POLITECNICO DI TORINO

Master's Degree in Computer engineering

Master's Degree Thesis

Cloudifying Desktop Applications: Your Laptop is your Data Center

Supervisors

Prof. Fulvio RISSO

Dott. Alex PALESANDRO

Candidate

Lorenzo CAMICIOLA

Academic year 2020-2021

To each one of the Liqoers, who showed me an open world of knowledge, hopes and possibilities and never left a friend behind during the hardest time of his life.

To Ugo, for everything else.

Summary

In the last decades the ICT world has undergone profound transformations, influencing and being influenced by the socioeconomic paradigm shift from a *ownership*-centered model to a *sharing*-oriented one, where the property of product is no longer important and what matters is the access to the service it provides.

What proved to be a real game changer was the transition from physical to virtual servers with the affirmation of virtualization at first and then with the introduction of containerization and container orchestrators, among which the Kubernetes project is becoming a standard platform for developers and operators. Released by Google in 2015, this open-source system provides a robust platform to automate the deployment and scaling of containerized applications in a clustered environment.

Nowadays, therefore, it is a common practice to embrace the *cloud computing* model: on-demand consuming digital services offloaded somewhere, aiming to achieve the maximum flexibility and a significant cost reduction with respect of a proprietary data center.

The progress made in this field, however, have always had a focus on professional users and enterprises, while affecting just indirectly the target of common desktop users where the lack of technical backgrounds and a limited need of virtualization usually led them to a Software-as-a-Service (SaaS) solution, by consuming out-of-the-box digital services which provide simple and friendly interfaces.

In the last years, the success of the IoT world, the massive spread of *smart* devices and a bigger pervasiveness of IT in everyday life have opened a different scenario, where everyone is surrounded by lots of digital devices, often small and always on, each one just performing few tasks. This moved the project in which this thesis is involved to explore the potential of **Liquid computing**, a distributed computing paradigm which aims to provide to the final user a unified virtual environment across devices where applications and data flow seamlessly.

This work proposes a desktop application which helps even non professional users to easily leverage computational power and storage provided by their own devices or offered by an organization (e.g. corporate, university) to run even heavy applications, thus becoming the managers of their own "data center".

Table of Contents

Chapter 1

Introduction

The search for more power and more powerful tools has always been intrinsically coupled with the industrial and technological progress. This is especially true in the ICT world, where in a few decades we observed a quantum leap even in the meaning itself of "*owning an electronic device*" in terms of availability, spreading, dimensions, capabilities and role in everyday life. Among with the internet development and the growth of available bandwidth, this process also brought to a paradigm shift about how and where the digital services are provided.

1.1 From the mainframe mountain to the cloud... and back: the «water cycle» of computing

1.1.1 The computing fragmentation

If the early adoption of electronic computational resources by an enterprise consisted in few huge expensive mainframes, the last decades of 20th century saw a trend of democratization in the access and availability of IT devices, mostly due to the virtuous cycle of constant engineering innovation, cheaper products and high demand. It was in 1965 when Gordon Moore made those first predictions on the exceptional growth of integrated circuits capabilities that would soon become known as his eponymous "*law*"[1]. This brought to a scenario amid the 90's where lots of servers and PCs spread across workplaces. The phenomenon was emphasized by the application of the «*One application per server*» rule, i.e. the good practice of running each service or application on a single underused machine, trying to prevent security and compatibility issues. This brought to a far by optimal configuration with waste of space and money, both *directly* by purchasing hardware and *indirectly* by running idle machines. It is worth remembering, in fact, that CPU utilization has only a minor impact on the global power consumption of a machine (as shown in figure 1.1) which is itself usually around 30% of the total amount.

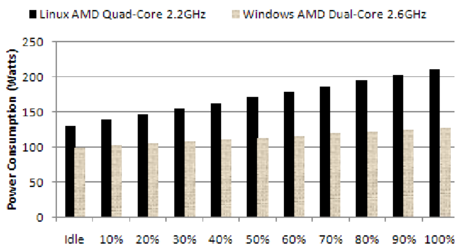


Figure 1.1: Power consumption on different CPU loads [2].
In the *Linux* case, moving the CPU from idle to full load only causes a change in power consumption of around 35%.

1.1.2 Up to the Cloud: The virtualization era and development of cloud computing

The 2000's represent the real change in the computing model with the widespread adoption of virtualization, on the push of several factors [3]:

- the urge to rationalize hardware resources and its management with the realization of data centers and IT departments, with the aim to overcome the *One application per server* rule.
- a new trend in the production process increasing the number of available cores per CPU, thus making possible to actually parallelize the execution of different virtual instances.
- the desire to obtain always more flexibility.

The transition from physical to virtual servers proved to be a real game changer, allowing to rethink data centers as a common resources pool to be shaped and scaled according to the needs of the moment, «*one massive warehouse-scale computer*» [4]. This concept evolved along the years to what is generically called "Cloud Computing": consuming on-demand a digital service offloaded somewhere, with the maximum flexibility. The definition nowadays covers a large number of flavors in terms of *ownership*, *kind of provided service* (e.g. the *X*-as-a-Service models [5]) and *virtualization type* (from the heavyweight virtualization of virtual machines to the lightweight one of containerization).

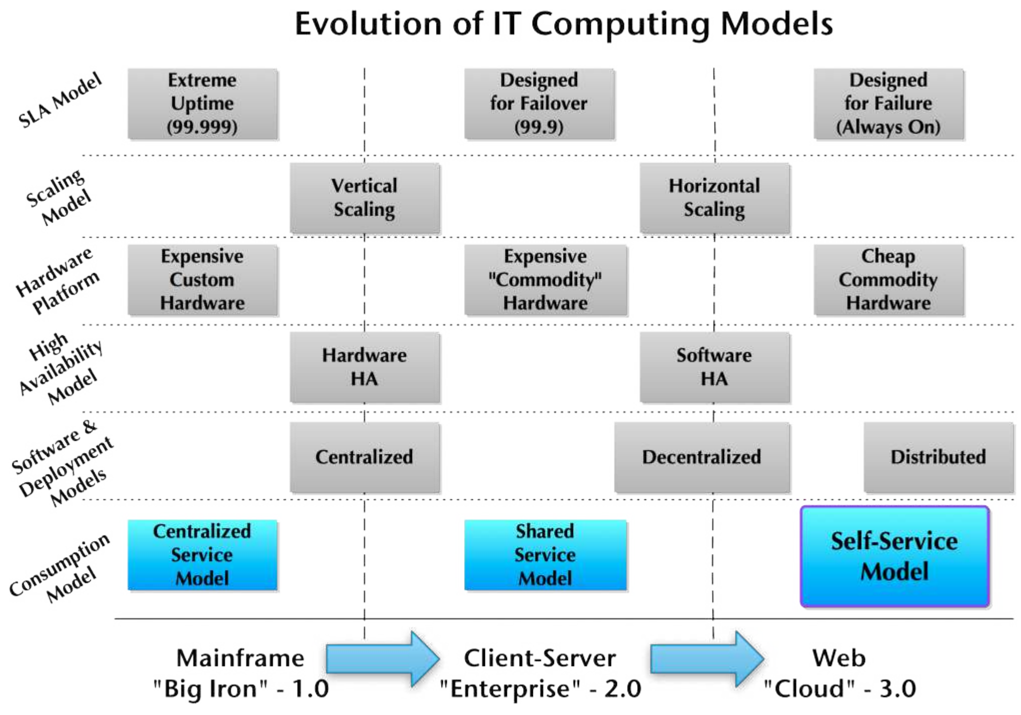


Figure 1.2: Evolution of IT computing models [6]

1.2 Liquid computing

With the introduction and rapid affirmation of the *Kubernetes* project [7], an orchestrator by Google aiming to automate and efficiently manage the deployment and scaling of containerized applications, there is substantially a new standard overlay stack for developing and deploying, with an automatic resource aggregation in the logic abstraction of a "cluster". It should be noted that the above description of computing models evolution has always been focused on the enterprise side, for the innovative drive, economic traction, features and market share.
The desktop market, on the contrary, has always got indirect benefit from the progress made for professional users, e.g. with free and limited versions of enterprise software. This was mainly due to the lack of a significant market share (both supply and demand sides):

- Common users are not supposed to have any kind of technical background and are often inclined to consume *out-of-the-box* digital services with the most simple and friendly interfaces.
- There is only a small fraction of desktop users who actually needs virtualization and its use is mostly related to take advantage of a parallel execution of different

OS.

- The average scenario of a home or small office environment usually involves just a few devices with limited capabilities with respect to a real data center in the enterprise field.

In the last decade, however, the explosion of the IoT world, the massive spread of *smart* devices and a bigger pervasiveness of IT in everyday life (and maybe even a bit more of awareness in this field) have brought to a different scenario, where everyone is surrounded by a large number of digital devices, often small and almost always on, each one just performing those few tasks it is designed for.

The project in which this thesis is involved aims to explore the potential of **Liquid computing**, by leveraging the Kubernetes environment and related tools to shape a unified, virtual environment where applications can be deployed and run seamlessly across federated clusters sharing storage and resources.

1.2.1 Goal of the thesis

The goal of this thesis is to make this federation mechanism available also to the desktop market, allowing even users who are not technical experts to finally benefit from the agility and elasticity of containers orchestration, becoming themselves managers of their own "data center" with several interesting (and even profitable) use cases:

- Take advantage of the underused and maybe outdated devices at home, thus optimizing costs and possibly avoiding to buy services from third party providers (which could also raise security concerns in some cases [8] [9]).
- Break the physical constraints of desktop solutions (e.g. PC, laptop), thus allowing to run even resource-consuming applications.
- Small organizations and universities could easily provide to employees and students controlled access even to licensed applications, with a new mechanism that could save significant money.

The analysis of this scenario brought to the development of a desktop application called "*Liqo Agent*" aiming to help even non professional users to manage the architecture of the cluster federation and to easily understand basic related information. The app has been designed to interact with the user with a clear, simple interface. The choice of a tray menu, in combination with desktop notifications tries to reduce the learning curve by emulating the usage of other common managers, like for wi-fi, antivirus or cloud storage providers. Originally developed for GNU-Linux distributions but with a cross-platform design, the application provides the cluster federation process (called "*peering*") by leveraging the *Liqo* project, an open-source framework developed inside the Computer Networks Group at Politecnico di Torino which runs on Kubernetes vanilla and has the complete support for *K3s* [10], a light Kubernetes distribution designed to work on IoT devices and in resource-constrained locations.

Chapter 2

Kubernetes

In this chapter we analyse Kubernetes architecture, showing also its history and evolution through time, in order to lay the foundations for all the work which will be exposed later on.[1] Kubernetes (often shortened as K8s) is a huge framework and a deep examination of it would require much more time and discussion, hence we only provide here a description of its main concepts and components. Further details can be found in the official documentation [12].

The chapter continues with an introduction to other technologies and tools used to develop the solution, in particular **Virtual-Kubelet** [13], a project which allows to create virtual nodes with a particular behaviour, and **Kubebuilder** [14], a tool to build custom resources.

2.1 Kubernetes: a bit of history

Around 2004, Google created the **Borg** [15] system, a small project with less than 5 people initially working on it. The project was developed as a collaboration with a new version of Google's search engine. Borg was a large-scale internal cluster management system, which "ran hundreds of thousands of jobs, from many thousands of different applications, across many clusters, each with up to tens of thousands of machines" [15].

In 2013 Google announced **Omega** [16], a flexible and scalable scheduler for large compute clusters. Omega provided a "parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, in order to achieve both implementation extensibility and performance scalability".

[1]This chapter is freely adapted from a similar one contained in the master thesis of another member of this project team [11]

In the middle of 2014, Google presented **Kubernetes** as on open-source version of Borg. Kubernetes was created by Joe Beda, Brendan Burns, and Craig McLuckie, and other engineers at Google. Its development and design were heavily influenced by Borg and many of its initial contributors previously used to work on it. The original Borg project was written in C++, whereas for Kubernetes the Go language was chosen.

In 2015 Kubernetes v1.0 was released. Along with the release, Google set up a partnership with the Linux Foundation to form the **Cloud Native Computing Foundation** (CNCF) [17]. Since then, Kubernetes has significantly grown, achieving the CNCF graduated status and being adopted by nearly every big company. Nowadays it has become the de-facto standard for container orchestration [18, 19].

2.2 Applications deployment evolution

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

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

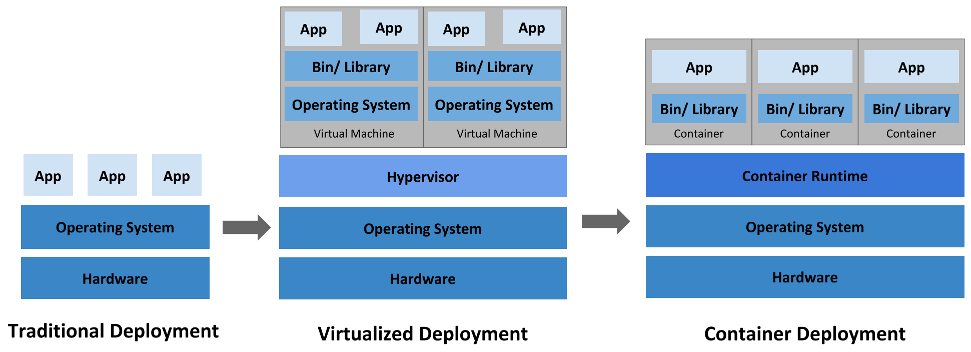


Figure 2.1: Evolution in applications deployment.

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

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

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

2.3 Container orchestrators

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

Kubernetes provides many services, including:

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

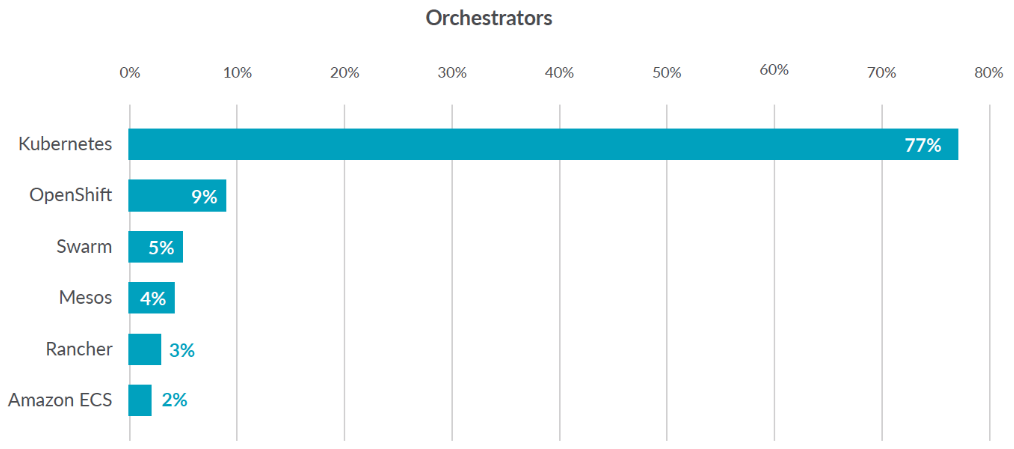


Figure 2.2: Container orchestrators use [20].

- **Automated rollouts and rollbacks** The desired state for the deployed containers can be described, and the actual state can be changed to the desired state at a controlled rate. For example, it is possible to automate the creation of new containers of a deployment, remove existing containers and adopt all their resources to the new container.

- **Automatic bin packing** Kubernetes is provided with a cluster of nodes that can be used to run containerized tasks. It is possible to set how much CPU and memory (RAM) each container needs, and automatically the containers are sized to fit in the nodes to make the best use of the resources.

- **Secret and configuration management** It is possible to store and manage sensitive information in Kubernetes, such as passwords, OAuth tokens, and SSH keys. It is possible to deploy and update secrets and application configuration without rebuilding the container images, and without exposing secrets in the stack configuration.

2.4 Kubernetes architecture

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

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

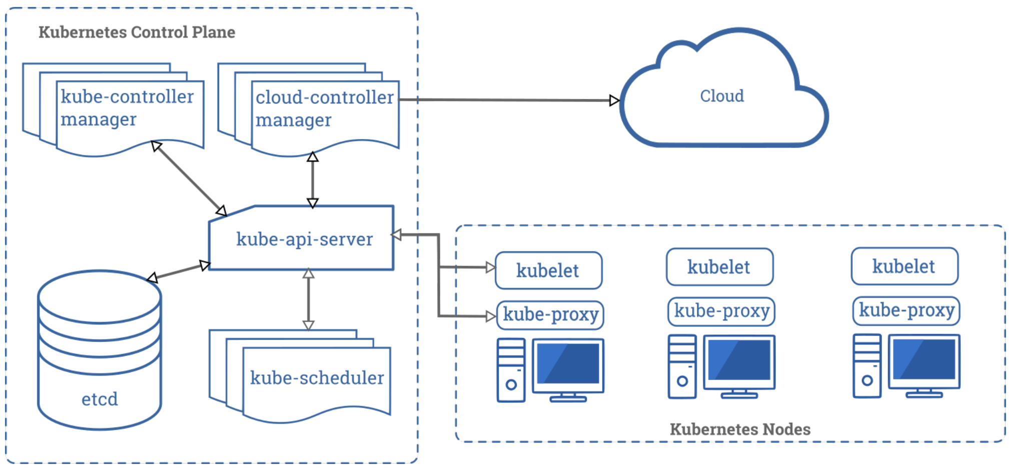


Figure 2.3: Kubernetes architecture

2.4.1 Control plane components

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

API server

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

etcd

`etcd` is a distributed, consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. It is based on the Raft consensus algorithm [21], which allows different machines to work as a coherent group and survive to the breakdown of one of its members. `etcd` can be stacked in the master node or external, installed on dedicated host. Only the API server can communicate with it.

Scheduler

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

kube-controller-manager

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

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

cloud-controller-manager

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

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

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

2.4.2 Node components

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

Container Runtime

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

kubelet

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

kube-proxy

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

Addons

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

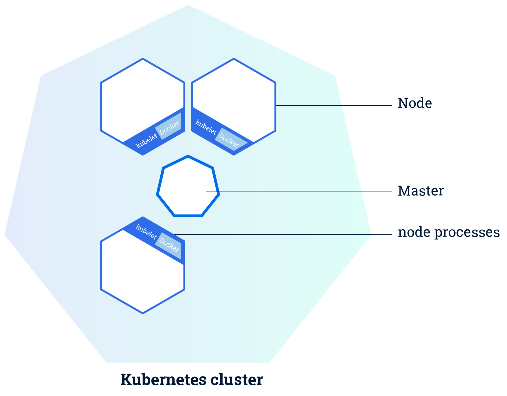


Figure 2.4: Kubernetes master and worker nodes [12].

2.5 Kubernetes objects

Kubernetes defines several types of objects, which constitutes its building blocks. Usually, a K8s resource object contains the following fields [22]:

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

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

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

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

2.5.1 Label & Selector

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

2.5.2 Namespace

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

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

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

2.5.3 Pod

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

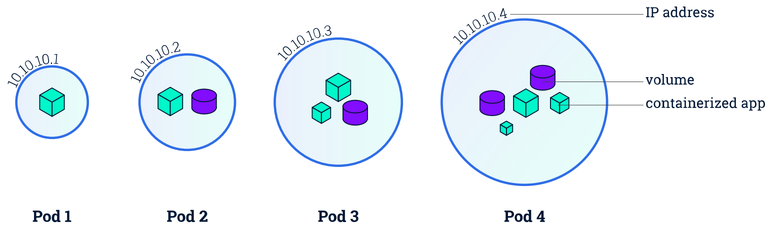


Figure 2.5: Kubernetes pods [12].

2.5.4 ReplicaSet

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

2.5.5 Deployment

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

Listing 2.1: Basic example of Kubernetes Deployment [12].

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

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

2.5.6 Service

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

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

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

Listing 2.2: Basic example of Kubernetes Service [12].

```
apiVersion: v1
kind: Service
```

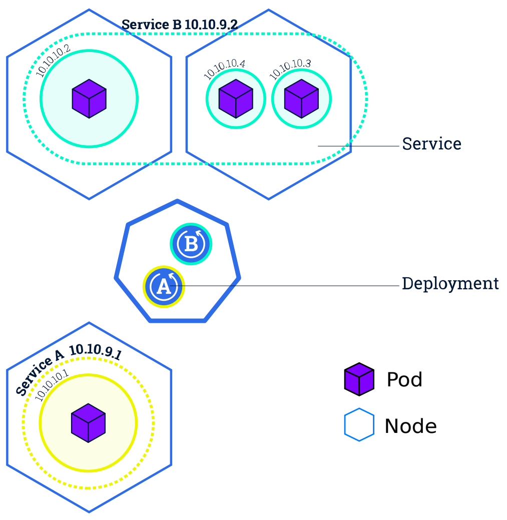


Figure 2.6: Kubernetes Services [12].

```
metadata:
  name: my-service
spec:
  selector:
    app: myApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

2.6 Virtual-Kubelet

Two Kubernetes-based tools which have been used during the development of this project are Virtual-Kubelet and Kubebuilder. Virtual Kubelet is an open source Kubernetes kubelet implementation that masquerades a cluster as a kubelet for the purposes of connecting Kubernetes to other APIs [13]. Virtual Kubelet is a Cloud Native Computing Foundation sandbox project.

The project offers a provider interface that developers need to implement in order to use it. The official documentation [13] says that “providers must provide the following functionality to be considered a supported integration with Virtual Kubelet:

1. Provides the back-end plumbing necessary to support the lifecycle management of pods, containers and supporting resources in the context of Kubernetes.
2. Conforms to the current API provided by Virtual Kubelet.
3. Does not have access to the Kubernetes API Server and has a well-defined callback mechanism for getting data like secrets or configmaps".

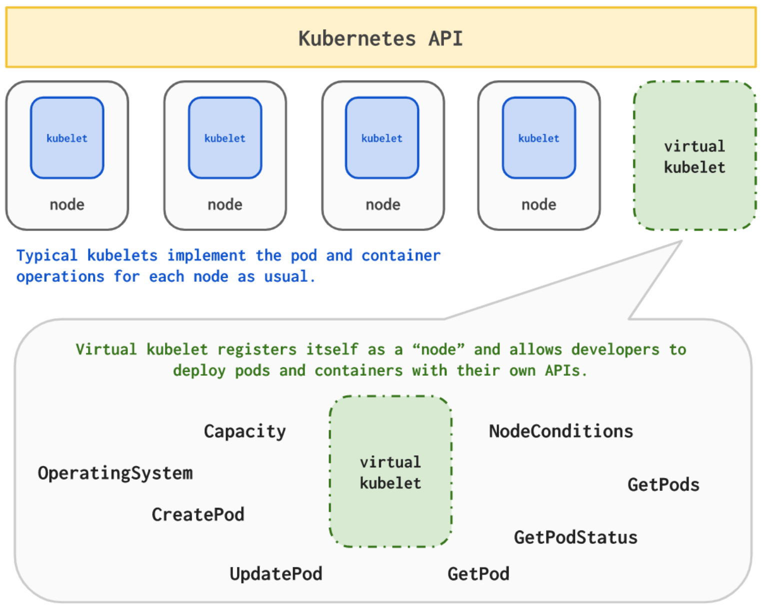

Kubernetes API
kubelet
node
kubelet
node
kubelet
node
kubelet
node
virtual kubelet
Typical kubelets implement the pod and container operations for each node as usual.
Virtual kubelet registers itself as a "node" and allows developers to deploy pods and containers with their own APIs.
Capacity
NodeConditions
OperatingSystem
virtual kubelet
GetPods
CreatePod
GetPodStatus
UpdatePod
GetPod

Figure 2.7: Virtual-Kubelet concept [13].

2.7 Kubebuilder

Kubebuilder is a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs) [14].

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

A **Custom Resource** is an endpoint in the Kubernetes API that is not available in a default Kubernetes installation and which frees users from writing their own API server to handle them [12]. On their own, custom resources simply let you store and retrieve structured data. In order to have a more powerful management, you also need to provide a custom controller which executes a control loop over the custom resource it watches: this behaviour is called Operator pattern [23].

Kubebuilder helps a developer in defining his Custom Resource, taking automatically basic decisions and writing a lot of boilerplate code. These are the main actions operated by Kubebuilder [14]:

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

Chapter 3

Resource sharing across Kubernetes clusters: the Liqo project

The main goal of the developed desktop application is to bring to common users an easy orchestration mechanism to aggregate and share resources between different Kubernetes clusters.

The **Kubernetes Cluster Federeation** (Kubefed) project [14] of the *Kubernetes Special Interest Groups* partially addresses this problem, but is still in a limited alpha version.

The choice fell to the **Liqo**[1] project [24], an open-source framework developed inside the Computer Networks Group at Politecnico di Torino which runs on Kubernetes vanilla and has the complete support for *K3s* [10], a light Kubernetes distribution designed to work on IoT devices and in resource-constrained environments.

3.1 Features

The key point of Liqo is to provide a powerful, dynamic resource sharing tool which, although, hides most of the complexity of clusters federation, by leveraging Kubernetes standard operations, both internally (automatic scheduling) and on the user side where a joined cluster is simply seen as a node.

Here are its most important features:

[1]The information contained in this chapter is mainly retrieved and revised from the documentation present on the Liqo repository [24] and the Liqo official website [25].

Peer-to-peer protocol

Each Liqo instance running on a cluster is seen as an equal peer which can request or receive peerings with others. A decentralized network is an important aspect for the project. This allows users, in fact, to be in full control of their choices, without requiring a central management unit which might also be offered as a paid service. Moreover, A distributed paradigm proves really helpful in all those context where there is no or limited internet connectivity but it is still possible to connects clusters under the same local network.

Multiple cluster discovery modes

The peering operation requires the involved clusters to be known to each other. Liqo provides several *discovery* methods:

- **Automatic LAN discovery**: all Liqo clusters present inside the same local network are automatically discovered via *multicast DNS* (*mDNS*). This is useful in those scenarios where a user needs to aggregate resources of their own home devices or lend a quota in a monitored environment, like a university laboratory.
- **Automatic discovery on geographic networks**: it is also possible to discover clusters present on geographic networks by associating them to a specific *DNS* domain. In this case, the process is simply based on DNS queries, with a 3-steps protocol similar to the one used for the discovery of SIP servers. This method could be preferred in large organizations, where a DNS-based mechanism provides an excellent unique point of management for several clusters.
- **Manual discovery**: for every other case, there is also the possibility to activate a manual discovery procedure, by explicitly creating a Kubernetes Custom Resource (*ForeignCluster*) and passing to it the hostname or the URL of the peer's authentication server.

Transparent offloading

The core of the Liqo framework is the easy join procedure (*peering*) between Kubernetes clusters. Each peer can send peering requests or handle received ones just by controlling a couple values of a *ForeignCluster* resource, which is the star point of all the collected data for each remote peer.

After a successful peering, the offloading of scheduled pods on remote clusters is handled completely automatically, unless explicit scheduling policies are enforced. In fact, one peer's remote resources are mapped on a virtual node (by means of a customized *virtual kubelet* [13]) of the home cluster (which allows the Kubernetes scheduler to take care of the scheduling).

Pod resilience

All the pods offloaded to a remote cluster can be controlled by simply performing operations on their shadow replicas (remote replicaset) running on the virtual kubelet in the home cluster. Thanks to a Liqo component called *CRDReplicator*, all state changes of the running pods are projected back on their virtual copy inside the home cluster, while all actions taken on them are also applied on the original resources on the remote cluster.

Secure inter-cluster networking

The connection between clusters is based on a reliable, encrypted tunnel by **Wireguard** [26], a general purpose, cross-platform VPN designed to be extremely fast and secure. Moreover, thanks to the Liqo networking module, between two peered clusters it is possible to achieve the same interconnection as inside a single cluster. All the pods in one cluster, in fact, can seamlessly communicate both with all pods and nodes of another one, with the module taking care of eventual pod and service CIDRs conflicts by means of a NAT.

Protection policies

The interconnection of clusters - especially when under different administrative domains - poses an important security issue. In order to address this aspect, Liqo leverages a wide set of security policies and features:

- Liqo networking module only exposes the CIDR subnet of the offloaded pods on the remote cluster, without affecting the entire network;
- all available Kubernetes features regarding security and accounting are enforced, such as *Role-Based Access Control* (to minimize the control of the external cluster admin), *Pod Security Policies* and hardened *Container Runtime Interfaces.*

CNI independence

Liqo supports a wide range of *Cloud Network Interfaces* (CNI) [27], e.g. the network plugins in charge of configuring connectivity for Linux containers and that are built upon the standard libraries and specifications of the *Cloud Native Computing Foundation.* Liqo clusters can also mount different CNIs.

3.2 General Architecture

3.2.1 Cluster representation

Each Liqo cluster is identified by a unique identifier (*ClusterID* and is characterized by several attributes and properties. On the local side it is represented by the **ClusterConfig** CRD (listing 3.1), which contains all configuration data and allows the customization of several aspects:

- **Acceptance policies for incoming peering requests**: users can automatically accept all incoming requests (useful for IoT devices) or decide manually on each one;
- **Outgoing peerings policies**, like the shared resources quota for each peering and automatic requests dispatch towards different clusters kind (e.g. only when they can be considered trusted);
- **Discovery management**: the home cluster can go silent or interrupt the discovery process of other clusters.
- **Cluster attributes**, like the mnemonic name used in combination with the clusterID.

Listing 3.1: Helm template of Liqo **ClusterConfig** CRD (version *0.2*)

```
apiVersion: config.liqo.io/v1alpha1
kind: ClusterConfig
metadata:
  name: {{ include "liqo.prefixedName" $config }}
  labels:
    {{- include "liqo.labels" $config  | nindent 4 }}
spec:
  advertisementConfig:
    ingoingConfig:
      acceptPolicy: AutoAcceptMax
      maxAcceptableAdvertisement: 1000000
    keepaliveThreshold: 3
    keepaliveRetryTime: 20
    outgoingConfig:
      {{- .Values.advertisement.config | toYaml | nindent 6 }}
  discoveryConfig:
    {{- .Values.discovery.config | toYaml | nindent 4 }}
    domain: local.
    name: MyLiqo
    port: 6443
    service: _liqo_api._tcp
  authConfig:
    {{- .Values.auth.config | toYaml | nindent 4 }}
  liqonetConfig:
    {{- .Values.networkManager.config | toYaml | nindent 4 }}
  dispatcherConfig:
    resourcesToReplicate:
    - group: net.liqo.io
      version: v1alpha1
      resource: networkconfigs
  agentConfig:
    dashboardConfig:
      namespace: {{ .Release.Namespace }}
      appLabel: "liqo-dashboard"
```

3.2.2 Discovery

On the other side, a Liqo cluster is represented by a Liqo **ForeignCluster** CRD which is the star point of all data related to a specific peer.

A *ForeignCluster* instance represents a discovered peer and contains information:

- **on the cluster itself**, like identification attributes (e.g. *ClusterID*, *ClusterName* or *DiscoveryType*) and parameters constraints to perform authentication on that cluster first and then a complete peering towards it, e.g. if it is recognized as trusted (listing 3.2);
- **on the active peerings towards it, both incoming and outgoing**: while in the *spec* section it is possible to trigger the peering process, in the *status* one the user can monitor all active peerings in the correspondent outgoing or incoming part and also get indirect access to all related Custom Resources, e.g. the *PeeringRequest* CR associated to an active incoming connection.

Listing 3.2: Extract of the *spec* field of Liqo **ForeignCluster** CRD (version *0.2*)

```
  group: discovery.liqo.io
  names:
    kind: ForeignCluster
          spec:
            description: ForeignClusterSpec defines the desired state
   of ForeignCluster
            properties:
              authUrl:
                description: URL where to contact foreign Auth
  service
                type: string
              clusterIdentity:
                description: Foreign Cluster Identity
                properties:
                  clusterID:
                    description: Foreign Cluster ID, this is a unique
 identifier of
                      that cluster
                    type: string
                  clusterName:
                    description: Foreign Cluster Name to be shown in
GUIs
                    type: string
                required:
                - clusterID
                type: object
              discoveryType:
                default: Manual
                description: How this ForeignCluster has been
discovered
                enum:
                - LAN
                - WAN
                - Manual
                - IncomingPeering
                type: string
              join:
                default: false
                description: Enable join process to foreign cluster
                type: boolean
              namespace:
                description: Namespace where Liqo is deployed
                type: string
              trustMode:
                default: Unknown
                description: Indicates if this remote cluster is
trusted or not
                enum:
                - Unknown
                - Trusted
                - Untrusted
                type: string
```

3.2.3 Peering

The peering process is the keystone of the Liqo framework.

(1) A user starts the peering request towards a peer by updating the correspondent ForeignCluster resource.

(2) If the authorization process on the foreign cluster has previously succeeded, the CRDReplicator component creates in there a new PeeringRequest CRD. On

the receiving peer's side, this Custom Resource is the star point of all the related components handling the sharing process. Its creation triggers the PeeringRequest operator to generate the related Broadcaster, which is the component in charge of keeping the active connection with the home cluster.

(3) According to the information contained in the ClusterConfig, the Broadcaster sends back to the home cluster an Advertisement CRD. This resource has two main scopes. On one hand, in fact, it contains the *offer* of the shared resources for the home cluster, among with their prices (listing 3.3). On the other hand, instead, it represents the core of a keep-alive mechanism implemented by the Broadcaster which periodically refreshes the Advertisements, similarly to what happens between network devices. The *status* section of the PeeringRequest contains both the references to the Broadcaster component and the Advertisement resource (listing 3.4). It is also possible to monitor its acceptance status.

(4) If the home cluster accepts the Advertisement, then on its side a **virtual kubelet** (see [13]) is created in the cluster in the form of a virtual node, having an amount of available resources equal to the accepted shared quota. Now it is up to the standard Kubernetes scheduler to deploy user resources on the available nodes.

Listing 3.3: Offer of shared resources quota in the *spec* field of Liqo **Advertisement** CRD (version *0.2*)

```
group: sharing.liqo.io
names:
  kind: Advertisement
        spec:
            prices:
              additionalProperties:
                anyOf:
                - type: integer
                - type: string
                pattern: ^(\+|-)?(([0-9]+(\.[0-9]*)?)|(\.[0-9]+))(([KMGTPE]i)|[numkMGTPE]|([eE](\+|-)?(([0-9]+(\.[0-9]*)?)|(\.[0-9]+))))?$
                x-kubernetes-int-or-string: true
              description: Prices contains the possible prices for every kind of
                resource (cpu, memory, image).
              type: object
            properties:
              additionalProperties:
                type: string
              description: Properties can contain any additional information about
```

```
                the cluster.
              type: object
            resourceQuota:
              description: ResourceQuota contains the quantity of resources made
                available by the cluster.
              properties:
                hard:
                  additionalProperties:
                    anyOf:
                    - type: integer
                    - type: string
                    pattern: ^(\+|-)?(([0-9]+(\.[0-9]*)?)|(\.[0-9]+))(([KMGTPE]i)|[numkMGTPE]|([eE](\+|-)?(([0-9]+(\.[0-9]*)?)|(\.[0-9]+))))?$
                    x-kubernetes-int-or-string: true
                  description: 'hard is the set of desired hard limits for each
                    named resource. More info: https://kubernetes.io/docs/concepts/policy/resource-quotas/'
                  type: object
```

Listing 3.4: Extract of *status* section of Liqo **Advertisement** CRD (version *0.2*)

```
group: sharing.liqo.io
names:
  kind: Advertisement
        status:
          description: AdvertisementStatus defines the observed state of Advertisement
          properties:
            advertisementStatus:
              description: AdvertisementStatus is the status of this Advertisement.
                When the adv is created it is checked by the operator, which sets
                this field to "Accepted" or "Refused" on tha base of cluster configuration.
                If the Advertisement is accepted a virtual-kubelet for the foreign
                cluster will be created.
              enum:
              - ""
              - Accepted
              - Refused
              type: string
```

Chapter 4

Liqo Agent: application design and general architecture

This thesis aims to design and develop a desktop application that could respond to two main goals:

1. to provide a single entry point that allows users to **act as managers of their own "data center"**, by aggregating their own computational resources spread across devices or joining the ones offered by different entities, like a colleague or an organization;

2. to target common desktop users - with little or no knowledge at all of the underlying technology - by offering a clear graphic user interface (GUI) and few simple interaction mechanisms, thus trying to achieve the best user experience.

Design choices led to the development of a tray bar application that, by means of a tray menu and desktop notifications, allows to perform simple *peering* operations between devices, by leveraging the *Liqo* framework running on a Kubernetes cluster.

4.1 Use cases

By exploring and extending the potential of the **Liquid Computing** paradigm, this application, in combination with the Liqo framework running underneath, envisions a further evolution in the fruition of web services.

We live in a world, in fact, where everything is becoming "a service" you can only consume but, on the other hand, almost everyone nowadays manages several aspects of their life with just a few taps on their own phone. There is a schism that could be healed by combining the technical benefits of the new distributed cloud-based web, with the democratization wave that brought “the computer” from large corporate rooms to the homes of common people, as shown in fig 4.1.

As a consequence of this increasing trend, computational resources could become a shareable commodity not only in professional context - where already is somehow like this, with the Infrastructure-as-a-Service model - but in everyday life, with the ability to run almost every kind of application (in terms of size, resource constraints and so on) everywhere, deciding a bit more whose is the cloud we are flying on.

In the design of this first version of the application, called **Liqo Agent**, the focus has been on making available to users features that could address some core use cases:

- when connected to their local network, a user can easily view all their devices (e.g. old PCs, IoT devices) running a Liqo instance and perform a peering towards them, creating their own resource pool to run several application and thus avoiding to buy third party services;
- a student or employee can peer to the Liqo cluster of the organization they belong to (e.g. university department, corporate), possibly with an authentication mechanism, and consume their quota of shared resources, running in the remote cluster application that could be heavy or with environmental requirements (e.g. license policies);
- skilled users can enjoy an easy entry point to manage their Liqo instances and to control even advanced configurations, with a direct access link to the *LiqoDash*, a general purpose dashboard developed inside the Liqo team that also acts as a web based UI for Liqo [28];
- a user can receive a peering request from another peer (e.g. a friend) and decide to share or not its resources.

4.2 Application design and user experience

The application is mainly intended for desktop users with low technical skills and thus moved us to put special efforts in order to provide the best **user experience** which includes all the aspects influencing the human interaction with a program interface and that have been categorized after recent studies [29] under three main factors: (1) *user’s state and previous experience* (2) *system properties* (3) *usage context.* This motivation led to some design choices for the user interface:

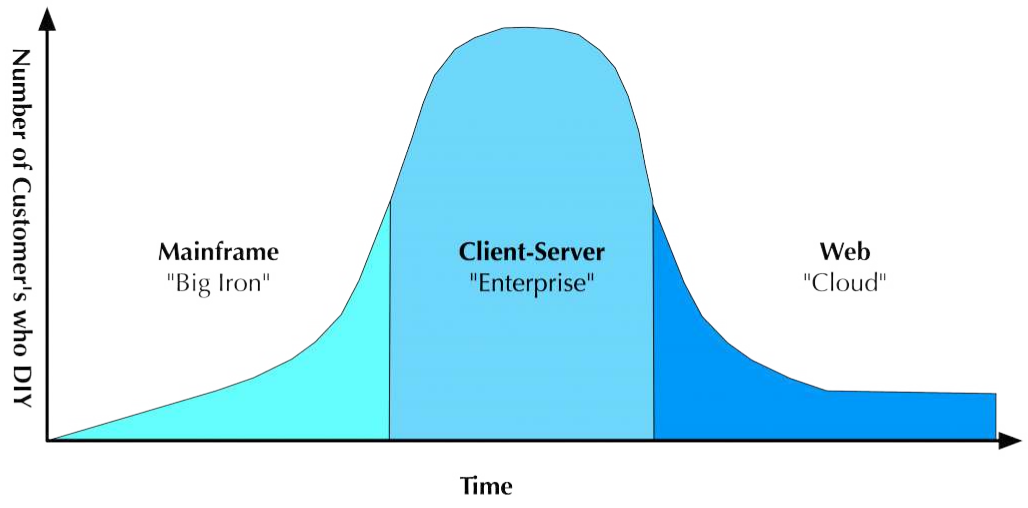


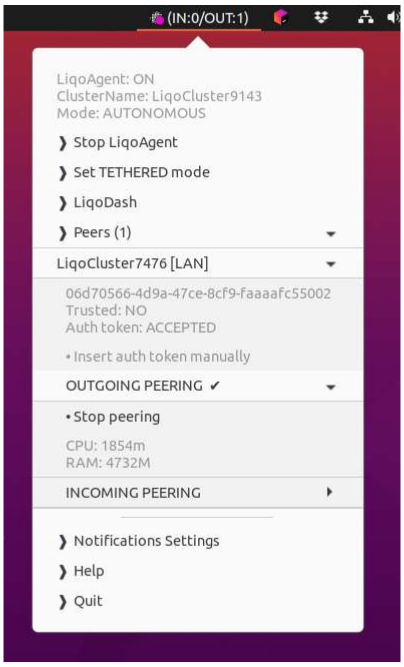
Figure 4.1: Adoption curve in the evolution of IT infrastructure models [6]. Technological innovations made possible to a larger number of people to take competitive advantage first and then provide high value professional services ("client-server"), until market reached the maturity to provide large scale consumable services. With the new Liquid computing paradigm, the 3.0 web could become the basis of a new escalating trend of early adopters starting to decentralize even more the cloud computing model.

- the application starts silently at boot in the form of a tray bar agent (hence its name) to emulate the look and feel of other common desktop managers which users should be used to, like the ones for WiFi, antivirus or cloud storage providers. The learning curve should thus be reduced;

- *Liqo Agent*'s tray icon remains always visible during the device activity and by means of some changes in its appearance, it lets users have immediate overview of their Liqo Cluster conditions, e.g. it there is an active connection, the presence of active peerings or special errors occurred needing their attention;

- with a simple click on the tray icon, users can access a tray menu presenting a set of useful shortcuts, a control panel with basic information about the program state and a list with all the *peers* discovered automatically by the Liqo framework, so that they can manage each single peering operation with them similarly to the well known "available networks" workflow (see 4.2);

- *Liqo Agent* can send desktop notifications to clearly inform users about important events.



(IN:0/OUT:1)

LiqoAgent: ON
ClusterName: LiqoCluster9143
Mode: AUTONOMOUS

06d70566-4d9a-47ce-8cf9-faaaafc55002
Trusted: NO
Auth token: ACCEPTED

CPU: 1854m
RAM: 4732M

Figure 4.2: *Liqo Agent* menu preview: while the icon and its associated text box shows that some peerings are active, a user can click on a specific peer's label and control information regarding it.

4.3 General architecture

Liqo Agent is a desktop application installed on the local file system and has three interconnection points:

- with a **Kubernetes cluster** where a **Liqo** application is running (hereinafter called *Liqo Cluster*). The app uses a configuration file (*kubeconfig*) retrieved from the file system to open a set of clients towards the cluster. By using these connections, the *Agent* is able to retrieve information from the Liqo framework, watch events regarding main aspects of the foreign clusters discovery and peering procedure and perform operations like requesting an outgoing peering or accepting an incoming one.

- with the Operating System **Display server**, whose primary task is to coordinate communications between its clients and I/O devices (via kernel). *Liqo Agent* connects to the server to show the tray menu and tray icon and send desktop notifications, while listening to the events of mouse clicks on menu choices.

- as little as possible with the **file system**, primarily for accessing the *kubeconfig* file and the *Agent* configuration file, containing persistent data of its settings. Desktop notifications are designed to also contain small symbols that are read from the disk when needed, but this behavior can be avoided transparently to limit expensive calls to the file system, especially on old devices.

The application has been designed following the *Model-View-Viewmodel* (MVVM) pattern, a variation of the original Model-View-Controller which aims to facilitate the separation between internal logic and user interface implementation, thus simplifying cross-platform support.

The core of the application is represented by the **Indicator** which manages the tray menu and contains internal representation of peers and peerings data. It connects to the *Liqo Cluster* through an instance of the **AgentController** component that handles all the connections towards the Kubernetes API server. All user interactions are delegated instead to the **GUIProvider**, which exports a set of standard graphic primitives towards the Indicator and connects internally with the display server.

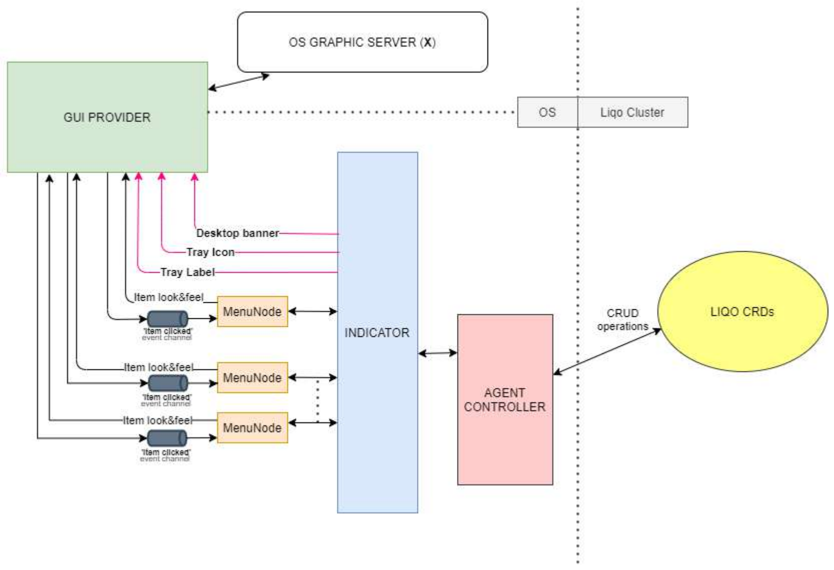


Figure 4.3: Liqo Agent architecture design.

4.3.1 GUI Provider

The GUIProvider is the component that provides the application user interface, hiding the complexity of different internal implementations according to the Operating System where the *Liqo Agent* is installed on.

By connecting to the OS display server by means of the right communication protocol, it manages both I/O operations and the *look and feel* of graphical items:

- the set of tray icons, used to inform users of the current status of the application;
- the desktop banner notifications warning about important events;
- input boxes that simplify application usage avoiding users the need to access command line tools, e.g. *kubectl*;
- the tray menu items, both in the content update and in the creation of the event listeners for each entry, triggered by click events.

4.3.2 Indicator

The indicator is the main component of *Liqo Agent*, representing the star point between the GUIProvider and the AgentController. It is a stateful data structure containing the internal representation of both graphical components (tray menu and icons) and abstract entities related to Kubernetes objects and Liqo abstract entities, like discovered foreign clusters (*peers*) and active connections (*peerings*) towards them. All communications flows between user and Kubernetes cluster are filtered and interpreted by the Indicator, mainly by means of event handler associated *globally* either to the entire component, called **Listeners**, or to single menu items, in the form of connected callbacks.

From an architectural point of view, it may be seen as a composite object, aggregating several sub components hereafter described.

Status

The Status component acts as the internal *Liqo Agent* database, storing and managing information about the running Liqo instance (associated to the so called *Home cluster*) and statistics on Liqo objects:

- **Home Liqo cluster details**, like (1) *common (i.e. human-readable) name* (2) *ClusterID*, a unique cluster identifier (3) *operating state* and *working mode*;
- **Discovered peers details** about (1) their identifier (2) *state of the authentication process* to the correspondent foreign cluster (3) *active peerings state*, including the amount of the shared resources quota;

Tray Menu

The Indicator contains the entire tray menu hierarchy implementation. Each menu item is either used as information panel or as active button with an associated callback which the Indicator uses as the main input flow from the users.

Listeners

The Indicator instantiates a set of event handlers called Listeners, containing all the logic for the data flow from the Liqo Cluster towards the user. each one triggered by signals sent by the AgentController when specific conditions in the Liqo cluster are met. According to internal implementation, they also send notifications out.

Timers

If requested, the Indicator can also activate custom Listeners, called Timers, whose activation depends on a repeatable countdown instead of generic events.

4.3.3 Agent Controller

The AgentController is the *Liqo Agent* component that interfaces with the Liqo cluster. It handles all Kubernetes related operations, hiding the complexity that is usually left to the operators via command line tools like *kubectl.* Its main task is to manage data and control flows between the cluster and the Indicator:

- *from user input to Liqo cluster*: by clicking on tray menu items, users can perform simple Kubernetes operations (e.g. connect to different clusters) or request abstract Liqo operations internally mapped to multiple CRUD calls, like the discover new peers or establishing new peerings. The result of these operations are then displayed as information on the tray menu or notified via desktop banner;
- *from Liqo cluster to users*: the AgentController watches internally several events inside the cluster, combining single conditions on different resources into logical Liqo events.

4.4 Available Features and Execution Workflow

After the operating system start up, *Liqo Agent* loads the Indicator and the associated tray menu within, then allocates the proper GUIProvider instance according to the local environment, thus displaying the tray icon and enabling some choices in the tray menu.

The application then seeks a *kubeconfig* file that could be even requested to the user if no valid hints for a valid file path are provided. If found, the Indicator then creates an AgentController instance based on that file and opens all the required connections towards the API server of the Liqo cluster specified in that.

If the connection results successful, the Indicator enables also the tray menu choices related to Liqo operations, e.g. the peers list, and starts its Listeners, waiting for signals sent by the AgentController. The complete list of available features are described below.

4.4.1 LiqoDash integration

LiqoDash [28] is a general purpose web based dashboard, developed inside the same *Liqotech* [30] team, that provides an advanced tool for professional users to deeply analyse a Kubernetes cluster and fully customize vanilla or custom resources running inside it. Among its useful features, it offers:

- a fully dynamic ***CRD*** (Kubernetes *Custom Resource Definition*) **form generator** that allows to create new resources or updating the existing ones without any knowledge of the YAML language;
- a powerful **custom View editor** that can be used to create custom views to access and monitor just the needed components, even with graphical tools like graphs and charts.

In particular, LiqoDash provides a built-in view for the Liqo application, showing detailed information on each Liqo component and a dedicated graph of the discovered peers and the active peerings between them. This tool, mainly meant for skilled users, needs an authorized access, usually through an OIDC provider, by means of the *OAuth* framework. In order to provide a simplified method to use the LiqoDash, *Liqo Agent* offers an extremely simplified two steps procedure, triggered by clicking on the correspondent tray menu button:

1. If the LiqoDash has been deployed (hopefully by using the simplified procedure inside the Liqo installer), *Liqo Agent* takes care of building the correct URL, according to information retrieved inside the Liqo cluster. It then automatically opens that address inside the default browser;
2. to simplify the authentication and authorization procedure, it uses a secondary token based process, by retrieving a secret token generated inside the cluster. Since the token length is significantly larger with respect of a common password, the application loads it into the clipboard buffer, so that users can easily paste it inside the input form on the LiqoDash web page.

4.4.2 Available Peers

The tray menu provides the "*Peers*" panel, resembling the look and feel of the *available networks* manager. By internally leveraging the *discovery* procedure of the Liqo framework, *Liqo Agent* can show a dynamic list of the Liqo clusters currently known by the home cluster, including just the most important details for the common user.

Discovery type

By means of a label, *Liqo Agent* allows to distinguish between clusters inside the Home cluster local network, with respect to the ones discovered on geographical networks by means of a DNS-based methods.

Trusted clusters

Discovered clusters are classified according on the certification of their data. In the details panel of each peer, *Liqo Agent* shows a **TRUSTED** or **UNTRUSTED** tag to identify whether the certificate exchanged by the correspondent foreign cluster has been signed by a trusted Certification Authority, with respect, for example, to a self-signed one. This allows users to perform an additional security check before requesting or accepting a new peering.

Liqo cluster Authentication

After the discovery of a new peer, the Liqo framework starts the authentication process towards the correspondent foreign cluster, in order to receive special rights to communicate with its API server. The successful completion of this procedure is required in order to perform subsequent peering request and it may be accomplished by means of two different methods:

- **Empty Token**: if the foreign cluster allows this procedure, it automatically provides the home cluster a valid temporary token, so no actual authentication is required;
- **OOB Token**: the home cluster must authenticate by sending to the foreign cluster a *valid* authN token previously obtained *Out Of Band*.

In the details panel of each peer, *Liqo Agent* shows the current state of the authentication process by means of the tags **PENDING, ACCEPTED, REFUSED**. In presence of a **REFUSED** state, the tray menu enables the tray menu choices to manually insert a new valid token, by directly pasting it in an input box, in order to perform a new authentication.

4.4.3 Peerings

Inside each peer's view, users can control the state of both outgoing and incoming peerings between that foreign cluster and theirs. If the authentication process resulted successful (ACCEPTED), it is possible to request an *outgoing* peering with just one click and the entire procedure is internally handled by the *Liqo Agent*. When a connection (of both types) is successfully established, users receive a desktop notification with general information about it, while the correspondent panel in the peer's view shows the amount of the shared resources quota. In order to tear down the peering, one has to just click on the correspondent "Stop Peering" button.

4.4.4 Secondary options

In addition to the core features above described, the *Liqo Agent* interface allows the user to perform some minor operations, like (1) a direct link to Liqo and *Liqo Agent* documentation (2) a notification settings panel, where it is possible to disable the desktop notifications and even the tray icon indicators.

Chapter 5

Liqo Agent: implementation

Since the application is intended to be used also on resource-constrained devices, efforts have been made also in the implementation phase to keep it light and fast.

The *Go* language played a role thanks to its key features about resource consumption, run-time efficiency and support to concurrent programming based on its custom lightweight processes: the goroutines. This has made it possible to develop even the entire GUI by using Go and to make the application binary cross-platform with minor adjustments.

The software is composed of 4 main libraries: (1) *Client*, providing to the Indicator the connectivity to the Kubernetes clusters and the implementation of the AgentController component (2) *Indicator*, containing the implementations of the Indicator and the GuiProvider and controlling the user interaction. (3) *Logic* which orchestrates the entire execution flow, containing the implementations of almost all the callbacks, e.g. the main routine (event loop), the actions associated with each tray menu entry and the *goroutines* listening to messages sent by the AgentController. (4) *Icon* which contains the set of tray icons used by the Agent to notify status changes.

The entire *Liqo Agent* code can be found on its public GitHub repository [31].

5.1 Efficiency improvements

The key of the *Liqo Agent* project is to allow users to acquire computational resources even on low powered and outdated devices. This led to pay more attention both on the choice of the programming language (and frameworks) and in the implementation of each component (which will be discussed later).

5.1.1 The Go language

Since it was publicly released by Google in 2009, the Go language has been growing interest around it, with a pretty interesting adoption curve, thanks also to big companies starting to use it to create new projects and migrate old and even famous ones. This was mainly due to its unique features about clarity, resource usage and efficient concurrency management. In fact, if on one hand it is a statically typed, compiled language developed based on C, on the other hand it was given powerful tools like garbage collection, interfaces, a simplified object-oriented programming style. This led, moreover, to most of the cloud native projects being written in Go, including Kubernetes.

With respect to the *Liqo Agent* development, the main factors that led to choose Go as programming language are the following:

- Google created *Go* starting from *C* style and main principles, making it feasible also to build software for operating systems, thanks also to the possibility to easily integrate C code in Go programs and vice versa. Go aims, in fact, especially to solve scalability issues, almost due to hardware constraints.

- Since it is a high level language supporting a large number of system architectures, there is an increasing trend among developers into making Go a "Swiss Army Knife", by creating wrapper libraries [32] that contain (1) useful widespread frameworks originally implemented in other languages (2) creating cross-platform frameworks that hide several platform-dependent implementations of the same feature, thus providing great code reuse and compatibility. This was very useful in the realization of the desktop notifications system.

- Even if it is mainly intended for *back-end* solutions, there is an incoming trend of developing graphical libraries, either web based (HTML/CSS/JS) or connected to the OS display server, as shown in sec5.2.1.

- Go provides native support to concurrent programming by means of its lightweight processes structures, the *goroutines*. Differently from other languages solutions, they are extremely versatile and with minimum costs for their creation and deletion.

- Google wrote the entire Kubernetes project in Go, providing powerful up-to-date clients towards clusters.

5.2 App-Indicator library

The **App-Indicator** is the core package of the application, containing:

- the implementation of the *Indicator* and all related data;
- the *GuiProvider* interface and its current implementation.

5.2.1 GuiProvider

GuiProvider is the component in charge of the app interaction with the OS Display server, hiding all the complexity of different platform-dependent implementations.

Listing 5.1: app-indicator **GuiProviderInterface**

```
//GuiProviderInterface wraps the methods to interact with the OS display server and manage a tray icon with its menu.
type GuiProviderInterface interface {
    //Run initializes the GUI and starts the event loop, then invokes the onReady callback. It blocks until
    //Quit() is called. After Quit() call, it runs onExit() before exiting. It should be called before
    //any other method of the interface.
    Run(onReady func(), onExit func())
    //Quit exits the GUI runtime execution after Run() has been called.
    Quit()
    //AddSeparator adds a separator bar to the tray menu.
    AddSeparator()
    //SetIcon sets the tray icon.
    SetIcon(iconBytes []byte)
    //SetTitle sets the content of the label next to the tray icon.
    SetTitle(title string)
    /*
        AddMenuItem creates and returns an Item, e.g. an entry of the tray menu. The menu works as a stack with only 'push'
        operation available. Use Item methods (e.g. Item.Hide()) to emulate 'pop' behavior.

            withCheckbox = true has to be used on Linux builds to force the creation of an Item with an actual checkbox.
            Otherwise the graphical behavior of Item.Check() is demanded to internal implementation.
    */
    AddMenuItem(withCheckbox bool) Item
    /*
        AddSubMenuItem creates and returns a child Item for a parent Item so that it can be displayed as
        a submenu element in the tray menu. Each Item submenu works as a stack with only 'push'
        method available. Use Item methods (e.g. Item.Hide()) to emulate 'pop' behavior.

```

```
		withCheckbox = true has to be used on Linux builds to force the creation of an Item with an actual checkbox.
		Otherwise the graphical behavior of Item.Check() is demanded to internal implementation.
	*/
	AddSubMenuItem(parent Item, withCheckbox bool) Item
	//Mocked returns whether the interaction with the OS display server is mocked.
	Mocked() bool
	//NewEventTester resets and return the EventTester. You can then call EventTester.Test() to start the testing
	//mechanism for the events handled by the current Indicator instance. Read more on EventTester documentation.
	NewEventTester() *EventTester
	//GetEventTester returns current GuiProvider EventTester instance. Read more on EventTester documentation.
	//
	//If testing==true, the EventTester is currently registering the events handled by the Indicator instance in test mode.
	GetEventTester() (eventTester *EventTester, testing bool)
}
```

As shown in listing 5.1, the GuiProvider takes charge of the following key points:

- starting and exiting the main routine (*event loop*) of the tray menu, actually managing its execution;
- managing the output flows from the application, i.e. (1) the tray icon (2) the tray title (small text near the icon);
- handling the test mode for the application workflow, e.g. with the NewEventTester method;
- allocating the right types of elements on the tray menu (separators and items). The interface methods related to this aspect were shaped around the characteristics of the **systray** package which was chosen as the keystone of the component implementation. More details about this can be found below.

The Systray package: features and limitations

The **systray** [33] Go package is the most complete cross-platform library providing a simple API to manage an icon and a menu in the tray bar. Thanks to the internal use of C snippets to interact with each operating system libraries, it greatly simplified the *Liqo Agent* development, acting as a single entry point to perform all graphical operations but the desktop notifications.

The most important advantage brought by systray is in a menu items management that does not require any direct interaction with widget toolkits for the correspondent display server, like GNU *GTK* for X11 windowing systems:

(1) It is possible to change all the aspects of each menu item (e.g. visibility, text content), as can be seen in listing 5.2. During real application executions, the *Item* interface leverages internally the `systray.MenuItem` type, exposed by the systray package.

Listing 5.2: app-indicator **Item** interface

```go
//Item is an interface representing the actual item that gets
    pushed (and displayed) in the stack of the tray menu.
type Item interface {
    //Check checks the Item.
    Check()
    //Uncheck unchecks the Item.
    Uncheck()
    //Checked returns whether the Item is checked.
    Checked() bool
    //Enable enables the Item, making it clickable.
    Enable()
    //Disable disables the Item, preventing it to be clickable.
    Disable()
    //Disabled returns whether the Item is disabled, i.e. not
    clickable.
    Disabled() bool
    //Show makes the Item visible in the menu.
    Show()
    //Hide hides the Item from the menu.
    Hide()
    //SetTitle sets the content of the Item that will be
    displayed in the menu.
    SetTitle(title string)
    //SetTooltip sets a tooltip for the Item displayed after a '
    mouse hover' event.
    //Currently, this is ineffective on Linux builds.
    SetTooltip(tooltip string)
}
```

(2) It is extremely simple to intercept "item clicked" events thanks to the exposed `systray.MenuItem.ClickedChan` field, a Go *chan* signalling each click on the correspondent menu item.

Listing 5.3: systray **MenuItem** type

```go
type MenuItem struct {
    // ClickedCh is the channel which will be notified when the
    menu item is clicked
    ClickedCh chan struct{}
    // contains filtered or unexported fields
}
```

There are, nevertheless, a few limitations put by the systray internal implementations, mainly involving the allocation of new *Item*s that have been addressed in the design of the **MenuNode** (see 5.2.2):

- the MenuNode type does not provide getters for all kinds of properties, e.g. its visibility (check listing 5.2), requiring an external storing of set values;
- as shown in listing 5.1, systray (and so the GuiProviderInterface) does not provide a *DELETE* operation for an *Item* created and registered in the tray menu. This is due to systray internal memory management. The tray menu is, in fact, handled as a LIFO data structure (i.e. a stack), where elements can be inserted into but no removed. Not only introduces this some difficulties in the orchestration of the menu views, but may represent a critical problem, since it may bring to memory leaks as the up-time goes on. This brought to the creation of type **LIST** MenuNodes whose allocated memory can be easily recycled (see 5.2.3).

5.2.2 Indicator

The Indicator is the *Liqo Agent* main component, acting as star point between the *GuiProvider* and the *AgentController*. All three components are implemented as *singleton*, since there has to be always only one instance of each of them. Given its role of model and entry point, it contains the references of the GuiProvider and AgentController that are initialized during Indicator creation by calling the `app−indicator.GetGuiProvider()` and `agent/client.GetGuiProvider()` functions.

Listing 5.4: app-indicator **Indicator** type

```
//Indicator singleton
var root *Indicator

//Indicator is a stateful data structure that controls the app
    indicator and its related menu. It can be obtained
//and initialized calling GetIndicator()
type Indicator struct {
    //root node of the menu hierarchy.
    menu *MenuNode
    //indicator label showed in the tray bar along the tray icon
    label string
    //indicator icon−id
    icon Icon
    //TITLE MenuNode used by the indicator to show the menu header
    menuTitleNode *MenuNode
    //title text currently in use
    menuTitleText string
    //STATUS MenuNode used to display status information.
    menuStatusNode *MenuNode
    //map that stores QUICK MenuNodes, associating them with their tag
    quickMap map[string]*MenuNode
    //reference to the node of the ACTION currently selected. If none, it defaults to the ROOT node
    activeNode *MenuNode
    //data struct containing indicator config
    config *config
    //guiProvider to interact with the graphic server
    gProvider GuiProviderInterface
    //data struct containing Liqo Status, used to control the menuStatusNode
    status StatusInterface
    //controller of all the application goroutines
    quitChan chan struct{}
    //if true, quitChan is closed and Indicator can gracefully exit
    quitClosed bool
    //data struct that controls Agent interaction with the cluster
    agentCtrl *client.AgentController
    //map of all the instantiated Listeners
    listeners map[client.NotifyChannel]*Listener
    //map of all the instantiated Timers
    timers map[string]*Timer
    //graphicResource is the map containing the mutex to protect access to the graphic resources handled by the Indicator
    //(e.g. tray icon, tray label and desktop notifications).
    graphicResource map[graphicResource]*sync.RWMutex
}
```

Below is the description of Indicator main tasks, based on its internal data structure fields.

Graphical resources orchestration

The Indicator manage the access to the three graphical resources (listing 5.5) of the *Liqo Agent*: (1) the tray icon (icon field) (2) the text box next to the icon (label field) (3) desktop notifications, by means of the Notify() function (listing 5.6).

In order to avoid race condition, each resource access is protected by a corresponding *mutex* stored in the graphicResource field.

Listing 5.5: app-indicator **graphicResource** type, enumerating the three resources handled by the Indicator.

```
//graphicResource defines a graphic interaction handled by the
    Indicator.
type graphicResource int

//graphicResource currently handled by the Indicator.
const (
    resourceIcon graphicResource = iota
    resourceLabel
    resourceDesktop
)
```

Listing 5.6: app-indicator **Notify** function to send desktop notifications and optionally change the tray icon.

```
//Notify manages Indicator notification logic. Depending on the
    current NotifyLevel of the Indicator,
//it changes the Indicator tray icon and displays a desktop banner,
    having title 'title' and 'message' as body.
//If present in client.EnvLiqoPath, also 'notifyIcon' is shown inside
     the banner.
//
//The "nil" values can be used for both 'notifyIcon' and '
    indicatorIcon':
//
//  NotifyIconNil : don't show a notification icon
//
//  IconLiqoNil : don't change current Indicator icon
func (i *Indicator) Notify(title string, message string, notifyIcon
    NotifyIcon, indicatorIcon Icon) {
    gr := i.graphicResource[resourceDesktop]
    gr.Lock()
    defer gr.Unlock()
    ...........
    ...........
        if !i.gProvider.Mocked() {
            /*The golang guidelines suggests error messages should
    not start with a capitalized letter.
            Therefore, since Notify sometimes receives an error as '
    message', the Capitalize() function
            overcomes this problem, correctly displaying the string
    to the user.*/
            _ = bip.Notify(title, stringUtils.Capitalize(message),
    filepath.Join(i.config.notifyIconPath, icoName))
        }
    default:
        return
    }
}
```

Tray menu orchestration

The Indicator contains the internal implementation of the try menu, following a hiearchical structure, whose root is represented by the `Indicator.menu` field.

All menu items (**MenuNode**s) are provided a tag to allow direct access in further searches. In particular, the first level ones - i.e. always visible in the menu frontpage - that are associated with specific callbacks are called *QUICKS* and its references are mapped with their tags inside the `Indicator.quickMap` (more on this on section 5.2.3). The addition of a *QUICK* MenuNode on the menu is possible by calling the `Indicator.AddQuick()` method.

Listing 5.7: app-indicator **Indicator.AddQuick()** method to add first level items to the tray menu, binding them with callbacks.

```go
//AddQuick adds a QUICK to the indicator menu. It is visible by
    default.
//
//  title : label displayed in the menu
//
//  tag : unique tag for the QUICK
//
//  callback : callback function to be executed at each 'clicked'
    event. If callback == nil, the function can be set
//  afterwards using (*MenuNode).Connect() .
func (i *Indicator) AddQuick(title string, tag string, callback func(
    args ...interface{}), args ...interface{}) *MenuNode {
    q := newMenuNode(NodeTypeQuick, false, nil)
    q.parent = q
    q.SetTitle(title)
    q.SetTag(tag)
    if callback != nil {
        q.Connect(false, callback, args...)
    }
    q.SetIsVisible(true)
    i.quickMap[tag] = q
    return q
}
```

Listeners and Timers

The Indicator registers *Listener*s and *Timer*s, objects that triggers the execution of an associated callback at a specific event or time. Each listener, in particular, has the task to listen on a certain `client.NotifyChannelGeneric` to signals coming from the **AgentController** and trigger the execution of its associated callback.

Listing 5.8: app-indicator **Listener** type.

```go
//Listener is an event listener that can react calling a specific callback.
type Listener struct {
    //Tag specifies the type of notification channel on which it listens to
    Tag client.NotifyChannel
    //StopChan lets control the Listener event loop
    StopChan chan struct{}
    //NotifyChan is the client.NotifyChannel on which it listens to
    NotifyChan chan client.NotifyDataGeneric
}

//Listen starts a Listener for a specific channel, executing callback when a notification arrives.
func (i *Indicator) Listen(tag client.NotifyChannel, callback func(data client.NotifyDataGeneric, args ...interface{}), args ...interface{}) {
    l := newListener(tag)
    i.listeners[tag] = l
    go func() {
        for {
            select {
            //exec handler
            case data, open := <-l.NotifyChan:
                /*While the Agent is OFF, the callback is not executed, in order not to update information
                on status and tray menu or trigger notifications.*/
                if open && i.Status().Running() == StatRunOn {
                    callback(data, args...)
                    //signal callback execution in test mode
                    if et, testing := GetGuiProvider().GetEventTester(); testing {
                        et.Done()
                    }
                }
                //closing application
            case <-i.quitChan:
                return
                //closing single listener. Channel controlled by Indicator
            case <-l.StopChan:
                delete(i.listeners, tag)
                return
            }
        }
    }()
}
```

All handlers are executed in their own *goroutine* and in any case terminated at the program exit, as a result of the implicit call of the `Indicator.Quit()` method.

5.2.3 MenuNode

As described in section 5.2.1, the `systray.MenuItem` type has some limitations, both in the access of this properties (no getters) and in the lack of a delete operation to dealloc unsed elements.

This moved to the creation of the `MenuNode` type which wraps a `MenuItem` object and provides an enhanced interface with additional features. It is the building block of the tray menu and has some sub types that differ their behavior according to their scope, as in listing 5.9.

Listing 5.9: app-indicator **MenuNode** type.

```go
/*NodeType defines the kind of a MenuNode, each one with specific features.

NodeType distinguishes different kinds of MenuNodes:

        ROOT:   root of the Menu Tree.

        QUICK:  simple shortcut to perform quick actions, e.g. navigation commands. It is always visible.

        ACTION: launch an application command. It can open command submenu (if present).

        OPTION: submenu choice.

        LIST:   placeholder item used to dynamically display application output.

        TITLE:  node with special text formatting used to display menu header.

        STATUS: non clickable node that displays status information.
*/
type NodeType int
// MenuNode is a stateful wrapper type that provides a better management of the
// Item type and additional features, such as submenus.
type MenuNode struct {
    // the Item actually allocated and displayed on the menu stack. It also contains
    //the internal ClickedChan channel that reacts to the 'item clicked' event.
    item Item
    // the type of the MenuNode
    nodeType NodeType
    // unique tag of the MenuNode that can be used as a key to get access to it, e.g. using (*Indicator)
    tag string
    // the kill switch to disconnect any event handler connected to the node.
    stopChan chan struct{}
    // flag that indicates whether a Disconnect() operation has been called on the MenuNode
    stopped bool
    // parent MenuNode in the menu tree hierarchy
    parent *MenuNode
    // nodeList stores the MenuNode children of type LIST. The node uses them to dynamically display to the user
    // the output of application functions. Use these kind of MenuNodes by calling UseListChild, FreeListChild
    // and FreeListChildren methods:
    //
    //      child1 := node.UseListChild(childTitle,childTag)
    //
    //      node.FreeListChild(childTag)
    //
    //      node.FreeListChildren()
    nodeList *nodeList
    //map that stores ACTION MenuNodes, associating them with their tag. This map is actually used only by the ROOT node.
    actionMap map[string]*MenuNode
    //map that stores OPTION MenuNodes, associating them with their tag. These nodes are used to create submenu choices
    optionMap map[string]*MenuNode
    //if isVisible==true, the MenuItem of the node is shown in the menu to the user
    isVisible bool
    //if isInvalid==true, the content of the LIST MenuNode is no more up to date and has to be refreshed by application
    //logic
    isInvalid bool
    //hasCheckbox determines whether the internal item has an embedded graphic checkbox that can be directly managed
    //by the internal Item. If not, (un)check operations are performed by using MenuNode own implementations.
    hasCheckbox bool
    //text prefix that is prepended to the MenuNode title when it is shown in the menu
    icon string
    //text content of the menu item. This redundancy of information is due to the fact Item does not provide getters
    //for the data.
    title string
    //protection for concurrent access to MenuNode attributes.
    sync.RWMutex
}
```

With respect to the MenuItem, the MenuNode offers three important improvements:

(1) An expanded interface of getters and setters to manage all Item properties.

(2) A simple mechanism to associate event handlers to a MenuNode. Thanks to the MenuNode.Connect() and MenuNode.Disconnect() methods, with just one function call it is possible to register to a MenuNode a callback that will be automatically executed when the correspondent tray menu element is clicked. The listening loop is performed on a different *goroutine* which is closed after a Disconnect() call or in any case at the program exit.

Listing 5.10: app-indicator MenuNode.**Connect()** method to associate a callback to the click of a tray menu item.

```
//Connect instantiates a listener for the 'clicked' event of the
    node.
//If once == true, the event handler is at most executed once.
func (n *MenuNode) Connect(once bool, callback func(args ...
    interface{}), args ...interface{}) {
    n.Lock()
    if n.stopped {
        n.stopChan = make(chan struct{})
        n.stopped = false
    }
    n.Unlock()
    var clickCh chan struct{}
    switch n.item.(type) {
    case *systray.MenuItem:
        clickCh = n.item.(*systray.MenuItem).ClickedCh
    case *mockItem:
        clickCh = n.item.(*mockItem).ClickedCh()
    default:
        clickCh = make(chan struct{}, 2)
    }
    go func() {
        for {
            select {
            case <-clickCh:
                callback(args...)
                if et, testing := GetGuiProvider().GetEventTester
    (); testing {
                    et.Done()
                }
                if once {
                    return
                }
            case <-n.stopChan:
                return
            case <-root.quitChan:
                return
            }
        }
    }()
}
```

(3) A reallocation mechanism to overcome the systray memory allocation problem (see section 5.2.1). Since it is not possible to free the memory of an unused MenuNode, the **LIST** MenuNode sub-type provides two methods, UseListChild() and FreeListChild, that, by emulating the *realloc* mechanism of several languages, acquire and release MenuNode elements from an internal FIFO queue, allocating new memory only when this structure is empty. This proves to be very useful in the creation of dynamic lists (e.g. the one of discovered peers), where a to be removed item is (a) cleared in all its internal attributes (b) hidden to the user and then the same procedure is applied to all its children (i.e. nested LIST MenuNodes).

5.2.4 Status

The **Status** is an Indicator sub-component that acts as an internal database for *Liqo Agent* and *Liqo* runtime information, protecting the access from concurrent operations (since the huge number of *goroutines* used).

By calling the Status.GoString() method, it also produces a textual digest of the current state which is then displayed in the tray menu inside the **STATUS** MenuNode.

Listing 5.11: app-indicator **Status** interface

```
//StatusInterface wraps the methods to manage the Indicator status.
type StatusInterface interface {
    //User returns the Liqo Name of the home cluster connected to the
     Agent.
    User() string
    //SetUser sets the Liqo Name of the home cluster connected to the
     Agent.
    SetUser(user string)
    //Running returns the running status of Liqo.
    Running() StatRun
    //SetRunning changes the running status of Liqo. Transition to
   StatRunOff
    //implies the end of all active peerings.
    SetRunning(running StatRun)
    //Mode returns the current working mode of Liqo.
    Mode() StatMode
    //SetMode sets the working mode for Liqo.
    //If the operation is not allowed for current configuration, it returns an error.
    SetMode(mode StatMode) error
    /*IsTetheredCompliant checks if the TETHERED mode is eligible
    accordingly to current status. The result can be used to display information.

    This method is not to be intended as a preliminary test
    for an actual mode change. In this case you must use SetMode() which exploits a
    "Compare&Change" protection.
    */
    IsTetheredCompliant() bool
    //Peerings returns the number of active peerings of type PeeringType.
    Peerings(peering PeeringType) int
    //ActivePeerings returns the amount of active peerings.
    ActivePeerings() int
    //Peers returns the number of Liqo peers discovered by the home cluster and currently available.
    Peers() int
    //Peer returns data related to a cluster if it is currently discovered by the home cluster.
    Peer(clusterId string) (peer *PeerInfo, present bool)
    //AddOrUpdatePeer updates the internal information on an existing or newly discovered peer.
    //In case no info about the peer's common name is provided, a placeholder "unknown identifier"
    //is assigned to allow the user to visually distinguish between different unknown peers.
    //When the number of unknown peers is decremented to 0, the identifier number is reset.
    AddOrUpdatePeer(data *client.NotifyDataForeignCluster) *PeerInfo
    //RemovePeer removes a peer from the currently registered ones.
    RemovePeer(data *client.NotifyDataForeignCluster) *PeerInfo
    //SetClusterName sets the common name of the cluster LiqoAgent is currently connected to.
    SetClusterName(clusterName string)
    //GoString produces a textual digest on the main status data managed by
    //a Status instance.
    GoString() string
}
```

5.3 Client library

The **client** library allows to manage all the interactions between the main routine of the *Liqo Agent* on the local device and the Kubernetes clusters involved.

The Indicator leverages the library for two main tasks:

- retrieve the correct *kubeconfig* file from the file system by calling the acquireKubeconfig function, potentially updating its config file accordingly;
- interact with Liqo clusters by means of the **AgentController**.

5.3.1 AgentController

The key component is the AgentController, which runs as a singleton instance bound to the *Home* Liqo cluster which the application is connected to.

Listing 5.12: client **AgentController** type

```
//AgentController is the data structure that manages Tray Agent interaction with the cluster.
type AgentController struct {
    //notifyChannels is a set of channels used by the cache logic to notify a watched event.
    notifyChannels map[NotifyChannel]chan NotifyDataGeneric
    //kubeClient is a standard kubernetes client.
    kubeClient kubernetes.Interface
    //agentConf contains Liqo Agent configuration parameters acquired from the cluster.
    agentConf *agentConfiguration
    //crdManager manages CRD operations.
    *crdManager
    //valid specifies whether the provided kubeconfig actually describes a correct configuration.
    valid bool
    //connected specifies whether all AgentController components are correctly up and running.
    connected bool
    mocked    bool
}
```

When the Indicator is initialized via the app−indicator.GetIndicator function, it creates and stores internally a reference to the AgentController by calling client .GetAgentController. If the connection to the designated cluster is successful, the Agent creates all the required resources and sub components to orchestrate a bi-directional data and control flow between application logic and the Liqo cluster.

CRDController

Almost all *Liqo Agent* operations towards a Kubernetes cluster involves the **CRD**s (Custom Resource Definition) of the Liqo framework. The AgentController registers all the required CRDs into its internal structures as CustomResource type variables and each of them is then managed by the crdManager (listing 5.13) by means of the correspondent **CRDController**.

Listing 5.13: client **crdManager** and **CustomResource** types

```go
//CustomResource defines the CRD managed by Liqo Agent.
type CustomResource string

const (
    //CRClusterConfig is the resource id for the ClusterConfig CRD.
    CRClusterConfig CustomResource = "clusterconfigs"
    //CRAdvertisement is the resource id for the Advertisement CRD.
    CRAdvertisement CustomResource = "advertisements"
    //CRForeignCluster is the resource id for the ForeignCluster CRD.
    CRForeignCluster CustomResource = "foreignclusters"
)

//crdManager stores the resources necessary to manage the CRDs.
type crdManager struct {
    //clientMap contains the Controllers for the CRDs managed by the
   Agent.
    clientMap map[CustomResource]*CRDController
}
```

Each CRDController (listing 5.14) is an enhanced custom client for a specific CustomResource, internally based on a CRDClient, the component provided by the Liqo framework which actually establishes the connections. Its main tasks are:

- to execute CRUD operations on *Custom Resource*s of CRDController.resource type, as consequence of user choices;
- to listen to specific events (*watch*) regarding *Custom Resource*s of CRDController .resource type and send back data to the Indicator using the NotifyChannel (more on this in the following section). The addFunc, updateFunc and deleteFunc contain the callbacks for the event handlers. It is worth noticing that, in order to monitor events and retrieve data, the CRDClient internally leverages a **Kubernetes cache to significantly reduce the interactions with the API server of the cluster**.

Listing 5.14: client **CRDController** type.

```go
//CRDController handles the Agent interaction with the cluster for a
   specific CRD.
```

```
type CRDController struct {
    //CRDClient to perform CRUD operations on the CRD.
    *crdClient.CRDClient
    //resource is the CRD literal identifier.
    resource string
    //running specifies whether the CRD cache is running.
    running bool
    //addFunc is the handler for the 'resource added' event.
    addFunc func(obj interface{})
    //updateFunc is the handler for the 'resource updated' event.
    updateFunc func(oldObj interface{}, newObj interface{})
    //deleteFunc is the handler for the 'resource deleted' event.
    deleteFunc func(obj interface{})
}
```

NotifyChannel

When a CRDController detects the realization of one of the watched events, it sends data back to an Indicator Listener through a specific Go chan, identified by its NotifyChannel type (listing 5.15).

The channels are designed to accept all possible kinds of data, thanks to the NotifyDataGeneric interface (listing 5.16) and its up to each CRDController logic to define and use custom sub types that will be correctly interpreted by the correspondent Listener.

Listing 5.15: client **NotifyChannel** type.

```
//NotifyChannel identifies a notification channel for a specific
    event.
type NotifyChannel int

//NotifyChannel identifiers.
const (
    //Notification channel id for an update of an available peer.
    ChanPeerAddedOrUpdated NotifyChannel = iota
    //Notification channel id for the removal of an available peer.
    ChanPeerDeleted
    //ChanClusterName os the NotifyChannel used to transmit the
    current ClusterName of the Liqo cluster the Agent is
    //connected to.
    ChanClusterName
)
```

Listing 5.16: client **NotifyDataGeneric** interface.

```
//NotifyChan is the wrapper type for generic data sent over a
    NotifyChannel. After receiving such element from a
```

```
//chan, it is then possible to try its conversion into a specific
    type.
type NotifyDataGeneric interface{}
```

5.4 Logic library

Logic package contains the code to actual orchestrate *Liqo Agent* execution. It contains the callbacks actually implementing all the looping goroutines:

- callbacks connected to tray menu MenuNode, like the QUICKs (first level) elements;
- event handlers for Indicator Listeners;
- the orchestration of the main (OnReady) and finalizer (OnExit) routines directly handled by the systray.Run function. OnReady, in particular is responsible for setting up the environment for the *Liqo Agent*, by creating the tray menu MenuNodes and starting all the goroutines.

Listing 5.17: logic **OnReady()** and **OnExit** routines: they define the operatoins performed before and after the main loop.

```
//OnReady is the routine orchestrating Liqo Agent execution.
func OnReady() {
    // Indicator configuration
    i := app.GetIndicator()
    i.RefreshStatus()
    startListenerClusterConfig(i)
    startListenerPeersList(i)
    startQuickOnOff(i)
    startQuickChangeMode(i)
    startQuickDashboard(i)
    startQuickShowPeers(i)
    i.AddSeparator()
    startQuickSetNotifications(i)
    startQuickLiqoWebsite(i)
    startQuickQuit(i)
    //try to start Liqo and main ACTION
    quickTurnOnOff(i)
}

//OnExit is the routine containing clean-up operations to be
    performed at Liqo Agent exit.
func OnExit() {
    app.GetIndicator().Disconnect()
}
```

5.5 Icon library

The Icon library contains the available Indicator iconset for the tray icon. The systray package offers a great opportunity to enhance the performance of the application. The icons, in fact, are not retrieved from the file system in their original image format (e.g. *png*). They are, instead, exported into Go variables as binary data (i.e. byte[]) by means of an external tool[1]. This way they are directly loaded into memory, thus bypassing expensive accesses (in terms of time) to the file system.

5.6 Installation on Linux distributions

Although the *Liqo Agent* codebase is already compatible with all major platforms, at the time this document is written the complete application is available only for GNU Linux distributions, released as a *desktop application* according the **freedesktop.org** standard, which provides compatibility guidelines for *X*-based (*X11*) and *Wayland* desktop environments, such as GNOME, KDE and Xfce.

This solution, despite being simple, provides no packaging system and requires the application resources (e.g. binaries, icons, manifests) to be moved to their appropriate "*well known*" folders. The *Liqo Agent* repository [31] provides a one line installer (via bash script) to simplify this process.

[1]Since Go 1.16, the built-in *embed* package allows to perform this conversion automatically by using code directives.

Chapter 6

Experimental evaluation

Liqo Agent has been developed as a desktop application which should comply with significant constraints:

- **limited CPU and memory consumption**, given that the application:
  - is designed to run even on heavy resource constrained devices, since it aims to provide an easy connections to other clusters;
  - is in the form of a tray agent, running in background most of the time and it should not consume resources destined to user tasks;
- **limited time overhead of the application with respect to Liqo**: the *Agent* provides an easy user interface to interact with the Liqo framework. Its use remains effective as long as it manages to keep a low delay between users and I/O operations directly performed on a terminal.

6.1 Test Environment and Procedures

In order to assess its performances, the application has been tested over a series of runs for:

(1) CPU percentage usage;

(2) memory usage, focusing on the *RSS* (Resident Set Size), i.e. the amount of memory actually held in main memory in a precise moment;

(3) duration of critical routines in the application workflow.

Tests have been performed by running a tailored version of *Liqo Agent* on a Linux Ubuntu 20.04 LTS distribution, by means of builtin Go packages (e.g. *runtime/metrics* or *time*) and third party libraries.

The analysis focused on two main scopes:

(1) the start-up phase, i.e. the initialization phase when the applications boots up, allocating resources and establishing connections towards the home Kubernetes cluster;

(2) the steady state behavior, when the application waits for user inputs or Kubernetes events; in this case the evaluation related to the progressive discovery of Liqo peers (foreign clusters) and the consequential response by the *Liqo Agent* in order to make the new resource available to the user, e.g. by updating the tray menu.

6.2 Liqo Agent Initialization

The first test related on the start-up phase of the application, when all main resources of the application are allocated. At the end of this transient period, the application goes stand by, waiting for user input and Kubernetes events.

This task is performed by the **OnReady** function (listing 6.1), which is mainly composed of the **GetIndicator** and **GetAgentController**, which respectively initialize the tray menu and the connections towards the Kubernetes cluster.

Listing 6.1: OnReady routine

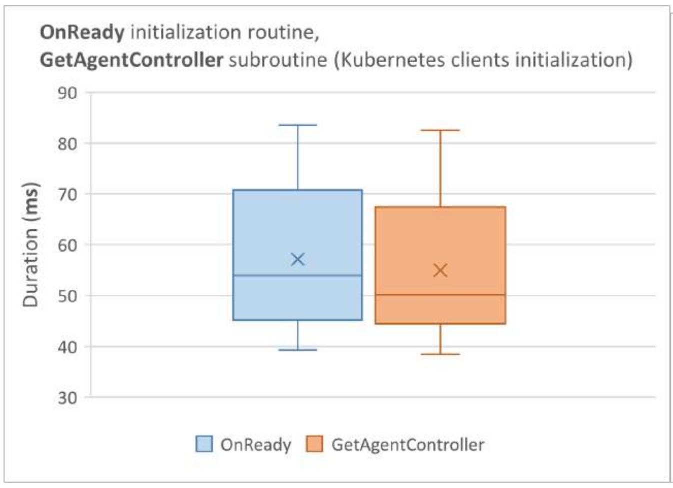
```
//OnReady is the routine orchestrating Liqo Agent execution.
func OnReady() {
    // Indicator configuration
    i := app.GetIndicator()
    i.RefreshStatus()
    startListenerClusterConfig(i)
    startListenerPeersList(i)
    startQuickOnOff(i)
    startQuickChangeMode(i)
    startQuickDashboard(i)
    startQuickShowPeers(i)
    i.AddSeparator()
    startQuickSetNotifications(i)
    startQuickLiqoWebsite(i)
    startQuickQuit(i)
    //try to start Liqo and main ACTION
    quickTurnOnOff(i)
}
```

Routine duration

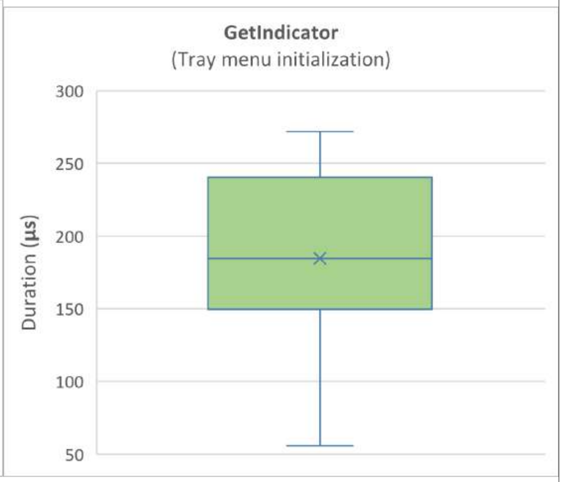
The duration of the **OnReady** routine and its two main subroutines have been measured over a series of iterations. Here are the results:

Duration of initialization routines (ms)			
Iteration	OnReady	GetIndicator	GetAgentController
1	47,43	0,17	47,05
2	85,53	0,272	75,28
3	42,39	0,266	41,18
4	66,75	0,173	64,87
5	55,08	0,145	54,66
6	82,84	0,10	82,51
7	55,39	0,163	53,35
8	39,25	0,221	38,43
9	46,06	0,247	45,50
10	52,77	0,207	46,93
AVG	57,15	0,196	54,98
S.D.	15,74	0,056	14,73

Table 6.1: Duration of **OnReady** initialization routine and its main components.



(a) OnReady, GetAgentController



(b) GetIndicator

Figure 6.1: Duration boxplots for the OnReady routine and its two main components.

Table 6.1 clearly shows how OnReady duration is mostly influenced by the GetAgentController sub routine (called by GetIndicator), showing on one hand the lower impact of the Model-View component - since all graphic operations are performed inside the GetIndicator routine - and on the other hand where future optimization efforts should be spent.

Since it is a one time operation, the overall duration of the initialization step seems to be pretty acceptable, with an average of ≈ *57 ms*.

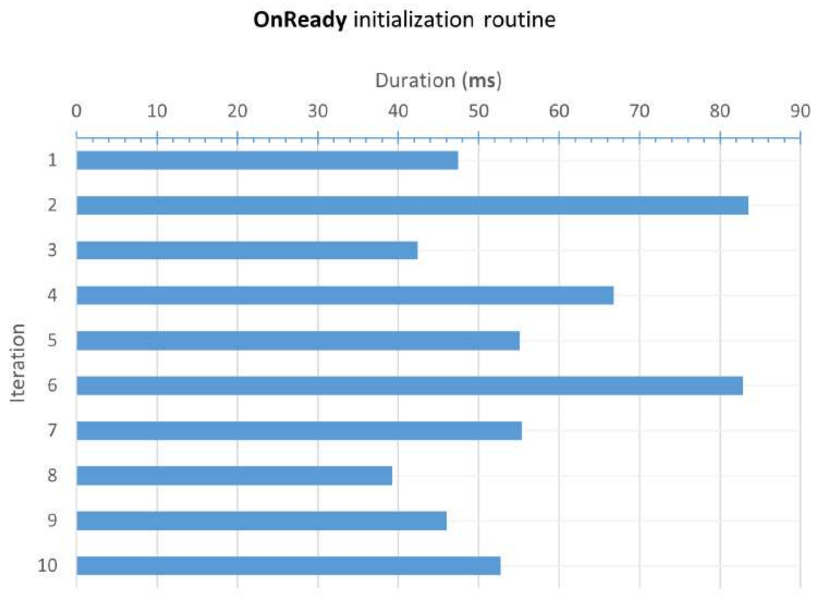


Figure 6.2: Average duration of OnReady initialization routine.

Resources usage

The initialization phase, carried on by the OnReady routine, involves three main actions, resulting in CPU consumption and new memory allocation:

(1) The initialization of the Indicator component, which builds the tray menu up, along with the basic structure of immutable entries. This operation mostly involves memory allocation for the Indicator's MenuNodes (see 5.2.2).

(2) The initialization of the AgentController component which performs the connections towards the Kubernetes cluster and loads all the CRDControllers with the associated watch routines to monitor Kubernetes events (see 5.3.1).

(3) The instantiation of the Indicator's Listeners and Timers (see 5.2.2) and the initialization of both graphical and internal sides of the tray menu choices, according to the current status.

The test sampled the CPU and memory usage during this routine over several iterations in order to analyse the impact of the application during the process and at the end of the transient period when it reaches its steady state. The results can be seen in table 6.2 and figures 6.3 and 6.4.

CPU and memory usage during OnReady initialization routine							
Iteration	Measure	Sample					
		1	2	3	4	5	6
1	CPU usage (%)	1,264	2,353	1	0	0,498	0
	Memory usage (**MB**)	71,121	73,089	73,218	73,471	73,624	73,908
2	CPU usage (%)	1,638	1,2	0,49	0	0	0
	Memory usage (**MB**)	72,521	73,411	73,299	73,509	73,693	73,948
3	CPU usage (%)	2,917	2,142	0,995	0	0,877	0,407
	Memory usage (**MB**)	70,178	73,761	74,133	74,54	74,54	74,54
4	CPU usage (%)	0,32	1	1,485	0	0,498	0
	Memory usage (**MB**)	70,02	73,202	73,361	73,315	73,479	73,684
5	CPU usage (%)	2,322	2,454	0	0	0,498	0
	Memory usage (**MB**)	70,35	74,158	73,146	73,203	74,312	74,315
6	CPU usage (%)	1,714	1,803	0	0,498	0	0
	Memory usage (**MB**)	71,17	73,277	73,30	73,748	74,116	74,27
7	CPU usage (%)	2,357	1,78	0	1	0	0
	Memory usage (**MB**)	72,534	73,221	73,625	73,426	73,727	73,842
8	CPU usage (%)	1,217	2,5	0,995	0,5	1,424	0
	Memory usage (**MB**)	71,014	73,33	73,412	73,535	73,834	73,802
9	CPU usage (%)	2,778	3,124	1,493	0	0	0,5
	Memory usage (**MB**)	71,74	71,93	73,643	73,43	73,76	73,69
10	CPU usage (%)	2,89	2,1	0,98	0,5	1,594	0
	Memory usage (**MB**)	72,30	73,358	73,217	73,36	73,38	73,38
AVG	CPU usage (%)	1,942	2,046	0,744	0,250	0,539	0.091
	Memory usage (**MB**)	71,30	73,30	73,40	73,60	73,80	73,90
S.D.	CPU usage (%)	0,858	0,631	0,585	0,353	0,192	0,594
	Memory usage (**MB**)	1,0	0,6	0,3	0,4	0,3	0,4

Table 6.2: CPU and Memory usage during OnReady initialization routine.

Figure 6.5 well indicates the consumption trends for these two resources:

- After an initial peak of $\approx 2\%$, mostly due to the GetAgentController execution, the **CPU usage** drops down to zero. This is consistent with the *Liqo Agent* workflow which blocks after the OnReady execution, waiting for user inputs or Kubernetes events.
- The OnReady routine allocates approximately $\approx 3MB\%$, reaching an average

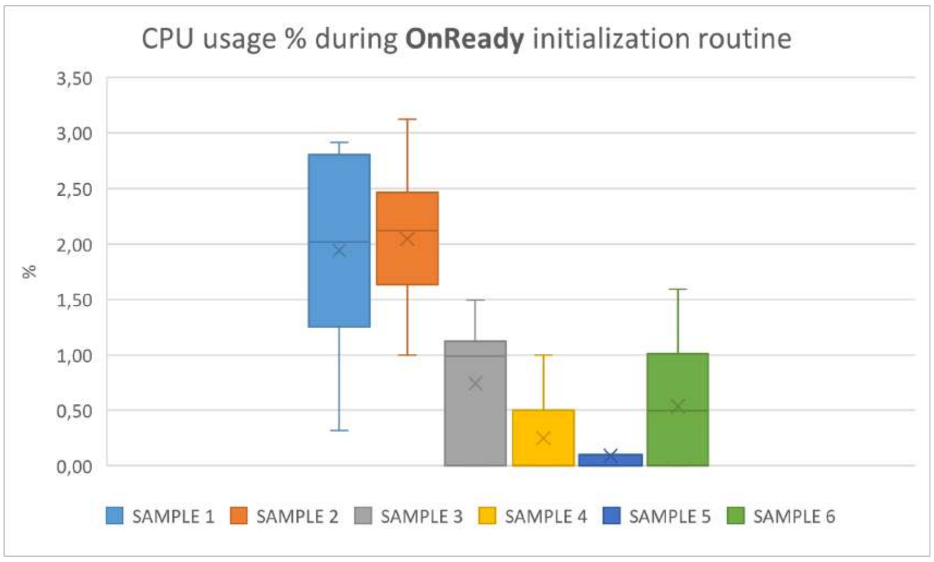


Figure 6.3: Boxplots of all CPU % usage samples taken during **OnReady** execution.

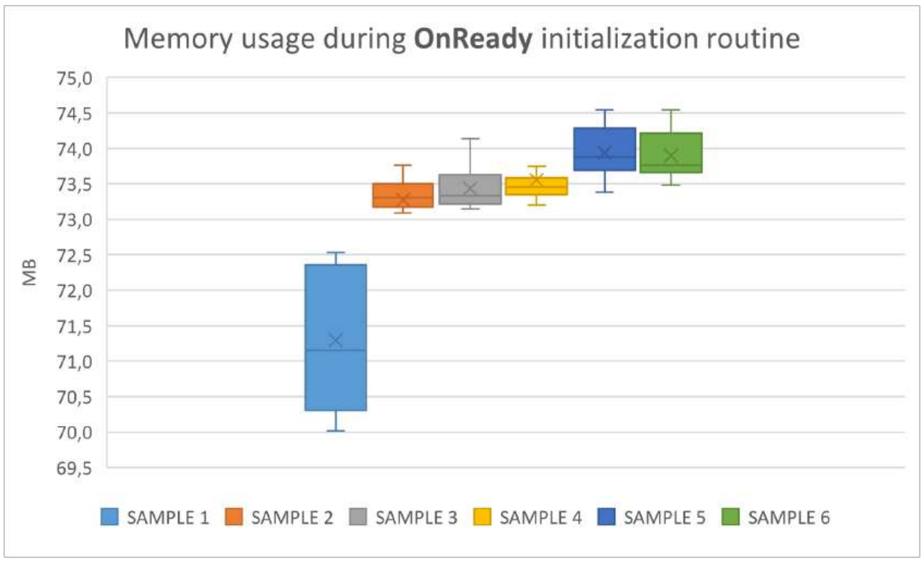


Figure 6.4: Boxplots of all Memory (Resident Set Size) usage samples taken during **OnReady** execution.

RSS of $74MB$, which is acceptable for a tray agent application.

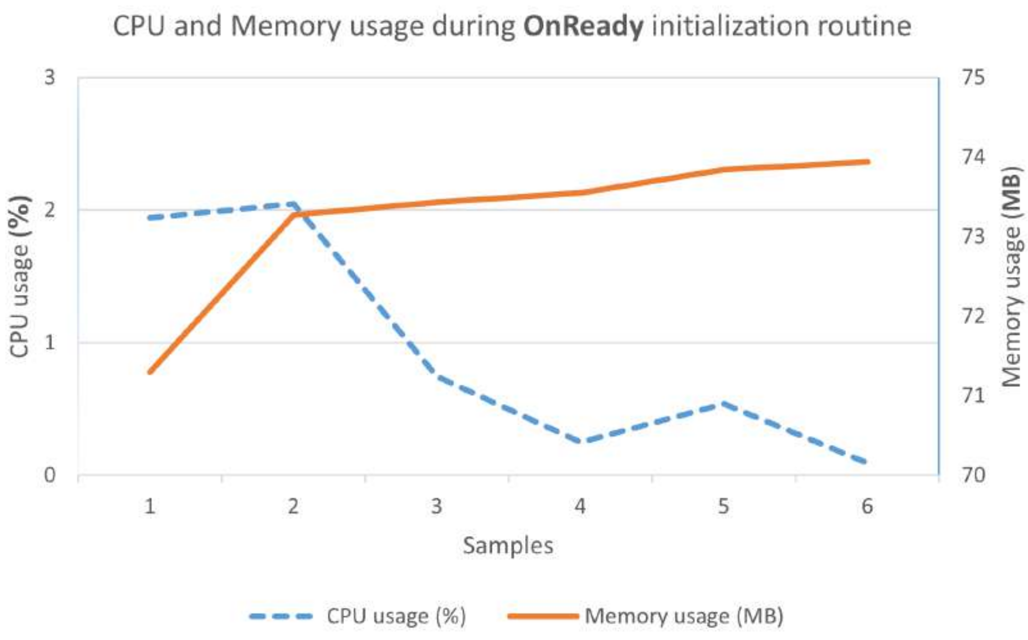


Figure 6.5: Trend lines of CPU usage % and memory (Resident Set Size) usage during **OnReady** execution.

6.3 Peers Discovery

After reaching the steady state, *Liqo Agent* waits for events coming either from user inputs or the Kubernetes cluster, by means of several goroutines.

The second phase of the test aimed to analyse the application response following a series of events regarding the discovery by the Liqo framework of new *Liqo peers*, i.e. new Kubernetes clusters that can be later joined. The focus was put on this kind of events since it is the most critical in terms of time and memory allocation, with respect, for example, to the establishment (or acceptance) of a new peering.

Table 6.3 and figures 6.6, 6.7 and 6.8 show the results.

Duration

Figure 6.9 shows the average duration of the response, which is essentially made up of the CRDController, watching for ForeignClusters events, and the correspondent Indicator Listener.

Since values reside in a range of around $348 \div 350\mu$s, results seem promising, given that the average duration of a new peer discovery by the Liqo framework is $600\mu s$, resulting in a time overhead by *Liqo Agent* of $\approx$ 0,058%.

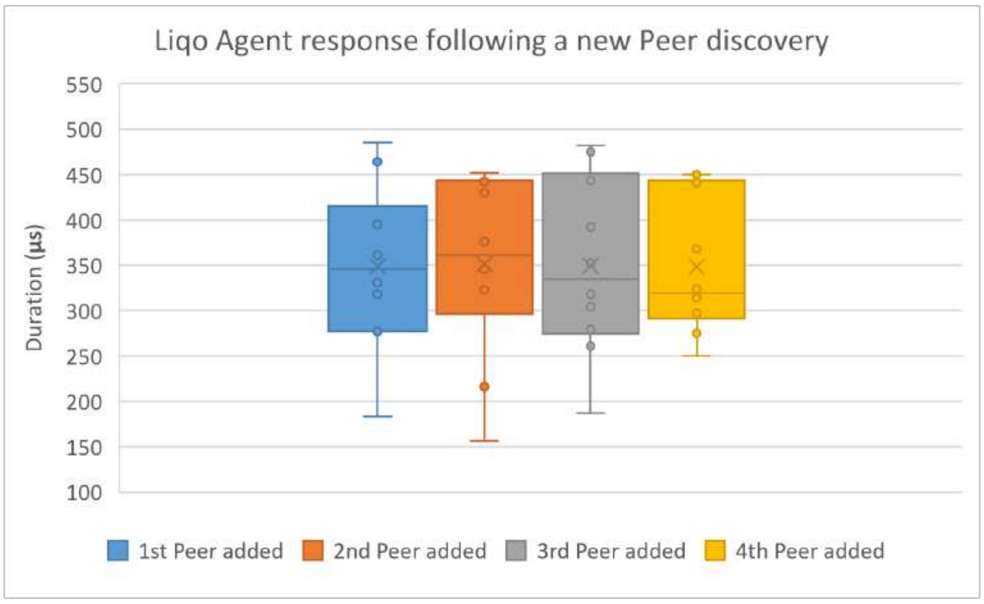


Figure 6.6: Boxplots of *Liqo Agent* response duration for each newly discovered peer.

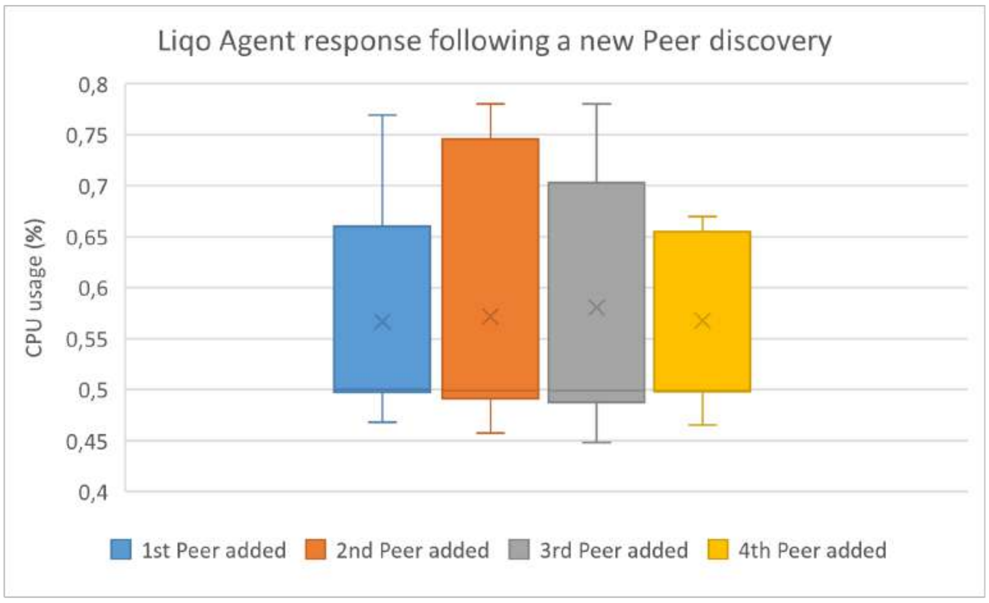


Figure 6.7: Boxplots of *Liqo Agent* average CPU % consumption during the response for each newly discovered peer.

CPU usage

Figure 6.10 shows the average CPU percentage usage during the *Liqo Agent* response. The consumption stands stably under the 1%, around ≈ 0,57%. This results in

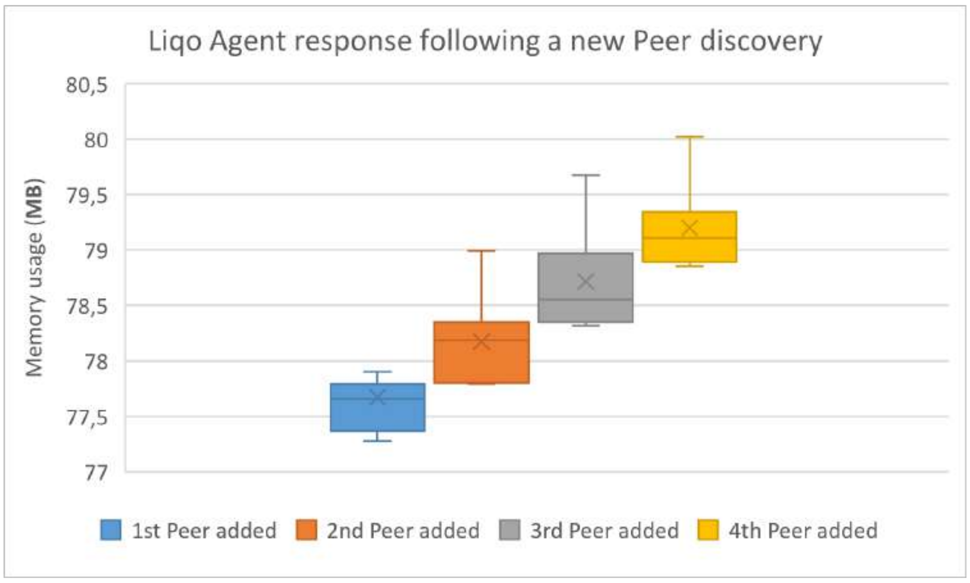


Figure 6.8: Boxplots of *Liqo Agent* memory RSS (Resident Set Size) following the response for each newly discovered peer.

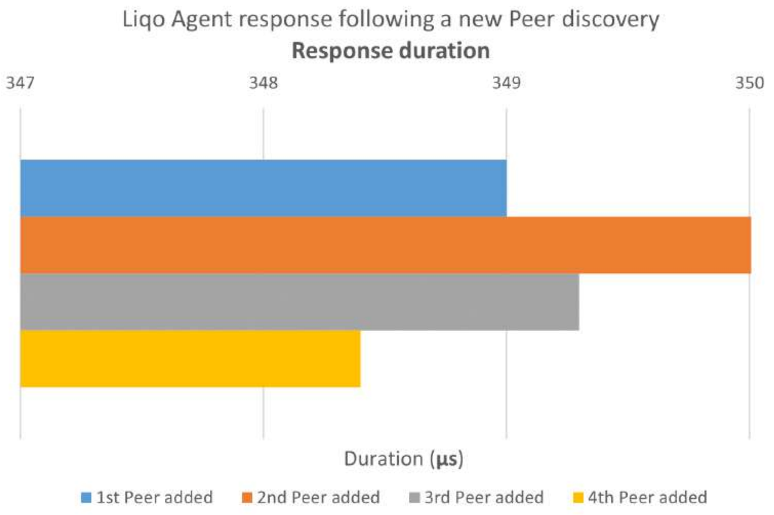


Figure 6.9: Average duration of *Liqo Agent* response following each discovery of a new peer.

two positive aspects, both in the stability and in the value itself.

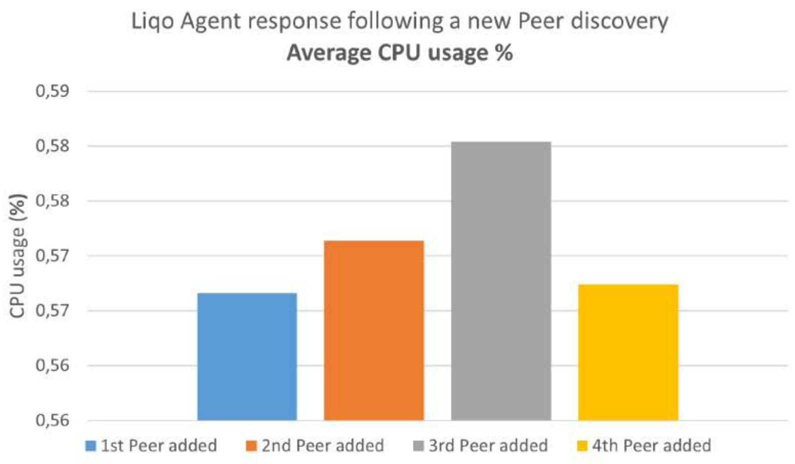


Figure 6.10: Average CPU % consumption during the *Liqo Agent* response following each newly discovered peer event.

Memory usage

Figure 6.11 shows the average memory allocation (for the RSS) after each *Liqo Agent* response to a newly discovered peer. The chart indicates the linear growth, mostly due to the addition of a new entry in the tray menu, which is itself made up of several MenuNodes. The trend seem pretty stable, with an average increment of $\approx 50MB$ for each peer. This low value, combined with a common use case of a reduced set of available peers, guarantees a good stability and a sustainable use of the application even on resource constrained devices.

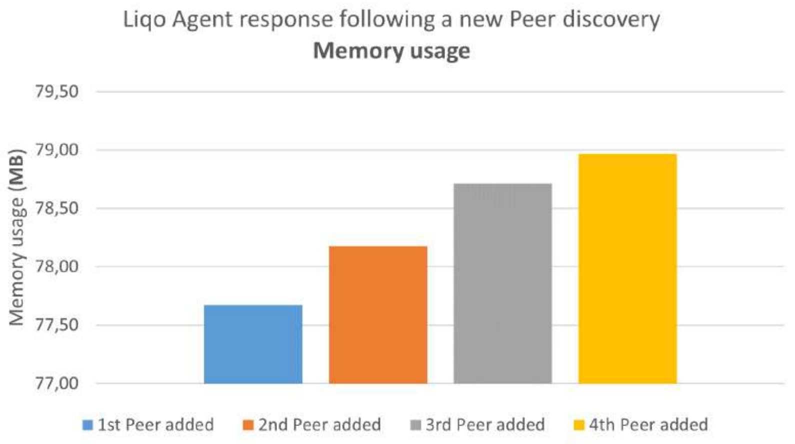


Figure 6.11: Memory RSS (Resident Set Size) trend after the *Liqo Agent* response for each newly discovered peer.

CPU & Memory usage and Duration of Liqo Agent response following a new Peer discovery					
Iteration	Measure	available Peers			
		1	2	3	4
1	Duration (μ)	485	346	261	324
	CPU usage (%)	0,495	0,498	0,469	0,5
	Memory usage (**MB**)	78,445	78,991	79,27	79,87
2	Duration (μ)	395	442	187	450
	CPU usage (%)	0,468	0,5	0,448	0,498
	Memory usage (**MB**)	77,642	78,264	78,665	79,127
3	Duration (μ)	464	376	352	315
	CPU usage (%)	0,5	0,78	0,772	0,67
	Memory usage (**MB**)	77,665	77,79	78,318	78,89
4	Duration (μ)	183	448	318	297
	CPU usage (%)	0,769	0,498	0,78	0,498
	Memory usage (**MB**)	77,275	77,805	78,325	78,853
5	Duration (μ)	277	323	392	368
	CPU usage (%)	0,498	0,469	0,5	0,65
	Memory usage (**MB**)	77,754	78,277	78,81	79,101
6	Duration (μ)	361	323	443	441
	CPU usage (%)	0,73	0,5	0,68	0,897
	Memory usage (**MB**)	77,904	78,159	78,44	78,89
7	Duration (μ)	318	216	475	450
	CPU usage (%)	0,569	0,457	0,498	0,5
	Memory usage (**MB**)	77,348	77,874	78,416	79,11
8	Duration (μ)	331	156	279	250
	CPU usage (%)	0,5	0,78	0,498	0,465
	Memory usage (**MB**)	77,597	78,218	78,871	79,168
9	Duration (μ)	399	452	482	314
	CPU usage (%)	0,637	0,734	0,665	0,5
	Memory usage (**MB**)	77,707	78,562	79,673	80,02
10	Duration (μ)	277	430	304	275
	CPU usage (%)	0,5	0,498	0,494	0,498
	Memory usage (**MB**)	77,375	77,797	78,356	78,97
AVG	Duration (μ)	349,00	351,20	349,30	348,40
	CPU usage (%)	0,57	0,57	0,58	0,57
	Memory usage (**MB**)	77,67	78,17	78,71	79,20
S.D.	Duration (μ)	91,71	101,43	97,83	74,69
	CPU usage (%)	0,11	0,13	0,13	0,14
	Memory usage (**MB**)	0,34	0,39	0,46	0,41

Table 6.3: CPU % and memory usage and duration of Liqo Agent response following each event of new peer discovery notified by the AgentController.

Chapter 7

Conclusion and future work

The work led to the realization of a lightweight desktop application with a simple interface that aims to bring to common PC users some of the advantages of enterprise world orchestrators in terms of power consumption optimization, scalability, elasticity.

Not only could this evolution have an economic impact - especially on final users and small organizations - but also on privacy, giving users an additional choice of where their applications can run.

The long term transition to the sharing economy has been changing the ICT sector for years, both on demand and supply side, with new development patterns, platforms and architecture. Projects like the application developed during this thesis could partially help to reconcile the dilemma of the black clouds of constant loss of control with the silver linings of having always more powerful applications available everywhere.

In the current release, the work has been focused on the development of the application architecture and its main features, like

- handling and monitoring main aspects of the peering/unpeering process, both outgoing and incoming, without any need to access the terminal
- a mixed system of notifications, icons and status bar so that the user is always aware of important events
- providing a quick access to the LiqoDash dashboard for advanced operations
- notification settings to let the user suppress desktop notifications.

The conducted tests showed that the application has no or very reduced CPU usage during the entire life cycle and, on the other hand, an incremental memory consumption of just $\approx 0{,}5MB$ for each known peer. These results give *Liqo Agent* a good scalability and make it suitable for executions on devices with reduced resources.

Future work

Future works will focus both on features improvement and in the release mechanism. Here are the main goals:

- to repackage the Linux version into the **AppImage** format which is extremely suitable for distributing portable software on all Linux distros. Not only allows it to wrap up all program resources in one file, but it needs no installation process.
- to release the application also on Windows and MacOs environments.
- to expand the list of available settings to allow some useful customizations, e.g. the autojoin option, Liqo cluster name customization and manual discovery of foreign clusters.

Bibliography

[1] *Moore's Law.* URL: http://www.mooreslaw.org/ (cit. on p. 1).

[2] Truong Duy, Yukinori Sato, and Y. Inoguchi. «Performance evaluation of a Green Scheduling Algorithm for energy savings in Cloud computing». In: May 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470908 (cit. on p. 2).

[3] Jim Wright. *Buy servers only if you really need them.* July 2004. URL: https://www.computerweekly.com/opinion/Buy-servers-only-if-you-really-need-them/ (cit. on p. 2).

[4] Luiz André Barroso and Urs Hölzle. «The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines». In: *Synthesis Lectures on Computer Architecture* 4.1 (2009), pp. 1–108. DOI: 10.2200/S00193ED1V01Y200905CAC006. eprint: https://doi.org/10.2200/S00193ED1V01Y200905CAC006. URL: https://doi.org/10.2200/S00193ED1V01Y200905CAC006 (cit. on p. 2).

[5] Michael Ogbole, Engr Ogbole, and Ayomide Olagesin. «Cloud Systems and Applications : A Review». In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* (Feb. 2021), pp. 142–149. DOI: 10.32628/CSEIT217131 (cit. on p. 2).

[6] Randy Bias. *Elasticity is NOT #Cloud Computing … Just Ask Google.* Nov. 2010. URL: http://cloudscaling.com/blog/cloud-computing/elasticity-is-not-cloud-computing-just-ask-google/ (cit. on pp. 3, 30).

[7] *Kubernetes project.* URL: https://kubernetes.io/ (cit. on p. 3).

[8] Andrew Joint, Edwin Baker, and Edward Eccles. «Hey, you, get off of that cloud?» In: *Computer Law & Security Review* 25 (Dec. 2009), pp. 270–274. DOI: 10.1016/j.clsr.2009.03.001 (cit. on p. 4).

[9] Kashim Kyari Mohammed, Aisha Abdulrahman Abba, and Aisha Muhammad. «Cloud Security». PhD thesis. Jan. 2021. DOI: 10.13140/RG.2.2.13876.58242 (cit. on p. 4).

[10] *K3s project.* URL: https://k3s.io/ (cit. on pp. 5, 20).

[11] Mattia Lavacca. «Scheduling Jobs on Federation of Kubernetes Clusters». master. Politecnico di Torino, 2020. Chap. 2 (cit. on p. 6).

[12] *Kubernetes official documentation.* URL: `https://kubernetes.io/docs/home/` (cit. on pp. 6, 13, 15–18).

[13] *Virtual-Kubelet project.* URL: `https://github.com/virtual-kubelet/virtual-kubelet` (cit. on pp. 6, 17, 18, 22, 26).

[14] *Kubebuilder git repository.* URL: `https://github.com/kubernetes-sigs/kubebuilder` (cit. on pp. 6, 18–20).

[15] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys).* Bordeaux, France, 2015 (cit. on p. 6).

[16] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. «Omega: flexible, scalable schedulers for large compute clusters». In: *SIGOPS European Conference on Computer Systems (EuroSys).* Prague, Czech Republic, 2013, pp. 351–364. URL: `http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf` (cit. on p. 6).

[17] Ferenc Hámori. *The History of Kubernetes on a Timeline.* June 2018. URL: `https://blog.risingstack.com/the-history-of-kubernetes/` (cit. on p. 7).

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

[19] Kalyan Ramanathan. *5 business reasons why every CIO should consider Kubernetes.* Oct. 2019. URL: `https://www.sumologic.com/blog/why-use-kubernetes/` (cit. on p. 7).

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

[21] Diego Ongaro and John Ousterhout. «In search of an understandable consensus algorithm». In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14).* 2014, pp. 305–319 (cit. on p. 11).

[22] *Kubernetes API official documentation.* URL: `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/` (cit. on p. 13).

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

[24] *Liqo project repository.* URL: `https://github.com/LiqoTech/liqo` (cit. on p. 20).

[25] *Liqo project website.* URL: `https://liqo.io/` (cit. on p. 20).

[26] *Wireguard website.* URL: `https://www.wireguard.com/` (cit. on p. 22).

[27] *CNI project.* URL: `https://www.cni.dev/` (cit. on p. 22).

[28] *LiqoDash project.* URL: `https://github.com/liqotech/dashboard` (cit. on pp. 29, 35).

[29] Marc Hassenzahl and Noam Tractinsky. «User experience - a research agenda». In: *Behaviour & Information Technology* 25.2 (2006), pp. 91–97. DOI: `10.1080/01449290500330331`. eprint: `https://doi.org/10.1080/01449290500330331`. URL: `https://doi.org/10.1080/01449290500330331` (cit. on p. 29).

[30] *Liqotech organization on Github.* URL: `https://github.com/liqotech` (cit. on p. 35).

[31] *LiqoAgent project.* URL: `https://github.com/liqotech/liqo-agent` (cit. on pp. 38, 57).

[32] *Awesome-Go repository.* URL: `https://github.com/avelino/awesome-go` (cit. on p. 39).

[33] *Systray repository.* URL: `https://github.com/getlantern/systray` (cit. on p. 41).