0.2
```

[3]External to the core Kubernetes resources, like Pod scheduling and managing

In both cases we have:

- **Well-Known Group or Domain**: *example.com.* and *_liqo_auth._tcp.local*

  This is an endpoint that we know is existing and where we will add our clusters, it can be a company domain (i.e. in WANs), or an mDNS domain (i.e. in LANs).

- **A cluster registered for that service**: *liqo-cluster.example.com.* and *ClusterID._liqo_auth._tcp.local*

  In the PTR record can be added how many clusters serving a Liqo service we want, each of them can be contacted to establish a peering.

- **The hostname where the Auth Service is exposed**: *auth.server.example.com.* and *nodename.local*

  For each cluster present in the PTR record we will have an SRV record telling us the hostname and the port to contact.

This information will be collected in an URL, that will be contacted in the future steps, and inserted in the new ForeignCluster CR. We can have a minimal (but working) example with the data taken from the "DNS example". All the other Cluster information will be retrieved from this URL, as described in Chapter 6.

"Minimal ForeignCluster example"

```
apiVersion: discovery.liqo.io/v1alpha1
kind: ForeignCluster
metadata:
  name: my-cluster
spec:
  authUrl: "https://auth.server.example.com.:443"
```

ForeignCluster list as Cluster Database

In a multi-cluster environment, we can see the list of the ForeignCluster resource as a Database of the clusters discovered and currently reachable from the home cluster. The discovery component constantly updates this list and manages also the peering status that can be in different phases: *Not Peered* when the cluster was discovered but no peering is active, *Outgoing* when we can schedule our pods in the remote cluster, *Incoming* when the remote cluster can schedule its pods in the home cluster, and *Bidirectional* when both the Incoming and the Outgoing has been activated.

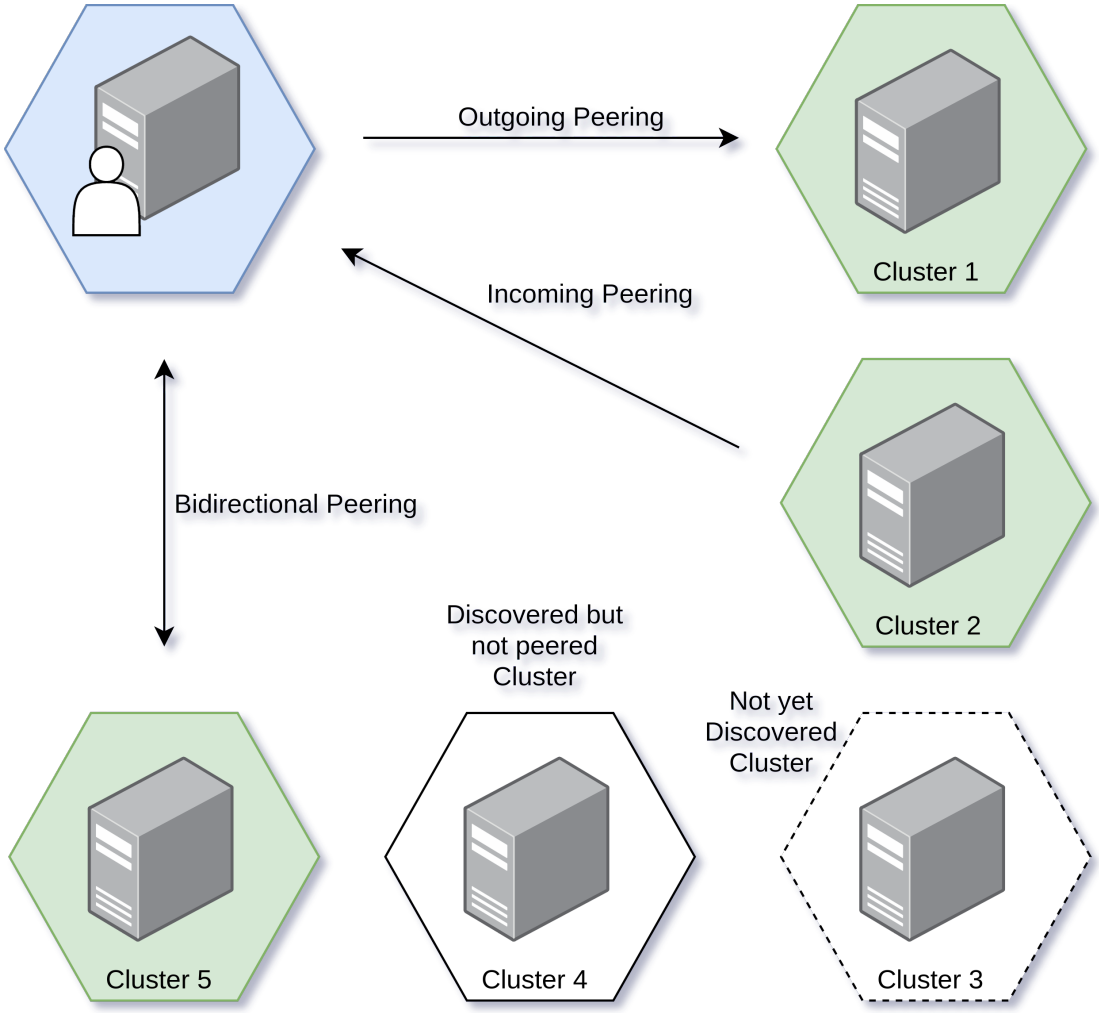


Figure 5.3: Environment with multiple Clusters

In the example in Figure 5.3, where we have five clusters, four of them discovered and three peered, we will have a ForeignCluster list like that (note that the Cluster 3 is not in the list):

Control Room for the Peering

The ForeignCluster CRD can be a control room for the peering with a remote cluster. We can act on this resource as an external user to ask the Liqo control plane to establish a peering or to tear down a previously established peering, simply setting the join flag inside the ForeignCluster specs.

The ForeignCluster Operator will react to the system events, as the creation of a new Advertisement or the successful setting of the VPN tunnel between the two

Name	Outgoing Peering	Incoming Peering
Cluster1	*Enabled*	Disabled
Cluster2	Disabled	*Enabled*
Cluster4	Disabled	Disabled
Cluster5	*Enabled*	*Enabled*

Table 5.1: Examples of ForeignCluster list

clusters, and updates the respective flags in the CR status.

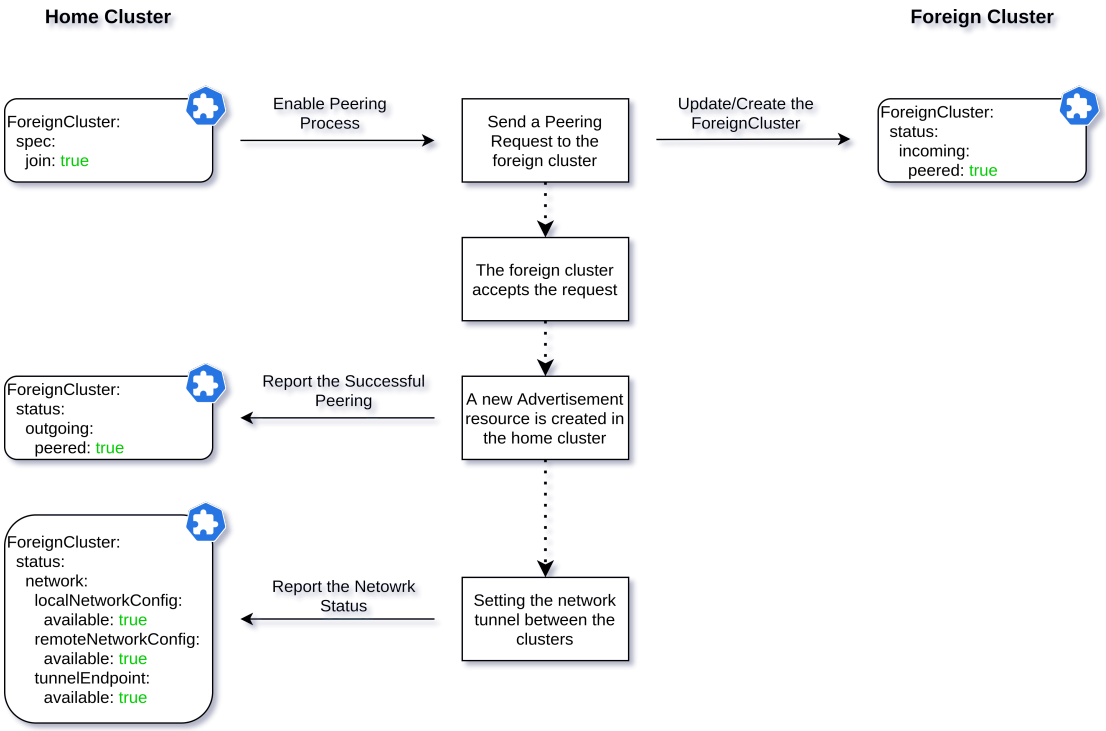


Figure 5.4: Peering Control Room (Join)

The ForeignCluster operator acts in a very similar way in the unjoin process. When the join flag is set to false, it reacts by deleting the PeeringRequest in the remote cluster. This action will trigger several actions in the Liqo components and the global status will be reported in the ForeingCluster status.

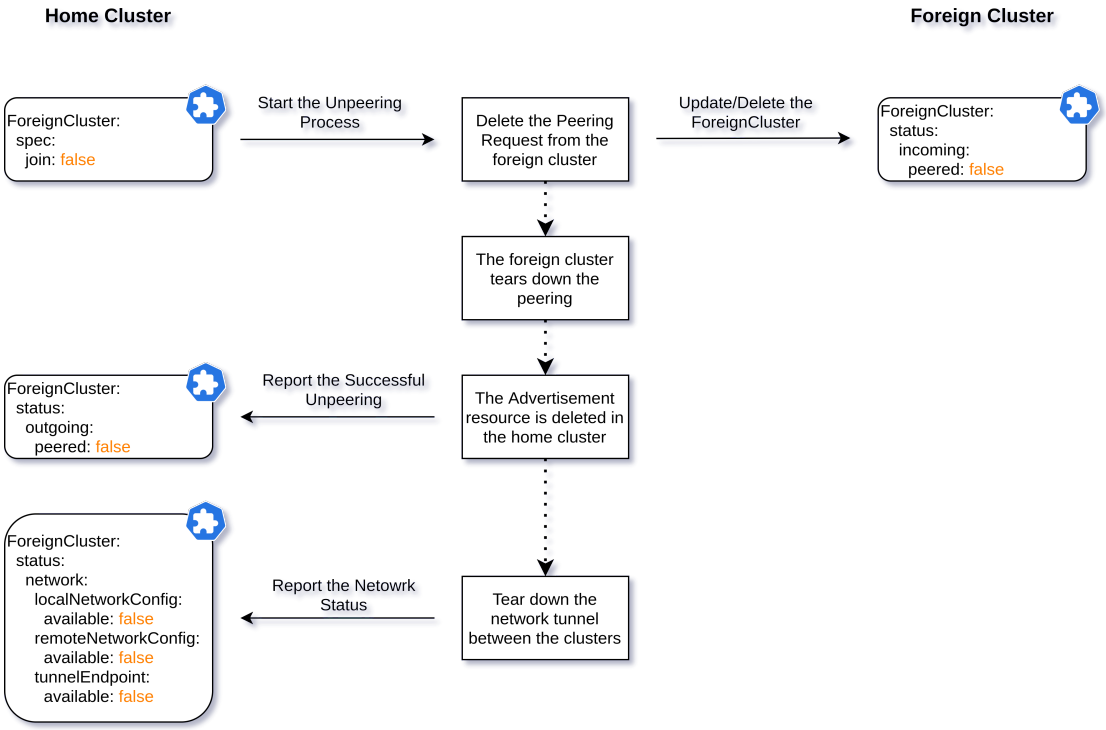


Figure 5.5: Peering Control Room (Unjoin)

This continuous update means that, at every moment, we can check the peering status reading the ForeignCluster CR. For each related resource, there is a field indicating the presence of that resource and where it is stored (i.e. the name and the namespace).

```
status:
  incoming:
    peered: true
    peeringRequest:
      name: <NAME>
      namespace: <NAMESPACE>
  network:
    localNetworkConfig:
      available: true
      reference:
        name: <NAME>
        namespace: <NAMESPACE>
    remoteNetworkConfig:
```

```
14        available: false
15      tunnelEndpoint:
16        available: false
17    outgoing:
18      peered: false
```

5.4.2 PeeringRequest

The PeeringRequest CR represents a request that the home cluster sends to a foreign cluster to start a peering. The existence of that resource means that a cluster is available for the peering, if the other cluster is also available to establish a connection, the request is accepted and the peering established.

Enabler for the Resource Sharing

When the remote cluster creates a new PeeringRequest on the local one, the home cluster has to accept this request to start the resource sharing. This can be done in two ways:

- **Automatically**

  This is the default option, when the home cluster receives a new request this is automatically accepted, this is not a security issue because the remote cluster was previously authenticated.

- **Manually**

  Every coming request will be in *pending* status until a manual action (or an action external to the Liqo control plane) accepts it changing its status to *accepted*.

Broadcaster Creation

When the PeeringRequest is accepted the PeeringRequest Operator will create a new Kubernetes Deployment containing the Advertisement Broadcaster that will manage the resources shared with the remote cluster.

On the new deployment is set an owner reference, in that way when the remote cluster needs to finish the peering, it is simple as deleting the PeeringRequest, its deletion will trigger the Broadcaster deployment deletion.

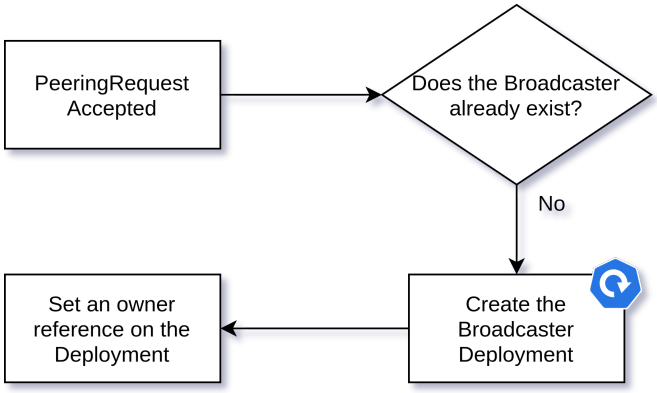


Figure 5.6: Broadcaster Creation

5.4.3 SearchDomain

The SearchDomain resource is required for the discovery on Wide Area Networks. With this resource, we provide a domain where to find a list of clusters available for the peering.

DNS Registry for Clusters in an organization

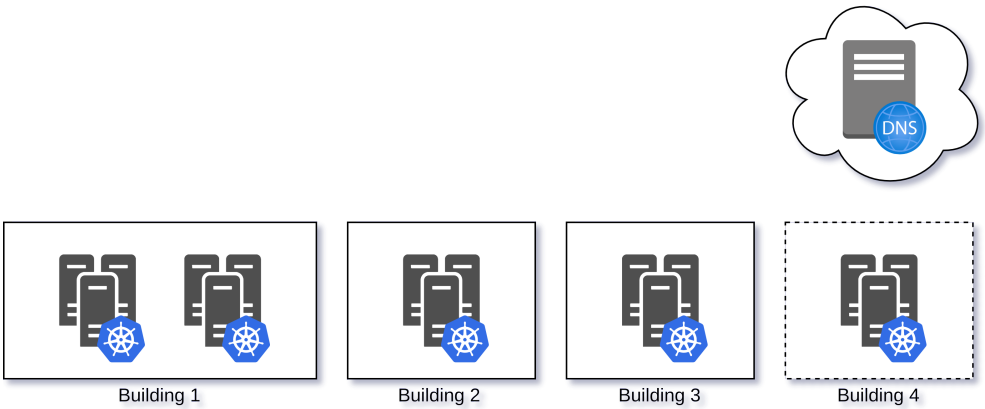


Figure 5.7: Search Domain Registry

In a company, we can have multiple clusters in multiple buildings distributed over different geographic regions. With the SearchDomain CR, this company can have a DNS PTR record containing all these clusters. When the company adds a new cluster, it can automatically discover all the other clusters.

Automatic Discovery and Deletion watching the DNS records

The DNS registry is periodically contacted to look for changes, both for new clusters that have been added and for clusters no more available.

For example, we can have a SearchDomain CR containing the company domain (in the following example *liqo.mycompany.com.*), when the list inside that PTR record changes, the list of created ForeignCluster CRs will change accordingly.

```
apiVersion: discovery.liqo.io/v1alpha1
kind: SearchDomain
metadata:
  name: mycompany
spec:
  domain: "liqo.mycompany.com."
  autojoin: false
```

In the example in Figure 5.7 when the cluster in the *Building 4* is added, a new entry in the DNS record will appear and in each cluster, a new ForeignCluster CR will appear too.

By default, the newly discovered foreign cluster will be peered according to the settings specified in the ClusterConfig CR[4], but settings the *autojoin* flag in this resource we can override the default behavior for our company's clusters.

[4]This is a Liqo CRD where are stored all the settings related to the local Liqo installation. They can be changed at runtime by the user and all the Liqo components will accept the new settings.

Chapter 6

Liqo Discovery Component

The Liqo Discovery Component has several goals, the main ones are:

- **Remote Clusters Discovery**: is there clusters available for peering?
- **Remote Clusters Garbage Collection**: are some of them no more available?
- **Starting the Peering Process**: ask to a remote cluster to share a part of its resources
- **Starting the Unpeering Process**: tell to a remote cluster that its resources are no more needed

6.1 Discovery Methods

The discovery process is in charge to take information from different sources, combining them, and creating a new ForeignCluster CR. Depending on the source of information, it can periodically check that the collected information is up to date, if not it can update the previously created ForeignCluster CR. In both cases we have to update the *lastUpdate* time, that will be used by the garbage collector.

The schema in Figure 6.1 can have many different implementations following it, with different data sources, with different technologies, and for different scenarios.

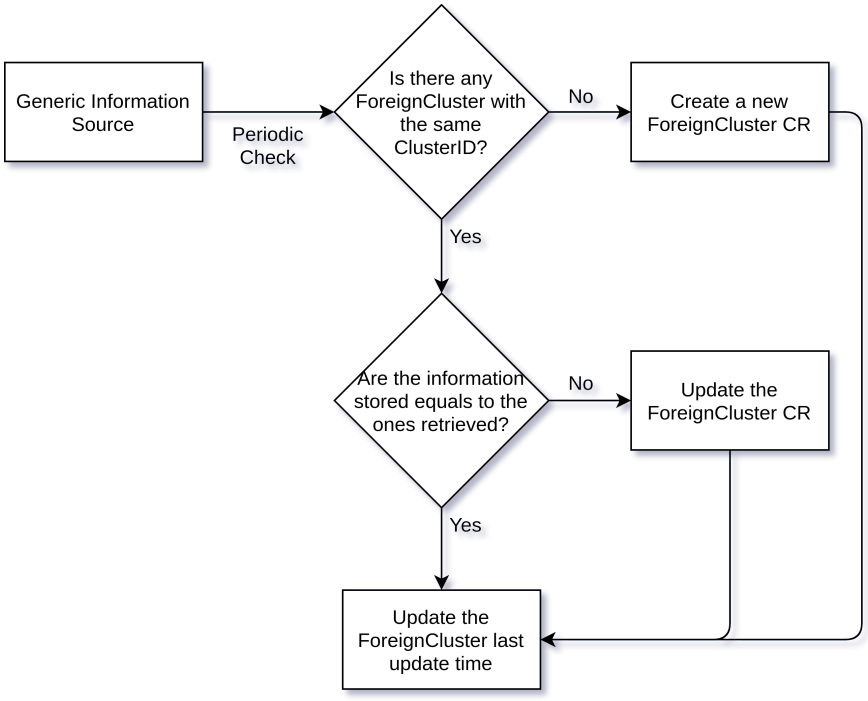


Figure 6.1: Generic Discovery Process

6.1.1 LAN

The first scenario is the discovery of clusters reachable in a LAN. This scenario can be achieved with a *Multicast DNS Service Discovery* over a specific service.

In this scenario, when a new Liqo instance starts (when the Discovery Component starts) creates three different execution threads (Figure 6.2):

- an mDNS server that registers itself to that multicast domain (configurable in the *ClusterConfig CR* to have multiple groups over the same LAN) and will answer to a discovery request from other clusters.

- an mDNS client that at a bootstrap time sends a discovery request and then starts listening for the answers.

- a thread that periodically sends a multicast message to inform other clusters to be still alive and running.

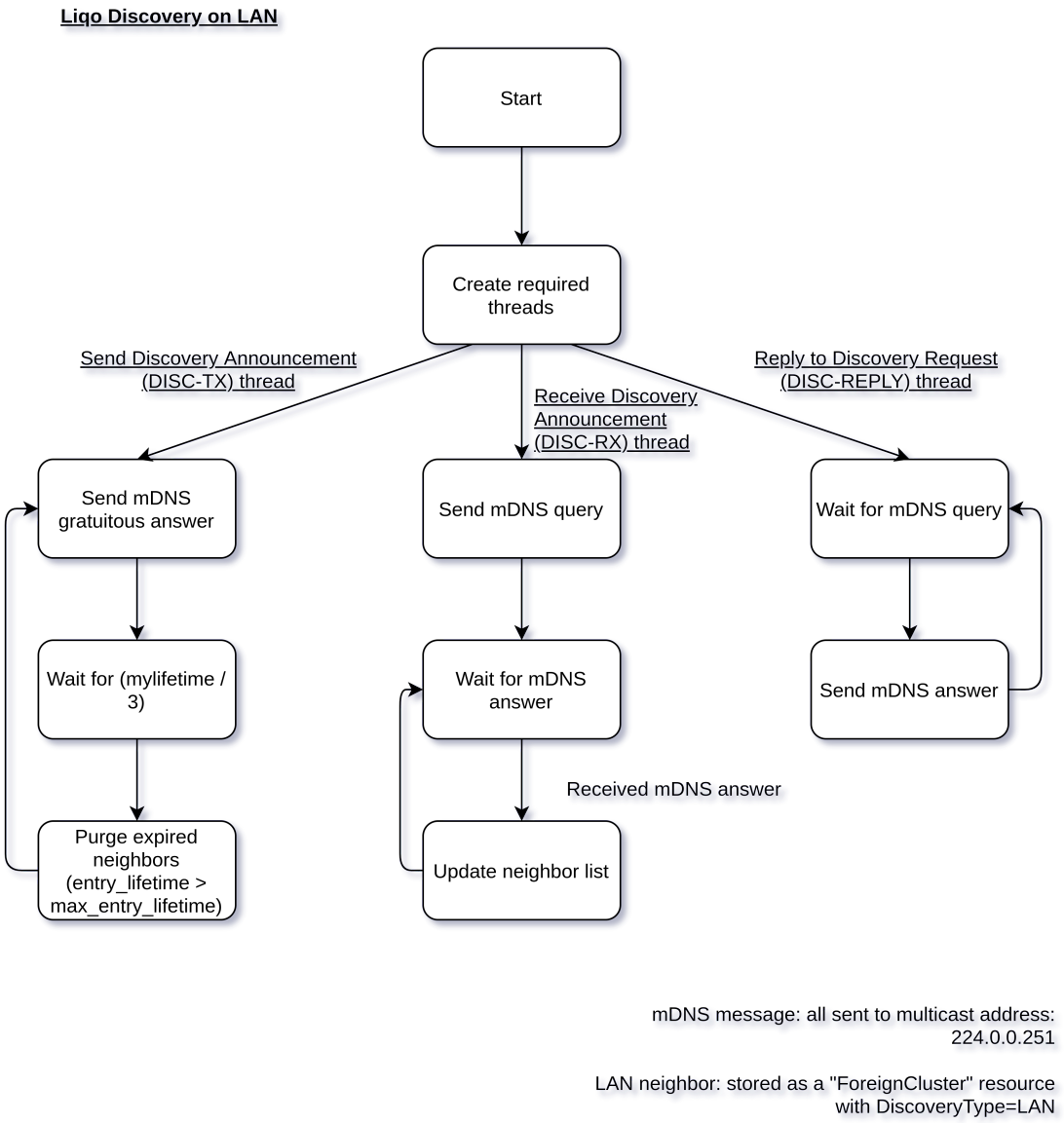


Figure 6.2: LAN Discovery Threads

The Discovery Component leverage on the mDNS service discovery to find the remote Authentication Service, and retrieves the information about the foreign clusters (see Chapter 7), this information is stored inside the new ForeignCluster CR (one for each discovered cluster). In addition, the label *discovery.liqo.io/discovery-type* is set to *LAN* over the CR to allow us to know its origin. In this way the list of discovered clusters is filterable by discovery type.

How is the Pod reachable?

The mDNS multicast packets have to be sent to the *224.0.0.251* multicast address at the port *5353*. Note that routers will not forward these packets outside the original LAN, hence the process that is sending and receiving them has to be in the main network namespace of the host.

To do that, in Kubernetes, we have to set in the Pod the **hostNetwork** flag to *true*. The process in this container will be isolated as usual in every Linux namespace except for the network namespace. In Figure 6.3 we can see in green the LANs where the multicast packets can be forwarded, only in the second scenario we are able to forward the packets between the hosts.

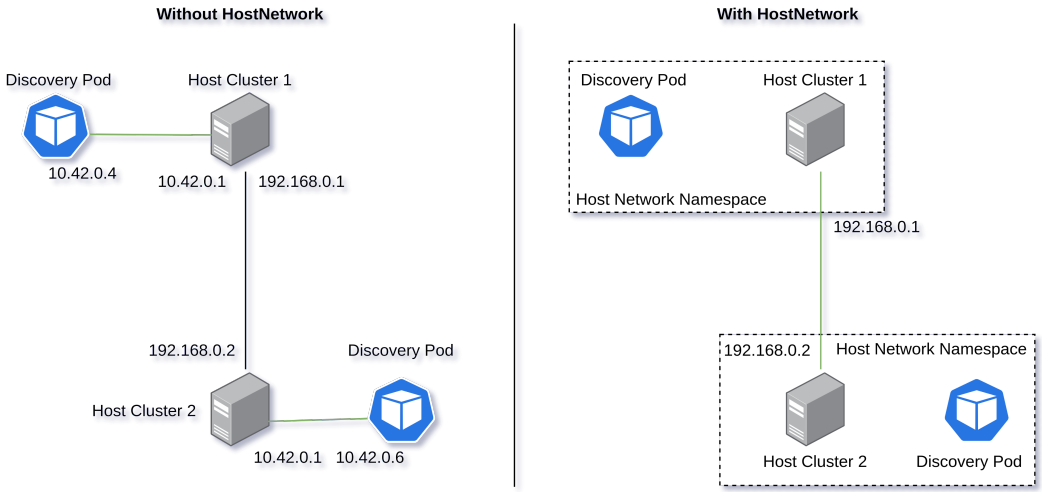


Figure 6.3: Pod Host Network

6.1.2 WAN

In a second scenario we can have multiple clusters, geographically distributes, that need to connect each other, or a separate pool of clusters that need to connect to another one.

For example, we can take a university that has multiple clusters distributed on multiple campuses and over different LANs. They need a simple way to interconnect their clusters. At the same time, there can be some students that need more resources and to connect to the university's clusters at their home. (Figure 6.4)

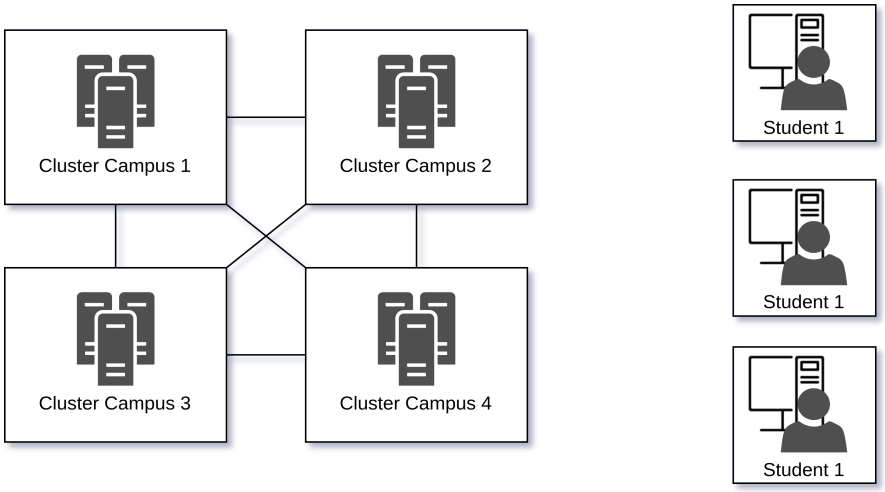


Figure 6.4: WAN Scenario Example

In this example, we can have a DNS server that is serving us a DNS registry for our clusters with a set of records like this:

```
example.com.                          PTR     campus-1.example.com.
                                              campus-2.example.com.
                                              campus-3.example.com.
                                              campus-4.example.com.

campus-1.example.com.       SRV       0 0 443 auth.campus-1.example.com.
campus-2.example.com.       SRV       0 0 443 auth.campus-2.example.com.
campus-3.example.com.       SRV       0 0 443 auth.campus-3.example.com.
campus-4.example.com.       SRV       0 0 443 auth.campus-4.example.com.

auth.campus-1.example.com.          A         1.2.3.4
auth.campus-2.example.com.          A         2.3.4.1
auth.campus-3.example.com.          A         3.4.1.2
auth.campus-4.example.com.          A         4.3.2.1
```

Applying a *SearchDomain CR* in each cluster, both campus ones and students ones, (as described in the Chapter 5 in the paragraph *SearchDomain*) we are able to discover them in a very easy way.

```yaml
apiVersion: discovery.liqo.io/v1alpha1
kind: SearchDomain
metadata:
  name: myuniversity
```

```
spec:
  domain: "example.com."
  autojoin: false
```

This discovery kind is simpler than the LAN discovery because the Liqo Control Plane has not to care about the DNS server, it only has a DNS client to resolve PTR and SRV records. After a configurable amount of time, the Discovery Component sends a new query for these DNS records to check that the information is still valid.

The Discovery Component retrieves the information about the foreign cluster form the remote Authentication Service endpoint (see Chapter 7), and stores this information inside the new ForeignCluster CR. The label *discovery.liqo.io/discovery-type* is set to *WAN* over the CR to allow us to know its origin. In addition, it sets an owner reference over it pointing to the *SearchDomain CR* to allow the garbage collection by the default garbage collector of Kubernetes when the parent CR will be deleted.

6.1.3 Manual

The *Manual* discovery method is useful for testing purposes, or small installations, because we have the least possible requirements: we don't need a single LAN, we don't need a DNS server or an own domain where to register our clusters.

The discovery of a remote cluster is simple as the creation of a *ForeignCluster CR* containing its authentication URL. The Discovery Component will contact this URL to get the cluster information that will be stored in the ForeignCluster CR. Due to the nature of the data source, there is no data periodic check over time.

```
apiVersion: discovery.liqo.io/v1alpha1
kind: ForeignCluster
metadata:
  name: my-cluster
spec:
  authUrl: "https://auth.server.example.com.:443"
```

6.1.4 IncomingPeering

The last available discovery method is the *IncomingPeering*. This method allows us to discover the clusters that are sending us a *PeeringRequest*.

We have a special field in each incoming *PeeringRequest* that indicates the authentication URL for the origin cluster. We can use that information to retrieve its cluster ID and compare it with the list of already known clusters. If it does not

exist we can create a new *ForeignCluster CR* for it. This resource will have a label *discovery.liqo.io/discovery-type* set to *IncomingPeering*.

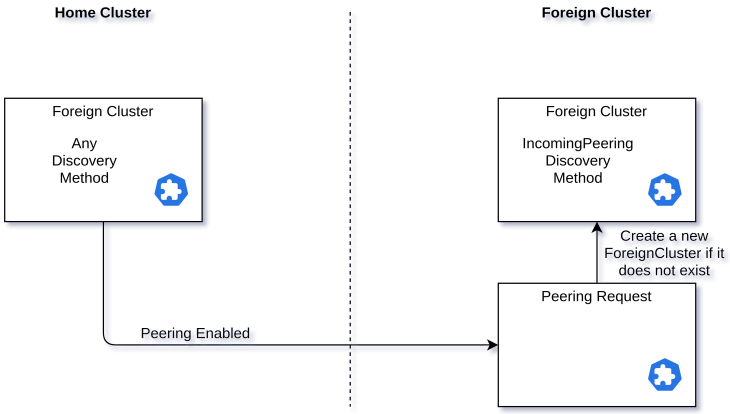


Figure 6.5: IncomingPeering Scenario

If a cluster discovered in this way has no more active peering (neither incoming nor outgoing), the *ForeignCluster CR* will be deleted. The idea is that this cluster was in our database only because of the peering, and, when this source of data is gone, we can no more check the validity of these data.

For this reason, the *IncomingPeering* discovery method is a **weak** method, and it is overwritable when we will discover the same cluster with a stronger method. We will change the *discoveryMethod* field inside the CR and the *discovery.liqo.io/discovery-type* label with the new value.

6.2 Garbage Collection

The Garbage Collection is another functionality of the Liqo Discovery Component. It is in charge to delete the ForeignCluster CRs that were not successfully updated before the time set in their Time To Live (TTL) field.

6.2.1 ForeignCluster Time to Live

With some kind of data sources (i.e. mDNS and DNS) we can have a TTL provided by the protocol chosen: in both the DNS packets have this field that is normally used to express the amount of time that the response will be valid. So we will use the information obtained from this packet (the ForeignCluster specifications) for this amount of time. If the discovery will not be refreshed the data will be considered no more valid and the ForeignCluster CR deleted.

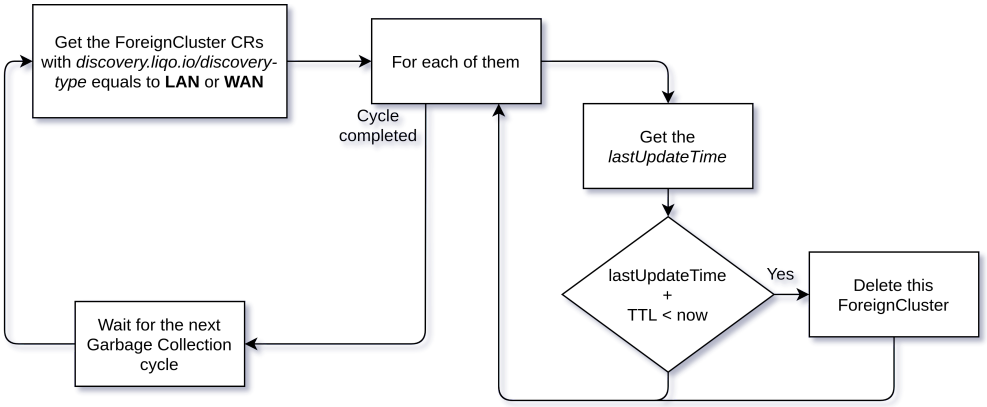


Figure 6.6: Garbage Collection

The Garbage Collector schedules a cycle every a configurable amount of time, in each cycle it follows the steps:

1. Get the list of clusters discovered through the LAN or the WAN method.
2. For each of them:
   (a) Get the *lastUpdateTime*, this is an annotation that the discovery module has to update each time it retrieves the information from its data source.
   (b) Check if the TTL is expired, if so delete this ForeignCluster CR.
3. Wait for the next Garbage Collection cycle.

6.3 Peering

The Discovery Component is in charge to start and monitor the peering process. When a user sets the *join* flag in a ForeignCluster CR, the related Operator reacts to this change starting the peering process.

It creates a PeeringRequest CR in the remote cluster if it is not existing yet, this resource has to contain the local Authentication URL required to create a new ForeignCluster in the remote cluster with the IncomingPeering discovery method.

In the remote cluster, other components will process this request, and a new *Advertisement CR* will be created in our home cluster containing the amount of resources that the remote cluster can share to the home cluster. This new Custom Resource will have an *ownerReference*[1] on it that will trigger the reconciliation in

[1]The *ownerReference* is a Kubernetes functionality that allows us to set an owner for a

the ForeignCluster Operator, it will check the new CR and set the *joined* flag in the *ForeignCluster Status* notifying the successful peering.

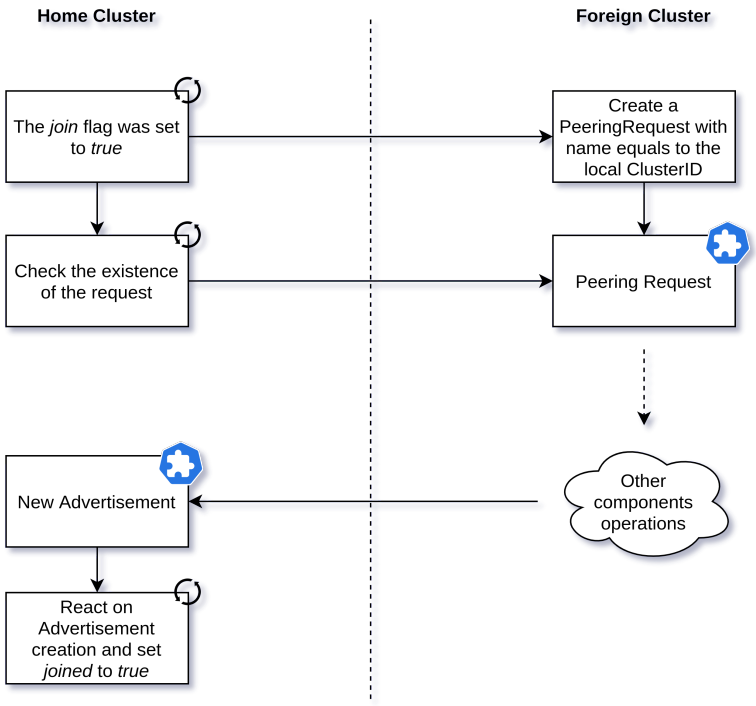


Figure 6.7: Peering Process

6.4 Unpeering

Specularly to the Peering process we have the Unpeering process. This can be triggered by the user setting the *join* flag to *false*, or by the foreign cluster scheduling the *Advertisement* for the deletion. When this flag is set to false, it triggers the deletion of the foreign *PeeringRequest*.

In addition, the Discovery Component adds a *Finalizer*[2] over each *ForeignCluster* resource with an active outgoing peering, in this way we are able to trigger the unpeering with that cluster when the resource has been deleted for any reason (by the user, for an expired TTL, ...).

---

particular resource. When the owner will be deleted also the child one will be. When child one is updated, the owner can be notified and triggers a reconciliation cycle.

[2]The *Finalizer* is a Kubernetes functionality that allows us to be notified when a resource will be deleted and to implement asynchronous pre-delete hooks.

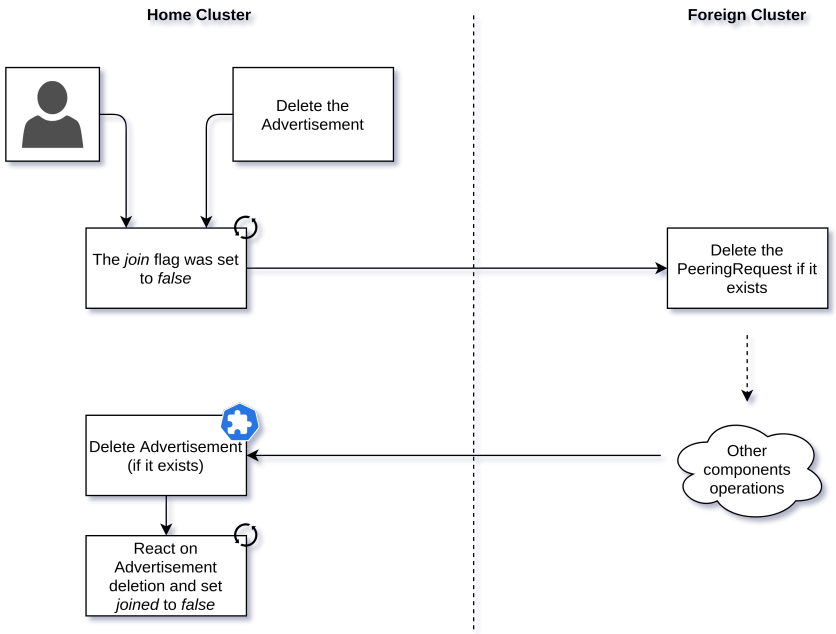


Figure 6.8: Unpeering Process

Chapter 7

Liqo Auth Component

When a process, both in a Pod or external to the cluster, has to contact the Kubernetes API Server, it needs an **Identity** (i.e. a *ServiceAccount*) with the associated **Permissions** (i.e. *Role*, *ClusterRole*).

The Auth Component is in charge to provide an Identity for the remote clusters that need to contact the home API Server for the first time. This component design is focused on four key points:

1. Check that who is asking us a new Identity is allowed to do that.
2. The new Identity has to have the least possible permissions.
3. Create one and only one Identity for each clusterID.
4. Keep the Identity confidential.

7.1 Authentication Methods

The first step is to authenticate who is asking us for a new Identity. We can have different scenarios with different Authentication Methods. The current design supports two of them:

- **Empty Token**: everyone knows the Auth Service URL can have an Identity, this is a no-authentication method.
- **Token Matching**: everyone knows the Auth Service URL and a secret token can have an Identity.

But other methods, like a private/public key authentication, can be added easily in a future work.

7.1.1 Empty Token

The Empty Token modality is with no-authentication and it is useful for testing purposes or if we have to expose a public resource sharing service where everyone can access and ask for resources.

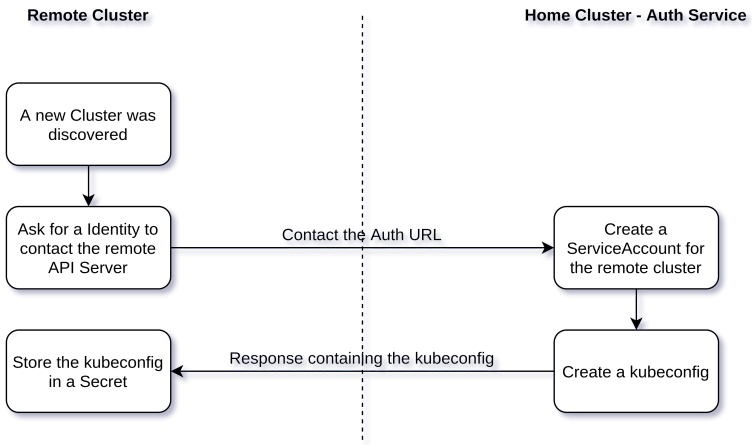


Figure 7.1: Empty Token

With this configuration, the Auth Service will always create a new Identity and a new kubeconfig will be returned to the cluster that is asking for it.

7.1.2 Token Matching

The Liqo Auth Component supports a basic authentication mechanism that consists of a secret token comparison, if the token provided by the remote client is equal to the one stored in a local Secret, it means that it can ask for an Identity. Note that this secret token has to be kept **confidential** and it has to be distributed **out-of-band**.

This can be useful to restrict access to the resources to a limited set of trusted clusters. We will not be able to revoke the access for a specific cluster, because the token is universal, if we change the token we have to share again the new one with all the clusters that need it.

This implements two levels of authentication: the first with the token is required to get the real identity that will be used in the second to contact the API Server. While the first is not authenticating uniquely a cluster, but a set of clusters, the second one is cluster-specific.

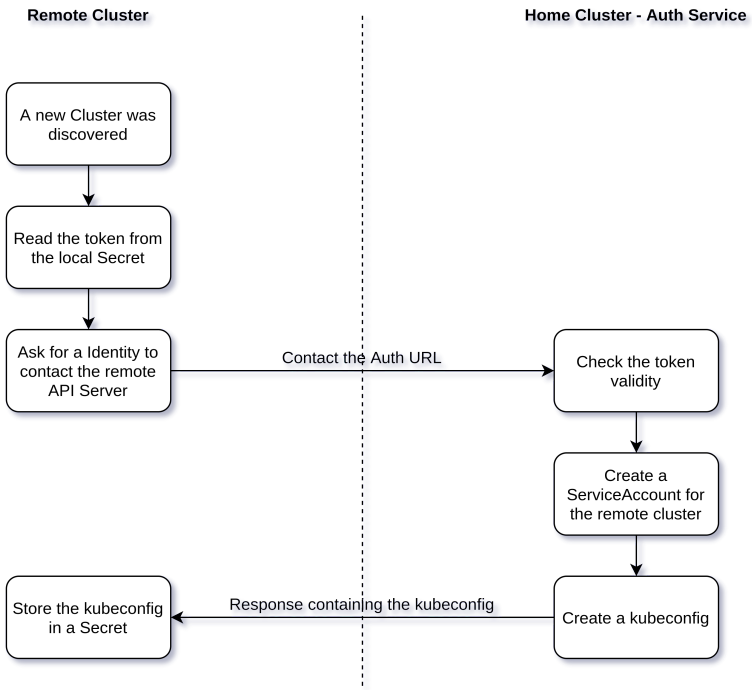


Figure 7.2: Token Authentication

Token Management

At the bootstrap time, the Auth Component checks for the existence of a Secret *auth-token*, if it does not exist or if it is empty, it creates a new random token and stores it in that Secret. At any time the user can modify it, this change will take effect for all the future requests since that moment.

A cluster Admin can get the token from this Secret with a simple command:

```
token=$(kubectl get secret -n liqo auth-token -o jsonpath="{.data.
    token}" | base64 -d)
echo -e "Token:\t$token"
```

On the remote cluster we have to create a new Secret containing the token:

```
kubectl create secret generic "token-for-remote-cluster" \
  -n liqo \
  --from-literal=token="$token"
```

Then, we have to label it, to tell to Liqo that it is an Auth Token and the clusterID of the cluster that it is referencing:

```
kubectl label secret "token-for-remote-cluster" \
  -n liqo \
   discovery.liqo.io/cluster-id="$clusterId" \
   discovery.liqo.io/auth-token=""
```

7.2 Service Account Management

The main goal of the Authentication Service is to provide an Identity to the remote cluster, to do so it has to manage the Service Account life cycle.

7.2.1 Creation

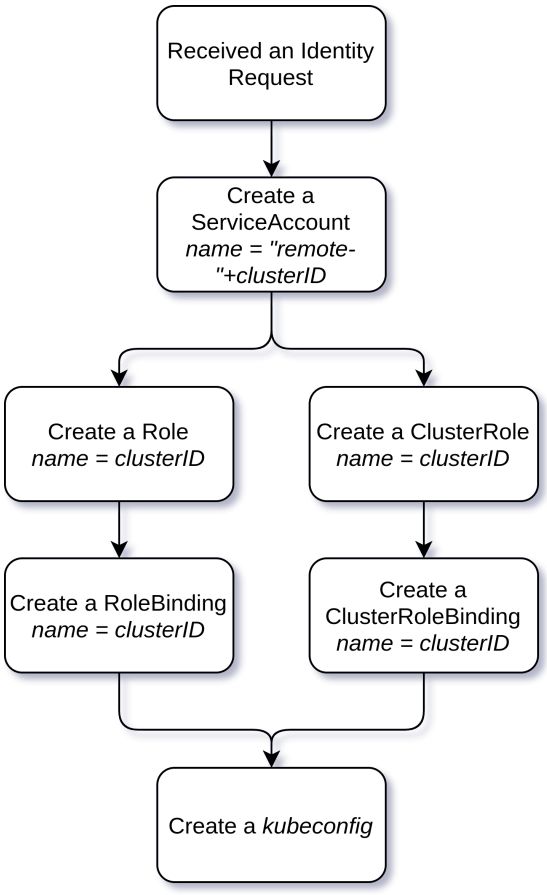


Figure 7.3: ServiceAccount Creation

After that the Authentication Service knows that the client can get an Identity, it has to create a *ServiceAccount* related to its *ClusterID*. The new resource will be created with the name equals to the remote ID, in that way it's simpler to check and refuse any future attempts to recreate the same Identity.

Figure 7.3 shows that when the **ServiceAccount** is available, we have to create several other Kubernetes resources to assign to this new identity the minimal permissions required to establish a new peering. In particular, we need to create a new **Role** to grant the permissions on namespaced resources and a **ClusterRole** to grant the permissions on cluster scoped resources, then we need to connect them with the ServiceAccount creating a **RoleBinding** and a **ClusterRoleBinding** respectively.

In this example we can see how they are linked together:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: remote-clusterID
  namespace: liqo
secrets:
- name: remote-clusterID-token-hjwj7
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: remote-clusterID
  namespace: liqo
rules:
  ...
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: remote-clusterID
  namespace: liqo
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
```

```
  name: remote-clusterID
subjects:
- kind: ServiceAccount
  name: remote-clusterID
  namespace: liqo
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: remote-clusterID
rules:
  ...
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: remote-clusterID
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: remote-clusterID
subjects:
- kind: ServiceAccount
  name: remote-clusterID
  namespace: liqo
```

When all these resources are ready, the Authentication Service will create a new kubeconfig containing the ServiceAccount token (the credentials to be authenticated by the API Server) and the API Server URL. This file is returned as the answer to successful identity requests, it will not be returned in any other case allowing us to keep its content confidential.

7.2.2 Permissions

Now, we have to understand which are these "minimal permissions" required to start the Liqo peering process.

Kubelet TLS Bootstrapping

In Kubernetes there is a quite similar scenario where an external device (a new node) wants to join an already existent cluster, this is the *Kubelet TLS Bootstrapping*.

In the bootstrap initialization process, the following occurs: [23]

1. kubelet begins
2. kubelet sees that it does not have a kubeconfig file
3. kubelet searches for and finds a bootstrap-kubeconfig file
4. kubelet reads its bootstrap file, retrieving the URL of the API server and a limited usage "token"
5. kubelet connects to the API server, authenticates using the token
6. kubelet now has **limited credentials to create and retrieve a certificate signing request (CSR)**
7. kubelet creates a CSR for itself with the signerName set to kubernetes.io/kube-apiserver-client-kubelet
8. CSR is approved in one of two ways:
   - If configured, kube-controller-manager automatically approves the CSR
   - If configured, an outside process, possibly a person, approves the CSR using the Kubernetes API or via kubectl
9. **Certificate is created for the kubelet**
10. Certificate is issued to the kubelet
11. **kubelet retrieves the certificate**
12. kubelet creates a proper kubeconfig with the key and signed certificate
13. kubelet begins normal operation

The key idea in this process is to give small privileges to someone external to the cluster, with these privileges it is able to ask for a new identity with full permissions.

Liqo Bootstrapping

With the Liqo Bootstrapping process, a remote cluster needs to have limited credentials too, but with a different goal: it does not have to authenticate as a node of the cluster, but as a service in order to manage the Liqo control plane.

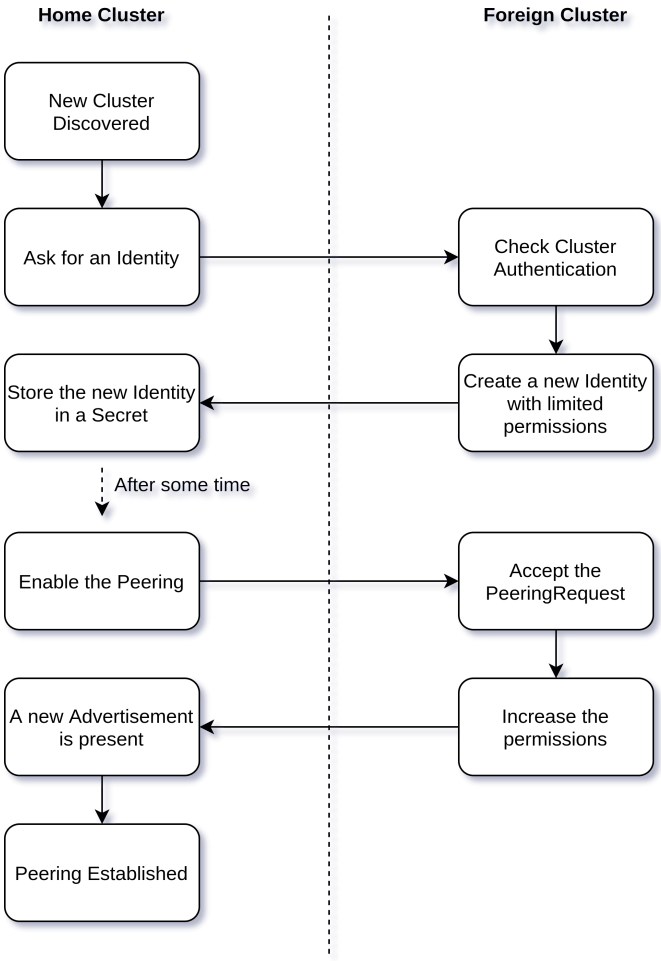


Figure 7.4: Permission Escalation During the Peering Process

In the Liqo bootstrapping process, the following occurs:

1. a new cluster is discovered
2. the home cluster sees that it does not have a kubeconfig file to contact this remote cluster
3. the home cluster contacts the remote AuthUrl

4. the home cluster authenticates to the remote cluster as described in *Authentication Methods*

5. the home cluster now has **limited credentials to create and manage PeeringRequests**

6. the remote cluster **creates a new ServiceAccount for the home cluster**

7. the **home cluster retrieves the certificate**

8. the home cluster has permissions to establish the peering

We can see very similar key points, starting from completely untrusted devices and arriving to trusted devices able to operate. At this point, the cluster is able to start the peering process. When the peering will be required and accepted by both sides, new permission will be granted to extends the previous one and to make the virtual node work.

But which are these required limited permissions?

First of all, we need the permissions to manage our (and only our) PeeringRequest, a cluster scoped resource, so in the ClusterRole, we will have the following rules:

```
...
rules:
- apiGroups:
  - discovery.liqo.io
  resourceNames:
  - clusterID
  resources:
  - peeringrequests
  verbs:
  - get
  - create
  - delete
  - update
```

Setting the *resourceNames* field, we can enforce the name of the resources that a remote cluster can manage to be equal to the remote cluster ID, this allows us to have multi-tenant complete support and isolation during the peering process (*Figure 7.5*).

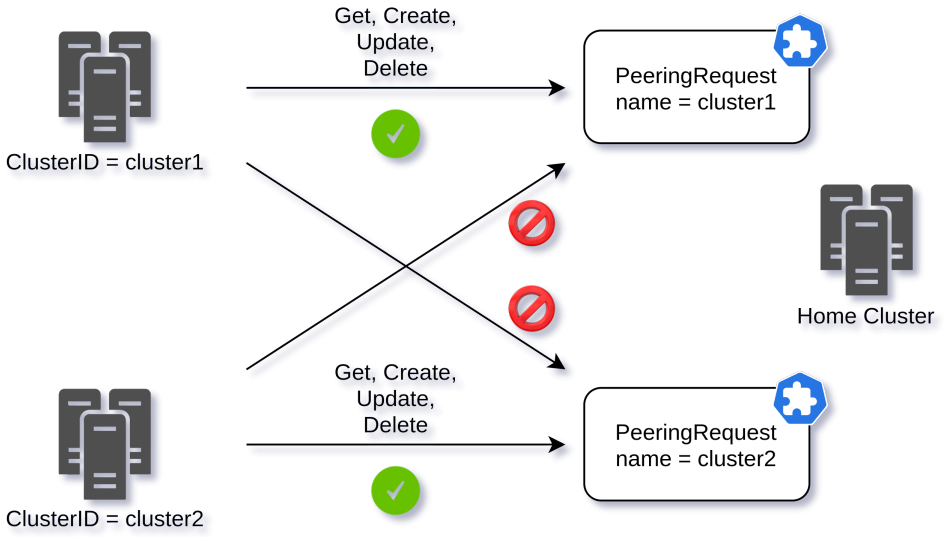


Figure 7.5: Multi-Tenant Peering

Alongside the PeeringRequest we also need to create a Secret containing the credentials required to the remote cluster to contact us back (if it accepts our peering offer) and to create a new Advertisement in the home cluster, then we need a similar mechanism for the Secrets too.

```
...
rules:
- apiGroups:
  - ""
  resourceNames:
  - clusterID
  resources:
  - secrets
  verbs:
  - get
  - create
  - delete
```

7.3 Observability

At every moment, we can check if our local cluster is authenticated with a remote one looking at the value contained in the status of the related ForeignCluster resource.

In fact, in the ForeignCluster status, we have a *AuthStatus* field containing the observed authentication state.

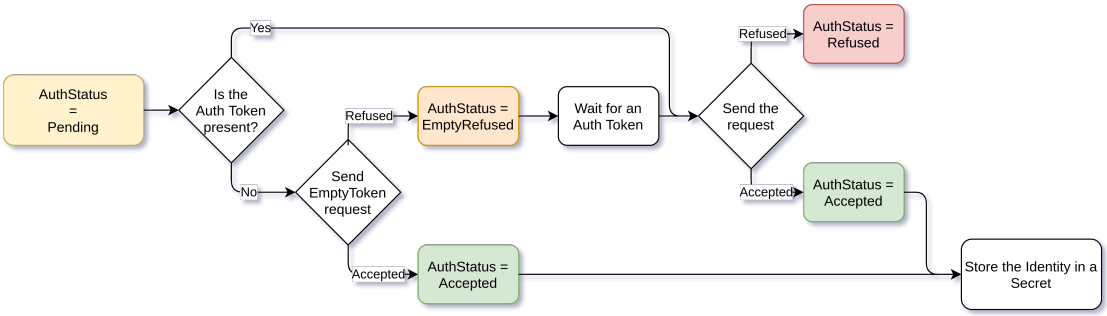


Figure 7.6: AuthStatus Flow Chart

When a new ForeignCluster is created in the local cluster, the ForeignCluster Operator will try to get an identity to contact this specific remote cluster. There can be four different possible values:

- **Pending**: the local cluster has not contacted the remote cluster yet. This is the default value.
- **Accepted**: the local cluster successfully contacted the remote cluster and there is an available identity.
- **EmptyRefused**: the local cluster sent a request with an empty token, but it was refused.
- **Refused**: the local cluster sent a request with the provided token, but it was refused.

If in the local cluster there is no secret containing a token for the cluster that we need to authenticate with, it assumes that the remote cluster is allowing us to have an identity with no previous authentication (empty token). If this request is refused, it will wait for an available token to send a new authenticated request.

7.4 How to Access it

The Liqo Authentication Service, because of its purpose, has to be exposed to the external of the cluster, to be reachable from other clusters.

The most used approach to expose an API in Liqo is to use CRDs. In this case is not a viable solution, because to use CRDs we need to authenticate with the API Server but we need the credentials that we will have at the end of the Authentication

process only. Note that even if Kubernetes offers us the feature[1] to assign to a particular user with username *system:anonymous* and group *system:unauthenticated* every request that comes with no authentication, this solution cannot give us the required isolation between different tenants. In fact, if we imagine a CRD that handles the Identity Request in this way, we are not able to keep these requests and the related responses confidential.

A viable solution is to expose it as a standalone HTTP service, in this way we can implement a service reachable without a previous authentication and that can implement the request/response mechanism required to have the multi-tenant isolation and to keep these identities confidential.

This service will create and return a kubeconfig, when the authentication is confirmed. An example request can be like this:

```
{
  "clusterID": "<ClusterID>",
  "token": "<TOKEN>"
}
```

7.4.1 Secured with TLS

This HTTP service needs to be protected against *Man In The Middle* or *Network Sniffing* attacks, where an attacker can stole the token in the request and use it again to have another Identity, or stole the kubeconfig in the response and then impersonate the victim. For that reason, it has to be exposed with HTTPS to satisfy the confidentiality requirement.

[1]The Anonymous Request feature needs to be enabled in some Kubernetes distribution (as in K3s), and is not available at all on many managed Kubernetes solution (as GKE, AKS, EKS).

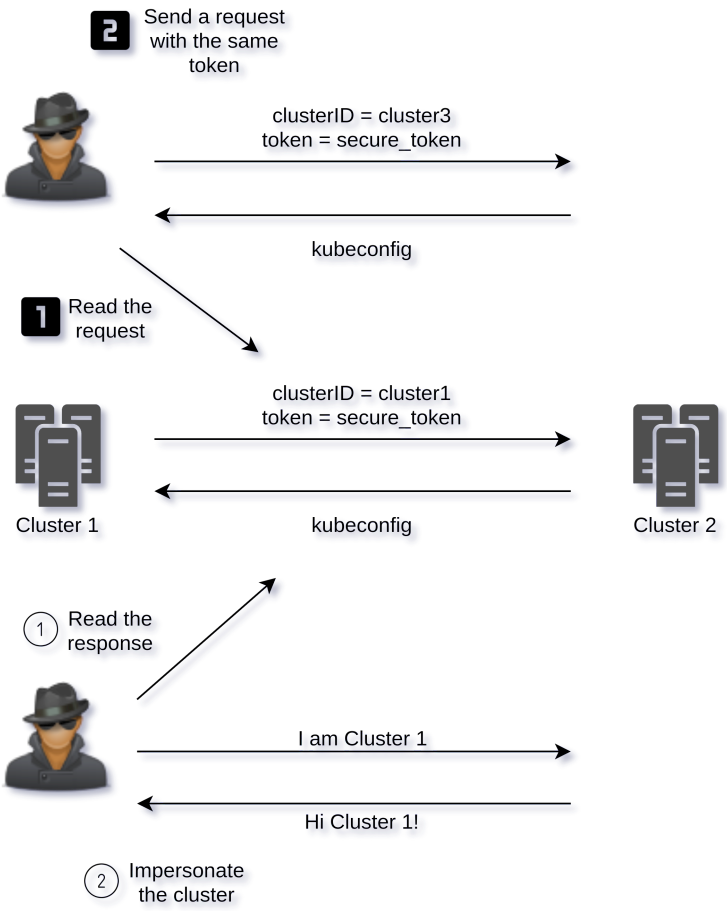


Figure 7.7: Authentication Attacks

7.4.2 NodePort

The simplest and widely usable way to expose a service with Kubernetes is to use a NodePort Service. With this configuration, it chooses a random port on each host in the cluster that will be mapped to a Service port and then to a set of Pods that is serving the HTTPS server, in this way the traffic is encrypted from the client to the Pod.

The Pod can serve it with a self-signed certificate, protecting against the *Network Sniffing* attacks, or with a provided trusted certificate that will protect against both *Network Sniffing* and *Man in the Middle* attacks.

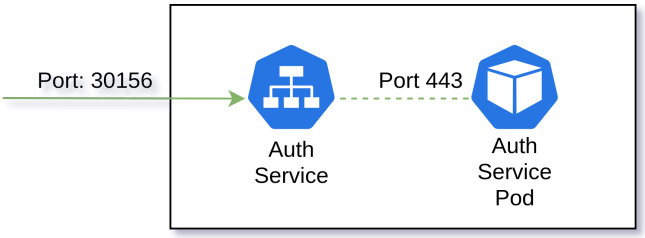


Figure 7.8: Authentication Service with NodePort

7.4.3 Ingress

We have an alternative way to serve it using an Ingress. This solution is the best one when it is available, the cluster needs an Ingress Controller, the cluster admin needs to assign a domain to this service. With solutions as *External DNS* is possible to automatically manage the DNS records to point to the IP address assigned to the Ingress resource, and with solutions as *CertManager* is possible to automatically manage trusted TLS certificates for these resources. This means that we are able to have a fully automated management of our HTTP service.

With this scenario, the traffic will be encrypted from the client to the Ingress Controller, but inside our cluster, it will be simple HTTP traffic. Normally it is not a security problem if we trust our environment and infrastructure, but, when we cannot, it is possible to have a second TLS connection from the Ingress Controller to the Pod setting our controller to do that. This configuration is provider-specific.

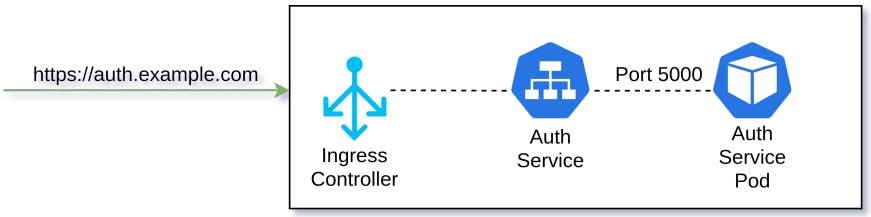


Figure 7.9: Authentication Service with Ingress

Chapter 8

Evaluation

The proposed work has to be validated in different ways. First of all, we need to validate that this approach is viable in as many Kubernetes distributions as possible, and to be able to join them each other, to demonstrate the possibility to use Liqo in widely different scenarios.

The second kind of evaluation will check the time needed to discover other clusters and to peer with them. We will evaluate if the time needed by Liqo is compatible with a scenario where we want to dynamically enable the peering during a workload.

8.1 Tested on various platforms

We tested the implementation validating that is general enough to be run in several different Kubernetes distributions, both on-premise and managed, both designed for the core and the edge. We see that we are able to peer clusters deployed with each of these distributions with all the others. We tested that the discovery is possible and we are able to authenticate to each of them and to create an identity in the proper way, even in the public and managed cloud providers.

In particular, the validation was focused on the following Kubernetes distributions:

- **K8s (kubeadm)**: the vanilla Kubernetes distribution.
- **K3s**: a Kubernetes distribution provided by Rancher, great for IoT and Edge devices thanks to the low resources request.
- **AKS**: the Azure managed Kubernetes service.
- **GKE**: the Google Cloud managed Kubernetes service.

8.2 Time Benchmarking

A second validation is the benchmarking of this solution, characterizing the time required to discover a cluster, to authenticate, and to peer with it, showing how it can be fast enough to allow us to make the peer on the fly when we need it even when our applications are already running. Some instrumentations have been added in the code to export additional metrics that can be collected by Prometheus.

Two different times have been analyzed:

1. The time needed from the discovery of a cluster (the time when a DNS or mDNS packet reached our Pod) to the creation of the ForeignCluster CR.

2. The time between the activation of a new peering and the final creation of a new node.

In the following tables and graphs, all the times are expressed in milliseconds. In the graphs in Figure 8.1, 8.2 and 8.3 use a logarithmic scale on the y axes (the time axes). We can see the maximum and the minimum value observed for each metric, the filled area represents the range where the 60% of values is.

8.2.1 Foreign Cluster Discovery

First of all, we want to check the time required by the Liqo control plane from the reception of an mDNS packet to the complete creation of a ForeignCluster CR, with all the required information and with a valid identity to be used on the remote cluster ready for the peering.

This process has been divided into four sub-steps:

- **mDNS Parsing**: the time required to parse and process the information contained in the packet.

- **Get Cluster Info**: the time required to contact the remote cluster and to retrieve its information.

- **Create Foreign Cluster**: the time required to create the new resource with the collected information.

- **Retrieve Identity**: the time required to contact the remote cluster and to get a new identity from it.

The metrics have been collected with two local KinD clusters (over the same LAN), with the automatic discovery enabled. The entire Discovery process has been repeated several times. Two different scenarios have been taken: the first is the discovery of a cluster that never met before, leading to the creation and the

sharing of a new Identity; the second is the discovery of a cluster that was already discovered in the past and then forgotten, in that scenario the identity is already present in the local cluster and we expect a shorter discovery time.

No Available Identity

Metric	Time 1	Time 2	Time 3	Time 4	Time 5	AVG	STD DEV	STD DEV %
mDNS Parsing	3	3	0	3	6	3	2,121	70,710
Get Cluster Info	797	308	262	263	213	368,6	241,829	65,607
Create Foreign Cluster	1	2	1	1	1	1,2	0,447	37,267
Retrieve Identity	307	200	308	222	216	250,6	52,562	20,974
Total Time	1108	513	571	489	436	623,4	275,205	44,145

Table 8.1: The time is expressed in ms (milliseconds). Some statistics have been collected on this data, like the average (AVG), the standard deviation (STD DEV), and the standard deviation in relation to the average (STD DEV %).

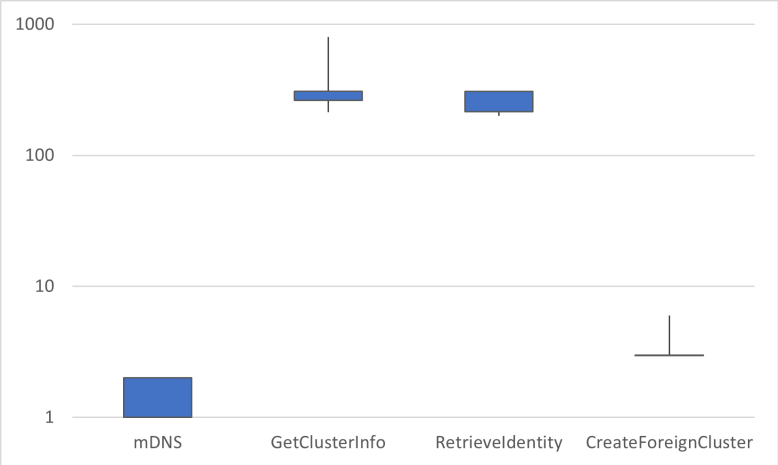


Figure 8.1: No Available Identity (Logarithmic Scale)

Already Available Identity

Metric	Time 1	Time 2	Time 3	Time 4	Time 5	AVG	STD DEV	STD DEV %
mDNS Parsing	7	0	3	3	0	2,6	2,880	110,806
Get Cluster Info	546	267	274	289	221	319,4	129,190	40,448
Create Foreign Cluster	5	0	1	0	1	1,4	2,0736	148,117
Retrieve Identity	0	0	0	0	0	0	0	0
Total Time	558	267	278	292	222	323,4	133,741	41,354

Table 8.2: The time is expressed in ms (milliseconds). Some statistics have been collected on this data, like the average (AVG), the standard deviation (STD DEV), and the standard deviation in relation to the average (STD DEV %).

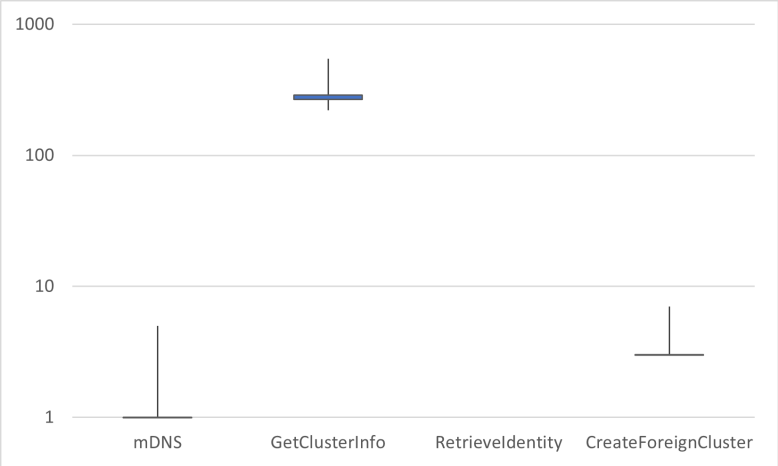


Figure 8.2: Available Identity (Logarithmic Scale)

As expected, in the two scenarios we have a comparable amount of time in all the phases, except for the Identity retrieval, which in the second one is faster thanks to the fact that it is already present in the local cluster.

8.2.2 Peering Process

The second interesting time is the one required by Liqo to establish a new Peering when the user requires it. We are now tracking the time between the activation of a peering and the moment when the new node pointing to the remote cluster will be ready.

This process has been divided into five sub-steps:

- **Start Peering**: the time required to handle the change of the control flag and to create a PeeringRequest in the remote cluster.

- **Process Peering Request**: the time required by the remote cluster to react to the creation of a new PeeringReqeust, to accept it, and to create the Broadcaster Deployment.

- **Advertisement Forging**: the time required by the Broadcaster to forge a new Advertisement.

- **Advertisement Processing**: the time required to process an incoming Advertisement and to create a new VirtualKubelet.

- **Create Virtual Node**: the time required to set up the new node and to make it ready for the Pod offloading.

Metric	Time 1	Time 2	Time 3	Time 4	Time 5	AVG	STD DEV	STD DEV %
Start Peering	25	23	28	28	27	26,2	2,167	8,274
Process Peering Request	10	2	4	3	4	4,6	3,130	68,054
Advertisement Forging	402	718	480	801	607	601,6	164,290	27,308
Advertisement Processing	555	21	38	575	447	327,2	276,154	84,399
Create Virtual Node	22700	24200	24400	23200	2020	19304	9687,470	50,183
Total Time	23692	24964	24950	24607	3105	20263,6	9605,892	47,404

Table 8.3: The time is expressed in ms (milliseconds). Some statistics have been collected on this data, like the average (AVG), the standard deviation (STD DEV), and the standard deviation in relation to the average (STD DEV %).

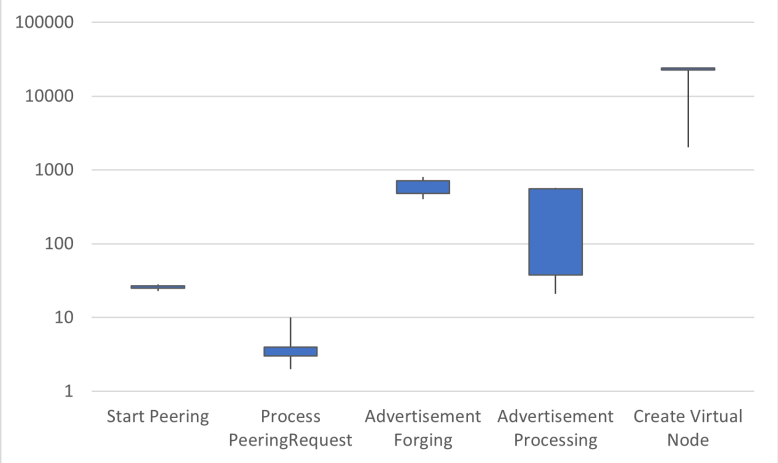


Figure 8.3: Peering Process (Logarithmic Scale)

8.3 Conclusions

This work proposed a mechanism for the automatic discovery of Kubernetes clusters. It has been validated in very different environments, starting from the edge with k3s to the core with managed Kubernetes solutions. It shows that it is possible to use the same APIs in all of them achieving good overall performance that allows us to enable the opportunistic peering that we was looking for.

Future Works

In future works, we aim to define further discovery methods and data sources for the discovery, like already existent clusters, sets created with other solutions (Rancher, JuJu, ...) and to import them into the Liqo environment in a simple way. Other authentication methods have to be implemented, like an RSA private/public key authentication, to have a stronger and per-user identity.

A certificates-driven authentication mechanism can allow us to not to have to create ServiceAccounts in the remote cluster, but the home cluster will create its own identity and a Certificate Signing Request (CSR) that the remote cluster will issue with its Certification Authority granting it an identity to authenticate to the API Server.

Another possible improvement can be the usage of different namespaces to store the resource related to each peering, in this way in a multi-peering scenario the debugging will be simpler. Also the management of permission will be simplified, not requiring anymore access to resources with a specific name but on a specific namespace.

Bibliography

[1] *Kubernetes official documentation*. URL: https://kubernetes.io/docs/home/ (cit. on p. 4).

[2] *Virtual-kubelet git repository*. URL: https://github.com/virtual-kubelet/virtual-kubelet (cit. on pp. 4, 17, 18).

[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 ({USENIX}{ATC}| 14)*. 2014, pp. 305–319 (cit. on p. 8).

[11] *Kubernetes official documentation*. URL: `https://kubernetes.io/docs/home/` (cit. on pp. 11, 13, 15–18).

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

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

[14] *DNS Service Discovery (DNS-SD)*. URL: `http://www.dns-sd.org/` (cit. on p. 20).

[15] S. Cheshire and M. Krochmal. *Requirements for a Protocol to Replace the AppleTalk Name Binding Protocol (NBP)*. RFC 6760. Feb. 2013 (cit. on p. 20).

[16] S. Cheshire and M. Krochmal. *DNS-Based Service Discovery*. RFC 6763. Feb. 2013 (cit. on pp. 20–22).

[17] *Internet of Things: A Survey on Enabling Technologies, Protocols and Applications - Scientific Figure on ResearchGate.* URL: `https://www.researchgate.net/figure/Discovering-print-service-by-DNS-SD_fig3_279177017` (cit. on p. 20).

[18] *Multicast DNS*. URL: `http://www.multicastdns.org/` (cit. on p. 22).

[19] S. Cheshire and M. Krochmal. *Multicast DNS*. RFC 6762. Feb. 2013 (cit. on p. 22).

[20] *Zero Configuration Networking (Zeroconf)*. URL: `http://www.zeroconf.org/` (cit. on p. 22).

[21] F. Borgogni. «Dynamic Sharing of Computing and Network Resources between Different Clusters». M. Eng. thesis. Torino, Italy: Politecnico di Torino, Mar. 2020 (cit. on p. 30).

[22] G. McCance. *CERN Data Centre Evolution*. Nov. 2012. URL: `https://www.slideshare.net/gmccance/cern-data-centre-evolution` (cit. on p. 32).

[23] *TLS Bootstrapping*. URL: `https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-tls-bootstrapping/` (cit. on p. 58).