Liqo

The Liqo Authors

Aug 21, 2023

FEATURES

Liqo is an open-source project that enables **dynamic and seamless Kubernetes multi-cluster topologies**, supporting heterogeneous on-premise, cloud and edge infrastructures.

CHAPTER

ONE

WHAT DOES IT PROVIDE?

Peering

Automatic peer-to-peer establishment of **resource and service consumption relationships** between independent and heterogeneous clusters. No need to worry about complex VPN configurations and certification authorities: everything is transparently **self-negotiated** for you.

Offloading

Seamless **workloads offloading** to remote clusters, without requiring any modification to Kubernetes or the applications themselves. **Multi-cluster is made native and transparent**: collapse an entire remote cluster to a **virtual node** compliant with the standard Kubernetes approaches and tools.

Network Fabric

A transparent **network fabric**, enabling multi-cluster **pod-to-pod** and **pod-to-service** connectivity, regardless of the underlying configurations and CNI plugins. Natively **access the services** exported by remote clusters, and spread interconnected application components across multiple infrastructures, with all cross-cluster traffic flowing through **secured network tunnels**.

Storage Fabric

A native **storage fabric**, supporting the remote execution of **stateful workloads** according to the **data gravity** approach. Seamlessly extend standard (e.g., database) **high availability deployment techniques** to the multi-cluster scenarios, for **increased guarantees**. All without the complexity of managing multiple independent cluster and application replicas.

CHAPTER

TWO

WHAT TO EXPLORE NEXT?

Features

New to Liqo? Would you like to know more? Here you can find an in-depth overview of what a **peering** is, how the *virtual node* abstraction enables **workload offloading**, as well as discover about the **network and storage fabric** subsystems, ensuring the seamless functioning of unmodified multi-cluster applications.

Peering · *Offloading* · *Network Fabric* · *Storage Fabric*

Installation

Ready to give Liqo a try? Learn about installation and connectivity **requirements**, discover how to download and install **liqoctl**, the CLI tool to streamline the installation and management of Liqo, and explore the **customization options**, based on the target environment characteristics.

Requirements · *Liqo CLI tool* · *Install* · *Uninstall*

Examples

Would you like to quickly join the fray and experiment with Liqo? Set up your playground and check out the **getting started examples**, which will guide you through a scenario-driven tour of the most notable features of Liqo. Discover how to **offload** (a subset of) your workloads, **access services** provided by remote clusters, **expose** multi-cluster applications, and more.

Quick Start · *Offloading with Policies* · *Offloading a Service* · *Stateful Applications* · *Global Ingress* · *Replicated Deployments* · *Provision with Terraform*

Usage

Do you want to make a step further and discover all the Liqo configuration options? These guides get you covered! Find out how to **establish and configure a peering** between two clusters, as well as how to enable and customize **namespace offloading**. Explore the details about which and how native resources are **reflected** to remote clusters, and learn more about the support for **stateful applications**.

Peer two Clusters · *Namespace Offloading* · *Resource Reflection* · *Stateful Applications Prometheus Metrics External Network Service Continuity*

CHAPTER

THREE

PEERING

In Liqo, we define **peering** a unidirectional resource and service consumption relationship between two Kubernetes clusters, with one cluster (i.e., the consumer) granted the capability to offload tasks to a remote cluster (i.e., the provider), but not vice versa. In this case, we say that the consumer establishes an **outgoing peering** towards the provider, which in turn is subjected to an **incoming peering** from the consumer.

This configuration allows for maximum flexibility in asymmetric setups, while transparently supporting bidirectional peerings through their combination. Additionally, the same cluster can play the role of provider and consumer in multiple peerings.

3.1 Overview

Overall, the establishment of a peering relationship between two clusters involves four main tasks:

- **Authentication**: each cluster, once properly authenticated through pre-shared tokens, obtains a valid identity to interact with the other cluster (i.e., its Kubernetes API server). This identity, granted only limited permissions concerning Liqo-related resources, is then leveraged to negotiate the necessary parameters, as well as during the offloading process.
- **Parameters negotiation**: the two clusters exchange the set of parameters required to complete the peering establishment, including the amount of resources shared with the consumer cluster, the information concerning the setup of the network VPN tunnel, and more. The process is completely automatic and requires no user intervention.
- **Virtual node setup**: the consumer cluster creates a new **virtual node** abstracting the resources shared by the provider cluster. This transparently enables the task offloading process detailed in the *offloading section*, and it is completely compliant with standard Kubernetes practice (i.e., it requires no API modifications for application deployment and exposition).
- **Network fabric setup**: the two clusters configure their **network fabric** and establish a secure cross-cluster VPN tunnel, according to the parameters negotiated in the previous phase (endpoints, security keys, address remappings, ...). Essentially, this enables pods hosted by the local cluster to seamlessly communicate with the pods offloaded to a remote cluster, regardless of the underlying CNI plugin and configuration. Additional details are presented in the *network fabric section*.

3.2 Approaches

Liqo supports two non-mutually exclusive peering approaches (i.e., the same cluster can leverage a different approach for different remote clusters), respectively referred to as **out-of-band control plane** and **in-band control plane**. The following sections briefly overview the differences among them, outlining the respective trade-offs. Additional in-depth details about the networking requirements are presented in the *installation requirements section*, while the *usage section* describes the operational commands to establish both types of peering.

3.2.1 Out-of-band control plane

The standard peering approach is referred to as **out-of-band control plane**, since the **Liqo control plane traffic** (i.e., including both the initial authentication process and the communication with the remote Kubernetes API server) **flows outside the VPN tunnel** interconnecting the two clusters (still, TLS is used to ensure secure communications). Indeed, this tunnel is dynamically started in a later stage of the peering process, and it is leveraged only for cross-cluster pods traffic.

The single cross-cluster traffic flow required by this approach is schematized at a high level in the figure below (agnostic from how services are exposed, which is presented in the *dedicated installation requirements section*).

Overall, the out-of-band control plane approach:

- Supports clusters under the control of **different administrative domains**, as each party interacts only with its own cluster: the provider retrieves an authentication token that is subsequently shared with and leveraged by the consumer to start the peering process.
- Is characterized by **high dynamism**, as upon parameters modifications (e.g., concerning VPN setup) the negotiation process ensures synchronization between clusters and the peering automatically re-converges to a stable status.
- Requires each cluster to expose **three different endpoints** (i.e., the Liqo authentication service, the Liqo VPN endpoint and the Kubernetes API server), making them accessible from the pods running in the remote cluster.

3.2.2 In-band control plane

The alternative peering approach is referred to as **in-band control plane**, since the **Liqo control plane traffic flows inside the VPN tunnel** interconnecting the two clusters. In this case, the tunnel is statically established at the beginning of the peering process (i.e., part of the negotiation process is carried out directly by the Liqo CLI tool), and it is leveraged from that moment on for all inter-cluster traffic. The three different cross-cluster traffic flows required by this approach are schematized at a high level in figure below (agnostic from how services are exposed, which is presented in the *dedicated installation requirements section*).

Overall, the in-band control plane approach:

- Requires the administrator starting the peering process to have **access to both clusters** (although with limited permissions), as the network parameters negotiation is performed through the Liqo CLI tool (which interacts at the same time with both clusters). The remainder of the peering process, instead, is completed as usual, although the entire communication flows inside the VPN tunnel.
- **Statically configures** the cross-cluster **VPN tunnel** at peering establishment time, hence requiring manual intervention in case of configuration changes causing connectivity loss.

- **Relaxes** the **connectivity requirements**, as only the Liqo VPN endpoint needs to be reachable from the pods running in the remote cluster. Specifically, the Kubernetes API service is not required to be exposed outside the cluster.

CHAPTER

FOUR

OFFLOADING

Workload offloading is enabled by a **virtual node**, which is spawned in the local (i.e., consumer) cluster at the end of the peering process, and represents (and aggregates) the subset of resources shared by the remote cluster.

This solution enables the **transparent extension** of the local cluster, with the new node (and its capabilities) seamlessly taken into account by the vanilla Kubernetes scheduler when selecting the best place for the workloads execution. At the same time, this approach is fully compliant with the **standard Kubernetes APIs**, hence allowing to interact with and inspect offloaded pods just as if they were executed locally.

4.1 Assigned resources

By default, the virtual node is assigned with 90% of the resources available in the remote cluster. For example:

- If the remote cluster has 100 vCPUs available, the virtual node created with 90 vCPUs.
- If now the remote cluster starts some applications that consume 50 vCPUs (i.e., pods *requesting* resources), the virtual node is resized to 45 vCPUs (i.e., 90% of (100-50)).
- If the remote cluster has some autoscaling mechanism that, at some point, double the size of the cluster, which reaches 200 vCPUs (all of them unused by any pod), the virtual node will be resized with 180 vCPUs.

This mechanism applies to all the physical resources available in the remote cluster, e.g., CPUs, RAM, GPUs and more. The percentage of sharing can be customized also at run-time using the `--sharing-percentage` option, as documented in the proper *section* of the Liqo installation.

> **Warning:** Pay attention to *math rounding*. For instance, if your remote cluster has 1 GPU, with default settings the virtual node will be set with 0.9 GPUs. Since numbers must be integers, you may end up with a virtual node with *zero* GPUs.

More granular resource definitions with external Resource Plugins

The `--sharing-percentage` option is a unique and global parameter for the cluster. Hence, currently Liqo cannot differentiate the resources assigned to different peered clusters. For a more granular definition of the resources, you should consider to instal an external Resource Plugin, or create your own.

4.2 Virtual kubelet

The virtual node abstraction is implemented by an extended version of the **Virtual Kubelet project**. A virtual kubelet replaces a traditional kubelet when the controlled entity is not a physical node. In the context of Liqo, it interacts with both the local and the remote clusters (i.e., the respective Kubernetes API servers) to:

1. Create the **virtual node resource** and reconcile its status with respect to the negotiated configuration.
2. **Offload the local pods** scheduled onto the corresponding (virtual) node to the remote cluster, while keeping their status aligned.
3. Propagate and synchronize the **accessory artifacts** (e.g., *Services*, *ConfigMaps*, *Secrets*, ...) required for proper execution of the offloaded workloads, a feature we call **resource reflection**.

For each remote cluster, a different instance of the Liqo virtual kubelet is started in the local cluster, ensuring isolation and segregating the different authentication tokens.

4.3 Virtual node

A **virtual node** summarizes and abstracts the **amount of resources** (e.g., CPU, memory, ...) shared by a given remote cluster. Specifically, the virtual kubelet automatically propagates the negotiated configuration into the *capacity* and *allocatable* entries of the node status.

Node conditions reflect the current status of the node, with periodic and configurable **healthiness checks** performed by the virtual kubelet to assess the reachability of the remote API server. This allows to mark the node as *not ready* in case of repeated failures, triggering the standard Kubernetes eviction strategies based on the configured *pod tolerations* (e.g., to enforce service continuity).

Finally, each virtual node includes a set of **characterizing labels** (e.g., geographical region, underlying provider, ...) suggested by the remote cluster. This enables the enforcement of **fine-grained scheduling policies** (e.g., through *affinity* constraints), in addition to playing a key role in the namespace extension process presented below.

4.4 Namespace extension

To enable seamless workload offloading, **Liqo extends Kubernetes namespaces** across the cluster boundaries. Specifically, once a given namespace is selected for offloading (see the *namespace offloading usage section* for the operational procedure), Liqo proceeds with the automatic creation of **twin namespaces** in the subset of selected remote clusters.

Remote namespaces host the actual **pods offloaded** to the corresponding cluster, as well as the **additional resources** propagated by the resource reflection process. This behavior is presented in the figure below, which shows a given namespace existing in the local cluster and extended to a remote cluster. A group of pods is contained in the local namespace, while a subset (i.e., those faded-out) is scheduled onto the virtual node and offloaded to the remote namespace. Additionally, the resource reflection process propagated different resources existing in the local namespace (e.g., *Services*, *ConfigMaps*, *Secrets*, ...) in the remote one (represented faded-out), to ensure the correct execution of offloaded pods.

The Liqo namespace extension process features a high degree of customization, mainly enabling to:

- Select a **specific subset of the available remote clusters**, by means of standard selectors matching the label assigned to the virtual nodes.

- Constraint whether pods should be scheduled onto **physical nodes only, virtual nodes only, or both**. The extension of a namespace, forcing at the same time all pods to be scheduled locally, enables the consumption of local services from the remote cluster, as shown in the service offloading *example*.
- Configure whether the **remote namespace name** should match the local one (although possibly incurring in conflicts), or be automatically generated, such as to be unique.

4.5 Pod offloading

Once a **pod is scheduled onto a virtual node**, the corresponding Liqo virtual kubelet (indirectly) creates a **twin pod object** in the remote cluster for actual execution. Liqo supports the offloading of both **stateless** and **stateful** pods, the latter either relying on the provided ***storage fabric*** or leveraging externally managed solutions (e.g., persistent volumes provided by the cloud provider infrastructure).

Remote pod resiliency (hence, service continuity), even in case of temporary connectivity loss between the two control planes, is ensured through a **custom resource** (i.e., *ShadowPod*) wrapping the pod definition, and triggering a Liqo enforcement logic running in the remote cluster. This guarantees that the desired pod is always present, without requiring the intervention of the originating cluster.

The virtual kubelet takes care of the automatic propagation of **remote status changes** to the corresponding local pod (remapping the appropriate information), allowing for complete **observability** from the local cluster. Advanced operations, such as **metrics and logs retrieval**, as well as **interactive command execution** inside remote containers, are transparently supported, to comply with standard troubleshooting operations.

Additional details concerning how pods are propagated to remote clusters are provided in the *resource reflection usage section*.

4.6 Resource reflection

The **resource reflection** process is responsible for the propagation and synchronization of selected control plane information into remote clusters, to enable the seamless execution of offloaded pods. Liqo supports the reflection of the resources dealing with **service exposition** (i.e., *Ingresses*, *Services* and *EndpointSlices*), **persistent storage** (i.e., *PersistentVolumeClaims* and *PersistentVolumes*), as well as those storing **configuration data** (i.e., *ConfigMaps* and *Secrets*).

All resources of the above types that live in a **namespace selected for offloading** are automatically propagated into the corresponding twin namespaces created in the selected remote clusters. Specifically, the local copy of each resource is the source of trust leveraged to realign the content of the **shadow copy** reflected remotely. Appropriate **remapping** of certain information (e.g., *endpoint IPs*) is transparently performed by the virtual kubelet, accounting for conflicts and different configurations in different clusters.

You can refer to the *resource reflection usage section* for a detailed characterization of how the different resources are reflected into remote clusters.

CHAPTER

FIVE

NETWORK FABRIC

The **network fabric** is the Liqo subsystem transparently extending the Kubernetes network model across multiple independent clusters, such that **offloaded pods can communicate with each other** as if they were all executed locally.

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

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

5.1 Network manager

The **network manager** (not shown in figure) represents the **control plane** of the Liqo network fabric. It is executed as a pod, and it is responsible for the **negotiation of the connection parameters** with each remote cluster during the peering process.

It features an **IP Address Management (IPAM) plugin**, which deals with possible **network conflicts** through the definition of high-level NAT rules (enforced by the data plane components). Additionally, it exposes an interface consumed by the reflection logic to handle **IP addresses remapping**. Specifically, this is leveraged to handle the *translation of pod IPs* (i.e., during the synchronization process from the remote to the local cluster), as well as during *EndpointSlices reflection* (i.e., propagated from the local to the remote cluster).

5.2 Cross-cluster VPN tunnels

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

Tunnels are set up by the **Liqo gateway**, a component of the network fabric that is executed as a *privileged* pod on one of the cluster nodes. Additionally, it appropriately populates the **routing table**, and configures, by leveraging *iptables*, the **NAT rules** requested to comply with address conflicts.

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

5.3 In-cluster overlay network

The **overlay network** is leveraged to **forward all traffic** originating from local pods/nodes, and directed to a remote cluster, **to the gateway**, where it will enter the VPN tunnel. The same process occurs on the other side, with the traffic that exits from the VPN tunnel entering the overlay network to reach the node hosting the destination pod.

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

CHAPTER

SIX

STORAGE FABRIC

The Liqo **storage fabric** subsystem enables the seamless offloading of **stateful workloads** to remote clusters. The solution is based on two main pillars:

- **Storage binding deferral** until its first consumer is scheduled onto a given cluster (either local or remote). This ensures that **new storage pools** are created in the exact location where their associated pods have just been scheduled onto for execution.
- **Data gravity**, entering in action in the subsequent scheduling processes, and involving a set of **automatic policies** to attract pods in the appropriate cluster. This guarantees that pods requesting **existing pools of storage** (e.g., following a restart) are scheduled onto the cluster physically hosting the corresponding data.

These approaches extend standard Kubernetes practice to multi-cluster scenarios, simplifying at the same time the configuration of **high availability** and **disaster recovery** scenarios. To this end, one relevant use-case is represented by database instances that need to be replicated and synchronized across different clusters, which is shown in the *stateful applications example*.

Under the hood, the Liqo storage fabric leverages a **virtual storage class**, which embeds the logic to **create the appropriate storage pools** in the different clusters. Whenever a new *PersistentVolumeClaim (PVC)* associated with the virtual storage class is created, and its consumer is bound to a (possibly virtual) node, the Liqo logic goes into action, based on the target node:

- If it is a **local node**, PVC operations are remapped to a second one, associated with the corresponding **real storage class**, to transparently **provision the requested volume**.
- In case of **virtual nodes**, the reflection logic is responsible for creating the **remote shadow PVC**, remapped to the negotiated storage class, and **synchronizing the PersistentVolume information**, to allow pod binding.

In both cases, **locality constraints** are automatically embedded within the resulting *PersistentVolumes (PVs)*, to make sure each pod is scheduled only onto the cluster where the associated storage pools are available.

Additional details about the **configuration of the Liqo storage fabric**, as well as concerning the possibility to **move storage pools** among clusters through the Liqo CLI tool, are presented in the *stateful applications usage section*.

Note

In addition to the provided storage class, Liqo supports the execution of pods leveraging cross-cluster storage managed by external solutions (e.g., persistent volumes provided by the cloud provider infrastructure).

CHAPTER

SEVEN

REQUIREMENTS

This page presents an overview of the main requirements, both in terms of **resources** and **network connectivity**, to use Liqo and successfully establish peerings with remote clusters.

7.1 Resources

Liqo requires very **limited resources** (i.e., CPU, RAM, network bandwidth), making it suitable for both traditional **K8s** clusters and **resource constrained** clusters, e.g., the ones running K3s on a Raspberry Pi.

While the exact numbers depend on the **number of established peerings**, **number of offloaded pods** and on the **size of the cluster**, as a ballpark figure the entire Liqo control plane, executed on a two-nodes KinD cluster, peered with one remote cluster, and while offloading 100 pods, requires less than:

- 0.5 CPU cores (only during transient periods, while CPU consumption is practically negligible in all the other instants).
- 200 MB of RAM (this metric increases the more pods are offloaded to remote clusters).
- 5 Mbps of cross-cluster control plane traffic (only during transient periods). Data plane traffic, instead, depends on the applications and their actual placements across the clusters.

However, to be on the safe side, we suggest installing Liqo on a cluster that has **at least 2 CPUs and 2 GB of RAM**, which takes into account also the resources used by standard Kubernetes components.

Liqo is compatible with **Kubernetes versions >= 1.22**.

An accurate analysis of the Liqo performance compared to vanilla Kubernetes, including the characterization of the resources consumed by Liqo, is presented in a dedicated blog post.

7.2 Connectivity

Liqo supports two alternative peering approaches, each characterized by **different requirements in terms of network connectivity** (i.e., mutually reachable endpoints):

- ***Out-of-band control plane peering***: it requires **three separated traffic flows** (hence, three exposed endpoints).
- ***in-band control plane peering***: it requires a **single endpoint**, as all control plane traffic is tunneled inside the cross-cluster VPN.

More details available in the *peering section*.

Note

The two peering approaches are **non-mutually exclusive**: a cluster can leverage different approaches toward different remote clusters, provided that the connectivity requirements are satisfied.

7.2.1 Out-of-band control plane peering

In order to successfully establish an out-of-band control plane peering with a remote cluster, the following three services need to be **reciprocally accessible** on both clusters (i.e., in terms of IP address/port):

- **Authentication service** (`liqo-auth`): the Liqo service used to authenticate incoming peering requests coming from other clusters.
- **Network gateway** (`liqo-gateway`): the Liqo service responsible for the setup of the cross-cluster VPN tunnels.
- **Kubernetes API server**: the standard Kubernetes API Server, that is used by the (remote) Liqo instance to create the resources required to start the peering process, and perform workload offloading.

Reciprocally accessible means that a first cluster must be able to connect to the *<IP/port>* of the above services exposed on the second cluster, and vice versa (i.e., from second cluster to the first); some exceptions that refer to the network gateway are detailed in the following of this page. This implies also that any network device (**NAT**, **firewall**, etc.) sitting in the path between the two clusters must be configured to **enable direct connectivity** toward the above services, as presented in the *network firewalls* section.

The tuple *<IP/port>* exported by the Liqo services (i.e., `liqo-auth`, `liqo-gateway`) depends on the Liqo configuration, chosen at installation time, which may depend on the physical setup of your cluster and the characteristics of your service. In particular:

- **Authentication Service**: when you install Liqo, you can choose to expose the authentication service through a *LoadBalancer* service, a *NodePort* service, or an *Ingress* (the last allows the service to be exposed as *ClusterIP*). This choice depends (1) on your necessities, (2) on the cluster configuration (e.g., a *NodePort* cannot be used if your nodes have private IP addresses, hence cannot be reached from the Internet), and (3) whether the above primitives (e.g., the *Ingress Controller*) are available in your cluster.
- **Network Gateway**: the same applies also for the network gateway, except that it cannot be exported through an *Ingress*. In fact, while the authentication service uses a standard HTTP/REST interface, the network gateway is the termination of a UDP-based network tunnel; hence only *LoadBalancer* and *NodePort* services are supported.

Note

Liqo supports scenarios in which, given two clusters, only one of the two **network gateways** is publicly reachable from the remote cluster (i.e., in terms of *<IP/port>* tuple), although communication must be allowed by possible firewalls sitting in the path.

By default, *liqoctl* exposes both the authentication service and the network gateway through a **dedicated *LoadBalancer* service**, falling back to a *NodePort* for simpler setups (i.e., KinD and K3s). However, more advanced configurations can be achieved by configuring the proper Helm chart parameters, either directly or by customizing the installation process *through* liqoctl.

An overview of the overall connectivity requirements to establish out-of-band control plane peerings in Liqo is shown in the figure below.

Additional considerations

The choice of the way you expose Liqo services to remote clusters may not be trivial in some cases. Here, we list some additional notes you should consider in your choice:

- **NodePort service**: although a *NodePort* service can be used to expose the authentication service and the network gateway, often the IP addresses of the nodes are configured with private IP addresses, hence not being suitable for connections originated from the Internet. This happens rather often in production clusters, and on public clusters as well.
- **Ingress controller**: in case the authentication service is exposed through an *Ingress*, you should remember that, by default, the authentication service uses the TLS protocol. Hence, either you configure your *Ingress Controller* to connect to backend services with TLS as well, or you disable TLS on the authentication service.

Finally, in some cases clusters may reside behind a NAT. Liqo transparently supports scenarios with **one cluster behind NAT** and the other publicly reachable. Yet, in such situations, we suggest leveraging the in-band peering, as it simplifies the overall configuration.

7.2.2 In-band control plane peering

In order to successfully establish an in-band control plane peering with a remote cluster, you need only the **network gateways to be *mutually* reachable**, since all the Liqo control plane traffic is then configured to flow inside the VPN tunnel. All considerations presented above and referring to the exposition of the network gateway apply also in this case.

Given the connectivity requirements are a subset of the previous case, this solution is compatible with the configurations that enable the out-of-band peering approach. Additionally, it:

- Supports scenarios characterized by a **non publicly accessible Kubernetes API Server**.
- Allows to expose the authentication service as a *ClusterIP* service, reducing the number of services exposed externally.
- Enables setups with one cluster **behind NAT**, since the VPN tunnel can be established successfully even in case only one of the two network gateways is publicly reachable from the other cluster.

An overview of the overall connectivity requirements to establish in-band control plane peerings in Liqo is shown in the figure below.

Warning: Due to current limitations, the establishment of an in-band peering may not complete successfully in case the authentication service is exposed through an Ingress to which the TLS termination is delegated (i.e., when TLS is disabled on the authentication service).

7.2.3 Network firewalls

In some cases, especially on production setups, additional network limitations are present, such as firewalls that may impair network connectivity, which must be considered in order to enable Liqo peerings.

Depending on your configuration and the selected peering approach, you may have to configure existing firewalls to enable remote clusters to contact either the `liqo-gateway` only or all the three endpoints (i.e., `liqo-auth`, `liqo-gateway` and Kubernetes API server) that need to be publicly accessible in the peering phase.

To know the network parameters (i.e., <IP/port>) used by `liqo-auth` and `liqo-gateway`, you can use standard Kubernetes commands (e.g., `kubectl get services -n liqo`), while the <IP/port> tuple used by your Kubernetes API server is the one written in the `kubeconfig` file.

Remember that the Kubernetes API server and authentication service use the HTTPS protocol (over TCP); vice versa, the network gateway uses the WireGuard protocol over UDP.

CHAPTER

EIGHT

LIQO CLI TOOL

8.1 Introduction

Liqoctl is the CLI tool to streamline the **installation** and **management** of Liqo. Specifically, it abstracts the creation and modification of Liqo defined custom resources, allowing to:

- **Install/uninstall** Liqo, wrapping the corresponding Helm commands and automatically retrieving the appropriate parameters based on the target cluster configuration.
- Establish and revoke **peering** relationships towards remote clusters.
- Enable and configure **workload offloading** on a per-namespace basis.
- Retrieve the **status** of Liqo, as well as of given peering relationships and offloading setups.

Warning: Make sure to always **use the *liqoctl* version matching that of Liqo** installed (or to be installed) in your cluster(s).

Note

liqoctl displays a *kubectl* compatible behavior concerning Kubernetes API access, hence supporting the `KUBECONFIG` environment variable, as well as all the standard flags, including `--kubeconfig` and `--context`. Moreover, subcommands interacting with two clusters (e.g., *liqoctl peer in-band*) feature a parallel set of flags concerning Kubernetes API access to the remote cluster, in the form `--remote-<flag>` (e.g., `--remote-kubeconfig`, `--remote-context`).

8.2 Install liqoctl with Homebrew

If you are using the Homebrew package manager, you can install *liqoctl* with Homebrew:

```
brew install liqoctl
```

When installed with Homebrew, autocompletion scripts are automatically configured and should work out of the box.

8.3 Install liqoctl with asdf

If you are using the asdf runtime manager, you can install *liqoctl* with asdf:

```
# Add the liqoctl plugin for asdf
asdf plugin add liqoctl

# List all installable versions
asdf list-all liqoctl

# Install the desired version
asdf install liqoctl <version>

# set it as the global version, unless a project declares it otherwise locally
asdf global liqoctl <version>
```

8.4 Install liqoctl manually

You can download and install *liqoctl* manually, following the appropriate instructions based on your operating system and architecture.

Linux

Download *liqoctl* and move it to a file location in your system `PATH`:

AMD64:

```
curl --fail -LS "https://github.com/liqotech/liqo/releases/download/master/liqoctl-linux-
↪amd64.tar.gz" | tar -xz
sudo install -o root -g root -m 0755 liqoctl /usr/local/bin/liqoctl
```

ARM64:

```
curl --fail -LS "https://github.com/liqotech/liqo/releases/download/master/liqoctl-linux-
↪arm64.tar.gz" | tar -xz
sudo install -o root -g root -m 0755 liqoctl /usr/local/bin/liqoctl
```

Note

Make sure `/usr/local/bin` is in your `PATH` environment variable.

MacOS

Download *liqoctl*, make it executable, and move it to a file location in your system `PATH`:

Intel:

```
curl --fail -LS "https://github.com/liqotech/liqo/releases/download/master/liqoctl-
↪darwin-amd64.tar.gz" | tar -xz
chmod +x liqoctl
sudo mv liqoctl /usr/local/bin/liqoctl
```

Apple Silicon:

```
curl --fail -LS "https://github.com/liqotech/liqo/releases/download/master/liqoctl-
↪darwin-arm64.tar.gz" | tar -xz
chmod +x liqoctl
sudo mv liqoctl /usr/local/bin/liqoctl
```

Note

Make sure `/usr/local/bin` is in your `PATH` environment variable.

Windows

Download the *liqoctl* binary:

```
curl --fail -LSO "https://github.com/liqotech/liqo/releases/download/master/liqoctl-
↪windows-amd64"
```

And move it to a file location in your system `PATH`.

Alternatively, you can manually download *liqoctl* from the Liqo releases page on GitHub.

8.5 Install Kubectl plugin with Krew

You can install liqoctl as a kubectl plugin by using the Krew plugin manager:

```
kubectl krew install liqo
```

Then, all commands shall be invoked with `kubectl liqo` rather than `liqoctl`, although all functionalities remain the same.

Warning: While the kubectl plugin is supported, it is recommended to use liqoctl as this enables a better experience via tab auto-completion. *Install it with Homebrew* if available on your system or by *manually downloading the binary from GitHub*.

8.6 Install liqoctl from source

You can install *liqoctl* building it from source. To do so, clone the Liqo repository, build the *liqoctl* binary, and move it to a file location in your system `PATH`:

```
git clone https://github.com/liqotech/liqo.git
cd liqo
make ctl
mv liqoctl /usr/local/bin/liqoctl
```

8.7 Enable shell autocompletion

liqoctl provides autocompletion support for Bash, Zsh, Fish, and PowerShell.

Bash

The *liqoctl* completion script for Bash can be generated with the `liqoctl completion bash` command. Sourcing the completion script in your shell enables *liqoctl* autocompletion.

However, the completion script depends on *bash-completion*, which means that you have to install this software first. You can test if you have *bash-completion* already installed by running `type _init_completion`. If it returns an error, you can install it via your OS's package manager.

To load completions in your current shell session:

```
source <(liqoctl completion bash)
```

To load completions for every new session, execute once:

```
source <(liqoctl completion bash) >> ~/.bashrc
```

After reloading your shell, *liqoctl* autocompletion should be working.

Zsh

The *liqoctl* completion script for Zsh can be generated with the `liqoctl completion zsh` command.

If shell completion is not already enabled in your environment, you will need to enable it. You can execute the following once:

```
echo "autoload -U compinit; compinit" >> ~/.zshrc
```

To load completions for each session, execute once:

```
liqoctl completion zsh > "${fpath[1]}/_liqoctl"
```

After reloading your shell, *liqoctl* autocompletion should be working.

Fish

The *liqoctl* completion script for Fish can be generated with the `liqoctl completion fish` command.

To load completions in your current shell session:

```
liqoctl completion fish | source
```

To load completions for every new session, execute once:

```
liqoctl completion fish > ~/.config/fish/completions/liqoctl.fish
```

After reloading your shell, *liqoctl* autocompletion should be working.

PowerShell

The *liqoctl* completion script for PowerShell can be generated with the `liqoctl completion powershell` command.

To load completions in your current shell session:

```
liqoctl completion powershell | Out-String | Invoke-Expression
```

To load completions for every new session, add the output of the above command to your PowerShell profile.

After reloading your shell, *liqoctl* autocompletion should be working.

CHAPTER

NINE

INSTALL

Liqo can be easily installed with *liqoctl*, which automatically handles all the customized settings required to set up the software on the multiple provider/distribution supported (e.g., AWS, EKS, GKE, Kubeadm, etc.). Under the hood, *liqoctl* uses Helm 3 to configure and install all the Liqo components, using the Helm chart available in the official repository.

Alternatively, *liqoctl* can also be configured to generate a local file with **pre-configured values**, which can be further customized and used for a manual installation with Helm.

9.1 Install with liqoctl

Below, you can find the basic information to install and configure Liqo, depending on the selected **Kubernetes distribution** and/or **cloud provider**. By default, *liqoctl install* installs the latest *stable* version of Liqo, although this can be changed with the `--version` flag.

The rest of this page presents **additional customization options** that apply to all setups, as well as advanced options that are cloud/distribution-specific.

Note

liqoctl implements a *kubectl* compatible behavior with respect to Kubernetes API access, hence supporting the `KUBECONFIG` environment variable, as well as all the standard flags, including `--kubeconfig` and `--context`. Hence, make sure you selected the correct target cluster before issuing *liqoctl* commands (as you would do with *kubectl*).

Kubeadm

Supported CNIs

Liqo supports Kubernetes clusters using the following CNIs: Flannel, Calico, Canal, Weave. Additionally, partial support is provided for Cilium, although with the limitations listed below.

> **Warning:** If you are installing Liqo on a cluster using the **Calico** CNI, you MUST read the *dedicated configuration section* to avoid unwanted misconfigurations.

Liqo + Cilium limitations

Currently, Liqo supports the Cilium CNI only when *kube-proxy* is enabled. Additionally, known limitations concern the impossibility of accessing the backends of *NodePort* and *LoadBalancer* services hosted on remote clusters, from a local cluster using Cilium as CNI.

Installation

Liqo can be installed on a Kubeadm cluster with the following command:

```
liqoctl install kubeadm
```

The name of the cluster is automatically generated, then used during the peering and offloading processes. Alternatively, you can manually specify a desired name with the `--cluster-name` flag.

Service Type

By default, the **kubeadm** provider exposes *liqo-auth* and *liqo-gateway* with **LoadBalancer** services. To change this behavior, check the *network flags*.

OpenShift

Supported versions

Liqo was tested on OpenShift Container Platform (OCP) 4.8.

Installation

Liqo can be installed on an OpenShift Container Platform (OCP) cluster with the following command:

```
liqoctl install openshift
```

The name of the cluster is automatically generated, then used during the peering and offloading processes. Alternatively, you can manually specify a desired name with the `--cluster-name` flag.

Service Type

By default, the **openshift** provider exposes *liqo-auth* and *liqo-gateway* with **LoadBalancer** services. To change this behavior, check the *network flags*.

AKS

Supported CNIs

Liqo supports AKS clusters using the following CNIs: Azure AKS - Kubenet and Azure AKS - Azure CNI.

Configuration

To install Liqo on AKS, you should first log in using the `az` CLI (if not already done):

```
az login
```

Before continuing, you should export the following variables with some information about your cluster:

```
# The resource group where the cluster is created
export AKS_RESOURCE_GROUP=resource-group
# The name of AKS cluster resource on Azure
export AKS_RESOURCE_NAME=cluster-name
# The name of the subscription associated with the AKS cluster
export AKS_SUBSCRIPTION_ID=subscription-name
```

Note

During the installation process, you need read-only permissions on the AKS cluster and on the Virtual Networks, if your cluster leverages the Azure CNI.

Installation

Liqo can be installed on an AKS cluster with the following command:

```
liqoctl install aks --resource-group-name "${AKS_RESOURCE_GROUP}" \
        --resource-name "${AKS_RESOURCE_NAME}" \
        --subscription-name "${AKS_SUBSCRIPTION_ID}"
```

The name of the cluster will be equal to the one specified in the `--resource-name` parameter. Alternatively, you can manually set a different name with the `--cluster-name` *liqoctl* flag.

Note

If you are running an AKS private cluster, you may need to set the `--disable-api-server-sanity-check` *liqoctl* flag, since the API Server in your kubeconfig may be different from the one retrieved from the Azure APIs.

Additionally, since your API Server is not accessible from the public Internet, you shall leverage the *in-band peering approach* towards the clusters not attached to the same Azure Virtual Network.

Service Type

By default, the **AKS** provider exposes *liqo-auth* and *liqo-gateway* with **LoadBalancer** services. To change this behavior, check the *network flags*.

EKS

Supported CNIs

Liqo supports EKS clusters using the default CNI: AWS EKS - amazon-vpc-cni-k8s.

Configuration

Liqo leverages **AWS credentials to authenticate peered clusters**. Specifically, in addition to the read-only permissions used to configure the cluster installation (i.e., retrieve the appropriate parameters), Liqo uses AWS users to map peering access to EKS clusters.

To install Liqo on EKS, you should first log in using the `aws` CLI (if not already done). This is widely documented on the official CLI documentation. In a nutshell, after having installed the CLI, you have to set up your identity:

```
aws configure
```

You can install Liqo even if you are not an EKS administrator. The minimum **IAM** permissions required by a user to install Liqo are the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "eks:DescribeCluster",
                "iam:CreateUser",
                "iam:CreateAccessKey",
                "ec2:DescribeVpcs"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreatePolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:AttachUserPolicy",
                "iam:GetUser",
                "iam:TagUser",
                "iam:ListAccessKeys"
            ],
            "Resource": [
                "arn:aws:iam::*:user/liqo-*",
                "arn:aws:iam::*:policy/liqo-*"
            ]
        }
    ]
}
```

Before continuing, you should export the following variables with some information about your cluster:

```
# The name of the target cluster
export EKS_CLUSTER_NAME=cluster-name
# The AWS region where the cluster is deployed
export EKS_CLUSTER_REGION=cluster-region
```

Then, you should retrieve the cluster's kubeconfig (if you have not done it already) with the following CLI command:

```
aws eks --region ${EKS_CLUSTER_REGION} update-kubeconfig --name ${EKS_CLUSTER_NAME}
```

Installation

Liqo can be installed on an EKS cluster with the following command:

```
liqoctl install eks --eks-cluster-region=${EKS_CLUSTER_REGION} \
        --eks-cluster-name=${EKS_CLUSTER_NAME}
```

The name of the cluster will be equal to the one specified in the `--eks-cluster-name` parameter. Alternatively, you can manually set a different name with the `--cluster-name` *liqoctl* flag.

Service Type

By default, the **EKS** provider exposes *liqo-auth* and *liqo-gateway* with **LoadBalancer** services. To change this behavior, check the *network flags*.

GKE

Supported CNIs

Liqo supports GKE clusters using the default CNI: Google GKE - VPC-Native.

> **Warning:** Liqo does NOT support:
>
> - GKE Autopilot Clusters
> - Container-Optimized OS with containerd (*cos_containerd*) as image type. Use Ubuntu with containerd (*ubuntu_containerd*) instead
> - Intranode visibility: make sure this option is disabled or use the `--no-enable-intra-node-visibility` flag.

Configuration

To install Liqo on GKE, you should create a service account for *liqoctl*, granting the read rights for the GKE clusters (you may reduce the scope to a specific cluster if you prefer).

First, you should export the following variables with some information about your cluster and the service account to create:

```
# The name of the service account used by liqoctl to interact with GCP
export GKE_SERVICE_ACCOUNT_ID=liqoctl
# The path where the GCP service account is stored
export GKE_SERVICE_ACCOUNT_PATH=~/.liqo/gcp_service_account

# The ID of the GCP project where your cluster was created
export GKE_PROJECT_ID=project-id
# The GCP zone where your GKE cluster is executed (if you are using zonal GKE clusters)
export GKE_CLUSTER_ZONE=europe-west1-b
# The GCP region where your GKE cluster is executed (if you are using regional GKE␣
↪clusters)
export GKE_CLUSTER_REGION=europe-west1
# The name of the GKE resource on GCP
export GKE_CLUSTER_ID=liqo-cluster
```

Second, you should create a GCP service account. This will represent the identity used by *liqoctl* to query the information required to properly configure Liqo on your cluster. The service account can be created using:

```
gcloud iam service-accounts create ${GKE_SERVICE_ACCOUNT_ID} \
    --project="${GKE_PROJECT_ID}" \
    --description="The identity used by liqoctl during the installation process" \
    --display-name="liqoctl"
```

Third, you should grant the service account the rights to inspect the cluster and the virtual networks parameters:

```
gcloud projects add-iam-policy-binding ${GKE_PROJECT_ID} \
    --member="serviceAccount:${GKE_SERVICE_ACCOUNT_ID}@${GKE_PROJECT_ID}.iam.
↪gserviceaccount.com" \
    --role="roles/container.clusterViewer"
gcloud projects add-iam-policy-binding ${GKE_PROJECT_ID} \
    --member="serviceAccount:${GKE_SERVICE_ACCOUNT_ID}@${GKE_PROJECT_ID}.iam.
↪gserviceaccount.com" \
    --role="roles/compute.networkViewer"
```

Fourth, you should create and download a set of valid service accounts keys, as presented by the official documentation. The keys will be used by liqoctl to authenticate to GCP:

```
gcloud iam service-accounts keys create ${GKE_SERVICE_ACCOUNT_PATH} \
    --iam-account=${GKE_SERVICE_ACCOUNT_ID}@${GKE_PROJECT_ID}.iam.gserviceaccount.com
```

Finally, you should retrieve the cluster's kubeconfig (if you have not done it already) with the following CLI command in case of **zonal** GKE clusters:

```
gcloud container clusters get-credentials ${GKE_CLUSTER_ID} \
        --zone ${GKE_CLUSTER_ZONE} --project ${GKE_PROJECT_ID}
```

or, in case of **regional** GKE clusters:

```
gcloud container clusters get-credentials ${GKE_CLUSTER_ID} \
        --region ${GKE_CLUSTER_REGION} --project ${GKE_PROJECT_ID}
```

The retrieved kubeconfig will be added to the currently selected file (i.e., based on the `KUBECONFIG` environment variable, with fallback to the default path `~/.kube/config`) or created otherwise.

Installation

Liqo can be installed on a zonal GKE cluster with the following command:

```
liqoctl install gke --project-id ${GKE_PROJECT_ID} \
    --cluster-id ${GKE_CLUSTER_ID} \
    --zone ${GKE_CLUSTER_ZONE} \
    --credentials-path ${GKE_SERVICE_ACCOUNT_PATH}
```

or, in case of regional GKE clusters:

```
liqoctl install gke --project-id ${GKE_PROJECT_ID} \
    --cluster-id ${GKE_CLUSTER_ID} \
    --region ${GKE_CLUSTER_REGION} \
    --credentials-path ${GKE_SERVICE_ACCOUNT_PATH}
```

The name of the cluster will be equal to the one defined in GCP. Alternatively, you can manually set a different name with the `--cluster-name` *liqoctl* flag.

Service Type

By default, the **GKE** provider exposes *liqo-auth* and *liqo-gateway* with **LoadBalancer** services. To change this behavior, check the *network flags*.

K3s

Note

By default, the K3s installer stores the kubeconfig to access your cluster in the non-standard path `/etc/rancher/k3s/k3s.yaml`. Make sure to properly refer to it when using *liqoctl* (e.g., setting the `KUBECONFIG` variable), and that the current user has permissions to read it.

Installation

Liqo can be installed on a K3s cluster with the following command:

```
liqoctl install k3s
```

You may additionally set the `--api-server-url` flag to override the Kubernetes API Server address used by remote clusters to contact the local one. This operation is necessary in case the default address (`https://<control-plane-node-ip>:6443`) is unsuitable (e.g., the node IP is externally remapped).

The name of the cluster is automatically generated, then used during the peering and offloading processes. Alternatively, you can manually specify a desired name with the `--cluster-name` flag.

Service Type

By default, the **k3s** provider exposes *liqo-auth* and *liqo-gateway* with **NodePort** services. To change this behavior, check the *network flags*.

KinD

Installation

Liqo can be installed on a KinD cluster with the following command:

```
liqoctl install kind
```

The name of the cluster is automatically generated, then used during the peering and offloading processes. Alternatively, you can manually specify a desired name with the `--cluster-name` flag.

Service Type

By default, the **kind** provider exposes *liqo-auth* and *liqo-gateway* with **NodePort** services. To change this behavior, check the *network flags*.

Other

Configuration

To install Liqo on alternative Kubernetes distributions, you should manually retrieve three main configuration parameters:

- **API Server URL**: the Kubernetes API Server URL (defaults to the one specified in the kubeconfig).
- **Pod CIDR**: the range of IP addresses used by the cluster for the pod network.
- **Service CIDR**: the range of IP addresses used by the cluster for service VIPs.

Installation

Once retrieved the above parameters, Liqo can be installed on a generic cluster with the following command:

```
liqoctl install --api-server-url=<API-SERVER-URL> \
      --pod-cidr=<POD-CIDR> --service-cidr=<SERVICE-CIDR>
```

The name of the cluster is automatically generated, then used during the peering and offloading processes. Alternatively, you can manually specify a desired name with the `--cluster-name` flag.

Service Type

By default, liqoctl exposes *liqo-auth* and *liqo-gateway* with **LoadBalancer** services. To change this behavior, check the *network flags*.

9.2 Customization options

This section lists the main **customization parameters** supported by the *liqoctl install* command, along with a brief description. Additionally, **all parameters** available in the Helm *values* file (the full list is provided in the dedicated repository page) can be modified through the `liqoctl install --set` flag, which supports the standard Helm syntax.

Finally, remember that:

- You can type `liqoctl install --help` to get the list of available options.
- Some of the above parameters can be changed after installation by simply updating their value and re-applying the Helm chart, or by re-issuing the proper `liqoctl install --set [param=value]` command. However, given that not all parameters can be updated at run-time, please check that the command triggered the desired effect; a precise list of commands that can be changed at run-time is left for our future work.

9.2.1 Global

The main global flags, besides those concerning the installation of *development versions*, include:

- `--enable-ha`: enables the support for **high-availability of the Liqo components**, starting two replicas (in an active/standby configuration) of the **gateway** to ensure no cross-cluster connectivity downtime in case one of the replicas is restarted, as well as of the **controller manager**, which embeds the Liqo control plane logic.
- `--enable-metrics`: exposes Liqo **metrics** through **Prometheus** (see the dedicated *Prometheus metrics page* for additional details).
- `--timeout`: configures the timeout for the completion of the installation/upgrade process. Once expired, the process is aborted and Liqo is rolled back to the previous version.

- `--verbose`: enables verbose logs, providing additional information concerning the installation/upgrade process (e.g., for troubleshooting).
- `--disable-telemetry`: disables the collection of telemetry data, which is enabled by default. The telemetry is used to collect anonymous usage statistics, which are used to improve Liqo. Additional details are provided here.

9.2.2 Control plane

The main control plane flags include:

- `--cluster-name`: configures a **name identifying the cluster** in Liqo. This name is propagated to remote clusters during the peering process, and used to identify the corresponding virtual nodes and the Liqo resources used in the peering process. Additionally, the cluster name is used as part of the suffix to ensure namespace names uniqueness during the offloading process. In case a cluster name is not specified, it is defaulted to that of the cluster in the cloud provider, if any, or it is automatically generated.
- `--cluster-labels`: a set of **labels** (i.e., key/value pairs) **identifying the cluster in Liqo** (e.g., geographical region, Kubernetes distribution, cloud provider, ...) and automatically propagated during the peering process to the corresponding virtual nodes. These labels can be used later to **restrict workload offloading to a subset of clusters**, as detailed in the *namespace offloading usage section*.
- `--sharing-percentage`: the maximum percentage of available **cluster resources** that could be shared with remote clusters. This is the Liqo's default behavior, which can be changed by deploying a custom resource plugin. More details about the amount of resources shared by a cluster is available in the *Resource Offloading* page. **Note**: the `--sharing-percentage` can be updated (e.g., via helm) dynamically, without reinstalling Liqo.

9.2.3 Networking

The main networking flags include:

- `--reserved-subnets`: the list of **private CIDRs to be excluded** from the ones used by Liqo to remap remote clusters in case of address conflicts, as already in use (e.g., the subnet of the cluster nodes). The Pod CIDR and the Service CIDR shall not be manually specified, as automatically included in the reserved list.
- `--service-type`: overrides the service type used by **liqo-gateway** and **liqo-auth** services. Possible values are: `LoadBalancer`, `NodePort`, and `ClusterIP`. By default, the service type is the one specified by the selected provider (check the provider's specific installation) or `LoadBalancer`.

9.3 Install with Helm

To install Liqo directly with Helm, you can proceed as follows:

1. Add the Liqo Helm repository:

```
helm repo add liqo https://helm.liqo.io/
```

2. Update the local Helm repository cache:

```
helm repo update
```

3. Generate a pre-configured values file with *liqoctl*:

```
liqoctl install <provider> [flags] --only-output-values
```

The resulting *values* file is saved in the current directory, as `values.yaml`, or in the path specified through the `--dump-values-path` flag.

Note

The current step is optional, but it relieves the user from the retrieval of the set of necessary parameters depending on the target provider/distribution. Alternatively, the upstream values file can be retrieved through:

```
helm show values liqo/liqo > values.yaml
```

4. Appropriately configure the *values* file. The full list of options is provided in the dedicated repository page.
5. Install Liqo:

```
helm install liqo liqo/liqo --namespace liqo \
       --values <path-to-values-file> --create-namespace
```

9.4 Install development versions

In addition to released versions (including alpha and beta candidates), *liqoctl* provides the possibility to install **development versions** of Liqo. Development versions include:

- All commits merged into the master branch of Liqo.
- The commits of *pull requests* to the Liqo repository, whose images have been built through the appropriate bot command.

The installation of a development version of Liqo can be triggered specifying a **commit *SHA*** through the `--version` flag. In this case, *liqoctl* proceeds to **clone the repository** (either from the official repository, or from a fork configured through the `--repo-url` flag) at the given revision, and to leverage the Helm chart therein contained:

```
liqoctl install <provider> --version <commit-sha> --repo-url <forked-repo-url>
```

Alternatively, the Helm chart can be retrieved from a **local path**, as configured through the `--local-chart-path` flag:

```
liqoctl install <provider> --version <commit-sha> --local-chart-path <path-to-local-
↪chart>
```

9.5 Check installation

After the installation, you can check the status of the Liqo components. In particular, the following command can be used to check the status of the Liqo **pods** and get **local information**:

```
liqoctl status
```

9.6 Liqo and Calico

Liqo adds several interfaces to the cluster nodes to handle cross-cluster traffic routing. Those interfaces are intended to not interfere with the normal CNI job.

However, by default, Calico scans all existing interfaces on a node to detect network configurations and establish the correct routes. To prevent misconfigurations, Calico shall then be configured to skip Liqo-managed interfaces during this process. This is required if Calico is configured in *BGP* mode, while not in case the *VPC native setup* is leveraged.

In Calico v3.17 and above, this can be performed by patching the *Installation CR*, adding the following:

```
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  calicoNetwork:
    nodeAddressAutodetectionV4:
      skipInterface: liqo.*
    ...
  ...
```

For Calico versions prior to 3.17, instead, you should modify the `calico-node` *DaemonSet*, adding the appropriate environment variable to the `calico-node` container.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: calico-node
  namespace: kube-system
spec:
  template:
    spec:
      containers:
      - name: calico-node
        env:
        - name: IP_AUTODETECTION_METHOD
          value: skip-interface=liqo.*
      ...
```

CHAPTER

TEN

UNINSTALL

Liqo can be uninstalled by leveraging the dedicated *liqoctl* command:

```
liqoctl uninstall
```

Alternatively, the same operation can be performed directly with Helm:

```
helm uninstall liqo --namespace liqo
```

Note

Due to current limitations, the uninstallation process might hang in case peerings are still established, or namespaces are selected for offloading. To this end, *liqoctl* performs a set of pre-checks and aborts the process in case any of the above is found, requesting the administrator to **unpeer all clusters and unoffload all namespaces** with the dedicated *liqoctl* commands.

10.1 Purge CRDs

By default, the uninstallation process does not remove the Liqo CRDs and the system namespaces. These operations can be performed by adding the `--purge` flag:

```
liqoctl uninstall --purge
```

CHAPTER

ELEVEN

REQUIREMENTS

Before starting the tutorials below, you should ensure the following software is installed on your system:

- **Docker**, the container runtime.
- **Kubectl**, the command-line tool for Kubernetes.
- **Helm**, the package manager for Kubernetes.
- **curl**, to interact with the tutorial applications through HTTP/HTTPS.
- **KinD**, the Kubernetes in Docker runtime.
- ***liqoctl*** command-line tool to interact with Liqo.

The following tutorials were tested on Linux, macOS, and Windows (WSL2 and Docker Desktop).

Warning: To prevent issues with tutorials leveraging more than two clusters, on some systems you may need to increase the maximum number of *inotify* watches:

```
sudo sysctl fs.inotify.max_user_watches=52428899
sudo sysctl fs.inotify.max_user_instances=2048
```

CHAPTER

TWELVE

QUICK START

This tutorial aims at presenting how to install Liqo and practicing with its most notable capabilities. You will learn how to create a *virtual cluster* by peering two Kubernetes clusters and how to deploy a simple application on it.

12.1 Provision the playground

First, check that you are compliant with the *requirements*.

Then, let's open a terminal on your machine and launch the following script, which creates a pair of clusters with KinD. Each cluster is made by two nodes (one for the control plane and one as a simple worker):

```
git clone https://github.com/liqotech/liqo.git
cd liqo
git checkout v0.9.1
cd examples/quick-start
./setup.sh
```

12.1.1 Explore the playground

You can inspect the deployed clusters by typing:

```
kind get clusters
```

You should see a couple of entries:

```
milan
rome
```

This means that two KinD clusters are deployed and running on your host.

Then, you can simply inspect the status of the clusters. To do so, you can export the `KUBECONFIG` variable to specify the identity file for *kubectl* and *liqoctl*, and then contact the cluster.

By default, the kubeconfigs of the two clusters are stored in the current directory (`./liqo_kubeconf_rome`, `./liqo_kubeconf_milan`). You can export the appropriate environment variables leveraged for the rest of the tutorial (i.e., `KUBECONFIG` and `KUBECONFIG_MILAN`), and referring to their location, through the following:

```
export KUBECONFIG="$PWD/liqo_kubeconf_rome"
export KUBECONFIG_MILAN="$PWD/liqo_kubeconf_milan"
```

Note

We suggest exporting the kubeconfig of the first cluster as default (i.e., `KUBECONFIG`), since it will be the entry point of the virtual cluster and you will mainly interact with it.

On the first cluster, you can get the available pods by merely typing:

```
kubectl get pods -A
```

Similarly, on the second cluster, you can observe the pods in execution:

```
kubectl get pods -A --kubeconfig "$KUBECONFIG_MILAN"
```

If the above commands return each an output similar to the following, your clusters are up and ready.

```
NAMESPACE            NAME                                      READY   STATUS    ␣
↪RESTARTS   AGE
kube-system          coredns-558bd4d5db-9vdr9                  1/1     Running   0    ␣
↪       3m58s
kube-system          coredns-558bd4d5db-tzdxg                  1/1     Running   0    ␣
↪       3m58s
kube-system          etcd-rome-control-plane                   1/1     Running   0    ␣
↪       4m10s
kube-system          kindnet-fcspl                             1/1     Running   0    ␣
↪       3m58s
kube-system          kindnet-q6qkm                             1/1     Running   0    ␣
↪       3m42s
kube-system          kube-apiserver-rome-control-plane         1/1     Running   0    ␣
↪       4m10s
kube-system          kube-controller-manager-rome-control-plane   1/1     Running   0    ␣
↪       4m11s
kube-system          kube-proxy-2c9bl                          1/1     Running   0    ␣
↪       3m42s
kube-system          kube-proxy-7nngv                          1/1     Running   0    ␣
↪       3m58s
kube-system          kube-scheduler-rome-control-plane         1/1     Running   0    ␣
↪       4m11s
local-path-storage   local-path-provisioner-85494db59d-skd55   1/1     Running   0    ␣
↪       3m58s
```

12.2 Install Liqo

You will now install Liqo on both clusters, using the following characterizing names:

- **rome**: the *local* cluster, where you will deploy and control the applications.
- **milan**: the *remote* cluster, where part of your workloads will be offloaded to.

You can install Liqo on the *Rome* cluster by launching:

```
liqoctl install kind --cluster-name rome
```

This command will generate the suitable configuration for your KinD cluster and then install Liqo.

Similarly, you can install Liqo on the *Milan* cluster by launching:

```
liqoctl install kind --cluster-name milan --kubeconfig "$KUBECONFIG_MILAN"
```

On both clusters, you should see the following output:

```
 INFO  Kubernetes clients successfully initialized
 INFO  Installer initialized
 INFO  Cluster configuration correctly retrieved
 INFO  Installation parameters correctly generated
 INFO  All Set! You can now proceed establishing a peering (liqoctl peer --help for more␣
↪information)
```

And the Liqo pods should be up and running:

```
kubectl get pods -n liqo
```

```
NAME                                      READY   STATUS    RESTARTS   AGE
liqo-auth-74c795d84c-x2p6h                1/1     Running   0          2m8s
liqo-controller-manager-6c688c777f-4lv9d  1/1     Running   0          2m8s
liqo-crd-replicator-6c64df5457-bq4tv      1/1     Running   0          2m8s
liqo-gateway-78cf7bb86b-pkdpt             1/1     Running   0          2m8s
liqo-metric-agent-5667b979c7-snmdg        1/1     Running   0          2m8s
liqo-network-manager-5b5cdcfcf7-scvd9     1/1     Running   0          2m8s
liqo-proxy-6674dd7bbd-kr2ls               1/1     Running   0          2m8s
liqo-route-7wsrx                          1/1     Running   0          2m8s
liqo-route-sz75m                          1/1     Running   0          2m8s
```

In addition, you can check the installation status, and the main Liqo configuration parameters, using:

```
liqoctl status
```

12.3 Peer two clusters

Once Liqo is installed in your clusters, you can establish new *peerings*. In this example, since the two API Servers are mutually reachable, you will use the *out-of-band peering approach*.

First, get the *peer command* from the *Milan* cluster:

```
liqoctl generate peer-command --kubeconfig "$KUBECONFIG_MILAN"
```

Second, copy and paste the command in the *Rome* cluster:

```
liqoctl peer out-of-band milan --auth-url [redacted] --cluster-id [redacted] --auth-
↪token [redacted]
```

Now, the Liqo control plane in the *Rome* cluster will contact the provided authentication endpoint providing the token to the *Milan* cluster to get its Kubernetes identity.

You can check the peering status by running:

```
kubectl get foreignclusters
```

The output should look like the following, indicating that the cross-cluster network tunnel has been established, and an outgoing peering is currently active (i.e., the *Rome* cluster can offload workloads to the *Milan* one, but not vice versa):

```
NAME    TYPE        OUTGOING PEERING   INCOMING PEERING   NETWORKING    AUTHENTICATION  ␣
↪AGE
milan   OutOfBand   Established        None               Established   Established     ␣
↪12s
```

At the same time, you should see a virtual node (`liqo-milan`) in addition to your physical nodes:

```
kubectl get nodes
```

```
NAME                 STATUS   ROLES           AGE     VERSION
liqo-milan           Ready    agent           27s     v1.25.0
rome-control-plane   Ready    control-plane   7m6s    v1.25.0
rome-worker          Ready    <none>          6m33s   v1.25.0
```

In addition, you can check the peering status, and retrieve more advanced information, using:

```
liqoctl status peer milan
```

12.4 Leverage remote resources

Now, you can deploy a standard Kubernetes application in a multi-cluster environment as you would do in a single cluster scenario (i.e. no modification is required).

12.4.1 Start a hello world application

If you want to deploy an application that is scheduled onto Liqo virtual nodes, you should first create a namespace where your pod will be started. Then tell Liqo to make this namespace eligible for the pod offloading.

```
kubectl create namespace liqo-demo
liqoctl offload namespace liqo-demo
```

The `liqoctl offload namespace` command enables Liqo to offload the namespace to the remote cluster. Since no further configuration is provided, Liqo will add a suffix to the namespace name to make it unique on the remote cluster (see the dedicated *usage page* for additional information concerning namespace offloading configurations).

Note

The virtual nodes have a taint that prevents the pods from being scheduled on them. The Liqo webhook will add the toleration for this taint to the pods created in the liqo-enabled namespaces.

Then, you can deploy a demo application in the `liqo-demo` namespace of the local cluster:

```
kubectl apply -f ./manifests/hello-world.yaml -n liqo-demo
```

The `hello-world.yaml` file represents a simple `nginx` service. It contains two pods running an `nginx` image and a Service exposing the pods to the cluster. One pods is running in the local cluster, while the other is forced to be scheduled on the remote cluster.

Info

Differently from the traditional examples, the above deployment introduces an *affinity* constraint. This forces Kubernetes to schedule the first pod (i.e. `nginx-local`) on a physical node and the second (i.e. `nginx-remote`) on a virtual node. Virtual nodes are like traditional Kubernetes nodes, but they represent remote clusters and have the `liqo.io/type: virtual-node` label.

When the affinity constraint is not specified, the Kubernetes scheduler selects the best hosting node based on the available resources. Hence, each pod can be scheduled either in the *local* cluster or in the *remote* cluster.

Now you can check the status of the pods. The output should be similar to the one below, confirming that one `nginx` pod is running locally; while the other is hosted by the virtual node (i.e., `liqo-milan`).

```
kubectl get pod -n liqo-demo -o wide
```

And the output should look like this:

```
NAME           READY   STATUS    RESTARTS   AGE   IP            NODE          NOMINATED␣
↪NODE   READINESS GATES
nginx-local    1/1     Running   0          10s   10.200.1.11   rome-worker   <none>    ␣
↪       <none>
nginx-remote   1/1     Running   0          9s    10.202.1.10   liqo-milan    <none>    ␣
↪       <none>
```

Check the pod connectivity

Once both pods are correctly running, it is possible to check one of the abstractions introduced by Liqo. Indeed, Liqo enables each pod to be transparently contacted by every other pod and physical node (according to the Kubernetes model), regardless of whether it is hosted by the *local* or by the *remote* cluster.

First, let's retrieve the IP address of the `nginx` pods:

```
LOCAL_POD_IP=$(kubectl get pod nginx-local -n liqo-demo --template={{.status.podIP}})
REMOTE_POD_IP=$(kubectl get pod nginx-remote -n liqo-demo --template={{.status.podIP}})
echo "Local Pod IP: ${LOCAL_POD_IP} - Remote Pod IP: ${REMOTE_POD_IP}"
```

You can fire up a pod and run `curl` from inside the cluster:

```
kubectl run --image=curlimages/curl curl -n default -it --rm --restart=Never -- curl $
↪{LOCAL_POD_IP}
kubectl run --image=curlimages/curl curl -n default -it --rm --restart=Never -- curl $
↪{REMOTE_POD_IP}
```

Both commands should lead to a successful outcome (i.e., return a demo web page), regardless of whether each pod is executed locally or remotely.

12.4.2 Expose the pods through a Service

The above `hello-world.yaml` manifest additionally creates a Service which is designed to serve traffic to the previously deployed pods. This is a traditional Kubernetes Service and can work with Liqo with no modifications.

Indeed, inspecting the Service, it is possible to observe that both `nginx` pods are correctly specified as endpoints. Nonetheless, it is worth noticing that the first endpoint (i.e. `10.200.1.10:80` in this example) refers to a pod running in the *local* cluster, while the second one (i.e. `10.202.1.9:80`) points to a pod hosted by the *remote* cluster.

```
kubectl describe service liqo-demo -n liqo-demo
```

```
Name:              liqo-demo
Namespace:         liqo-demo
Labels:            <none>
Annotations:       <none>
Selector:          app=liqo-demo
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.94.41.143
IPs:               10.94.41.143
Port:              web  80/TCP
TargetPort:        web/TCP
Endpoints:         10.200.1.11:80,10.202.1.10:80
Session Affinity:  None
Events:
  Type    Reason                Age                From                      Message
  ----    ------                ----               ----                      -------
  Normal  SuccessfulReflection  51s (x2 over 51s)  liqo-service-reflection  Successfully␣
↪reflected object to cluster "milan"
```

Check the Service connectivity

It is now possible to contact the Service: as usual, Kubernetes will forward the HTTP request to one of the available back-end pods. Additionally, all traditional mechanisms still work seamlessly (e.g. DNS discovery), even though one of the pods is actually running in a *remote* cluster.

You can fire up a pod and run `curl` from inside the cluster:

```
kubectl run --image=curlimages/curl curl -n default -it --rm --restart=Never -- \
    curl --silent liqo-demo.liqo-demo.svc.cluster.local | grep 'Server'
```

Note

Executing the previous command multiple times, you will observe that part of the requests are answered by the pod running in the *local* cluster, and in part by that in the *remote* cluster (i.e., the Server value changes).

12.5 Play with a microservice application

It is very common in a cloud-based environment to deploy microservices applications composed of many pods interacting among each other. This pattern is transparently supported by Liqo and the virtual cluster abstraction.

You can play with a microservices application provided by Google, which includes multiple cooperating Services leveraging different networking protocols:

```
kubectl apply -k ./manifests/demo-application -n liqo-demo
```

By default, Kubernetes schedules each pod either in the local or in the remote cluster, optimizing each deployment based on the available resources. However, you can play with *affinity* constraints to force Kubernetes to schedule of each component in a specific location, and see that everything continues to work smoothly. Specifically, the manifest above forces the frontend component to be executed in the *local* cluster, as this is required to enable *port-forwarding*, which is leveraged below.

Each demo component is exposed as a Service and accessed by other components. However, given that nobody knows, a priori, where each Service will be deployed (either locally or in the remote cluster), Liqo replicates all Kubernetes Services across both clusters, although the corresponding pod may be running only in one location. Hence, each microservice deployed across clusters can reach the others seamlessly: independently of the cluster a pod is deployed in, each pod can contact other Services and leverage the traditional Kubernetes discovery mechanisms (e.g., DNS discovery and environment variables).

Additionally, several other objects (e.g. `ConfigMaps` and `Secrets`) inside a namespace are replicated in the remote cluster within the *twin namespace*, thus, ensuring that complex applications can work seamlessly across clusters.

12.5.1 Observe the application deployment

Once the demo application manifest is applied, you can observe the creation of the different pods:

```
watch kubectl get pods -n liqo-demo -o wide
```

At steady-state, you should see an output similar to the following. Different pods may be hosted by either the local nodes (*rome-worker* in the example below) or remote cluster (*liqo-milan* in the example below), depending on the scheduling decisions.

```
NAME                                 READY   STATUS    RESTARTS        AGE     IP␣
↪         NODE          NOMINATED NODE   READINESS GATES
adservice-84cdf76d7d-6s8pq           1/1     Running   0               5m1s    10.
↪202.1.11   liqo-milan    <none>           <none>
cartservice-5c9c9c7b4-w49gr          1/1     Running   0               5m1s    10.
↪202.1.12   liqo-milan    <none>           <none>
checkoutservice-6cb9bb8cd8-5w2ht     1/1     Running   0               5m1s    10.
↪202.1.13   liqo-milan    <none>           <none>
currencyservice-7d4bd86676-5b5rq     1/1     Running   0               5m1s    10.
↪202.1.14   liqo-milan    <none>           <none>
emailservice-c9b45cdb-6zjrk          1/1     Running   0               5m1s    10.
↪202.1.15   liqo-milan    <none>           <none>
frontend-58b9b98d84-hg4xz            1/1     Running   0               5m1s    10.
↪200.1.13   rome-worker   <none>           <none>
loadgenerator-5f8cd58cd4-wvqqq       1/1     Running   0               5m1s    10.
↪202.1.16   liqo-milan    <none>           <none>
nginx-local                          1/1     Running   0               7m35s   10.
↪200.1.11   rome-worker   <none>           <none>
```

(continues on next page)

(continued from previous page)

```
nginx-remote                              1/1     Running   0               7m34s   10.
↪202.1.10   liqo-milan    <none>           <none>
paymentservice-69558cf7bb-v4zjw           1/1     Running   0               5m      10.
↪202.1.17   liqo-milan    <none>           <none>
productcatalogservice-55c58b57cb-k8mfq    1/1     Running   0               5m      10.
↪202.1.18   liqo-milan    <none>           <none>
recommendationservice-55cd66cf64-6fz9w    1/1     Running   0               5m      10.
↪202.1.19   liqo-milan    <none>           <none>
redis-cart-5d45978b94-wjd97               1/1     Running   0               5m      10.
↪202.1.20   liqo-milan    <none>           <none>
shippingservice-5df47fc86-f867j           1/1     Running   0               4m59s   10.
↪202.1.21   liqo-milan    <none>           <none>
```

12.5.2 Access the demo application

Once the deployment is up and running, you can start using the demo application and verify that everything works correctly, even if its components are distributed across multiple Kubernetes clusters.

By default, the frontend web-page is exposed through a `LoadBalancer` Service, which can be inspected using:

```
kubectl get service -n liqo-demo frontend-external
```

Leverage `kubectl port-forward` to forward the requests from your local machine (i.e., `http://localhost:8080`) to the frontend Service:

```
kubectl port-forward -n liqo-demo service/frontend-external 8080:80
```

Open the http://localhost:8080 page in your browser and enjoy the demo application.

12.6 Tear down the playground

Our example is finished; now we can remove all the created resources and tear down the playground.

12.6.1 Unoffload namespaces

Before starting the uninstallation process, make sure that all namespaces are unoffloaded:

```
liqoctl unoffload namespace liqo-demo
```

Every pod that was offloaded to a remote cluster is going to be rescheduled onto the local cluster.

12.6.2 Revoke peerings

Similarly, make sure that all the peerings are revoked:

```
liqoctl unpeer out-of-band milan
```

At the end of the process, the virtual node is removed from the local cluster.

12.6.3 Uninstall Liqo

Now you can uninstall Liqo from your clusters with *liqoctl*:

```
liqoctl uninstall
liqoctl uninstall --kubeconfig="$KUBECONFIG_MILAN"
```

Purge

By default the Liqo CRDs will remain in the cluster, but they can be removed with the `--purge` flag:

```
liqoctl uninstall --purge
liqoctl uninstall --purge --kubeconfig="$KUBECONFIG_MILAN"
```

12.6.4 Destroy clusters

To teardown the KinD clusters, you can issue:

```
kind delete cluster --name rome
kind delete cluster --name milan
```

CHAPTER

THIRTEEN

OFFLOADING WITH POLICIES

This tutorial aims to guide you through a tour to learn how to use the core Liqo features. You will learn how to tune namespace offloading, and specify the target clusters through the *cluster selector* concept.

More specifically, you will configure a scenario composed of a *single entry point cluster* leveraged for the deployment of the applications (i.e., the *Venice* cluster, located in *north* Italy) and two *worker clusters* characterized by different geographical regions (i.e., the *Florence* and *Naples* clusters, respectively located in *center* and *south* Italy). Then, you will offload a given namespace (and the applications contained therein) to a subset of the worker clusters (i.e., only to the *Naples* cluster), while allowing pods to be also scheduled on the local cluster (i.e., the *Venice* one).

13.1 Provision the playground

First, check that you are compliant with the *requirements*.

Then, let's open a terminal on your machine and launch the following script, which creates the three above-mentioned clusters with KinD and installs Liqo on all of them. Each cluster is made by a single combined control-plane + worker node.

```
git clone https://github.com/liqotech/liqo.git
cd liqo
git checkout v0.9.1
cd examples/offloading-with-policies
./setup.sh
```

Export the kubeconfigs environment variables to use them in the rest of the tutorial:

```
export KUBECONFIG="$PWD/liqo_kubeconf_venice"
export KUBECONFIG_FLORENCE="$PWD/liqo_kubeconf_florence"
export KUBECONFIG_NAPLES="$PWD/liqo_kubeconf_naples"
```

Note

We suggest exporting the kubeconfig of the first cluster as default (i.e., `KUBECONFIG`), since it will be the entry point of the virtual cluster and you will mainly interact with it.

At this point, you should have three clusters with Liqo installed on them. The setup script named them **venice**, **florence** and **naples**, and respectively configured the following cluster labels:

- *venice*: `topology.liqo.io/region=north`
- *florence*: `topology.liqo.io/region=center`

- *naples*: `topology.liqo.io/region=south`

You can check that the clusters are correctly labeled through:

```
liqoctl status
liqoctl --kubeconfig $KUBECONFIG_FLORENCE status
liqoctl --kubeconfig $KUBECONFIG_NAPLES status
```

These labels will be propagated to the virtual nodes corresponding to each cluster. In this way, you can easily identify the clusters through their characterizing labels, and define the appropriate scheduling policies.

13.2 Peer the clusters

Once Liqo is installed in your clusters, you can establish new *peerings*. In this example, since the two API Servers are mutually reachable, you will use the *out-of-band peering approach*.

To implement the desired scenario, let's first retrieve the *peer command* from the *Florence* and *Naples* clusters:

```
PEER_FLORENCE=$(liqoctl generate peer-command --only-command --kubeconfig $KUBECONFIG_
↪FLORENCE)
PEER_NAPLES=$(liqoctl generate peer-command --only-command --kubeconfig $KUBECONFIG_
↪NAPLES)
```

Then, establish the peerings from the *Venice* cluster:

```
echo "$PEER_FLORENCE" | bash
echo "$PEER_NAPLES" | bash
```

When the above commands return successfully, you can check the peering status by running:

```
kubectl get foreignclusters
```

The output should look like the following, indicating that an outgoing peering is currently active towards both the *Florence* and the *Naples* clusters, as well as the cross-cluster network tunnels have been established:

```
NAME       TYPE        OUTGOING PEERING   INCOMING PEERING   NETWORKING   ␣
↪AUTHENTICATION   AGE
florence   OutOfBand   Established        None               Established   Established  ␣
↪   111s
naples     OutOfBand   Established        None               Established   Established  ␣
↪   98s
```

Additionally, you should have two new virtual nodes in the *Venice* cluster, characterized by the install-time provided labels:

```
kubectl get node --selector=liqo.io/type=virtual-node --show-labels
```

```
NAME            STATUS   ROLES   AGE   VERSION   LABELS
liqo-florence   Ready    agent   19s   v1.25.0   liqo.io/remote-cluster-id=5f3b5abd-cccb-
↪4f75-931b-d6b1ca95fa7d,liqo.io/type=virtual-node,topology.liqo.io/region=center
liqo-naples     Ready    agent   14s   v1.25.0   liqo.io/remote-cluster-id=edc8c24a-4c11-
↪48b8-8b0e-2a95cf7464af,liqo.io/type=virtual-node,topology.liqo.io/region=south
```

Note

Some of the default labels were omitted for the sake of clarity.

13.3 Tune namespace offloading

Now, let's suppose you want to deploy an application that needs to be scheduled in the *north* and in the *south* region, but not in the *center* one. This constraint needs to be respected at the infrastructural level: the dev team does not need to be aware of required affinities and/or node selectors, nor it should be able to bypass them.

First, you should create a new namespace in the *Venice* cluster, which will host the application:

```
kubectl create namespace liqo-demo
```

Then, enable Liqo offloading for that namespace:

```
liqoctl offload namespace liqo-demo \
  --namespace-mapping-strategy EnforceSameName \
  --pod-offloading-strategy LocalAndRemote \
  --selector 'topology.liqo.io/region=south'
```

The above command configures the following aspects (see the dedicated *usage page* for additional information concerning namespace offloading configurations):

- the `liqo-demo` namespace is replicated with the same name in the other clusters.
- the `liqo-demo` namespace, and the contained resources, are offloaded only to the clusters with the `topology.liqo.io/region=south` label.
- the pods living in the `liqo-demo` namespace are free to be scheduled onto both physical and virtual nodes.

The *NamespaceOffloading* resource created by *liqoctl* in the `liqo-demo` namespace exposes the status of the offloading process, including a global *OffloadingPhase*, which is expected to be `Ready`, and a list of *RemoteNamespaceConditions*, one for each remote cluster.

In this case:

- the *Florence* cluster has not been selected to offload the namespace `liqo-demo`, since it does not match the cluster selector;
- the *Naples* cluster has been selected to offload the namespace `liqo-demo`, and the namespace has been correctly created.

```
kubectl get namespaceoffloadings offloading -n liqo-demo -o yaml
```

```
...
status:
  observedGeneration: 1
  offloadingPhase: Ready
  remoteNamespaceName: liqo-demo
  remoteNamespacesConditions:
    florence-7ab115:
    - lastTransitionTime: "2023-01-30T09:50:05Z"
      message: The remote cluster has not been selected through the ClusterSelector field
```

(continues on next page)

(continued from previous page)

```
      reason: ClusterNotSelected
      status: "False"
      type: OffloadingRequired
    naples-5eada1:
    - lastTransitionTime: "2023-01-30T09:50:05Z"
      message: The remote cluster has been selected through the ClusterSelector field
      reason: ClusterSelected
      status: "True"
      type: OffloadingRequired
    - lastTransitionTime: "2023-01-30T09:50:05Z"
      message: Namespace correctly offloaded to the remote cluster
      reason: NamespaceCreated
      status: "True"
      type: Ready
```

Indeed, if you query for the namespaces in the *Naples* cluster, you should see the following output, confirming that the remote namespace has been correctly created by Liqo:

```
kubectl get namespaces liqo-demo --kubeconfig="$KUBECONFIG_NAPLES"
```

```
NAME        STATUS   AGE
liqo-demo   Active   70s
```

Instead, the same command executed in the *Florence* cluster should return an error, as the namespace has not been replicated:

```
kubectl get namespaces liqo-demo --kubeconfig="$KUBECONFIG_FLORENCE"
```

```
Error from server (NotFound): namespaces "liqo-demo" not found
```

13.4 Deploy applications

All constraints specified during namespace offloading are automatically enforced by Liqo, and merged with other pod-level specifications.

To verify this, you can now create two deployments in the `liqo-demo` namespace, characterized by additional *NodeAffinity* constraints. More precisely, one (`app-south`) is forced to be scheduled onto the virtual node representing the *Naples* cluster, while the other (`app-center`) is forced onto the *Florence* virtual cluster (which is incompatible with the namespace-level constraints).

```
kubectl apply -f ./manifests/deploy.yaml -n liqo-demo
```

Checking the pod status, it is possible to verify that one has been scheduled onto the *Naples* cluster, and it is correctly running, while the other remained *Pending* due to conflicting requirements (i.e., no node is available to satisfy all its constraints).

```
kubectl get pod -n liqo-demo -o wide
```

```
NAME                           READY   STATUS    RESTARTS   AGE   IP            NODE     ␣
↪    NOMINATED NODE   READINESS GATES
app-center-58d8ff79c9-xf6pz    0/1     Pending   0          27s   <none>        <none>   ␣
↪    <none>           <none>
app-south-545766885-zn4nx      1/1     Running   0          27s   10.204.0.13   liqo-
↪naples   <none>           <none>
```

Note

You can remove the conflicting node affinity from the `app-center` deployment, and check that the generated pod gets scheduled onto either the *Venice* (i.e., locally) or the *Naples* cluster, as constrained by the namespace offloading configuration.

13.5 Tear down the playground

Our example is finished; now we can remove all the created resources and tear down the playground.

13.5.1 Unoffload namespaces

Before starting the uninstallation process, make sure that all namespaces are unoffloaded:

```
liqoctl unoffload namespace liqo-demo
```

Every pod that was offloaded to a remote cluster is going to be rescheduled onto the local cluster.

13.5.2 Revoke peerings

Similarly, make sure that all the peerings are revoked:

```
liqoctl unpeer out-of-band florence
liqoctl unpeer out-of-band naples
```

At the end of the process, the virtual nodes are removed from the local cluster.

13.5.3 Uninstall Liqo

Now you can uninstall Liqo from your clusters with *liqoctl*:

```
liqoctl uninstall
liqoctl uninstall --kubeconfig="$KUBECONFIG_FLORENCE"
liqoctl uninstall --kubeconfig="$KUBECONFIG_NAPLES"
```

Purge

By default the Liqo CRDs will remain in the cluster, but they can be removed with the `--purge` flag:

```
liqoctl uninstall --purge
liqoctl uninstall --kubeconfig="$KUBECONFIG_FLORENCE" --purge
liqoctl uninstall --kubeconfig="$KUBECONFIG_NAPLES" --purge
```

13.5.4 Destroy clusters

To teardown the KinD clusters, you can issue:

```
kind delete cluster --name venice
kind delete cluster --name florence
kind delete cluster --name naples
```

CHAPTER

FOURTEEN

OFFLOADING A SERVICE

In this tutorial you will learn how to create a multi-cluster Service and how to consume it from each connected cluster.

Specifically, you will deploy an application in a first cluster (*London*) and then offload the corresponding Service and transparently consume it from a second cluster (*New York*).

14.1 Provision the playground

First, check that you are compliant with the *requirements*.

Then, let's open a terminal on your machine and launch the following script, which creates the two above-mentioned clusters with KinD and installs Liqo on them. Each cluster is made by a single combined control-plane + worker node.

```
git clone https://github.com/liqotech/liqo.git
cd liqo
git checkout v0.9.1
cd examples/service-offloading
./setup.sh
```

Export the kubeconfigs environment variables to use them in the rest of the tutorial:

```
export KUBECONFIG="$PWD/liqo_kubeconf_london"
export KUBECONFIG_NEWYORK="$PWD/liqo_kubeconf_newyork"
```

Note

We suggest exporting the kubeconfig of the first cluster as default (i.e., `KUBECONFIG`), since it will be the entry point of the virtual cluster and you will mainly interact with it.

At this point, you should have two clusters with Liqo installed on them. The setup script named them **london** and **newyork**.

14.2 Peer the clusters

Once Liqo is installed in your clusters, you can establish new *peerings*. In this example, since the two API Servers are mutually reachable, you will use the *out-of-band peering approach*.

To implement the desired scenario, let's first retrieve the *peer command* from the *New York* cluster:

```
PEER_NEW_YORK=$(liqoctl generate peer-command --only-command --kubeconfig $KUBECONFIG_
↪NEWYORK)
```

Then, establish the peering from the *London* cluster:

```
echo "$PEER_NEW_YORK" | bash
```

When the above command returns successfully, you can check the peering status by running:

```
kubectl get foreignclusters
```

The output should look like the following, indicating that an outgoing peering is currently active towards the *New York* cluster, as well as that the cross-cluster network tunnel has been established:

```
NAME      TYPE        OUTGOING PEERING   INCOMING PEERING   NETWORKING    AUTHENTICATION␣
↪   AGE
newyork   OutOfBand   Established        None               Established   Established   ␣
↪   61s
```

14.3 Offload a service

Now, let's deploy a simple application composed of a *Deployment* and a *Service* in the *London* cluster.

First, you should create a hosting namespace in the *London* cluster:

```
kubectl create namespace liqo-demo
```

Then, deploy the application in the *London* cluster:

```
kubectl apply -f manifests/app.yaml -n liqo-demo
```

At this moment, you have an HTTP application serving JSON data through a Service, and running in the *London* cluster (i.e., locally). If you look at the *New York* cluster, you will not see the application yet.

To make it visible, you need to enable the Liqo offloading of the Services in the desired namespace to the *New York* cluster:

```
liqoctl offload namespace liqo-demo \
    --namespace-mapping-strategy EnforceSameName \
    --pod-offloading-strategy Local
```

This command enables the offloading of the Services in the *London* cluster to the *New York* cluster and sets:

- the namespace mapping strategy to *EnforceSameName*, which means that the namespace in the remote cluster is created with the same name as of the local one. This is particularly useful when you want to consume the Services in the remote cluster using the Kubernetes DNS service discovery (i.e. with `svc-name.namespace-name.svc.cluster.local`).

- the pod offloading strategy to *Local*, which means that the pods running in this namespace will be kept local and not scheduled on virtual nodes (i.e., no pod is offloaded to remote clusters).

Refer to the dedicated *usage page* for additional information concerning namespace offloading configurations.

Some seconds later, you should see that the *Service* has been replicated by the *resource reflection process*, and is now available in the *New York* cluster:

```
kubectl get services --namespace liqo-demo --kubeconfig "$KUBECONFIG_NEWYORK"
```

```
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
flights-service   ClusterIP   10.81.139.132   <none>        7999/TCP   14s
```

The Service is characterized by a different *ClusterIP* address in the two clusters, since each cluster handles it independently. Additionally, you can also check that there is no application pod running in the *New York* cluster:

```
kubectl get pods --namespace liqo-demo --kubeconfig "$KUBECONFIG_NEWYORK"
```

```
No resources found in liqo-demo namespace.
```

14.3.1 Consume the service

Let's now consume the Service from both clusters from a different pod (e.g., a temporary shell).

Starting from the *London* cluster:

```
kubectl run consumer --rm -i --tty --image dwdraju/alpine-curl-jq -- /bin/sh
```

When the shell is ready, you can access the Service with `curl`:

```
curl -s -H 'accept: application/json' http://flights-service.liqo-demo:7999/schedule |␣
↪jq .
```

A similar result is obtained executing the same command in a shell running in the *New York* cluster, although the backend pod is effectively running in the *London* cluster:

```
kubectl run consumer --rm -i --tty --image dwdraju/alpine-curl-jq \
    --kubeconfig $KUBECONFIG_NEWYORK -- /bin/sh
```

```
curl -s -H 'accept: application/json' http://flights-service.liqo-demo:7999/schedule |␣
↪jq .
```

This quick example demonstrated how Liqo can **upgrade *ClusterIP* Services to multi-cluster Services**, allowing your local pods to transparently serve traffic originating from remote clusters with no additional configuration neither in the local cluster and/or applications nor in the remote ones.

14.4 Tear down the playground

Our example is finished; now we can remove all the created resources and tear down the playground.

14.4.1 Unoffload namespaces

Before starting the uninstallation process, make sure that all namespaces are unoffloaded:

```
liqoctl unoffload namespace liqo-demo
```

Every pod that was offloaded to a remote cluster is going to be rescheduled onto the local cluster.

14.4.2 Revoke peerings

Similarly, make sure that all the peerings are revoked:

```
liqoctl unpeer out-of-band newyork
```

At the end of the process, the virtual node is removed from the local cluster.

14.4.3 Uninstall Liqo

Now you can uninstall Liqo from your clusters with *liqoctl*:

```
liqoctl uninstall
liqoctl uninstall --kubeconfig="$KUBECONFIG_NEWYORK"
```

Purge

By default the Liqo CRDs will remain in the cluster, but they can be removed with the `--purge` flag:

```
liqoctl uninstall --purge
liqoctl uninstall --kubeconfig="$KUBECONFIG_NEWYORK" --purge
```

14.4.4 Destroy clusters

To teardown the KinD clusters, you can issue:

```
kind delete cluster --name london
kind delete cluster --name newyork
```

CHAPTER

FIFTEEN

STATEFUL APPLICATIONS

This tutorial demonstrates how to use the core Liqo features to deploy stateful applications. In particular, you will deploy a multi-master *mariadb-galera* database across a multi-cluster environment (composed of two clusters, respectively identified as *Turin* and *Lyon*), hence replicating the data in multiple regions.

15.1 Provision the playground

First, check that you are compliant with the *requirements*.

Then, let's open a terminal on your machine and launch the following script, which creates the two above-mentioned clusters with KinD and installs Liqo on them. Each cluster is made by a single combined control-plane + worker node.

```
git clone https://github.com/liqotech/liqo.git
cd liqo
git checkout v0.9.1
cd examples/stateful-applications
./setup.sh
```

Export the kubeconfigs environment variables to use them in the rest of the tutorial:

```
export KUBECONFIG="$PWD/liqo_kubeconf_turin"
export KUBECONFIG_LYON="$PWD/liqo_kubeconf_lyon"
```

Note

The install script creates two clusters with no overlapping pod CIDRs. This is required by the *mariadb-galera* application to work correctly. Given it needs to know the real IP of the connected masters, it will not work correctly when natting is enabled.

Note

We suggest exporting the kubeconfig of the first cluster as default (i.e., `KUBECONFIG`), since it will be the entry point of the virtual cluster and you will mainly interact with it.

15.2 Peer the clusters

Once Liqo is installed in your clusters, you can establish new *peerings*. In this example, since the two API Servers are mutually reachable, you will use the *out-of-band peering approach*.

To implement the desired scenario, let's first retrieve the *peer command* from the *Lyon* cluster:

```
PEER_LYON=$(liqoctl generate peer-command --only-command --kubeconfig $KUBECONFIG_LYON)
```

Then, establish the peering from the *Turin* cluster:

```
echo "$PEER_LYON" | bash
```

When the above command returns successfully, you can check the peering status by running:

```
kubectl get foreignclusters
```

The output should look like the following, indicating that an outgoing peering is currently active towards the *Lyon* cluster,, as well as that the cross-cluster network tunnel has been established:

```
NAME   TYPE        OUTGOING PEERING   INCOMING PEERING   NETWORKING    AUTHENTICATION  ␣
↪AGE
lyon   OutOfBand   Established        None               Established   Established     ␣
↪1m28s
```

15.3 Deploy a stateful application

In this step, you will deploy a *mariadb-galera* database using the Bitnami helm chart.

First, you need to add the helm repository:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

Then, create the namespace and offload it to remote clusters:

```
kubectl create namespace liqo-demo
liqoctl offload namespace liqo-demo --namespace-mapping-strategy EnforceSameName
```

This command will create a twin `liqo-demo` namespace in the *Lyon* cluster. Refer to the dedicated *usage page* for additional information concerning namespace offloading configurations.

Now, deploy the helm chart:

```
helm install db bitnami/mariadb-galera -n liqo-demo -f manifests/values.yaml
```

The release is configured to:

- have two replicas;
- spread the replicas across the cluster (i.e., a hard pod anti-affinity is set);
- use the `liqo` *virtual storage class*.

Check that these constraints are met by typing:

```
kubectl get pods -n liqo-demo -o wide
```

After a while (the startup process might require a few minutes), you should see two replicas of a *StatefulSet* spread over two different nodes (one local and one remote).

```
NAME                  READY   STATUS    RESTARTS   AGE     IP             NODE          ␣
↪      NOMINATED NODE   READINESS GATES
db-mariadb-galera-0   1/1     Running   0          3m13s   10.210.0.15    liqo-lyon     ␣
↪      <none>           <none>
db-mariadb-galera-1   1/1     Running   0          2m6s    10.200.0.17    turin-control-
↪plane   <none>           <none>
```

15.4 Consume the database

When the database is up and running, check that it is operating as expected executing a simple SQL client in your cluster:

```
kubectl run db-mariadb-galera-client --rm --tty -i \
    --restart='Never' --namespace default \
    --image docker.io/bitnami/mariadb-galera:10.6.7-debian-10-r56 \
    --command \
    -- mysql -h db-mariadb-galera.liqo-demo -uuser -ppassword my_database
```

And then create an example table and insert some data:

```
CREATE TABLE People (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);

INSERT INTO People
    (PersonID, LastName, FirstName, Address, City)
    VALUES
    (1, 'Smith', 'John', '123 Main St', 'Anytown');
```

You are now able to query the database and grab the data:

```
SELECT * FROM People;
```

```
+----------+----------+-----------+-------------+---------+
| PersonID | LastName | FirstName | Address     | City    |
+----------+----------+-----------+-------------+---------+
|        1 | Smith    | John      | 123 Main St | Anytown |
+----------+----------+-----------+-------------+---------+
1 row in set (0.000 sec)
```

15.4.1 Database failures toleration

With this setup the applications running on a cluster can tolerate failures of the local database replica.

This can be checked deleting one of the replicas:

```
kubectl delete pod db-mariadb-galera-0 -n liqo-demo
```

And querying again for your data:

```
kubectl run db-mariadb-galera-client --rm --tty -i \
    --restart='Never' --namespace default \
    --image docker.io/bitnami/mariadb-galera:10.6.7-debian-10-r56 \
    --command \
      -- mysql -h db-mariadb-galera.liqo-demo -uuser -ppassword my_database \
      --execute "SELECT * FROM People;"
```

Pro-tip

Try deleting the other replica and query again.

NOTE: at least one of the two replicas should be always running, be careful deleting all of them.

Note

You can run exactly the same commands to query the data from the other cluster, and you will get the same results.

15.5 Tear down the playground

Our example is finished; now we can remove all the created resources and tear down the playground.

15.5.1 Unoffload namespaces

Before starting the uninstallation process, make sure that all namespaces are unoffloaded:

```
liqoctl unoffload namespace liqo-demo
```

Every pod that was offloaded to a remote cluster is going to be rescheduled onto the local cluster.

15.5.2 Revoke peerings

Similarly, make sure that all the peerings are revoked:

```
liqoctl unpeer out-of-band lyon
```

At the end of the process, the virtual node is removed from the local cluster.

15.5.3 Uninstall Liqo

Now you can remove Liqo from your clusters with *liqoctl*:

```
liqoctl uninstall
liqoctl uninstall --kubeconfig="$KUBECONFIG_LYON"
```

Purge

By default the Liqo CRDs will remain in the cluster, but they can be removed with the `--purge` flag:

```
liqoctl uninstall --purge
liqoctl uninstall --kubeconfig="$KUBECONFIG_LYON" --purge
```

15.5.4 Destroy clusters

To teardown the KinD clusters, you can issue:

```
kind delete cluster --name turin
kind delete cluster --name lyon
```

CHAPTER

SIXTEEN

GLOBAL INGRESS

In this tutorial, you will learn how to leverage Liqo and K8GB to deploy and expose a multi-cluster application through a *global ingress*. More in detail, this enables improved load balancing and distribution of the external traffic towards the application replicated across multiple clusters.

The figure below outlines the high-level scenario, with a client consuming an application from either cluster 1 (e.g., located in EU) or cluster 2 (e.g., located in the US), based on the endpoint returned by the DNS server.

16.1 Provision the playground

First, check that you are compliant with the *requirements*. Additionally, this example requires k3d to be installed in your system. Specifically, this tool is leveraged instead of KinD to match the K8GB Sample Demo.

To provision the playground, clone the Liqo repository and run the setup script:

```
git clone https://github.com/liqotech/liqo.git
cd liqo
git checkout v0.9.1
cd examples/global-ingress
./setup.sh
```

The setup script creates three k3s clusters and deploys the appropriate infrastructural application on top of them, as detailed in the following:

- **edgedns**: this cluster will be used to deploy the DNS service. In a production environment, this should be an external DNS service (e.g. AWS Route53). It includes the Bind Server (manifests in `manifests/edge` folder).
- **gslb-eu** and **gslb-us**: these clusters will be used to deploy the application. They include:
  - ExternalDNS: it is responsible for configuring the DNS entries.
  - Ingress Nginx: it is responsible for handling the local ingress traffic.
  - K8GB: it configures the multi-cluster ingress.
  - *Liqo*: it enables the application to spread across multiple clusters, and takes care of reflecting the required resources.

Export the kubeconfigs environment variables to use them in the rest of the tutorial:

```
export KUBECONFIG_DNS=$(k3d kubeconfig write edgedns)
export KUBECONFIG=$(k3d kubeconfig write gslb-eu)
export KUBECONFIG_US=$(k3d kubeconfig write gslb-us)
```

Note

We suggest exporting the kubeconfig of the *gslb-eu* as default (i.e., `KUBECONFIG`), since it will be the entry point of the virtual cluster and you will mainly interact with it.

16.2 Peer the clusters

Once Liqo is installed in your clusters, you can establish new *peerings*. In this example, since the two API Servers are mutually reachable, you will use the *out-of-band peering approach*.

Specifically, to implement the desired scenario, you should enable a peering from the *gslb-eu* cluster to the *gslb-us* cluster. This will allow Liqo to *offload workloads and reflect services* from the first cluster to the second cluster.

To proceed, first generate a new *peer command* from the *gslb-us* cluster:

```
PEER_US=$(liqoctl generate peer-command --only-command --kubeconfig $KUBECONFIG_US)
```

And then, run the generated command from the *gslb-eu* cluster:

```
echo "$PEER_US" | bash
```

When the above command returns successfully, you can check the peering status by running:

```
kubectl get foreignclusters
```

The output should look like the following, indicating that an outgoing peering is currently active towards the *gslb-us* cluster, as well as that the cross-cluster network tunnel has been established:

```
NAME      TYPE        OUTGOING PEERING   INCOMING PEERING   NETWORKING    AUTHENTICATION␣
↪  AGE
gslb-us   OutOfBand   Established        None               Established   Established   ␣
↪  57s
```

Additionally, you should see a new virtual node (`liqo-gslb-us`) in the *gslb-eu* cluster, and representing the whole *gslb-us* cluster. Every pod scheduled onto this node will be automatically offloaded to the remote cluster by Liqo.

```
kubectl get node --selector=liqo.io/type=virtual-node
```

The output should be similar to:

```
NAME           STATUS   ROLES   AGE   VERSION
liqo-gslb-us   Ready    agent   14s   v1.25.0+k3s1
```

16.3 Deploy an application

Now that the Liqo peering is established, and the virtual node is ready, it is possible to proceed deploying the *podinfo* demo application. This application serves a web-page showing different information, including the name of the pod; hence, easily identifying which replica is generating the HTTP response.

First, create a hosting namespace in the *gslb-eu* cluster, and offload it to the remote cluster through Liqo.

```
kubectl create namespace podinfo
liqoctl offload namespace podinfo --namespace-mapping-strategy EnforceSameName
```

At this point, it is possible to deploy the *podinfo* helm chart in the `podinfo` namespace:

```
helm upgrade --install podinfo --namespace podinfo \
    podinfo/podinfo -f manifests/values/podinfo.yaml
```

This chart creates a *Deployment* with a *custom affinity* to ensure that the two frontend replicas are scheduled on different nodes and clusters:

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: node-role.kubernetes.io/control-plane
          operator: DoesNotExist
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: app.kubernetes.io/name
          operator: In
          values:
          - podinfo
      topologyKey: "kubernetes.io/hostname"
```

Additionally, it creates an *Ingress* resource configured with the `k8gb.io/strategy: roundRobin` annotation. This annotation will instruct the K8GB Global Ingress Controller to distribute the traffic across the different clusters.

16.4 Check application spreading

Let's now check that Liqo replicated the ingress resource in both clusters and that each *Nginx Ingress Controller* was able to assign them the correct IPs (different for each cluster).

Note

You can see the output for the second cluster appending the `--kubeconfig $KUBECONFIG_US` flag to each command.

```
kubectl get ingress -n podinfo
```

The output in the *gslb-eu* cluster should be similar to:

```
NAME      CLASS   HOSTS                   ADDRESS                 PORTS   AGE
podinfo   nginx   liqo.cloud.example.com   172.19.0.3,172.19.0.4   80      6m9s
```

While the output in the *gslb-us* cluster should be similar to:

```
NAME      CLASS   HOSTS                   ADDRESS                 PORTS   AGE
podinfo   nginx   liqo.cloud.example.com   172.19.0.5,172.19.0.6   80      6m16s
```

With reference to the output above, the `liqo.cloud.example.com` hostname is served in the demo environment on:

- `172.19.0.3`, `172.19.0.4`: addresses exposed by cluster *gslb-eu*
- `172.19.0.5`, `172.19.0.6`: addresses exposed by cluster *gslb-us*

Each local *K8GB* installation creates a *Gslb* resource with the Ingress information and the given strategy (*RoundRobin* in this case), and *ExternalDNS* populates the DNS records accordingly.

On the *gslb-eu* cluster, the command:

```
kubectl get gslbs.k8gb.absa.oss -n podinfo podinfo -o yaml
```

should return an output along the lines of:

```
apiVersion: k8gb.absa.oss/v1beta1
kind: Gslb
metadata:
  annotations:
    k8gb.io/strategy: roundRobin
  name: podinfo
  namespace: podinfo
spec:
  ingress:
    ingressClassName: nginx
    rules:
    - host: liqo.cloud.example.com
      http:
        paths:
        - backend:
            service:
              name: podinfo
              port:
                number: 9898
          path: /
          pathType: ImplementationSpecific
  strategy:
    dnsTtlSeconds: 30
    splitBrainThresholdSeconds: 300
    type: roundRobin
status:
  geoTag: eu
  healthyRecords:
    liqo.cloud.example.com:
    - 172.19.0.3
    - 172.19.0.4
    - 172.19.0.5
```

(continues on next page)

(continued from previous page)

```
    - 172.19.0.6
  serviceHealth:
    liqo.cloud.example.com: Healthy
```

Similarly, when issuing the command from the *gslb-us* cluster:

```
kubectl get gslbs.k8gb.absa.oss -n podinfo podinfo -o yaml --kubeconfig $KUBECONFIG_US
```

```
apiVersion: k8gb.absa.oss/v1beta1
kind: Gslb
metadata:
  annotations:
    k8gb.io/strategy: roundRobin
  name: podinfo
  namespace: podinfo
spec:
  ingress:
    ingressClassName: nginx
    rules:
    - host: liqo.cloud.example.com
      http:
        paths:
        - backend:
            service:
              name: podinfo
              port:
                number: 9898
          path: /
          pathType: ImplementationSpecific
  strategy:
    dnsTtlSeconds: 30
    splitBrainThresholdSeconds: 300
    type: roundRobin
status:
  geoTag: us
  healthyRecords:
    liqo.cloud.example.com:
    - 172.19.0.5
    - 172.19.0.6
    - 172.19.0.3
    - 172.19.0.4
  serviceHealth:
    liqo.cloud.example.com: Healthy
```

In both clusters, the *Gslb* resources are pretty identical; they only differ for the *geoTag* field. The resource status also reports:

- the *serviceHealth* status, that should be *Healthy* for both clusters
- the list of IPs exposing the HTTP service: they are the IPs of the nodes of both clusters since the *Nginx Ingress Controller* is deployed in *HostNetwork DaemonSet* mode.

16.5 Check service reachability

Since *podinfo* is an HTTP service, you can contact it using the *curl* command. Use the `-v` option to understand which of the nodes is being targeted.

You need to use the DNS server in order to resolve the hostname to the IP address of the service. To this end, create a pod in one of the clusters (it does not matter which one) overriding its DNS configuration.

```
HOSTNAME="liqo.cloud.example.com"
K8GB_COREDNS_IP=$(kubectl get svc k8gb-coredns -n k8gb -o custom-columns='IP:spec.
↪clusterIP' --no-headers)

kubectl run -it --rm curl --restart=Never --image=curlimages/curl:7.82.0 --command \
    --overrides "{\"spec\":{\"dnsConfig\":{\"nameservers\":[\"${K8GB_COREDNS_IP}\"]},\
↪"dnsPolicy\":\"None\"}}" \
    -- curl $HOSTNAME -v
```

Note

Launching this pod several times, you will see different IPs and different frontend pods answering in a round-robin fashion (as set in the *Gslb* policy).

```
*   Trying 172.19.0.3:80...
* Connected to liqo.cloud.example.com (172.19.0.3) port 80 (#0)
...
{
  "hostname": "podinfo-67f46d9b5f-xrbmg",
  "version": "6.1.4",
  "revision": "",
...
```

```
*   Trying 172.19.0.6:80...
* Connected to liqo.cloud.example.com (172.19.0.6) port 80 (#0)
...
{
  "hostname": "podinfo-67f46d9b5f-xrbmg",
  "version": "6.1.4",
  "revision": "",
...
```

```
*   Trying 172.19.0.3:80...
* Connected to liqo.cloud.example.com (172.19.0.3) port 80 (#0)
...
{
  "hostname": "podinfo-67f46d9b5f-cmnp5",
  "version": "6.1.4",
  "revision": "",
...
```

This brief tutorial showed how you could leverage Liqo and *K8GB* to deploy and expose a multi-cluster application. In addition to the *RoundRobin* policy, which provides load distribution among clusters, *K8GB* allows favoring closer

endpoints (through the *GeoIP* strategy), or adopt a *Failover* policy. Additional details are provided in its official documentation.

16.6 Tear down the playground

Our example is finished; now we can remove all the created resources and tear down the playground.

16.6.1 Unoffload namespaces

Before starting the uninstallation process, make sure that all namespaces are unoffloaded:

```
liqoctl unoffload namespace podinfo
```

Every pod that was offloaded to a remote cluster is going to be rescheduled onto the local cluster.

16.6.2 Revoke peerings

Similarly, make sure that all the peerings are revoked:

```
liqoctl unpeer out-of-band gslb-us
```

At the end of the process, the virtual node is removed from the local cluster.

16.6.3 Uninstall Liqo

Now you can remove Liqo from your clusters with *liqoctl*:

```
liqoctl uninstall
liqoctl uninstall --kubeconfig="$KUBECONFIG_US"
```

Purge

By default the Liqo CRDs will remain in the cluster, but they can be removed with the `--purge` flag:

```
liqoctl uninstall --purge
liqoctl uninstall --kubeconfig="$KUBECONFIG_US" --purge
```

16.6.4 Destroy clusters

To teardown the k3d clusters, you can issue:

```
k3d cluster delete gslb-eu gslb-us edgedns
```

CHAPTER

SEVENTEEN

REPLICATED DEPLOYMENTS

In this tutorial you will learn how to deploy an application, and use Liqo to replicate it on multiple clusters.

In this example you will configure a scenario composed of a *single entry point cluster* used for the deployment of the applications (called *origin cluster*) and two *destination clusters*. The deployed application will be replicated on all *destination clusters* in order to deploy exactly **one** identical application on each destination cluster.

17.1 Provision the playground

First, make sure that the *requirements* for Liqo are satisfied.

Then, let's open a terminal on your machine and launch the following script, which creates the three above-mentioned clusters with KinD and installs Liqo on all of them.

```
git clone https://github.com/liqotech/liqo.git
cd liqo
git checkout v0.9.1
cd examples/replicated-deployments
./setup.sh
```

Export the kubeconfigs environment variables to use them in the rest of the tutorial:

```
export KUBECONFIG=liqo_kubeconf_europe-cloud
export KUBECONFIG_EUROPE_ROME_EDGE=liqo_kubeconf_europe-rome-edge
export KUBECONFIG_EUROPE_MILAN_EDGE=liqo_kubeconf_europe-milan-edge
```

Note

We suggest exporting the kubeconfig of the *origin* cluster as default (i.e., `KUBECONFIG`), since you will mainly interact with it.

Now you should have three clusters with Liqo running. The setup script named them **europe-cloud**, **europe-rome-edge** and **europe-milan-edge**, and respectively configured the following cluster labels:

- *origin*: `topology.liqo.io/type=origin`
- *europe-rome-edge*: `topology.liqo.io/type=destination`
- *europe-milan-edge*: `topology.liqo.io/type=destination`

You can check that the clusters are correctly labeled through:

```
liqoctl status
liqoctl status --kubeconfig "$KUBECONFIG_EUROPE_ROME_EDGE"
liqoctl status --kubeconfig "$KUBECONFIG_EUROPE_MILAN_EDGE"
```

17.2 Peer the clusters

Now, you can establish new Liqo *peerings* from *origin* to *destination* clusters, e.g., using the *out-of-band peering approach*:

To implement the desired scenario, let's first retrieve the *peer command* from the *destination* clusters:

```
PEER_EUROPE_ROME_EDGE=$(liqoctl generate peer-command --only-command --kubeconfig
↪$KUBECONFIG_EUROPE_ROME_EDGE)
PEER_EUROPE_MILAN_EDGE=$(liqoctl generate peer-command --only-command --kubeconfig
↪$KUBECONFIG_EUROPE_MILAN_EDGE)
```

Then, establish the peerings from the *origin* cluster:

```
echo "$PEER_EUROPE_ROME_EDGE" | bash
echo "$PEER_EUROPE_MILAN_EDGE" | bash
```

When the above commands return successfully, you can check the peering status by running:

```
kubectl get foreignclusters
```

The output should look like the following, indicating that an outgoing peering is currently active towards both the *europe-rome-edge* and the *europe-milan-edge* clusters, and that the cross-cluster network tunnels have been established:

```
NAME                 TYPE        OUTGOING PEERING   INCOMING PEERING   NETWORKING   ␣
↪AUTHENTICATION   AGE
europe-rome-edge     OutOfBand   Established        None               Established  ␣
↪Established      41s
europe-milan-edge    OutOfBand   Established        None               Established  ␣
↪Established      7s
```

Additionally, you should have two new virtual nodes in the *origin* cluster, characterized by the labels set at install-time:

```
kubectl get node --selector=liqo.io/type=virtual-node --show-labels
```

```
NAME                     STATUS   ROLES   AGE   VERSION   LABELS
liqo-europe-milan-edge   Ready    agent   27s   v1.25.0   liqo.io/remote-cluster-
↪id=9636366f-2718-464e-b1df-3eca5a71aaf6,liqo.io/type=virtual-node,topology.liqo.io/
↪type=destination
liqo-europe-rome-edge    Ready    agent   34s   v1.25.0   liqo.io/remote-cluster-
↪id=7a0f5f75-e98e-4927-b65f-d0274ca03d9c,liqo.io/type=virtual-node,topology.liqo.io/
↪type=destination
```

Note

Some of the default labels were omitted for the sake of clarity.

17.3 Tune namespace offloading

Now, let's pretend you want to deploy an application that needs to be scheduled on all *destination* clusters, but not in the *origin* one. First, we create a new namespace, then enable Liqo offloading to it:

```
kubectl create namespace liqo-demo
```

Then, enable Liqo offloading for that namespace:

```
liqoctl offload namespace liqo-demo \
  --namespace-mapping-strategy EnforceSameName \
  --pod-offloading-strategy Remote \
  --selector 'topology.liqo.io/type=destination'
```

The above command configures Liqo with the following behaviour (see the dedicated *usage page* for additional information concerning namespace offloading configurations):

- the `liqo-demo` namespace, and the contained resources, are offloaded only to the clusters with the `topology.liqo.io/type=destination` label.
- the pods living in the `liqo-demo` namespace only on virtual nodes.

Selectors

This example uses **selectors**, but they are not strictly necessary here, as all *peered* clusters have been targeted as *destination*. **Selectors** become necessary in case you want to target a subset of *peered* clusters. More information are available in the *offloading with policies* example.

You can now query for the namespaces either in the *europe-rome-edge* or *europe-milan-edge* cluster to see if the remote namespace has been correctly created by Liqo:

```
kubectl get namespaces liqo-demo --kubeconfig="$KUBECONFIG_EUROPE_ROME_EDGE"
kubectl get namespaces liqo-demo --kubeconfig="$KUBECONFIG_EUROPE_MILAN_EDGE"
```

If everything is correct, both commands should return an output similar to the following:

```
NAME        STATUS   AGE
liqo-demo   Active   70s
```

17.4 Deploy applications

Now it is time to deploy the application.

In order to create a replica of the application in each *destination* cluster, you need to enforce the following conditions:

- The *deployment* resource must produce at least one *pod* for each *destination* cluster.
- Each *destination* cluster must schedule at most one *pod* on its nodes.

To obtain this result you can leverage the following features available in *kubernetes*:

- Set a number of replicas in the *deployment* which is equal to the number of *destination* clusters
- Set **topologySpreadConstraints** inside the *deployment*'s template, which sets the **maxSkew** equal to **1**.

The file `./manifests/deploy.yaml` contains an example of a *deployment* which satisfies these conditions. Let's deploy it:

```
kubectl apply -f ./manifests/deploy.yaml -n liqo-demo
```

More replicas

If the deployment uses a number of replicas which is higher than the number of *virtual nodes*, the pods will be scheduled respecting the **maxSkew** value, which guarantees that the difference between the maximum number of pods (scheduled on a single node) and the minimum will be **1**.

We can check the pod status and verify that each *destination* cluster has scheduled one *pod* on its nodes, i.e., one pod has been scheduled onto the *europe-rome-edge* cluster, and the other on *europe-milan-edge*, and they are both correctly running:

```
kubectl get pod -n liqo-demo -o wide
```

```
NAME                              READY   STATUS    RESTARTS   AGE     IP            NODE ␣
↪              NOMINATED NODE   READINESS GATES
liqo-demo-app-777fb9fc8-bbt4d   1/1     Running   0          7m28s   10.113.0.65   liqo-
↪europe-rome-edge   <none>           <none>
liqo-demo-app-777fb9fc8-wrjph   1/1     Running   0          7m28s   10.109.0.62   liqo-
↪europe-milan-edge   <none>           <none>
```

17.5 Tear down the playground

Our example is finished; now we can remove all the created resources and tear down the playground.

17.5.1 Unoffload namespaces

Before starting the uninstallation process, make sure that all namespaces are unoffloaded:

```
liqoctl unoffload namespace liqo-demo
```

17.5.2 Revoke peerings

Similarly, make sure that all the peerings are revoked:

```
liqoctl unpeer out-of-band europe-rome-edge
liqoctl unpeer out-of-band europe-milan-edge
```

At the end of the process, the virtual nodes are removed from the local cluster.

17.5.3 Uninstall Liqo

Now you can uninstall Liqo from your clusters:

```
liqoctl uninstall
liqoctl uninstall --kubeconfig="$KUBECONFIG_EUROPE_ROME_EDGE"
liqoctl uninstall --kubeconfig="$KUBECONFIG_EUROPE_MILAN_EDGE"
```

Purge

By default the Liqo CRDs will remain in the cluster, but they can be removed with the `--purge` flag.

17.5.4 Destroy clusters

To teardown the KinD clusters, you can issue:

```
kind delete cluster --name origin
kind delete cluster --name europe-rome-edge
kind delete cluster --name europe-milan-edge
```

CHAPTER

EIGHTEEN

PROVISION WITH TERRAFORM

Terraform is a widely used Infrastructure as Code (IaC) tool that allows engineers to define their software infrastructure in code.

This tutorial aims at presenting how to set up an environment with Liqo installed via Terraform.

You will learn how to create a *virtual cluster* by peering two Kubernetes clusters and offload a namespace using the *Generate*, *Peer* and *Offload* resources provided by the Liqo provider.

18.1 Provision the infrastructure

First, check that you are compliant with the *requirements*. Additionally, this example requires Terraform to be installed in your system.

Then, let's open a terminal on your machine and launch the following script, which creates the infrastructure used in this example (i.e., two KinD clusters, peered with Liqo), with all playground already set up.

This tutorial will present a detailed description about how this result is achieved, by analyzing the most notable parts of the Terraform infrastructure definition file that refer to Liqo.

```
git clone https://github.com/liqotech/liqo.git
cd liqo
git checkout v0.9.1
cd examples/provision-with-terraform
terraform init
terraform apply
```

18.2 Analyze the infrastructure and code

Inspecting `Terraform main` file within the examples/provision-with-terraform folder you can see the Terraform configuration file analyzed below. With the previous command you created two KinD clusters, installed Liqo and established an outgoing peering from local to remote cluster. Furthermore, you offloaded a namespace to virtual node (i.e., remote cluster). In this way the namespace will leverage on both local and remote clusters resources following offloading configuration.

This example is provisioned on KinD, since it requires no particular configurations (e.g., concerning accounts), and does not lead to resource costs. Yet, all the presented functionalities work also on other clusters, e.g., the ones operated by public cloud providers.

18.2.1 Provision the clusters

The first step executed by Terraform is the creation of the two KinD clusters: the resource in charge of building them is the `kind_cluster` resource of the provider tehcyx that, starting from configuration parameters (such as *cluster_name*, *service_subnet/pod_subnet*), generates the clusters and related config files needed by other providers to set up the infrastructure.

You can inspect the deployed clusters by typing on your workstation:

```
kind get clusters
```

You should see a couple of entries:

```
milan
rome
```

This means that two KinD clusters are deployed and running on your host.

Then, you can simply inspect the status of the clusters. To do so, you can export the `KUBECONFIG` variable to specify the identity file for *kubectl* and *liqoctl*, and then contact the cluster.

By default, the kubeconfigs of the two clusters are stored in the current directory (`./rome-config`, `./milan-config`). You can export the appropriate environment variables leveraged for the rest of the tutorial (i.e., `KUBECONFIG` and `KUBECONFIG_MILAN`), and referring to their location, through the following:

```
export KUBECONFIG="$PWD/rome-config"
export KUBECONFIG_MILAN="$PWD/milan-config"
```

Note

We suggest exporting the kubeconfig of the first cluster as default (i.e., `KUBECONFIG`), since it will be the entry point of the virtual cluster and you will mainly interact with it.

18.2.2 Install Liqo

After creating the two KinD clusters, Terraform will install Liqo using the `helm_release` resource of the Helm provider configured with the cluster config files. Once the installation is complete, you should see the Liqo system pods up and running on both clusters:

```
kubectl get pods -n liqo
```

```
NAME                                      READY   STATUS    RESTARTS   AGE
liqo-auth-74c795d84c-x2p6h                1/1     Running   0          2m8s
liqo-controller-manager-6c688c777f-4lv9d  1/1     Running   0          2m8s
liqo-crd-replicator-6c64df5457-bq4tv      1/1     Running   0          2m8s
liqo-gateway-78cf7bb86b-pkdpt             1/1     Running   0          2m8s
liqo-metric-agent-5667b979c7-snmdg        1/1     Running   0          2m8s
liqo-network-manager-5b5cdcfcf7-scvd9     1/1     Running   0          2m8s
liqo-proxy-6674dd7bbd-kr2ls               1/1     Running   0          2m8s
liqo-route-7wsrx                          1/1     Running   0          2m8s
liqo-route-sz75m                          1/1     Running   0          2m8s
```

18.2.3 Extract the peering parameters

Once the Liqo installation in the remote cluster is complete, Terraform will extract the authentication parameters required to peer the local (i.e., *Rome*) cluster with the remote one (i.e., *Milan*).

This is achieved with the `liqo_generate` resource of the liqo provider instance, configured with either the config file or the full list of parameters of the remote cluster:

```
provider "liqo" {
  alias = "milan"
  kubernetes = {
    config_path = kind_cluster.milan.kubeconfig_path
  }
}

resource "liqo_generate" "generate" {

  depends_on = [
    helm_release.install_liqo_milan
  ]

  provider = liqo.milan

}
```

18.2.4 Run the peering procedure

Once the `generate_resource` is created, Terraform will continue with the *out-of-band peering* procedure leveraging the output parameters of the previous resource.

This is achieved with the `liqo_peer` resource of the liqo provider instance, configured with either the config file or the full list of parameters of the local cluster:

```
provider "liqo" {
  alias = "rome"
  kubernetes = {
    config_path = kind_cluster.rome.kubeconfig_path
  }
}

resource "liqo_peer" "peer" {

  depends_on = [
    helm_release.install_liqo_rome
  ]

  provider = liqo.rome

  cluster_id      = liqo_generate.generate.cluster_id
  cluster_name    = liqo_generate.generate.cluster_name
  cluster_authurl = liqo_generate.generate.auth_ep
  cluster_token   = liqo_generate.generate.local_token

}
```

You can check the peering status by running:

```
kubectl get foreignclusters
```

The output should look like the following, indicating that the cross-cluster network tunnel has been established, and an outgoing peering is currently active (i.e., the *Rome* cluster can offload workloads to the *Milan* one, but not vice versa):

```
NAME    TYPE        OUTGOING PEERING   INCOMING PEERING   NETWORKING    AUTHENTICATION  ␣
↪AGE
milan   OutOfBand   Established        None               Established   Established     ␣
↪12s
```

At the same time, you should see a virtual node (`liqo-milan`) in addition to your physical nodes:

```
kubectl get nodes
```

```
NAME                STATUS   ROLES                  AGE     VERSION
liqo-milan          Ready    agent                  14s     v1.25.0
rome-control-plane  Ready    control-plane,master   7m56s   v1.25.0
rome-worker         Ready    <none>                 7m25s   v1.25.0
```

18.2.5 Offload a namespace

If you want to deploy an application that is scheduled on a Liqo virtual node (hence, it is offloaded on a remote cluster), you should first create a namespace where your pod will be started.

This can be achieved with the `kubernetes_namespace` resource of the kubernetes provider, configured with either the config file or the full list of parameters of the local cluster. Then tell Liqo to make this namespace eligible for the pod offloading.

The resource in charge of doing this is `liqo_offload` of the same liqo provider instance of `liqo_peer` resource:

```
resource "liqo_offload" "offload" {

  depends_on = [
    helm_release.install_liqo_rome,
    kubernetes_namespace.namespace
  ]

  provider = liqo.rome

  namespace = "liqo-demo"

}
```

Note

Liqo virtual nodes have a taint that prevents the pods from being scheduled on them. The Liqo webhook will add the toleration for this taint to the pods created in the liqo-enabled namespaces.

Since no further configuration is provided, Liqo will add a suffix to the namespace name to make it unique on the remote cluster (see the dedicated *usage page* for additional information concerning namespace offloading configurations).

You can now test the infrastructure you have just created by deploying an application. For this, you can follow the *proper example section* in the Quick Start page.

18.3 Tear down the infrastructure

To tear down all the infrastrucure you only need to run the following command:

```
terraform destroy
```

This command will destroy all the resources starting from the last one created, from bottom to top.

If you want to destroy a specific resource (for example to unpeer a cluster or to unoffload a namespace) you can leverage on the `-target` flag of `destroy` command. For example, you can run the following command to unpeer two clusters:

```
terraform destroy -target="liqo_peer.peer"
```

Warning: The Terraform `destroy` command will destroy all resources that have a dependence on the one that has to be destroyed.

CHAPTER

NINETEEN

PEER TWO CLUSTERS

This section describes the procedure to **establish a peering** with a remote cluster, using one of the two alternative approaches featured by Liqo. You can refer to the *dedicated features section* for a high-level presentation of their characteristics, and the associated trade-offs.

Warning: The establishment of a peering with a remote cluster leveraging a **different version of Liqo**, net of patch releases, is currently **not supported**, and could lead to unexpected results.

19.1 Overview

The peering process leverages ***liqoctl*** to interact with the clusters, abstracting the creation and update of the appropriate custom resources. To this end, the most important one is the ***ForeignCluster*** resource, which **represents a remote cluster**, including its identity, the associated authentication endpoint, and the desired peering state (i.e., whether it should be established, and in which directions). Additionally, its status reports a **summary of the current peering status**, detailing whether the different phases (e.g., authentication, network establishment, resource negotiation, ...) correctly succeeded.

The following sections present the respective procedures to **peer a local cluster A** (i.e., the *consumer*), with a **remote cluster B** (i.e., the *provider*). At the end of the process, a new **virtual node** is created in the consumer, abstracting the resources shared by the provider, and enabling seamless **pod offloading** to the remote cluster. Additional details are also provided to enable the reverse peering direction, hence achieving a **bidirectional peering**, allowing both clusters to offload a part of their workloads to the other.

By default, Liqo shares a configurable percentage of the currently available resources of the **provider** cluster with **consumers**. You can change this behavior by using a custom resource plugin.

All examples leverage two different *contexts* to refer to *consumer* and *provider* clusters, respectively named `consumer` and `provider`.

Note

liqoctl displays a *kubectl* compatible behavior concerning Kubernetes API access, hence supporting the `KUBECONFIG` environment variable, as well as all the standard flags, including `--kubeconfig` and `--context`. Ensure you selected the correct target cluster before issuing *liqoctl* commands (as you would do with *kubectl*).

19.2 Out-of-band control plane

Briefly, the procedure to establish an *out-of-band control plane peering* consists of a first step performed on the *provider*, to **retrieve the set of information** required (i.e., authentication endpoint and token, cluster ID, ...), followed by the creation, on the *consumer*, of the necessary resources to **start the actual peering**. The remainder of the process, including identity retrieval, resource negotiation and network tunnel establishment is **performed automatically** by Liqo, through a mutual exchange of information and negotiation between the two clusters involved.

19.2.1 Information retrieval

To proceed, ensure that you are operating in the *provider* cluster, and then issue the *liqoctl generate peer-command* command:

```
liqoctl --context=provider generate peer-command
```

This retrieves the information concerning the *provider* cluster (i.e., authentication endpoint and token, cluster ID, ...) and generates a command that can be executed on a *different* cluster (i.e., the *consumer*) to establish an out-of-band outgoing peering towards the *provider* cluster.

An example of the resulting command is the following:

```
liqoctl peer out-of-band <cluster-name> --auth-url <auth-url> \
    --cluster-id <cluster-id> --auth-token <auth-token>
```

19.2.2 Peering establishment

Once obtained the peering command, it is possible to execute it in the *consumer* cluster, to kick off the peering process.

Warning: Pay attention to operate in the correct cluster, possibly adding the appropriate flags to the generated command (e.g., `--context=consumer`).

```
liqoctl --context=consumer peer out-of-band <cluster-name> --auth-url <auth-url> \
    --cluster-id <cluster-id> --auth-token <auth-token>
```

The above command configures the appropriate authentication token, and then creates a new *ForeignCluster* resource in the *consumer cluster*. Finally, it waits for the different peering phases to complete (this might require a few seconds, depending on the download time of the Liqo virtual kubelet image).

The *ForeignCluster* resource can be inspected through *kubectl*:

```
kubectl --context=consumer get foreignclusters
```

If the peering process completed successfully, you should observe an output similar to the following, indicating that the cross-cluster network tunnel has been established, and an outgoing peering is currently active (i.e., the *consumer* cluster can offload workloads to the *provider* one, but not vice versa):

```
NAME       TYPE        OUTGOING PEERING   INCOMING PEERING   NETWORKING    AUTHENTICATION
provider   OutOfBand   Established        None               Established   Established
```

At the same time, a new *virtual node* should have been created in the *consumer* cluster. Specifically:

```
kubectl --context=consumer get nodes -l liqo.io/type=virtual-node
```

Should return an output similar to the following:

```
NAME            STATUS   ROLES   AGE    VERSION
liqo-provider   Ready    agent   179m   v1.23.4
```

In addition, you can check the peering status, and retrieve more advanced information, using:

```
liqoctl status peer provider
```

Note

The name of the *ForeignCluster* resource, as well as that of the *virtual node*, reflects the cluster name specified with the *liqoctl peer out-of-band* command.

19.2.3 Bidirectional peering

Once the peering from the *consumer* to the *provider* has been established, the reverse direction (i.e., leading to a bidirectional peering) can be enabled through a simpler command, since the *ForeignCluster* resource is already present:

```
liqoctl --context=provider peer consumer
```

19.2.4 Tear down

An out-of-band peering can be disabled leveraging the symmetric *liqoctl unpeer* command, causing the local virtual node (abstracting the remote cluster) to be destroyed, and all offloaded workloads to be rescheduled:

```
liqoctl --context=consumer unpeer out-of-band
```

Note

The reverse peering direction, if any, is preserved, and the remote cluster can continue offloading workloads to its virtual node representing the local cluster. In this case, the command *emits a warning*, and it does not proceed deleting the *ForeignCluster* resource. Hence, the same command shall be executed on both clusters to completely tear down a bidirectional peering.

In case only one peering direction shall be teared down, while preserving the opposite, it is suggested to leverage the appropriate *liqoctl unpeer* command to disable the outgoing peering (e.g., on the *provider* cluster):

```
liqoctl --context=provider unpeer consumer
```

19.3 In-band control plane

Briefly, the procedure to establish an *in-band control plane peering* consists of a first step performed by *liqoctl*, which interacts alternatively with both clusters to **establish the cross-cluster VPN tunnel**, exchange the **authentication tokens** and configure the Liqo control plane traffic to flow inside the VPN. The remainder of the process, including identity retrieval and resource negotiation, is **performed automatically** by Liqo, through a mutual exchange of information and negotiation between the two clusters involved.

Note

The host used to issue the *liqoctl peer in-band* command must have **concurrent access to both clusters** (i.e., *consumer* and *provider*) while carrying out the in-band control plane peering process. To this end, these subcommands feature a parallel set of flags concerning Kubernetes API access to the remote cluster, in the form `--remote-<flag>` (e.g., `--remote-kubeconfig`, `--remote-context`).

19.3.1 Peering establishment

The in-band control plane peering process can be started leveraging a single *liqoctl* command:

```
liqoctl peer in-band --context=consumer --remote-context=provider
```

The above command outputs a set of information concerning the different operations performed on the two clusters. Notably, it exchanges the appropriate authentication tokens, establishes the cross-cluster VPN tunnel, and then creates a new *ForeignCluster* resource in *both clusters*. Finally, it waits for the different peering phases to complete (this might require a few seconds, depending on the download time of the Liqo virtual kubelet image).

The *ForeignCluster* resource can be inspected through *kubectl* (e.g., on the *consumer*):

```
kubectl --context=consumer get foreignclusters
```

If the peering process completed successfully, you should observe an output similar to the following, indicating that the cross-cluster network tunnel has been established, and an outgoing peering is currently active (i.e., the *consumer* cluster can offload workloads to the *provider* one, but not vice versa):

```
NAME       TYPE     OUTGOING PEERING   INCOMING PEERING   NETWORKING    AUTHENTICATION
provider   InBand   Established        None               Established   Established
```

At the same time, a new *virtual node* should have been created in the *consumer* cluster. Specifically:

```
kubectl --context=consumer get nodes -l liqo.io/type=virtual-node
```

Should return an output similar to the following:

```
NAME             STATUS   ROLES   AGE    VERSION
liqo-provider    Ready    agent   179m   v1.23.4
```

In addition, you can check the peering status, and retrieve more advanced information, using:

```
liqoctl status peer provider
```

Note

The name of the *ForeignCluster* resource, as well as that of the *virtual node*, reflects the cluster name specified by the remote cluster administrators at install time.

19.3.2 Bidirectional peering

A bidirectional in-band peering can be established adding the `--bidirectional` flag to the *liqoctl peer* command invocation:

```
liqoctl peer in-band --context=consumer --remote-context=provider --bidirectional
```

Note

The *liqoctl peer in-band* command is idempotent, and can be re-executed without side effects to enable a bidirectional peering.

Alternatively, the reverse peering can be also activated executing the following on the *provider* cluster:

```
liqoctl --context=provider peer consumer
```

19.3.3 Tear down

An in-band peering can be disabled leveraging the symmetric *liqoctl unpeer* command, causing both virtual nodes (if present) to be destroyed, all offloaded workloads to be rescheduled, and finally tearing down the cross-cluster VPN tunnel:

```
liqoctl unpeer in-band --context=consumer --remote-context=provider
```

In case only one peering direction shall be teared down, while preserving the opposite, it is possible to leverage the appropriate *liqoctl unpeer* command to disable the outgoing peering (e.g., on the *provider* cluster):

```
liqoctl --context=provider unpeer consumer
```

CHAPTER

TWENTY

NAMESPACE OFFLOADING

This section presents the operational procedure to **offload a namespace** to (possibly a subset of) the **remote clusters** peered with the local cluster. Hence, enabling **pod offloading**, as well as triggering the ***resource reflection*** process: additional details about namespace extension in Liqo are provided in the dedicated *namespace extension features section*.

20.1 Overview

The offloading of a namespace can be easily controlled through the dedicated ***liqoctl*** commands, which abstract the creation and update of the appropriate custom resources. In this context, the most important one is the ***NamespaceOffloading*** resource, which enables the offloading of the corresponding namespace, configuring at the same time the subset of target remote clusters, additional constraints concerning pod offloading and the naming strategy. Moreover, different namespaces can be characterized by different configurations, hence achieving a high degree of flexibility. Finally, the *NamespaceOffloading* status reports for each remote cluster a **summary about its status** (i.e., whether the remote cluster has been selected for offloading, and the twin namespace has been correctly created).

20.2 Offloading a namespace

A given namespace *foo* can be offloaded, leveraging the default configuration, through:

```
liqoctl offload namespace foo
```

Alternatively, the underlying *NamespaceOffloading* resource can be generated and output (either in *yaml* or *json* format) leveraging the dedicated `--output` flag:

```
liqoctl offload namespace foo --output yaml
```

Then, the resulting manifest can be applied with *kubectl*, or through automation tools (e.g., by means of GitOps approaches).

Note

Possible race conditions might occur in case a *NamespaceOffloading* resource is created at the same time (e.g., as a batch) as pods (or higher level abstractions such as *Deployments*), preventing them from being considered for offloading until the *NamespaceOffloading* resource is not processed.

This situation can be prevented manually labeling in advance the hosting namespace with the *liqo.io/scheduling-enabled=true* label, hence enabling the Liqo mutating webhook and causing pod creations to be rejected until pod

offloading is possible. Still, this causes no problems, as the Kubernetes abstractions (e.g., *Deployments*) ensure that the desired pods get eventually created correctly.

Regardless of the approach adopted, namespace offloading can be further configured in terms of the three main parameters presented below, each one exposed through a dedicated CLI flag.

20.2.1 Namespace mapping strategy

The *namespace mapping strategy* defines the naming strategy used to create the remote namespaces, and can be configured through the `--namespace-mapping-strategy` flag. The accepted values are:

- **DefaultName** (default): to **prevent conflicts** on the target cluster, remote namespace names are generated as the concatenation of the local namespace name, the cluster name of the local cluster and a unique identifier (e.g., *foo* could be mapped to *foo-lively-voice-dd8531*).
- **EnforceSameName**: remote namespaces are named after the local cluster's namespace. This approach ensures **naming transparency**, which is required by certain applications, as well as guarantees that **cross-namespace DNS queries** referring to reflected services work out of the box (i.e., without adapting the target namespace name). Yet, it can lead to **conflicts** in case a namespace with the same name already exists inside the selected remote clusters, ultimately causing the remote namespace creation request to be rejected.

Note

Once configured for a given namespace, the *namespace mapping strategy* is **immutable**, and any modification is prevented by a dedicated Liqo webhook. In case a different strategy is desired, it is necessary to first *unoffload* the namespace, and then re-offload it with the new parameters.

20.2.2 Pod offloading strategy

The *pod offloading strategy* defines high-level constraints about pod scheduling, and can be configured through the `--pod-offloading-strategy` flag. The accepted values are:

- **LocalAndRemote** (default): pods deployed in the local namespace can be scheduled **both onto local nodes and onto virtual nodes**, hence possibly offloaded to remote clusters.
- **Local**: pods deployed in the local namespace are enforced to be scheduled onto **local nodes only**, hence never offloaded to remote clusters. The extension of a namespace, forcing at the same time all pods to be scheduled locally, enables the consumption of local services from the remote cluster, as shown in the service offloading *example*.
- **Remote**: pods deployed in the local namespace are enforced to be scheduled onto **remote nodes only**, hence always offloaded to remote clusters.

Note

The *pod offloading strategy* applies to pods only, while the other objects that live in namespaces selected for offloading, and managed by the resource refletion process, are always replicated to (possibly a subset of) the remote clusters, as specified through the *cluster selector* (more details below).

Warning: Due to current limitations of Liqo, the pods violating the *pod offloading strategy* are not automatically evicted following an update of this policy to a more restrictive value (e.g., *LocalAndRemote* to *Remote*) after the initial creation.

20.2.3 Cluster selector

The *cluster selector* provides the possibility to **restrict the set of remote clusters** (in case more than one peering is active) selected as targets for offloading the given namespace. The *twin* namespace is not created in clusters that do not match the cluster selector, as well as the resource reflection mechanism is not activated for those namespaces. Yet, different *cluster selectors* can be specified for different namespaces, depending on the desired configuration.

The cluster selector follows the standard **label selector** syntax, and refers to the Kubernetes labels characterizing the **virtual nodes**. Specifically, these include both the set of labels suggested by the remote cluster during the peering process and automatically propagated by Liqo, as well as possible additional ones added by the local cluster administrators.

The cluster selector can be expressed through the `--selector` flag, which can be optionally repeated multiple times to specify alternative requirements (i.e., in logical OR). For instance:

- `--selector 'region in (europe,us-west), !staging'` would match all clusters located in the *europe* or *us-west* region, *AND* not including the *staging* label.
- `--selector 'region in (europe,us-west)' --selector '!staging'` would match all clusters located in the *europe* or *us-west* region, *OR* not including the *staging* label.

In case no *cluster selector* is specified, all remote clusters are selected as targets for namespace offloading. In other words, an empty *cluster selector* matches all virtual clusters.

20.3 Unoffloading a namespace

The offloading of a namespace can be disabled through the dedicated *liqoctl* command, causing in turn the deletion of all resources reflected to remote clusters (including the namespaces themselves), and triggering the rescheduling of all offloaded pods locally:

```
liqoctl unoffload namespace foo
```

Warning: Disabling the offloading of a namespace is a **destructive operation**, since all resources created in remote namespaces (either automatically or manually) get removed, including possible **persistent storage volumes**. Before proceeding, double-check that the correct namespace has been selected, and ensure no important data is still present.

CHAPTER

TWENTYONE

RESOURCE REFLECTION

This section characterizes the ***resource reflection*** process (including also ***pod offloading***), detailing how the different resources are propagated to remote clusters and which fields are mutated.

Briefly, the set of supported resources includes (by category):

- ***Workload***: *Pods*
- ***Exposition***: *Services*, *EndpointSlices*, *Ingresses*
- ***Storage***: *PersistentVolumeClaims*, *PresistentVolumes*
- ***Configuration***: *ConfigMaps*, *Secrets*

Note

The reflection of a given object belonging to the *Exposition* or *Configuration* categories, and living in a namespace enabled for offloading, can be manually disabled adding the `liqo.io/skip-reflection` annotation to the object itself.

Additionally, the reflection of a given type of resources (e.g., *Secrets*) towards remote clusters can be completely disabled setting the corresponding `--<resource>-reflection-workers=0` virtual kubelet flag at install time:

```
liqoctl install ... --set "virtualKubelet.extra.args={--secret-reflection-workers=0}"
```

21.1 Pods offloading

Liqo leverages a custom resource, named *ShadowPod*, combined with an appropriate enforcement logic to ensure **remote pod resiliency** even in case of temporary connectivity loss between the local and remote clusters.

Pod specifications are propagated to the remote cluster **verbatim**, except for the following fields that are mutated:

- Removal of **scheduling constraints** (e.g., *Affinity*, *NodeSelector*, *SchedulerName*, *Preemption*, ...), as referring to the local cluster.
- Mutation of **service account** related information, to allow offloaded pods to transparently interact with the local (i.e., origin) API server, instead of the remote one.
- Enforcement of the properties concerning the usage of **host namespaces** (e.g., network, IPC, PID) to *false* (i.e., disabled), as potentially invasive and troublesome.

Note

Anti-affinity presets can be leveraged to specify predefined scheduling constraints for offloaded pods, spreading them across different nodes in the remote cluster. This feature is enabled through the `liqo.io/anti-affinity-preset` pod annotation, which can take three values:

- `propagate`: the anti-affinity constraints of the pod are propagated *verbatim* when offloaded to the remote cluster. Make sure that they match both the virtual node in the local cluster and at least one physical node in the remote cluster, otherwise the pod will fail to be scheduled (i.e., remain in pending status).
- `soft`: the pods sharing the same labels are *preferred* to be scheduled on different nodes (i.e., it is translated into a *preferredDuringSchedulingIgnoredDuringExecution* anti-affinity constraint).
- `hard`: the pods sharing the same labels are *required* to be scheduled on different nodes (i.e., it is translated into a *requiredDuringSchedulingIgnoredDuringExecution* anti-affinity constraint).

When set to *soft* or *hard*, the `liqo.io/anti-affinity-labels` annotation allows to select a subset of the pod label keys to build the anti-affinity constraints:

```
annotations:
  liqo.io/anti-affinity-preset: soft
  liqo.io/anti-affinity-labels: app.kubernetes.io/name,app.kubernetes.io/instance
```

Given that affinity constraints are *immutable*, the addition/removal of the annotations to/from an already existing pod *does not have any effect*. Make sure that the annotations are configured appropriately in the template of the managing object (e.g., *Deployment*, or *StatefulSet*).

Differently, **pod status** is propagated from the remote cluster to the local one, performing the following modifications:

- The *PodIP* is **remapped** according to the network fabric configuration, such as to be reachable from the other pods running in the same cluster.
- The *NodeIP* is replaced with the one of the corresponding virtual kubelet pod.
- The number of **container restarts** is augmented to account for the possible deletions of the remote pod (whose presence is enforced by the controlling *ShadowPod* resource).

Note

A pod living in a namespace not enabled for offloading, but manually forced to be scheduled in a virtual node, remains in *Pending* status, and it is signaled with the *OffloadingBackOff* reason. For instance, this can happen for system *DaemonSets* (e.g., CNI plugins), which tolerate all *taints* (hence, including the one associated with virtual nodes) and thus get scheduled on *all nodes*.

To prevent this behavior, it is necessary to explicitly modify the involved *DaemonSets*, adding a suitable *affinity* constraint excluding virtual nodes:

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
          - key: liqo.io/type
            operator: NotIn
            values:
            - virtual-node
```

21.2 Service exposition

The reflection of **Service** and **EndpointSlice** resources is a key element to allow the seamless **intercommunication** between microservices spread across multiple clusters, enabling the usage of standard DNS discovery mechanisms. In addition, the propagation of **Ingresses** enables the definition of multiple points of entrance for the external traffic, especially when combined with additional tools such as K8GB (see the *global ingress example* for additional details).

21.2.1 Services

Services are reflected **verbatim** into remote clusters, except for what concerns the *ClusterIP*, *LoadBalancerIP* and *NodePort* fields (when applicable), which are left empty (hence defaulted by the remote cluster), as likely conflicting. Still, the usage of **standard DNS discovery** mechanisms (i.e., based on service name/namespace) abstracts away the *ClusterIP* differences, with each pod retrieving the correct IP address.

Note

In case *node port* correspondence across clusters is required, its propagation can be enforced adding the `liqo.io/force-remote-node-port=true` annotation to the involved service.

21.2.2 EndpointSlices

In the local cluster, Services are transparently handled by the vanilla Kubernetes control plane, since it has **full visibility of all pods** (even those offloaded), hence leading to the creation of the corresponding **EndpointSlice** entries. Differently, the control plane of each remote cluster perceives **only the pods running in that cluster**, and the standard *EndpointSlice* creation logic alone is not sufficient (as it would not include the pods hosted by other clusters).

This gap is filled by the Liqo **EndpointSlice reflection** logic, which takes care of propagating all *EndpointSlice* entries (i.e. endpoints) not already present in the destination cluster. During the propagation process, endpoint addresses are appropriately **remapped** according to the **network fabric** configuration, ensuring that the resulting IPs are reachable from the destination cluster.

Thanks to this approach, **multiple replicas** of the same microservice spread across different clusters, and backed by the same service, are handled transparently. Each pod, no matter where it is located, contributes with a distinct *EndpointSlice* entry, either by the standard control plane or through resource reflection, hence becoming eligible during the **Service load-balancing process**.

Note

Even in a scenario where a single cluster is peered with multiple remote ones, the **EndpointSlice reflection** logic ensures that a **pod** scheduled **remotely** is reachable from every cluster through its **service**.

21.2.3 Ingresses

The propagation of **Ingress** resources enables the configuration of multiple points of entrance for **external traffic**. *Ingress* resources are propagated **verbatim** into remote clusters, except for the *IngressClassName* field, which is left empty. Hence, selecting the default *ingress class* in the remote cluster, as the local one (i.e., the one in the origin cluster) might not be present.

21.3 Persistent storage

The reflection of **PersistentVolumeClaims (PVCs)** and **PersistentVolumes (PVs)** is a key to enable the cross-cluster *Liqo storage fabric*. Specifically, the process is triggered when a PVC requiring the *Liqo storage class* is bound for the first time, and the requesting pod is scheduled in a virtual node (i.e., remote cluster). Upon this event, the **PVC is propagated verbatim** to the remote cluster, replacing the requested *StorageClass* with the one negotiated during the peering process.

Once created, the **resulting PV is reflected backwards** (i.e., from the remote to the local cluster), and the proper **affinity selectors** are added to **bind it to the virtual node**. Hence, subsequent pods mounting that *PV* will be scheduled on that virtual node, and eventually offloaded to the same remote cluster.

21.4 Configuration data

ConfigMaps and **Secrets** typically hold **configuration data** consumed by pods, and both types of resources are propagated by Liqo **verbatim** into remote clusters. In this respect, Liqo features also the propagation of **ServiceAccount tokens**, to enable offloaded pods to contact the Kubernetes API server of the origin cluster, as well as to support those applications leveraging *ServiceAccounts* for internal authentication purposes.

Warning: *ServiceAccount* tokens are stored within *Secret* objects when propagated to the remote cluster. This implies that any entity authorized to access *Secret* objects (or the mounting pods) might **retrieve the tokens and impersonate the offloaded workloads**. Hence, gaining the possibility to interact with the Kubernetes API server of the origin cluster, with the same permissions granted to the corresponding service account.

If this is a security concern in your scenario (e.g., the clusters are under the control of different administrative domains), it is possible to disable this feature setting the `--enable-apiserver-support=false` virtual kubelet flag at install time:

```
liqoctl install ... --set "virtualKubelet.extra.args={--enable-apiserver-
↪support=false}"
```

CHAPTER

TWENTYTWO

STATEFUL APPLICATIONS

As introduced in the *storage fabric features section*, Liqo supports **multi-cluster stateful applications** by extending the classical approaches adopted in standard Kubernetes clusters.

22.1 Liqo virtual storage class

The Liqo virtual storage class is a *Storage Class* that embeds the logic to create the appropriate *PersistentVolumes*, depending on the target cluster the mounting pod is scheduled onto. All operations performed on virtual objects (i.e., *PersistentVolumeClaims (PVCs)* and *PersistentVolumes (PVs)* associated with the *liqo* storage class) are then automatically propagated by Liqo to the corresponding real ones (i.e., associated with the storage class available in the target cluster).

Additionally, once a real *PV* gets created, the corresponding virtual one is enriched with a set of policies to attract mounting pods in the appropriate cluster, following the **data gravity** approach.

The figure below shows an application pod consuming a virtual *PVC*, which in turn led to the creation of the associated virtual *PV*. This process is **completely transparent** from the management point of view, with the only difference being the name of the storage class.

> **Warning:** The deletion of the virtual *PVC* will cause the deletion of the real *PVC/PV*, and the stored data will be **permanently lost**.

The Liqo control plane handles the binding of virtual PVC resources (i.e., associated with the *liqo* storage class) differently depending on the cluster where the mounting pod gets eventually scheduled onto, as detailed in the following.

22.1.1 Local cluster binding

In case a virtual *PVC* is bound to a pod initially **scheduled onto the local cluster** (i.e., a physical node), the Liqo control plane takes care of creating a twin *PVC* (in turn originating the corresponding twin *PV*) in the *liqo-storage* namespace, while mutating the *storage class* to that configured at Liqo installation time (with a fallback to the default one). A virtual *PV* is eventually created by Liqo to mirror the real one, effectively allowing pods to mount it and enforcing the *data gravity* constraints.

The resulting configuration is depicted in the figure below.

Current Limitations

Currently, the virtual storage class does not support the configuration of Kubernetes mount options and parameters.

22.1.2 Remote cluster binding

In case a virtual *PVC* is bound to a pod initially **scheduled onto a remote cluster** (i.e., a virtual node), the Liqo control plane takes care of creating a twin *PVC* (in turn originating the corresponding twin *PV*) in the *offloaded* namespace, while mutating the *storage class* to that negotiated at peering time (i.e., configured at Liqo installation time in the remote cluster, with a fallback to the default one). A virtual *PV* is eventually created by Liqo to mirror the real one, effectively allowing pods to mount it and enforcing the *data gravity* constraints.

The resulting configuration is depicted in the figure below.

Warning: The tearing down of the peering and/or the deletion of the offloaded namespace will cause the deletion of the real PVC, and the stored data will be **permanently lost**.

22.1.3 Move PVCs across clusters

Once a PVC is created in a given cluster, subsequent pods mounting that volume will be forced to be **scheduled onto the same cluster** to achieve storage locality, following the *data gravity* approach.

Still, if necessary, you can **manually move** the storage backing a virtual *PVC* (i.e., associated with the *liqo* storage class) from a cluster to another, leveraging the appropriate *liqoctl* command. Then, subsequent pods will get scheduled in the cluster the storage has been moved to.

Warning: This procedure requires the *PVC/PV* not to be bound to any pods during the entire process. In other words, live migration is currently not supported.

A given *PVC* can be moved to a target node (either physical, i.e., local, or virtual, i.e., remote) through the following command:

```
liqoctl move volume $PVC_NAME --namespace $NAMESPACE_NAME --target-node $TARGET_NODE_NAME
```

Where:

- `$PVC_NAME` is the name of the *PVC* to be moved.
- `$NAMESPACE_NAME` is the name of the namespace where the *PVC* lives in.
- `$TARGET_NODE_NAME` is the name of the node where the *PVC* will be moved to.

Under the hood, the migration process leverages the Liqo cross-cluster network fabric and the Restic project to back up the original data in a temporary repository, and then restore it in a brand-new *PVC* forced to be created in the target cluster.

Warning: *Liqo* and *liqoctl* **are not** backup tools. Make sure to properly back up important data before starting the migration process.

22.2 Externally managed storage

In addition to the virtual storage class, Liqo supports the offloading of pods that bind to ***cross-cluster*** **storage managed by external solutions** (e.g., managed by the cloud provider, or manually provisioned). Specifically, the *volumes* stanza of the pod specification is propagated verbatim to the offloaded pods, hence allowing to specify volumes available only remotely.

Note

In case a piece of externally managed storage is available only in one remote cluster, it is likely necessary to manually force pods to get scheduled exactly in that cluster. To prevent scheduling issues (e.g., the pod is marked as *Pending* since the local cluster has no visibility on the remote *PVC*), it is suggested to configure the target *NodeName* in the pod specifications to match that of the corresponding virtual nodes, hence bypassing the standard Kubernetes scheduling logic.

Warning: Due to current Liqo limitations, the remote namespace, including any *PVC* therein contained, will be **deleted** in case the local namespace is unoffloaded/deleted, or the peering is torn down.

CHAPTER

TWENTYTHREE

PROMETHEUS METRICS

This section presents the metrics exposed by Liqo, using the Prometheus format. Although in this page we suppose Prometheus is running in your cluster, please note that this is not strictly required: metrics can be scraped also by an external Prometheus server, with Liqo metrics exposed through a dedicated endpoint.

23.1 Scraping metrics

Gathering of Liqo metrics is **disabled** by default. To enable the scraping of Liqo metrics, you should set the `--enable-metrics` *liqoctl* flag during installation (cf. *installation customization options*). In this case, Liqo assumes that you leverage the Prometheus Operator to run Prometheus, hence it creates also the proper **ServiceMonitor** and **PodMonitor** resources that are automatically associated to the components that export metrics (e.g., network gateway, virtual kubelet). Finally, metrics are scraped depending on how your **Prometheus server(s)** has been configured.

If you need to finely tune the above settings, you should use **Helm**. For example, this can be useful if your Prometheus server is external to your cluster, hence you want simply to export the Liqo metrics to a public endpoint and scrape them from there. Refer to the *Install with Helm* section for further details.

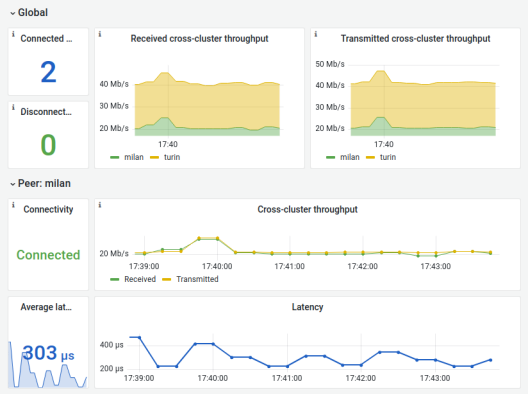
23.2 Cross-cluster network metrics

These metrics are available for each peered remote cluster, providing statistics about the cross-cluster network interconnections:

- **liqo_peer_receive_bytes_total**: the total number of bytes received from a remote cluster.
- **liqo_peer_transmit_bytes_total**: the total number of bytes transmitted to a remote cluster.
- **liqo_peer_latency_us**: the round-trip (RTT) latency between the local cluster and a remote cluster, in micro seconds, measured by a periodic UDP `ping` between the two Liqo gateways and sent within the Liqo tunnel itself.
- **liqo_peer_is_connected**: boolean keeping the status of the network interconnection between clusters, i.e., whether the peering is established and works properly, derived from the `ping` measurement above.

23.2.1 Grafana dashboard

We provide a `sample Grafana dashboard` to monitor the network interconnection of an arbitrary number of Liqo peerings. As presented in the screenshot below, it includes an overview section presenting the overall cross-cluster throughput, followed by detailed per-peering throughput and latency information.



23.3 Virtual kubelet metrics

These metrics are available for each peered remote cluster, providing statistics about the reflected resources:

- **liqo_virtual_kubelet_reflection_item_counter**: the number of resources that are currently successfully reflected (e.g., Pod, ConfigMap, Secret, Service, ServiceAccount, EndpointSlice, Ingress and PersistentVolumeClaim). This number can increase/decrease over time, and it may reach zero when two peered clusters have no reflected resources.
- **liqo_virtual_kubelet_reflection_error_counter**: the number of transient errors during the reflection phase. Errors can occur due to temporary race conditions that can be resolved by retrying the synchronization. These conditions mainly occur when some of the requested resources are not yet fully configured (e.g., no reflector is found for the given namespace and no fallback is configured, the fallback is not completely initialized this happens if namespace reflectors still need to be started, and the reflector is not completely initialized because only one of the two informer factories has synced).

23.3.1 Grafana dashboard

We offer a `sample Grafana dashboard` that allows you to monitor the reflected resources for each component of the virtual-kubelet. As shown in the screenshot below, it contains visual representations of the total number of reflected resources and the average rate per second. Additionally, there are detailed tables that provide information on the total number of each type of resource, as well as an overall summary of all reflected items during a certain time period.

CHAPTER

TWENTYFOUR

EXTERNAL NETWORK

Since Liqo v0.8.0, it is possible to enable ***resource reflection*** and ***namespace offloading*** without the need to establish network connectivity between the clusters. This feature is called **External Network**. In this section, we will see how to enable/disable this feature and how to use it.

24.1 Overview

The *External Network* feature allows resource reflection and namespaces offloading without requiring network connectivity between clusters.

This feature is useful in scenarios where the offloaded application **does not need to perform cross-cluster communication** but only needs to access local resources. For example, a batch processing application does not need to communicate with other clusters, but it needs to access a database external to the cluster. In this case, a network interconnection between clusters is not required and may lead to unnecessary **security issues** and **interdependencies** between clusters.

Another use case is when the clusters and the pods running on them are **already connected to the same network**. In this case, you may leverage the *External Network* feature to enable resource reflection and namespaces offloading without having to establish another network connection between the clusters.

24.2 Enable/Disable the External Network

This feature is **disabled** by default, and can be configured with **two** different **feature flags** at install time (see the *reference*):

- `--set networking.internal=false` to disable the internal network
- `--set networking.reflectIPs=false` to disable the reflection of the IP addresses

24.2.1 networking.internal=false

This flag disables the internal network. When this flag is set to `false`, the Liqo Gateway and the Liqo Route are not deployed on the cluster and the Liqo Network Manager is not started. The Liqo Network Manager is responsible for creating the `tunnel-endpoint` resource, which is used to establish the network connectivity between the clusters.

When the internal network is disabled, the Liqo Network Fabric is not enabled and **no parameter negotiation or IP remapping is performed**. The IP addresses of the remote pods are reflected as they are.

Note

The pod IPs are still reflected in the remote clusters, but they are not remapped. This means that the shadow pods will see the same IP in each cluster. It similarly happens with EndpointSlices resources. If you have an external network tool that handles the connection, you will be able to connect to the remote pods.

24.2.2 networking.reflectIPs=false

This flag disables the reflection of the IP addresses. When this flag is set to `false`, the **IP addresses** of the remote pods **are not reflected** and both local and remote EndpointSlices resources are not populated.

All shadow pods will have an **empty IP address**, and will not be selected as targets by any Kubernetes service.

CHAPTER

TWENTYFIVE

SERVICE CONTINUITY

This section provides additional details regarding service continuity in Liqo. It reports the main architectural design choices and the options to better handle eventual losses of components of the multi-cluster (e.g., control plane, nodes, network, liqo pods, etc.).

For simplicity, we consider a simple consumer-provider setup, where the consumer/local cluster offloads an application to a provider/remote cluster. Since a single peering is unidirectional and between two clusters, all the following considerations can be extended to more complex setups involving bidirectional peerings and/or multiple clusters.

25.1 Resilience to cluster failures/unavailability

Liqo performs periodic checks to ensure the availability and readiness of all peered clusters. In particular, for every peered cluster, it checks for:

- readiness of the foreign cluster's **API server**
- availability of the **VPN tunnel** for cross-cluster connectivity (*liqo-gateway*)

The **ForeignCluster** CR contains the status conditions indicating the current status of the above checks (named respectively *APIServerStatus* and *NetworkStatus*). A peered cluster is considered ready/healthy if **all** the above checks are successful.

25.1.1 Remote cluster failure

In this scenario the remote cluster is unavailable/unhealthy. Following the standard K8s protocol, the virtual node is marked as *NotReady* after `node-monitor-grace-period` seconds (default: 40s). This allows the control plane of the local cluster (which has visibility on all pods in a Liqo-enabled namespace, i.e. both local and remote pods) to mark all endpointslices associated to remote pods as not ready, preventing services to redirect traffic towards them in the same way services will not backend standards Kubernetes nodes. Also, new pods are scheduled on the remaining local nodes. As the virtual node transparently implements the standard Kubernetes interface, service continuity in the local cluster is guaranteed by Kubernetes in the event of unavailability of the remote cluster. Look at the official guide for further details.

25.1.2 Local cluster failure

In this scenario the local cluster is unavailable/unhealthy. Since the virtual node is not present on the remote cluster, Liqo logic ensures service continuity.

Remote pod resiliency

Remote pod resiliency (hence, service continuity) is ensured, even in case of temporary connectivity loss between the two control planes, through a custom resource (i.e., **ShadowPod**) wrapping the pod definition, and triggering a Liqo enforcement logic running in the remote cluster. This guarantees that the desired pod is always present, without requiring the intervention of the originating cluster. The virtual kubelet takes care of the automatic propagation of remote status changes to the corresponding local pod (remapping the appropriate information).

Fig. 1: Schematic representation of the pod offloading workflow. Solid lines refer to liqo-related tasks, while dashed ones to standard Kubernetes logic. Double circles indicate the pod in execution (i.e., whose containers are running). Blue rectangles refers to liqo-related resources.

Remote endpointslices

The endpointslices of all **local** pods must be disabled to prevent services to redirect traffic towards pods running on the local cluster to ensure service continuity on the remote cluster when the local cluster has a failure. Note that since the control plane of each remote cluster perceives only the pods running in its cluster, the Liqo EndpointSlice reflection fills the gaps by creating the necessary endpointslices associated with local pods (as explained *here*).

Liqo provides a more robust mechanism that offers better resiliency to cluster failures since version v0.8.2. It introduces an intermediate resource, the **ShadowEndpointSlice** CR, similar to the one adopted for the pods (i.e., ShadowPod). In this case, it is an abstraction that serves as a template for the desired configuration of the **remote** endpointslice. The virtual kubelet forges the remote shadow resource of a reflected endpointslice and creates it on the remote cluster. A controller in Liqo runs in the remote cluster and enforces the presence of the actual endpointslice, using the shadow resource as a source of truth. At the same time, it periodically checks the **local cluster status** (monitoring the above-described conditions) and dynamically updates the *Ready* condition of the endpoints in the endpointslices, depending on the cluster status. More specifically, endpoints are set ready only if both the VPN tunnel and the API server of the foreign cluster are ready. Note that a remote endpoint is updated only when the local endpointslice (and therefore the shadowendpointslice) has the *Ready* condition set to *True* or unspecified. If it is set to *False*, the remote endpoint condition is set to *False* regardless of the current status of the foreign cluster to preserve the local cluster's desired behavior.

Fig. 2: Schematic representation of the endpointslice reflection workflow. Solid lines refer to liqo-related tasks, while dashed ones to standard Kubernetes logic. Blue rectangles refer to liqo-related resources.

In summary, Liqo ensures that when a peered cluster is unavailable the endpoints of the local pods are temporarily disabled, and re-enabled when the cluster becomes ready again (if not explicitly disabled by the originating cluster).

25.2 Resilience to worker nodes failures

This section describes scenarios where one or more worker nodes are unavailable/unhealthy, with all control planes ready and the cross-cluster network up and running.

25.2.1 Worker node failure on the local cluster

Pods running on the local cluster are scheduled on regular worker nodes and therefore their entire lifecycle is handled by Kubernetes as explained in the official guide.

25.2.2 Worker node failure on the remote cluster

Offloaded pods are scheduled on the virtual node in the local cluster and run on regular worker nodes in the remote cluster. As explained in the *pod offloading section* the ShadowPod abstraction guarantees remote pod resiliency (hence, service continuity) in case of unavailability of the local cluster, enforcing the presence of the desired pod (scheduled on a regular worker node) without requiring the intervention of the originating cluster.

If a remote worker node becomes *NotReady* the Kubernetes control plane marks all pods scheduled on that node for deletion, leaving them in a *Terminating* state **indefinitely** (until the node becomes ready again or a manual eviction is performed). Due to design choices in Liqo, a pod that is (1) offloaded, (2) *Terminating*, (3) running on a failed node is **not** replaced by a new one on a healthy worker node (like in vanilla Kubernetes). The consequence is that in case of remote worker node failure, the expected workload (i.e., the number of replicas actively running) of a deployment could be less than expected.

Since Liqo v0.7.0, it is possible to overcome this issue. You can configure Liqo to make sure the expected workload is always running on the remote cluster, setting the Helm value `controllerManager.config.enableNodeFailureController=true` at install/upgrade time. This flag enables a custom Liqo controller that checks for all offloaded and *Terminating* pods running on *NotReady* nodes. A pod matching all conditions is force-deleted by the controller. This way, the ShadowPod controller will enforce the presence of the remote pod by creating a new one on a healthy remote worker node, therefore ensuring the expected number of replicas is actively running on the remote cluster.

As explained in the *pod reflection section*, the local cluster has the feedback on what is happening on the remote cluster because the remote pod status is propagated to the local pod and the number of **container restarts** is augmented to account for possible deletions of the remote pod (e.g., the Liqo controller force-deletes the *Terminating* pod on the failed node).

Warning: Enabling the controller can have some minor drawbacks: when the pod is force-deleted, the resource is removed from the K8s API server. This means that in the (rare) case that the failed node becomes ready again and without an OS restart, the containers in the pod will not be gracefully deleted by the API server because the entry is not in the database anymore. The side effect is that zombie processes associated with the pod will remain in the node until the next OS restart or manual cleanup.

25.3 High-availability Liqo components

Liqo allows to deploy the most critical Liqo components in high availability. This is achieved by deploying multiple replicas of the same component in an **active/standby** fashion. This ensures that, even after eventual pod restarts or node failures, exactly one replica is always active while the remaining ones run on standby.

The supported components in high availability are:

- ***liqo-gateway***: ensures no cross-cluster connectivity downtime. The number of replicas is configurable through the Helm value `gateway.replicas`
- ***liqo-controller-manager***: ensures the Liqo control plane logic is always enforced. The number of replicas is configurable through the Helm value `controllerManager.replicas`

Look at the *install customization options section* for further details on how to configure high availability during Liqo installation.

CHAPTER

TWENTYSIX

CONTRIBUTING TO LIQO

First off, thank you for taking the time to contribute to Liqo!

This page lists a set of contributing guidelines, including suggestions about the local development of Liqo components and the execution of the automatic tests.

26.1 Repository structure

The Liqo repository structure follows the Standard Go Project Layout.

26.2 Release notes generation

Liqo leverages the automatic release notes generation capabilities featured by GitHub. Specifically, PRs characterized by the following labels get included in the respective category:

- *kind/breaking*: Breaking Change
- *kind/feature*: New Features
- *kind/bug*: Bug Fixes
- *kind/cleanup*: Code Refactoring
- *kind/docs*: Documentation

26.3 Local development

While developing a new feature, it is typically useful to test the changes in a local environment, as well as debug the code to identify possible problems. To this end, you can leverage the *setup.sh* script provided for the *quick start example* to spawn two development clusters using KinD, and then install Liqo on both of them (you can refer to the *dedicated section* for additional information concerning the installation of development versions through liqoctl):

```
./examples/quick-start/setup.sh
liqoctl install kind --kubeconfig=./liqo_kubeconf_rome --version ...
liqoctl install kind --kubeconfig=./liqo_kubeconf_milan --version ...
```

Once the environment is properly setup, it is possible to proceed according to one of the following approaches:

- Building and pushing the Docker image of the component under development to a registry, and appropriately editing the corresponding *Deployment/DaemonSet* to use the custom version. This allows to observe the modified

component in realistic conditions, and it is mandatory for the networking substratum, since it needs to interact with the underlying host configuration.

- Scaling to 0 the number of replicas of the component under development, copying its current configuration (i.e., command-line flags), and executing it locally (while targeting the appropriate cluster). This allows for faster development cycles, as well as for the usage of standard debugging techniques to troubleshoot possible issues.

26.4 Automatic tests

Liqo features two major test suites:

- *End-to-end (E2E) tests*, which assess the correct functioning of the main Liqo features.
- *Unit Tests*, which focus on each specific component, in multiple operating conditions.

Both test suites are automatically executed through the GitHub Actions pipelines, following the corresponding slash command trigger. A successful outcome is required to make PRs eligible for being considered for review and merged.

The following sections provide additional details concerning how to run the above tests in a local environment, for troubleshooting.

26.4.1 End-to-end tests

We suggest executing the E2E tests on a system with at least 8 GB of free RAM. Additionally, please review the requirements presented in the *Liqo examples section*, which also apply in this case (including the suggestions concerning increasing the maximum number of *inotify* watches).

Once all requirements are met, it is necessary to export the set of environment variables shown below, to configure the tests. In most scenarios, the only variable that needs to be modified is `LIQO_VERSION`, which should point to the SHA of the commit referring to the Liqo development version to be tested (the appropriate Docker images shall have been built in advance through the appropriate GitHub Actions pipeline).

```
export CLUSTER_NUMBER=4
export K8S_VERSION=v1.21.1
export CNI=kindnet
export TMPDIR=$(mktemp -d)
export BINDIR=${TMPDIR}/bin
export TEMPLATE_DIR=${PWD}/test/e2e/pipeline/infra/kind
export NAMESPACE=liqo
export KUBECONFIGDIR=${TMPDIR}/kubeconfigs
export LIQO_VERSION=<YOUR_COMMIT_ID>
export INFRA=kind
export LIQOCTL=${BINDIR}/liqoctl
export POD_CIDR_OVERLAPPING=false
export TEMPLATE_FILE=cluster-templates.yaml.tmpl
```

Finally, it is possible to launch the tests:

```
make e2e
```

26.4.2 Unit tests

Most unit tests can be run directly using the *ginkgo* CLI, which in turn supports the standard testing API (*go test*, IDE features, ...). The only requirement is the controller-runtime envtest environment, which can be installed through `setup-envtest`:

```
go install sigs.k8s.io/controller-runtime/tools/setup-envtest@latest
setup-envtest use 1.25.x!
```

To enable the downloaded envtest, you can append the following line to your `~/.bashrc` or `~/.zshrc` file:

```
source <(setup-envtest use --installed-only --print env 1.25.x)
```

Some networking tests, however, require an isolated environment. To this end, you can leverage the dedicated *liqo-test* Docker image (the Dockerfile is available in *build/liqo-test*):

```
# Build the liqo-test Docker image
make test-container

# Run all unit tests, and retrieve coverage
make unit

# Run the tests for a specific package.
# Note, the package path must start with ./ to avoid the "package ... is not in GOROOT␣
↪error".
make unit PACKAGE_PATH=<package_path>
```

Debugging unit tests

When executing the unit tests from the *liqo-test* container, it is possible to use Delve to perform remote debugging:

1. Start the *liqo-test* container with an idle entry point, exposing a port of choice (e.g. 2345):

   ```
   docker run --name=liqo-test -d -p 2345:2345 --mount type=bind,src=$(pwd),dst=/go/
   ↪src/liqo \
      --privileged=true --workdir /go/src/liqo --entrypoint="" liqo-test tail -f /dev/
   ↪null
   ```

2. Open a shell inside the *liqo-test* container, and install Delve:

   ```
   docker exec -it liqo-test bash
   go install github.com/go-delve/delve/cmd/dlv@latest
   ```

3. Run a specific test inside the container:

   ```
   dlv test --headless --listen=:2345 --api-version=2  \
      --accept-multiclient ./path/to/test/directory
   ```

4. From the host, connect to *localhost:2345* with your remote debugging client of choice (e.g. GoLand), and enjoy!