Politecnico di Torino

POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master's Degree Thesis

Analysis, modeling and implementation of cost models for a multi-cloud Kubernetes context

Supervisor
prof. Fulvio Risso

Candidate
Federico Cicchiello

December 2021

Summary

In the last year more and more companies are moving toward the adoption of cloud, both public and private, to deploy their services. In order to better exploit the cloud capabilities, such as the horizontal scalability, they start developing applications to be cloud-native, that is made of a set of containerized micro-services cooperating each other. Additionally they're becoming more and more interested in adopting multi-cluster solution such as hybrid cloud and multi-cloud, to increase the resiliency and improve performance being present in multiple geographical regions. Kubernetes is becoming the defacto standard cloud management system, allowing to manage a cluster and orchestrate cloud-native application components life cycles.

This thesis aims at investigating what are Kubernetes cluster costs and how they are structured, either in case of hosting at public cloud (Amazon AWS, Google GCP and Microsoft Azure) and in case of private on-prem infrastructure. The identified cost components have been modeled in the two cases, proposing some formulas that describe how to compute the overall Kubernetes cluster cost.

A passenger model has been proposed to express how the container cost can be expressed. Such model has been applied in the context of the design of a multi-cluster solution to make the application components distribution costs aware, so improving the overall cost efficiency. The design proposes a component, the cost oracle, that maintains each cluster's cost model and interacts with the multi-cluster scheduler providing it the cost of allocating the container requests in the cluster.

A prototype of the cost oracle has been implemented and evaluated by means of some simulation to understand its impact over the scheduling performances and decisions.

Contents

List of Figures

Chapter 1

Introduction

In the last years cloud computing is becoming more and more important for companies. They start developing their applications to be cloud-native, composed by a bunch of containerized micro-services cooperating each other in order to exploit the horizontal scalability capabilities. Having many containers require a component able to manage their life cycle: the widely used solution is Kubernetes.

More and more companies are interested in diversifying their cloud infrastructure in two main ways:

1. **multi-cloud**, that is having clusters at different public cloud providers;

2. **hybrid cloud**, an owned physical data center and a virtual infrastructure at one or more cloud providers.

Diversifying the infrastructure allows to increase resiliency, to be present on different geographical regions, to avoid lock-in to a single provider and gives the opportunity of saving costs.

The last point is what this thesis wants to investigate. How can we exploit that kind of infrastructures optimizing costs?

Public cloud clusters are elastic: the number of nodes can increase and decrease in time to adapt their overall capacity to the actual workload in execution. This has to be kept into consideration when modeling their costs.

Knowing how Kubernetes cluster costs are structured can be very useful for many reasons: for teams to increase awareness on how much are spending, to compare what cloud provider is more convenient for some requirements, to estimate the costs of having a private data center and comparing it to having virtual infrastructures.

Cloud providers bills, unfortunately, does not give at all details on how costs are distributed, so having a model can make them more clear.

An effort has been made on modeling container costs since they are the smallest execution unit in Kubernetes. The knowledge of their model has been then applied to a multi-cluster scheduling scenario to increase the cost efficiency of the container distribution among the clusters thanks to their cost information.

The thesis is structured as the following:

- in chapter 2, Kubernetes, which is becoming the de-facto standard cloud management system, will be presented with a focus on why it is so useful and spread, what are its most important concepts; then the project Liqo will be presented;
- in chapter 3, an analysis will be done to discover what are the costs associated to a Kubernetes cluster, both the public and the private cloud will be investigated;
- in chapter 4, the cost components identified in the analysis will be modeled to obtain some formulas to compute the cost of a cluster and the cost of a Pod;
- in chapter 5, the design of a component implementing a cost model, the cost oracle, will be presented in the context of multi-cluster scheduling;
- in chapter 6, the main choices will be presented in the implementation of the cost oracle in a simulation scenario;
- in chapter 7, the implemented cost oracle will be evaluated in different scenario, with a focus on its performance impact and influence on the Pod distribution across clusters;
- the chapter 8 will present some conclusions related to the thesis work.

Chapter 2

Kubernetes and Liqo approach to multi-cloud

This chapter will present the main fundamentals of Kubernetes, currently the most widespread container orchestrator, and Liqo, a platform for managing multi-cluster networks developed at Politecnico di Torino.

2.1 Kubernetes

Kubernetes is a portable, extensible, open source platform for running and coordinating containerized applications across a cluster of machines. This section is inspired by the Kubernetes official online documentation[1].

2.1.1 Applications deployment evolution

In order to understand why is Kubernetes so useful and appreciated, it's useful to have an historical perspective about the applications deployment evolution over time.

Traditionally, organizations ran applications on physical servers. However there was no way to define resource boundaries for applications in that environment leading to resource allocation issues. This means that if multiple applications run on a physical server, there can be instances where one application would take up most of the available resources and, as a result, the other applications would underperform. The solution to these issues would

[1] https://kubernetes.io/docs/home/

be to run each application on a different physical server, but doing so many resources would be wasted, costs would increase a lot and maintenance of such machines would complicate as well.

Then virtualization was introduced: it allows to define and run multiple Virtual Machines (VMs) on a single physical server. Virtualization allows applications to be isolated between VMs providing at the same time an higher level of security, since one application cannot access resources and data outside of its virtual environment. Furthermore, virtualization allows better utilization of resources in a physical server, enabling better scalability because an application can be added or updated easily, reducing hardware costs. With virtualization a set of physical resources can be presented as a cluster of disposable virtual machines. Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

More recently, a new approach to application deployment has been proposed based on the concept of container. A container is similar to a VM, but it has relaxed isolation properties to share the operating system among the applications. Therefore, containers are considered lightweight so they allow to improve the hardware usage efficiency compared to the VM paradigm. A container has its own filesystem, share of CPU, memory, process space and more. An important property of containers is portability: they are decoupled from the actual underlying infrastructure so they can be portable across different machines. Deploying applications into containers is much simpler and faster than using VMs.

The container technology has changed not only applications deployment but also their development. Applications are developed as a set of smaller, independent pieces that can be deployed and managed dynamically, called micro-services.

2.1.2 Container orchestration

Managing the applications' containers life cycle is much complex. Containers have to be allocated, run, scaled, monitored and made resilient. Kubernetes comes into play, providing a solution for managing this scenario. It provides:

- service discovery and load balancing: containers can be exposed using a DNS name or their own IP addresses, the incoming network traffic is balanced among replicas to make the deployment stable;
- storage orchestration: it allow to mount arbitrarily selectable storage

systems such as local storage or public cloud providers' storage services;

- automated rollouts and rollbacks: the desired applications state is defined declaratively by the user, Kubernetes reconciles their actual state with the desired one;
- automatic bin packing: it manages cluster's resources by properly distributing containers;
- self-healing: containers status is automatically checked, in case of failures Kubernetes restarts them;
- secret and configuration management: Kubernetes provides a simple mechanism to handle applications configuration and sensible data.

2.1.3 Architecture

When Kubernetes is deployed, a cluster is created. A Kubernetes cluster consists of a set of worker 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 users. In production environments the control plane usually runs across multiple computers and a cluster usually run multiple nodes, providing fault-tolerance and high availability.

The figure 2.1 shows a diagram of Kubernetes cluster with all the components linked together.

Control plane components

The control plane's components make global decisions about the cluster (for example, container scheduling), as well as detecting and responding to cluster events (for example, starting up a new container). Control plane components can be run on any machine in the cluster, but typically are run on the same machine for simplicity. In this case such machine does not host user containers.

API server The API server is the component that exposes the Kubernetes REST API, acting as the front end for the Kubernetes control plane. It processes and validates REST requests and updates the state of objects, allowing clients to configure workloads and containers across worker nodes.

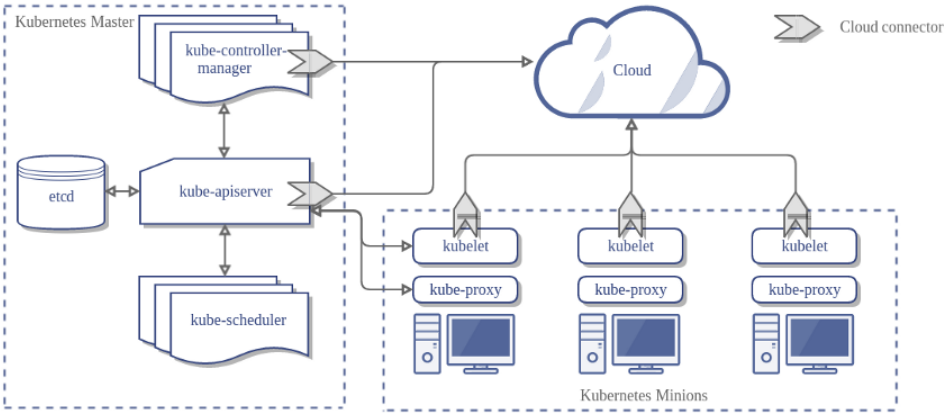

Figure 2.1. Kubernetes architecture

etcd etcd is the Kubernetes backing store, providing consistent and highly available key value store to be used for all cluster data. It hosts the configuration data of the cluster, representing its overall state at any given point of time.

Scheduler The Kubernetes scheduler watches for newly created containers with no assigned node, and selects a node for them to run on. To take the scheduling decision it takes into account individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

kube-controller-manager The kube-controller-manager is the component that runs controller processes. A controller is a control loop that watches the shared state of the cluster through the API server and makes changes attempting to move the current state towards the desired state. An example of controller is the node controller, responsible of noticing and responding when nodes go down.

cloud-controller-manager The cloud-controller-manager embeds the cloud-specific control logic. It allow to link the cluster with the cloud provider's API. As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that can

be run as one or more processes in order to improve performance and increase resiliency.

Node components

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

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

kubelet The kubelet is an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

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

2.1.4 Kubernetes objects

Kubernetes objects are persistent entities in the Kubernetes system that represent the state of the cluster. Specifically, they can describe:

- what containerized applications are running and on which nodes;
- the resources available to those applications;
- the policies around how those applications bahave, such as restart policies, upgrades and fault-tolerance.

Once an object has been created, Kubernetes by means of its controllers will move the cluster state to the desired one expressed in the object itself. Almost every Kubernetes object includes two nested object fields: the object spec and the object status. The spec is a description of the characteristics the resource the user wants to have: its desired state. The status describes the current state of the object, supplied and updated by the Kubernetes system

and its components. The Kubernetes control plane continually and actively manages every object's actual state to match the desired state expressed in the spec.

Namespace

In Kubernetes, namespaces provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects (e.g. Pods, Deployments, Services) and not for cluster-wide objects (e.g. StorageClass, Nodes, PersistentVolumes). Namespaces are intended for use in environments with many users spread across multiple teams, or projects. Namespaces cannot be nested inside one another and each namespaced Kubernetes resources can be only in one namespace.

Labels and Selectors

Labels are key/value pairs that are attached to objects. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system.

A label selector is a way the user can identify a set of objects.

Pod

Pods are the smallest deployable units of computing in Kubernetes. A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation. Within a Pod's context, the individual applications may have further sub-isolations applied. Usually Pods are not created directly from the users, but their creation is triggered by some "higher level" objects such as Deployments or Job.

Job

A Job is an object that creates one or more Pods and continues to retry their execution until a specified number of them successfully terminate. As Pods successfully complete, the Job tracks the successful completions. It is meant for carrying out batch processes.

ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods. It is a grouping mechanism that lets Kubernetes maintain the number of instances that have been declared for a given Pod. The definition of a ReplicaSet uses a selector, whose evaluation will result in identifying all Pods that are associated to it.

Deployment

A Deployment provides declarative updates for Pods and ReplicaSets. It allows the user to describe an application's life cycle, such as which images to use for the app, the number of Pods there should be, and the way in which they should be updated.

StatefulSet

The StatefulSet is the Kubernetes object used to manage stateful applications. It allows to manage the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These Pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

Service

The Service is an abstract way to expose an application running on a set of Pods as a network service. It allows to define a logical set of Pods and a policy by which to access them. The set of Pods targeted by a Service is usually determined by a selector. The Service's field Type allows to specify its kind among:

- ClusterIP: exposes the Service on a cluster-internal IP, making it reachable only from within the cluster;
- NodePort: exposes the Service on each Node's IP at a static port (the NodePort), making it reachable also from outside the cluster by requesting <NodeIP>:<NodePort>;
- LoadBalancer: exposes the Service externally using a cloud provider's Load Balancer;
- ExternalName: maps the Service to the contents of the externalName field, by returning a CNAME record with its value.

Ingress

An Ingress is the Kubernetes object that manages external access to the services in the cluster, typically HTTP. Ingress may provide load balancing, SSL termination and name-based virtual hosting. Ingress exposes HTTP and HTTPS routes from outside the cluster to Services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

StorageClass

A StorageClass provides a way for administrators to describe the classes of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators.

PersistentVolume

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. A PersistentVolume has a life cycle independent of any individual Pod that uses it.

PersistentVolumeClaim

A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod: Pods consume node resources and PVCs consume PV resources. Claims can request specific size, class (StorageClass) and access modes for the PV.

2.1.5 Kubernetes Cluster Autoscaler

The Cluster Autoscaler is a tool that automatically adjusts the size of the Kubernetes cluster when one of the following conditions is true:

- there are Pods that failed to run in the cluster due to insufficient resources;
- there are nodes in the cluster that have been underutilized for an extended period of time and their Pods can be placed on other existing nodes.[11]

It is designed to run on Kubernetes control plane node for resiliency reasons. The Cluster Autoscaler is configured at the moment of cluster definition, by means of three main parameters:

1. the minimum number of nodes to automatically scale;
2. the maximum number of nodes to automatically scale;
3. the scale-down utilization threshold.

The third parameter is the node utilization level, defined as sum of requested resources divided by capacity, below which a node can be considered for scale down.

2.1.6 Prometheus

The Prometheus project is a systems and service monitoring system, often used within a Kubernetes cluster. It collects metrics from configured targets at given intervals, evaluates rule expressions, displays the results, and can trigger alerts when specified conditions are observed. Prometheus collects data in the form of time series by querying a configured list of data sources, called exporters, at a specific polling frequency. It provides its own query language PromQL that lets users select and aggregate data.

Some of the most used exporters are cAdvisor, node-exporter and kube-state-metrics.

cAdvisor

cAdvisor provides container users an understanding of the resource usage and performance characteristics of their running containers. It is a running

daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics.[7]

node-exporter

node-exporter is a Prometheus exporter for hardware and Operating System metrics. It is designed to monitor the host system, that is the node. In a Kubernetes cluster, an instance of the node-exporter runs on every nodes in the cluster.

kube-state-metrics

kube-state-metrics is a simple service that listens to the Kubernetes API server and generates metrics about the state of the objects. It is not focused on the health of the individual Kubernetes components, but rather on the health of the various objects inside, such as Deployments, Nodes and Pods.[8]

2.2 Liqo

Liqo is an open source project developed at Politecnico di Torino, thought to enable dynamic and decentralized resource sharing across Kubernetes clusters, either on-prem or managed.[6] It allows to build and manage complex multi-cluster topologies (the so called ”big cluster”) in a simple way, without any modification of Kubernetes or to the applications. With Liqo it’s possible to extend the control plane of a Kubernetes cluster across the cluster’s boundaries, making multi-cluster native and transparent. An entire remote cluster is seen as local virtual node, allowing workloads offloading and resource management compliant with the standard Kubernetes approach. With Liqo the multi-cluster topology can be managed in a decentralized way: clusters are peers, there is no any centralized management entity.

2.2.1 Peering

The basic mechanism that allows to build a multi-cluster network is the peering. The peering is a relationship two clusters can establish in order to join. The peering is unidirectional: such process allows one cluster to share a subset of its resources to another one. During the peering phase

two clusters exchange some configuration parameters. From the perspective of a ”home” cluster, the peering process terminates by creating a new local virtual node that represents a subset of the resources available in the other ”foreign” cluster.

2.2.2 Liqo multi-cluster scheduler

After a multi-cluster network topology has been built, applications can be deployed over it. The Liqo multi-cluster scheduler is the component that takes as input an application made of a set of Pods, and decides how to distribute its components among the available clusters. The scheduler works per application: this means that the scheduling process succeeds only if all the Pods the application is composed of have been successfully scheduled. After the Liqo scheduler has chosen the destination cluster for a Pod, that cluster's Kubernetes scheduler will perform a second scheduling process local to the cluster. An important characteristic of the scheduler is its bandwidth finite capacity awareness: it models network links between the clusters and tracks their usage.

The Liqo scheduler models the multi-cluster infrastructure as a graph where clusters are nodes and network links between a pair of clusters are edges.

The Liqo scheduler tracks the allocation of the resources of all the clusters in the topology. Its algorithm uses a very simple cost model: each cluster defines some unitary costs for the resources (cost per CPU core, cost per GB of RAM) that is multiplied for the Pod resource request to compute its price quotation. Network links as well have a tariff for their usage.

However such simplified cost model does not fit how cluster resources are really charged, as we will see in the next chapters, so the Liqo scheduler is not capable of optimizing the cost of the application components distribution.

Chapter 3

Analysis of Kubernetes cluster costs

The goal of this chapter is to analyze what are the costs of building and maintaining a Kubernetes cluster with reference to public cloud (in particular Amazon Web Services, Microsoft Azure and Google Cloud Platform since they are the biggest players on the market) and on-premises cloud.

3.1 Public cloud

Public cloud providers offer a very wide choice of products and services to fit all the needs a customer can have, from the basic VM computing service to feature rich DBMS, from simple network functions to Identity and Access Management, with an unreachable degree of elasticity, flexibility, resiliency and agility in reacting to scalability needs. Non negligible at all the very short time-to-market in putting into operation a virtual infrastructure and low capital investments, so public cloud is particularly palatable for small and medium businesses as well as for big ones. In order to build and deploy a Kubernetes managed cluster at a cloud provider, a set of components must be chosen and combined, each with a different impact on the overall cost the virtual infrastructure will have.

3.1.1 Control plane cost

All the main public cloud providers charges for the deployment of the cluster's Kubernetes control plane: it is created and configured in a high availability way, possibly spanning across all the geographical zones within the customer selected region. The service offered includes some provider-specific Kubernetes components: the API server, the distributed key-value store etcd, the scheduler and an implementation of the cloud-controller-manager. The control plane resiliency and scalability is completely managed by the cloud provider: it offers some Service Level Agreements (SLA) for the cluster control plane uptime between the 99.5% and 99.95% on the base of the selected configuration (using single or multiple availability zones) and the specific provider. The cluster management fee applies to all the deployed clusters irrespective of the mode of operation, the actual size or topology. All the three cloud providers charges each cluster for 0.1$/hour for the cluster control plane and the previously described uptime SLAs; only at Azure it is possibly free of charge in case of uptime SLAs renunciation.

3.1.2 Node costs

In the Kubernetes architecture, the node is the component responsible of Pods execution, it is the target computing unit for the scheduler. A node is made of a bunch of resources: some vCPUs, an amount of RAM, possibly some vGPUs and some amount of local ephemeral storage. It is charged for the time it is turned on with generally a per second precision with a minimum fee of one minute, regardless how its resources are actually loaded. No other commitments such as upfront payments are needed. A node has a different tariff on the base of its characteristics, the provider, its geographical location and its provisioning mode. The public cloud providers under analysis offer two ways of deploying nodes: using VM instances (unmanaged way) or relying on the container serverless service (fully managed way).

VM-as-node

The first one was the traditional operational way in which the customer at the moment of cluster creation specifies the number and the profile of the VMs that will act as nodes, so an a priori design of the cluster composition and dimensions is required and successive updates to the topology are under his responsibility. Public cloud providers offer a very wide choice of VM instance types that differ in terms of hardware profile and also in terms of cost to fit

all possible needs one could have. Bare metal instances are available as well. Beside the VM profile, the operating system running on top of it is under customer choice, from vanilla Linux to Windows or MacOS (only on AWS), as well as the architecture (e.g. x86 or ARM), and impacts over the node tariff. Furthermore, a VM instance must be selected in one of three possible options: on-demand, preemptible or reserved.

- On-demand instances are the full price ones, they are charged for their uptime measured as the number of seconds between when they are powered on and when they are stopped.
- Reserved instances, instead, are charged for the time interval for which they have been reserved (from one to three years), regardless their actual uptime, but with an important discount with respect to the on-demand price.
- Preemptible instances are provided at lower prices than the on-demand one (up to 90% less in AWS, up to 66% less in GCP), but provides less quality guarantees: as their name suggests they can be deallocated from the customer availability at any time, with just a bit of time of alert so that some termination routine can be executed (60 seconds before in GCP, 3 minutes in AWS). Preemptible prices oscillates often in time on the base of the market trend, they depend on the actual availability of resources at every specific availability zone. GCP's preemptible instances are subject to a limitation: they cannot be on for a subsequent time greater than 24 hours, if this limit is reached they will be automatically stopped[12]. Preemptible instances are suitable for some stateless, fault-tolerant or flexible workloads that can deal with sudden interruption such as batch processing or some development environments; they are typically not suitable to deploy production applications. In AWS and Azure terminology this kind of instances is called "spot".

Since nodes are payed for the time they are up, in order to improve the cost efficiency the Kubernetes' Cluster Autoscaler can be configured to automatically handle the nodes powering to approximately fit the actual load but it is not so particularly dynamic.

Fully managed node deployment

The second way to deploy nodes is the simplest one from the customer's point of view: nodes are automatically created or destroyed each time a Pod

is scheduled or terminates, with so many resources as the Pod requires to execute. Nodes are allocated via the provider's container serverless service, such as AWS Fargate, GCP Autopilot GKE and Azure Container Instances. The advantages of this solution are the automatic infrastructure provisioning, configuration and health-checking, and a cost optimization since the customer pays only for the resources actually used and for nothing more.

The node provisioning mode used in the successive modeling and design is the VM-as-node since it is the one Liqo is based upon and the most spread scenario.

3.1.3 Storage costs

Cloud providers offer services to support persistence in the Kubernetes cluster context: in this case the most used and best integrated is the Block Storage one. This service allows customers to use a volume of freely selectable size, so the capacity is not limited to some predefined alternatives as in the physical world. A volume is charged for the time of allocation, regardless how much it has been loaded. Different rates applies to different kind of volumes: cloud providers offer volumes from high performance SSDs to cheaper alternatives such as HDDs. The Kubernetes' storage class identifies the volume type: their names are different from provider to provider. The volume is allocated into the same cluster region and zone after a storage volume of a given storage class is required, for example among a Pod's specifications.

3.1.4 Networking costs

There are many costs associated to the cluster networking.

Outgoing network traffic

From a node's point of view, the network traffic is charged only in one direction: the outgoing one. There are different rates on the base of from and to where the traffic is exchanged: all the three cloud provided under analysis offer free traffic within the same availability zone, a certain tariff for the traffic between zones in the same region, a set of tariffs for traffic between different regions and a last one for the outgoing traffic towards Internet. Additionally, Google Cloud Platform provides two different choices to select the network the traffic will traverse in the most of its path: the Standard Tier network or the Premium Tier one. The latter is the most expensive, but the more

performing since it will exploit the Google's owned network infrastructure, of course as far as possible (mainly backbone, almost never at the edge).

Public IP addresses

Another network cost is the one to obtain public IP addresses, that are fundamental to be reachable from the Internet, e.g. to expose a service to the end users. An address is charged with a per-hour tariff, from the time of allocation to that of deallocation. One exception to this is the case of AWS: its EC2 service (the one at the base of the VM-as-node mode) includes in the VM cost also a public Internet address.

Load Balancers

In the Kubernetes context there are two kind of load balancers (LB) of interest: the Network Load Balancer (NLB) and the Application Load Balancer (ALB). The first one comes into play in the moment in which a service of kind LoadBalancer is deployed into the cluster: the cloud-controller-manager intercepts this request and provides a Network Load Balancer to balance the incoming requests to the various replicas providing the service. The Application Load Balancer is deployed by the cloud provider in the moment in which an Ingress resource has been deployed in the cluster. NLBs and ALBs are charged for the time they are deployed with a per-instance fee plus another one that depends on the processed traffic amount, the number of established connections, the number of active connections and the number of configured rules. The actual tariff depends on the specific provider and the region in which the service is deployed.

3.2 Private cloud

In order to dig into the private cloud costs the approach of the Total Cost of Ownership (TCO) has been followed: the TCO is a financial estimate approved by Gartner to help companies calculate precisely economic impact during the whole life cycle of IT projects. [3] The computation of the TCO for an on-premise data center must be done over a defined time period: a good value in this case is to consider a 10 years time interval considering the technological obsolescence of the infrastructure. Costs can be divided into two main categories: the upfront and the recurring ones. There are numerous costs associated with an on-premises data center; a good way to identify

them is analyzing the various phases the infrastructure will pass during the considered time interval: design, build, deploy, maintain and upgrade. The first three phases account for the upfront costs, the last two instead for the recurring ones.

3.2.1 Design costs

The design phase involves the activities of identifying the future business needs, decide the dimensions and the composition of the data center, estimate the future growth, identifying the personnel that will administer the system. In this phase the starting point are the business requirements: identifying them is the very first activity that costs in terms of time. Expert personnel is required for the infrastructure design: if available internally implies some salary costs, otherwise some consulting fees must be payed.

3.2.2 Build costs

The build phase implies an important capital investment to buy hardware appliances - servers, storage units, network equipment, cables, cooling system and the space that will host the infrastructure. Costs can include also the purchase of software licenses, in case they are under the upfront category. Also the salary for the personnel employed in the build phase must be considered.

3.2.3 Deploy costs

The deploy phase associated costs are the one to setup, install and test the new infrastructure (both the hardware and the software), the personnel training and possibly migration from an older system.

3.2.4 Maintain costs

After a data center has been deployed, a set of recurring costs must be kept into account: the most important is the salary for the personnel employed to administer the system. Some expenses are incurred for performing regular monitoring, diagnostics and testing of the system. Electric and network bills are part of this phase's costs as well as the recurring fees for software licenses. The purchased hardware and software require ongoing support and maintenance, approximately 20 percent of the upfront cost per year. Amazon

Web Services (AWS) estimates the ongoing cost of maintenance to be 18 percent to 22 percent.

3.2.5 Upgrade costs

Most hardware and software has about three to five years of useful life, so they have to be renewed during the whole infrastructure lifetime. This implies also some costs to reassess the core business requirements.

Chapter 4

Modeling Kubernetes costs

After the costs of public and private cloud have been identified, a work of modeling has been made in order to formalize how to compute the overall infrastructure cost. Such a model can be useful for a set of usages: it can be used to estimate the costs of having a virtual infrastructure at a certain public cloud provider, to compare what is the most convenient provider for some certain requirements, or to support an analysis of the kind make-or-buy to evaluate if investing on a proprietary physical infrastructure. An effort has been made to identify some ways to associate cluster costs to Pods that are the Kubernetes abstraction for the smallest deployable unit of computing. This knowledge will be applied in the next chapters in a multi-cloud domain, specifically to enhance the Liqo scheduler with some models used to compute what are the more costs-efficient ways to place a Pod.

4.1 The Kubecost approach to cloud cost modeling: characteristics and limitations

Before diving into costs modeling, an existing approach has been studied: the Kubecost's one. Kubecost is a partially open-source project which aims at providing teams using Kubernetes clusters visibility into current and historical money spend and resource allocation. The advantage of having this detailed view is to provide awareness on how much each team is spending, since Kubernetes makes very easy to increase the expenses thanks to a number of functionalities to scale up and out, that is allocate more powerful

resources or increase their number. Cloud providers' bills are not so much detailed at all, so in a company context a tool like this can be very useful. In order to reach its goal, Kubecost uses its own open source Kubernetes cost model.[9]

4.1.1 The cost model

Kubecost is able to associate a cost in a given time window to each of the main Kubernetes abstractions: from the Pod to the Service, from the Deployment to the Namespace. Let's start understanding how the Pod cost model works since it is the smallest brick in Kubernetes model: all the higher level abstraction costs can be obtained by properly aggregating Pods' costs. Each Pod has among its specifications some requests, that is some amount of resources it needs to execute that will be allocated exclusively to it just after it has been successfully scheduled. As seen in the previous chapter, there are four main cost components in the public cloud case: control plane, node, storage volumes and networking. The control plane cost is the only one that can be associated only to the whole cluster: it's impractical and maybe useless to say how much each Pod accounts to it, so it is excluded from the Pod's cost model. A successfully scheduled Pod is bound to a specific node; such a node has a certain tariff that is retrieved via the cloud provider pricing API or, in case this is not possible (e.g. on-premises cloud), from some user provided configuration (it includes hourly CPU, spot CPU, RAM, spot RAM and GPU tariffs plus storage volumes and egress network traffic cost per GB). A Pod can require an arbitrary amount of resources, in most cases just a portion of the available ones among the node's resources. In order to compute how much of each resource to account to the Pod, Kubecost uses Prometheus collected metrics coming from exporters such as cAdvisor, node-exporter and kube-state-metrics, to obtain, in a given time window, the resource request and the average of its actual usage. Then it computes the maximum between the two values: the result is considered the resource amount to account to the Pod. The flow diagram of a Pod's resource cost is shown in figure 4.1. Then for each node of the infrastructure a normalization factor is computed so that:

$$\sum_i c_{container,i} + \sum_j c_{idle,j} = c_{node}$$

, where $c_{container,i}$ is the cost of the i-th container[1] on the node, $c_{idle,j}$ is the cost of the j-th idle resource on the node, and c_{node} is the node's cost computed by multiplying the actual node's tariff and its time of allocation. The container cost is computed as:

$$c_{container,i} = \sum_j (c_{resource,j}) \cdot f_{norm}$$

, where $c_{container,i}$ is i-th container's cost, $c_{resource,j}$ is the cost of the j-th resource, f_{norm} is the normalization factor. The normalization factor is needed

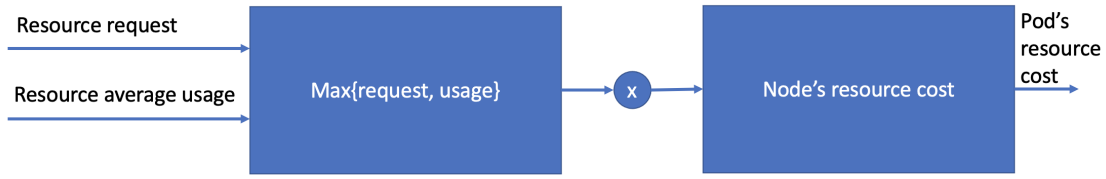

Figure 4.1. Kubecost pod resource cost computation

because cloud providers only gives an aggregated tariff, the node's one, and it is not decomposable into per-resource parts. So Kubecost uses as per-resource tariff some default values and then normalizes the costs so that the overall node cost (the one taken from the pricing API) is equal to the sum of the Pod's resources plus the cost for idle ones, since a node is almost never fully loaded.

Pods requiring storage volumes via Persistent Volume Claims are charged for them: the actual tariff for the selected storage class in the cluster's region is retrieved via the specific provider pricing API.

$$c_{storage} = \sum_i t_{allocation,i} \cdot size_{volume,i} \cdot c_{volume,i}$$

, where $c_{storage}$ is the overall storage cost, $t_{allocation,i}$ is the time of allocation of the i-th volume, $size_{volume,i}$ is the i-th volume size, $c_{volume,i}$ is the i-th volume tariff.

For what concerns networking costs, Kubecost retrieves via Prometheus each Pod's egress traffic amount and charges them for the respective tariff on the base of the destination (outside the zone, outside the region or towards

[1] A Pod is composed of a set of containers, in most cases just one

Internet). Such tariffs are not actually retrieved from the specific provider APIs, but are taken from the configuration. As seen in the previous chapter there are other networking costs such as Load Balancers and public IPs that, however, cannot be bound to single Pods: they are related to the Service Kubernetes abstraction. The Load Balancer cost model used in Kubecost is a simplified one that does not take into consideration how this computation is different at each provider and treates in the same way Network Load Balancers and Application Load Balancers. It works in this way: it applies a tariff on the first five forwarding rules (c_{F5FR} in the following formula), a different one to the next forwarding rules (c_{AFR}) and a fee to the ingress traffic amount (c_{LBID}), so that:

$$c_{LB} = \begin{cases} c_{F5FR} \cdot n_{rules} + c_{LBID} \cdot tr_{ingress} & \text{, if } n_{rules} \leq 5 \\ c_{F5FR} \cdot 5 + c_{AFR} \cdot (n_{rules} - 5) + c_{LBID} \cdot tr_{ingress} & \text{, if } n_{rules} > 5 \end{cases}$$

, where n_{rules} is the number of configured forwarding rules and $tr_{ingress}$ is the amount of ingress traffic. Public IP costs are not considered at all in the model.

To sum up, for the Kubecost cost model, the Pod cost is computed as:

$$\begin{aligned} c_{pod} = & \sum_i c_{container,i} + \sum_j c_{volume,j} + \\ & + \begin{pmatrix} t_{inter\text{-}zone} & t_{inter\text{-}region} & t_{internet} \end{pmatrix} \cdot \begin{pmatrix} c_{inter\text{-}zone} \\ c_{inter\text{-}region} \\ c_{internet} \end{pmatrix} \end{aligned}$$

$$\begin{aligned} c_{cluster} = & \sum_i c_{node,i} + \sum_j c_{volume,j} + \\ & + \begin{pmatrix} t_{inter\text{-}zone} & t_{inter\text{-}region} & t_{internet} \end{pmatrix} \cdot \begin{pmatrix} c_{inter\text{-}zone} \\ c_{inter\text{-}region} \\ c_{internet} \end{pmatrix} + \sum_k c_{LB,k} \end{aligned}$$

4.1.2 Considerations

Kubecost can be very useful to monitor resources allocation and costs, in particular to account them to each team or department in a multi-tenant scenario and can lead to money saves thanks to the awareness it brings and also the suggestions it proposes in order to reduce the wastes. However its model is not so accurate as some scenarios could need: network costs are roughly computed and mostly pod costs computed in a certain cluster are

not comparable with those computed in another one due to the usage of the normalization factor. Such factor is available only in the aftermath, so it is unavailable for predictive models. Furthermore the model Kubecost proposes has some relevant differences to the way cloud providers actually charges nodes: Kubecost's model is based on the per-resource prices, e.g. CPU/hour or GB_RAM/hour, but providers charge a node as a whole, that is its overall price is not decomposable into per-resource components. So using Kubecost for some usages involving multi-cloud use cases, as the case of Liqo, is not a good idea. Also using the same cost model for public and private cloud is not a good idea: as seen in the previous chapter associated costs are significantly different so the two cases do need different models.

4.2 Cluster cost model

By using the knowledge coming from the analysis of costs and the Kubecost's approach with its insights and limitations, it's time to model Kubernetes cluster costs separately for the case of public and private cloud. These models will aim to express how the compute the overall infrastructure cost.

4.2.1 Public cloud

By putting all together the costs identified in the analysis of the case of Kubernetes cluster hosted at public cloud providers, the following formula has been defined:

$$c_{cluster} = c_{ctrl\text{-}plane} + \sum_i t_{VM,i} \cdot c_{VM,i} + \sum_j t_{volume,j} \cdot size_{volume,j} \cdot c_{volume,j} + c_{network}$$

, where $c_{ctrl\text{-}plane}$ represents the control plane cost, $t_{VM,i}$ is the allocation time of the i-th VM, $c_{VM,i}$ is the tariff of the i-th VM, $t_{volume,j}$ is the allocation time of the j-th storage volume, $size_{volume,j}$ is the j-th volume's size, $c_{volume,j}$ is the tariff of the j-th volume, $c_{network}$ represents network related costs. Units of measure are expressly neglected: they are an arbitrary choice when applying the formula, of course they must be chosen in a consistent way(e.g. if $c_{volume,j}$ is in $USD \cdot GB/h$ then $t_{volume,j}$ will be in h and $size_{volume,j}$ will be in GB). The formula allows to compute the cluster's cost in a certain time unit (e.g. USD/h).

Networking costs are expressed as:

$$c_{network} = \boldsymbol{T} \cdot \boldsymbol{C_t} + c_{LB} + c_{publicIP}$$

, where

$$\boldsymbol{T} = \begin{pmatrix} t_{intra\text{-}zone} & t_{inter\text{-}zone} & t_{inter\text{-}region} & t_{internet} \end{pmatrix}$$

represents the outgoing traffic vector, and

$$\boldsymbol{C_t} = \begin{pmatrix} c_{intra\text{-}zone} \\ c_{inter\text{-}zone} \\ c_{inter\text{-}region} \\ c_{internet} \end{pmatrix}$$

is the outgoing traffic cost vector, where c_i is the cost of outgoing traffic of the type i. The possible outgoing traffic categories are intra-zone (to a destination in the same availability zone of the sender), inter-zone (to a destination in another zone but in the same region of the sender), inter-region (to a different region of the same provider) and Internet (to a destination outside the sender's provider boundaries), since cloud providers charges them differently. c_{LB} depends on the specific provider pricing model, while the public IP addresses cost is computed as:

$$c_{publicIP} = n_{publicIP} \cdot u_{publicIP}$$

, where $n_{publicIP}$ is the number of public IPs allocated, $u_{publicIP}$ is the unit cost for a public IP.

The proposed model is independent from the specific cloud provider: the only provider dependent component is the Load Balancer cost.

4.2.2 Private cloud

As argued in Kubecost's cost model considerations, the private cloud case is significantly different from the public one. So a specific model has been defined, including all the costs identifies during the analysis.

$$c_{cluster} = c_{upfront} + c_{recurring}$$

This first formula tells that the overall infrastructure cost for the whole time window chosen in TCO computation is the sum of some upfront costs and some recurring ones, from a very coarse-grained perspective. An approach to private cloud costs modeling can be that of computing the overall cost of the whole TCO time window, and then amortize it across the time to obtain an infrastructure cost for the interval of interest (e.g. hour, month, year...).

Generally, it's possible to categorize costs as upfront or recurring on the base of the phase they are involved in.

$$c_{upfront} = c_{design} + c_{build} + c_{deploy}$$

$$c_{recurring} = c_{maintain} + c_{upgrade}$$

4.3 Pod cost model

The Pod cost model plays a central role in Kubernetes costs modeling: it is the smallest deployable unit, all higher abstractions cost can be obtained by aggregating pod costs. In the following section two models will be proposed: the passenger and the slice one.

4.3.1 The passenger model

The passenger model is based on a metaphor: Pod's are considered passengers while nodes are vehicles. We have seen that a node is charged for the time it is on, regardless to its effective usage. So the most cost-efficient way to exploit is to make it perform the maximum amount of work, ideally filling all its resources completely. At the core of this model there is the idea that the total node's fee is charged fully to the first Pod that will be scheduled onto it, then the following Pods will not be charged for node's resources. It's like a car that has a path to go: it has some costs such as fuel and tolls that has to be paid when a driver decides to leave (the first Pod on the node), then having zero, one or more passengers (other Pods on the same node) has no impact over the cost of the trip, so it's like they travel free. Let's have for each node i of the cluster a set N_i of Pods scheduled on it: each time a Pod is placed on node i an element is added to N_i, each time a Pod is removed from the node i the corresponding element is removed from N_i.

$$c_{Pod}(i) = \begin{cases} c_{node} + \sum_j c_{volume,j} + \boldsymbol{T} \cdot \boldsymbol{C_t} & \text{, if } |N_i| = 0 \\ \sum_j c_{volume,j} + \boldsymbol{T} \cdot \boldsymbol{C_t} & \text{, if } |N_i| > 0 \end{cases}$$

$$\boldsymbol{T} = \begin{pmatrix} t_{intra\text{-}zone} & t_{inter\text{-}zone} & t_{inter\text{-}region} & t_{internet} \end{pmatrix}$$

$$\boldsymbol{C_t} = \begin{pmatrix} c_{intra\text{-}zone} \\ c_{inter\text{-}zone} \\ c_{inter\text{-}region} \\ c_{internet} \end{pmatrix}$$

, where $c_{Pod}(i)$ is the Pod's cost on node i, c_{node} is the node's cost, $c_{volume,j}$ is the j-th storage volume's cost, $\boldsymbol{T}$ is the outgoing traffic vector, and $\boldsymbol{C_t}$ is the outgoing traffic tariff vector. The node's cost is computed as:

$$c_{node} = t_{allocation} \cdot c_{VM}$$

, where c_{node} is the cost for the node the Pod is scheduled on, $t_{allocation}$ is the time interval in which the cost is computed, c_{VM} is the VM's tariff. A volume's cost is computed as:

$$c_{volume,j} = t_{allocation,j} \cdot size_{volume,j} \cdot c_{storage\text{-}class,j}$$

, where $c_{volume,j}$ is the j-th storage volume's cost, $t_{allocation,j}$ is the j-the volume's allocation time, $size_{volume,j}$ is the j-th volume's size, $c_{storage\text{-}class,j}$ is the j-th volume's storage class tariff.

This model is inspired from the behavior of the Kubernetes Cluster Autoscaler and fits very well to it: this component turns on and off nodes on the base of the actual cluster load, allowing money savings when some resources are no longer needed. Furthermore the passenger model is applicable to both the public and the private cloud: what changes is the meaning and the actual value associated to nodes and volumes costs, from the provider's tariff to an energetic cost to take the resource on. In case the model is applied to a private cloud it's useful to consider only the costs that are in a causal relationship with Pods, such as energetic and network costs, excluding in particular the upfront costs.

This is a stateful model: the cost of the j-th Pod depends on the state of the i nodes of the cluster tracked via the sets N_i.

4.3.2 The slice model

For the slice model the cost of a Pod is equal to the sum of the resources costs it requires: it pays only for the slice of resources it uses.

$$c_{Pod} = \sum_i a_{resource,i} \cdot c_{resource,i} + \sum_j c_{volume,j} + \boldsymbol{T} \cdot \boldsymbol{C_t}$$

, where $a_{resource,i}$ is the i-th node's resource amount required by the Pod, $c_{resource,i}$ is the cost of the i-th node's resource, $c_{volume,j}$ is the j-th volume's cost, $\boldsymbol{T}$ is the outgoing traffic vector, $\boldsymbol{C_t}$ is the outgoing traffic tariff vector.

$$c_{volume,j} = t_{allocation,j} \cdot size_{volume,j} \cdot c_{storage\text{-}class,j}$$

$$\boldsymbol{T} = \begin{pmatrix} t_{intra\text{-}zone} & t_{inter\text{-}zone} & t_{inter\text{-}region} & t_{internet} \end{pmatrix}$$

$$\boldsymbol{C_t} = \begin{pmatrix} c_{intra\text{-}zone} \\ c_{inter\text{-}zone} \\ c_{inter\text{-}region} \\ c_{internet} \end{pmatrix}$$

Storage volumes and networking costs are modeled exactly in the same way of the passenger model. Focusing on node's resources (so CPU cores, memory and possibly GPU), to the Pod are charged exactly the amount of resource it requires (e.g. 1/2 CPU and 1GB of RAM). This is a stateless model: the cost of each Pod does not depend at all on the infrastructure state, so it is simpler than the passenger one. But it is far from the actual nodes pricing model: in order to apply it per-resource costs are needed, but they are not provided this way from cloud providers. This model is very similar to the Kubecost's one but with a main difference: there is no normalization factor. This means that Pod costs computed on different clusters are comparable, so the slice model can be effectively applied to multi-cloud domains.

Chapter 5

Design of a multi-cloud costs-aware component: the cost oracle

In the previous chapter some models have been defined to express how Kubernetes cluster costs are modeled, both at whole infrastructure granularity and at Pod granularity. Now it's time to use them in the context of the design of a multi-cloud costs-aware component that will be based in particular on the passenger model.

5.1 Architecture

Before explaining in details the architecture of the cost oracle, a focus is put on the context in which it has been thought.

5.1.1 Context

Liqo is able to build a complex multi-cluster infrastructure (the so-called "big cluster") based on the peering operation between the single "small clusters". An example topology is shown in figure 5.1: there are four clusters (Cluster A, B, C and D) hosted on different infrastructures (A and B at cloud provider X, B at private cloud, D at cloud provider Y) that established a Liqo peering (represented by the black arrows) between some of them - in particular between the Cluster A and all the others. When an user wants to run some job such as a Pod, a specific component has to choose what "small

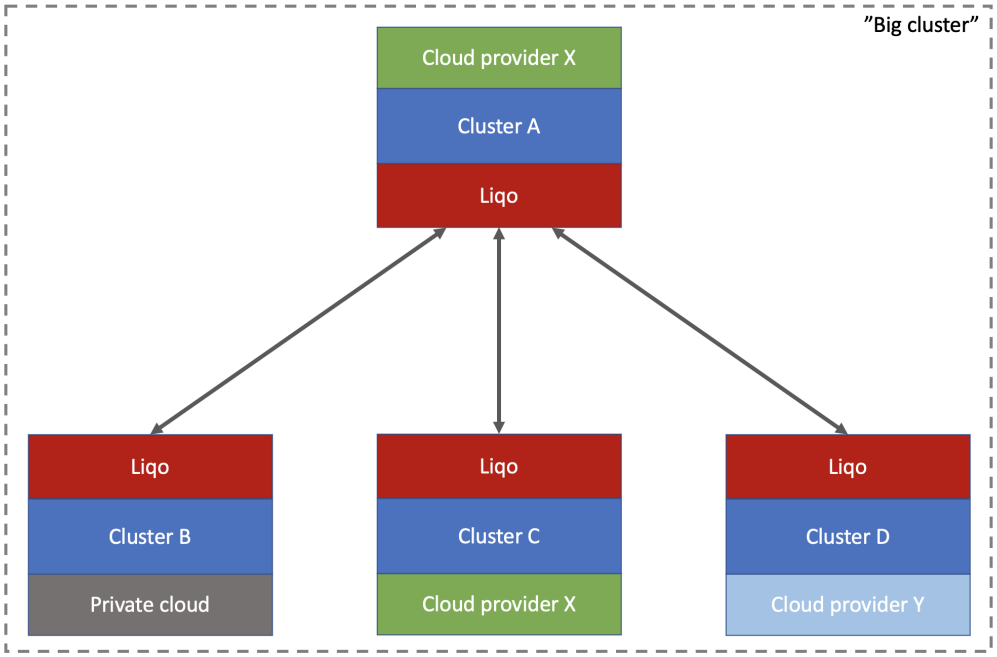

Figure 5.1. Multi-cluster topology based on Liqo peering

cluster" will host it. Such component is the Liqo scheduler. Every cluster with Liqo installed has an instance of the Liqo scheduler that will execute its logic every time a deploy operation is required.

The scheduler's cost model

In order to perform its job the Liqo scheduler tracks the resource capacity and load of the whole multi-cluster infrastructure and uses a cost model to evaluate the cost-efficiency of the possible alternatives for the scheduling solutions, that is the cost of placing that job on each possible "small cluster". The scheduler's cost model is an implementation of a slice model: each "small cluster" defines some per-resource costs (per-CPU core, per-GB of RAM, per-GB of network bandwidth) that are applied to the Pod's requests and to its network traffic estimates to compute a "price quotation". Its cost model is not so coherent with the real pricing model of Kubernetes clusters as seen in the previous chapters. The scheduler works per-application: in a micro-services scenario as cloud computing typically is, an application is made of a set of smaller components (the micro-services) that cooperate with each

other. A micro-service in Kubernetes is represented by a Pod. The network bandwidth estimates come from the knowledge of how much an application's micro-services are coupled. The goal of the design will be that of replacing the scheduler's slice cost model with an implementation of the passenger one in order to take better the scheduling decisions thanks to a more realistic model. The new implementation of the cost model will be named cost oracle.

5.1.2 The cost oracle: what it is and how it works

Each "small cluster" in the Liqo handled multi-cluster infrastructure in addition to having its Liqo scheduler instance will have its cost oracle instance. The cost oracle is a software module with the responsibility of keeping its cluster's cost model and working in concert with the corresponding Liqo scheduler providing it some "price quotations" for the Pods it wants to schedule. Two different scenarios have been identified in the Liqo multi-cluster context that will have different architectures for what concerns the cost oracle.

Multi owned "big cluster"

The first scenario is the most general one: the "big cluster" built through Liqo is composed of resources owned by different entities that want to share them but also want to keep hidden the details of their clusters such as the topology, the actual load and also the cost model. In this case each cluster's cost oracle will know all the details it needs of its cluster and nothing about the other ones. So in order for the Liqo scheduler to perform its job, it will need to contact different cost oracles to obtain different clusters' "price quotations".

Single owned centrally managed "big cluster"

The second scenario is a more specific one: the whole "big cluster" is owned by the same entity. This assumption simplifies the design: the multi-cluster infrastructure can be fully managed from a central point, such as from Cluster A in figure 5.1. Deploying load from a central point allows to divide the clusters into two categories: the master and the slaves. The master cluster is the one from where the "big cluster" is managed (Cluster A in figure 5.1), the slave clusters (Clusters B, C and D in figure 5.1) share their resources and receive their workload from the master one. In this scenario two different kind of cost oracles as well will exist: the master's oracle and the slaves' oracle. The masters' cost oracle will perform most of the work: it will collect

all the "small clusters"' cost models, track the allocated resources in each of them and will cooperate with the master cluster's Liqo scheduler to distribute Pods over the "big cluster". Each slave cost oracle will have reduced responsibilities: it will just send its cluster's cost model, tariffs and detailed topology to the master one and nothing more. An example of this scenario is shown in figure 5.2: the Cluster master is the only one users interact with to deploy the applications, the master's cost oracle collects all slave cluster's oracles cost models.

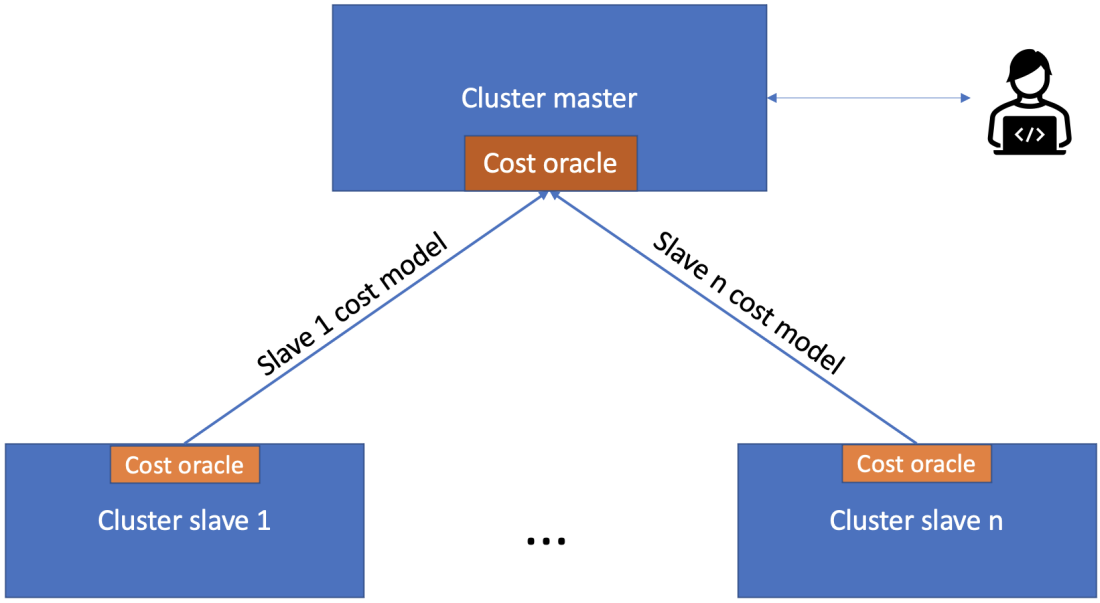

Figure 5.2. Architecture of the cost oracles in the single owner case

5.1.3 Architecture of the prototype

There are two slightly different architectures for the cost oracle in the multi-owned and in the single owned "big cluster". The architecture of the cost oracle in the multi-owned infrastructure is depicted in figure 5.3. The figure shows the components the oracle interacts with within a cluster, in the figure in the Cluster A. As previously said, the cost oracle is questioned from the corresponding Liqo scheduler instance that needs to know the "price quotation" of scheduling on the local infrastructure a certain Pod. The cost oracle is responsible of keeping a model representing the cluster's topology:

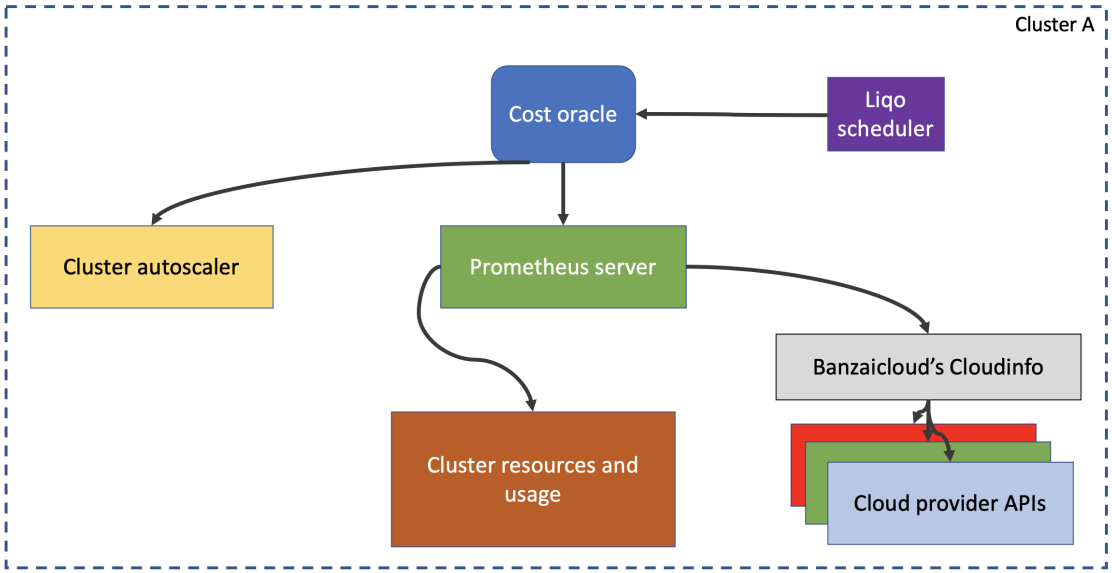

Figure 5.3. Architecture of the cost oracle in the multi-owned "big cluster"

it need to know the nodes number and their capacity in terms of resources. In order to build and keep updated the topology model it interacts with the Kubernetes' Cluster Autoscaler (CA) if enabled, to synchronize with its actual state: indeed the CA is responsible of powering on and off the nodes according to its configuration with the aim of adapting the actual resource availability to the actual workload. The cost oracle uses the cluster's Prometheus server in order to perform queries and retrieve useful metrics of two main kind. The first kind of metrics are related to the actual cluster resource capacity, since it can change across time (e.g. a node's profile can be changed), and to the actual resource allocations. The latter information is crucial for the passenger model since it is stateful: the cost of a Pod depends on the distribution of load at the time of the request. The second kind of metrics the oracle needs is concerned with the actual resources tariffs, only in case the cluster is hosted at a public cloud provider. In order to simplify their retrieval an open source project is used in the architecture: Bandzai Cloud's Cloudinfo[10]. It does the actual job of using the provider specific pricing APIs in order to retrieve the resources tariffs, manages this information in a consistent way and exposes the results as metrics in the Prometheus server. The architecture of the cost oracle in the single-owned "big cluster" is shown in figure 5.4. It is very similar to the previously presented one but

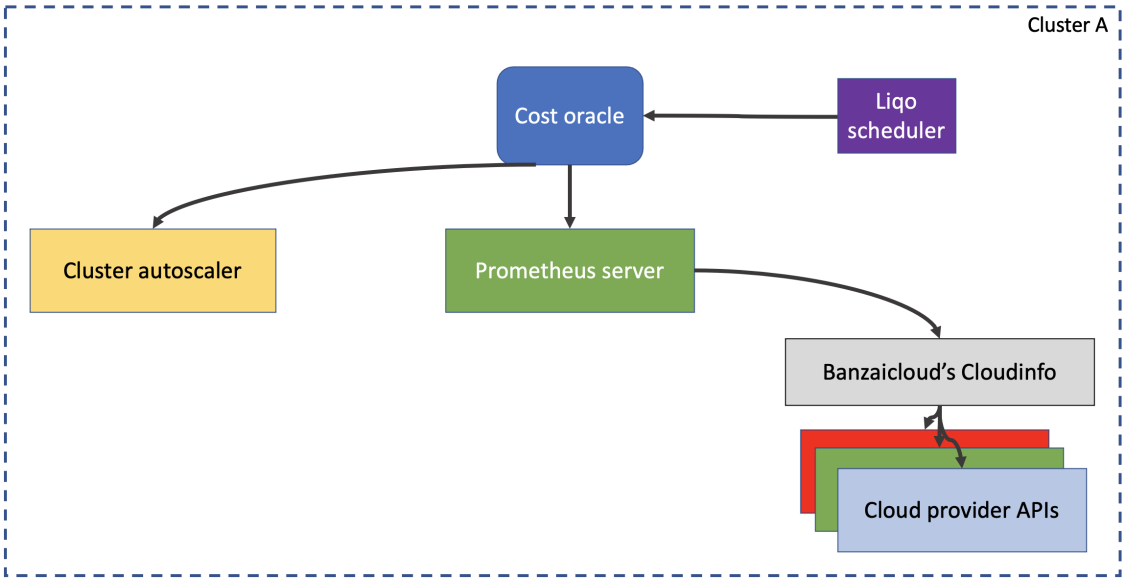

Figure 5.4. Architecture of the cost oracle in the single-owned "big cluster"

with a difference: there is no more the "Cluster resources and usage" block. This component is not needed because in this scenario the "big cluster" is used only from a central point (the master cluster) where the multi-cluster infrastructure topology model and the resources allocation can be tracked entirely within the central cluster's cost oracle. Indeed in order to perform this the master cluster's oracle updates the resource allocations model each time the master cluster's Liqo scheduler instance takes a scheduling decision.

Cost oracle's internal architecture

The figure 5.5 shows how the cost oracle is composed from a logical point of view. This architecture is the same for either the scenario discussed previously. The cost oracle is composed by two main logical blocks: a resource impact evaluation one and a costs aware one. The two blocks works in series in order to provide the Liqo scheduler the input Pod price quotation. Indeed there are two inputs coming from outside: the Pod specifications and its network traffic estimates. The first input contains the amount of resources to allocate to the Pod, the second one consists in some estimates of the network traffic it will exchange with other Pods. The only output coming from the cost oracle is the actual price quotation. Let's take a look at the two logical

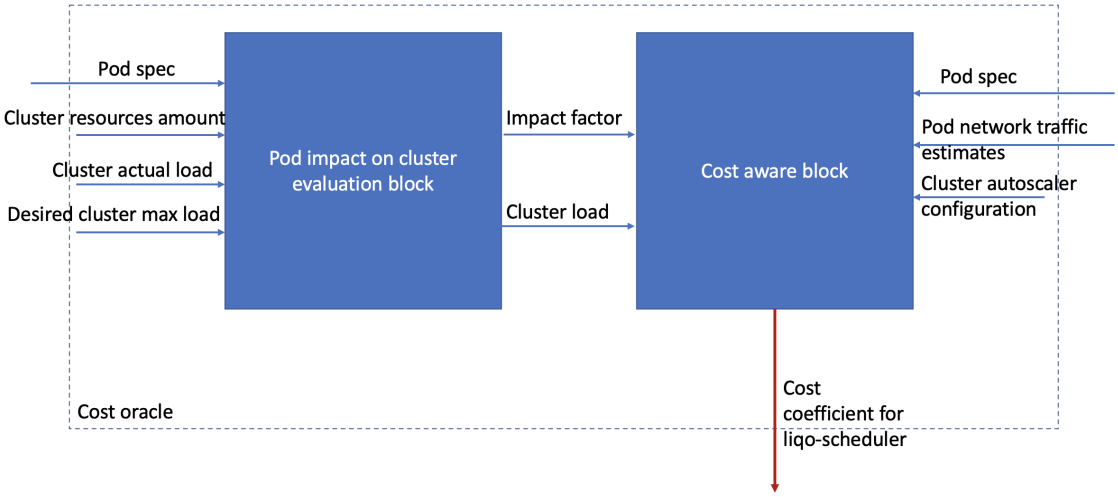

Figure 5.5. Internal architecture of the cost oracle

blocks. The "Pod impact on cluster evaluation" block is the oracle's part that evaluates what the impact of allocating Pod's request will be on the cluster's resources. In order to do it, the block receives the Pod's specifications, retrieves the cluster's resources capacity and their actual load from the Prometheus server (in the multi-owned case) or from some local data structures (in the single-owned case) and reads a user provided configuration parameter: the desired cluster max load. Such parameter represents how much of the cluster's resources the user wants to fill before concerning about the problem of resource contentions with the consequences of performance degradation. The oracle will compute each cluster's resource (CPU, memory) load as the ratio between the allocated amount including the input Pod's requests and the available amount. The overall cluster's load is the highest resource ratio. If the load is smaller than the desired cluster max load then the block will return as impact factor the multiplication neutral value (1), otherwise it will return a factor as big as the desired cluster max load has been exceeded. Such factor is the first output of this block. The second output is composed of the cluster's resources actual load previously computed. The second block is the one implementing the passenger cost model: it applies the cluster's cost model and its resources tariffs to the input Pod requests and network traffic estimates. The result of its computation is multiplied for the impact factor and provided to the Liqo scheduler as the cost oracle's output.

5.2 Interactions

There are two main kind of interactions in the cost oracle design, those between the Liqo scheduler and the Cost oracle and those between the oracles, with some differences between the cases multi-owned and single-owned centrally managed "big cluster".

5.2.1 Oracle - Liqo Scheduler

The interactions between the cost oracle and the Liqo scheduler are two: one named "ComputePodCost", one named "AllocatePodRequests".

ComputePodCost interaction

ComputePodCost is the interaction meant, for the Liqo scheduler, to request the price quotation of a Pod, to a cost oracle instance. The interaction, shown in figures 5.6 and 5.7, is slightly different depending on the scenario considered. In both cases the requests includes two parameters: the Pod's requests and its network traffic estimates with other Pods. In the single-owned scenario there is an additional parameter for the ComputePodCost method: the clusterID. Such additional parameter is needed because the cluster master's cost oracle maintains the resource and cost model of each of the clusters in the topology. So the clusterID will be used by the master's oracle to identify what is the actual cluster of interest for the request. The actual Liqo Scheduler and Cost oracle instances of this interaction depend on the design scenario considered. In the case of multi-owned "big cluster" the Liqo scheduler instance participating in the interaction is the one on the cluster where the user requesting the scheduling of the application is connected; the cost oracle instance that is questioned is the one on the cluster the Liqo scheduler wants to know the price quotation for the Pod. In the single-owned "big cluster" scenario instead, the Liqo scheduler and the Cost oracle instance are always the ones on the master cluster: the oracle in this case has and uses the cost model of all the slave clusters in addition to its one.

AllocatePodRequest interaction

AllocatePodRequests is the interaction meant for the Liqo Scheduler to inform the Cost oracle instance of a scheduling decision. It serves for the oracle to update it's resource allocation tracking, including the just scheduled

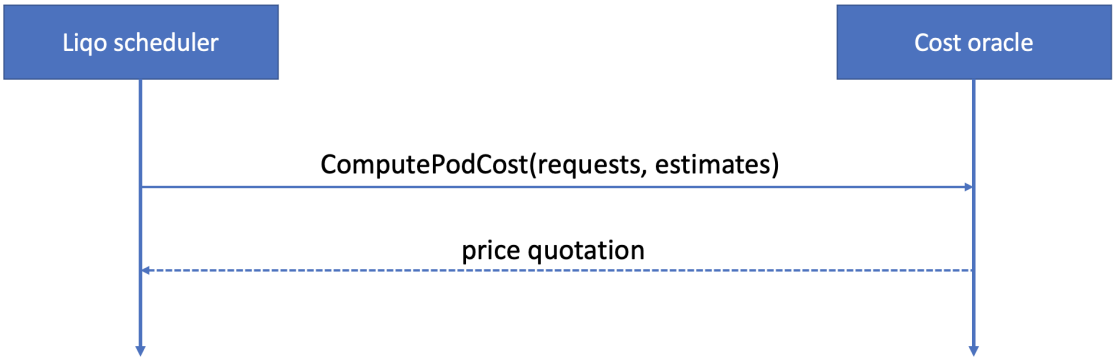

Figure 5.6. ComputePodCost interaction Liqo Scheduler-Cost Oracle in multi-owned ”big cluster”

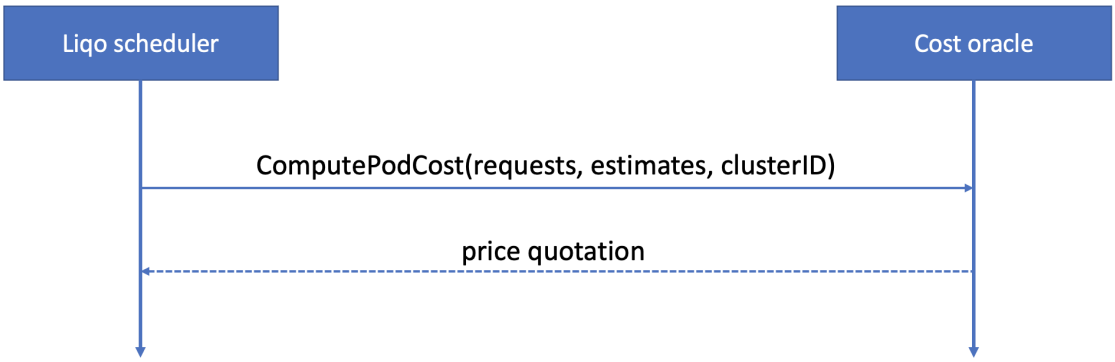

Figure 5.7. ComputePodCost interaction Liqo Scheduler-Cost Oracle in single-owned ”big cluster”

Pod’s requests in its resource model. As shown in figure 5.8, the invocation includes as parameters the Pod’s requests and the cluster identifier. In the multi owned ”big cluster” scenario, this interaction does not exist since each oracle synchronizes its resource model by querying to its corresponding Prometheus server as shown in the oracle prototype architecture in figure 5.3. In the single owned centrally managed ”big cluster” this interaction allows to simplify the oracle prototype (figure 5.4) since all the workloads of the whole infrastructure are distributed by the Liqo scheduler instance on the cluster master.

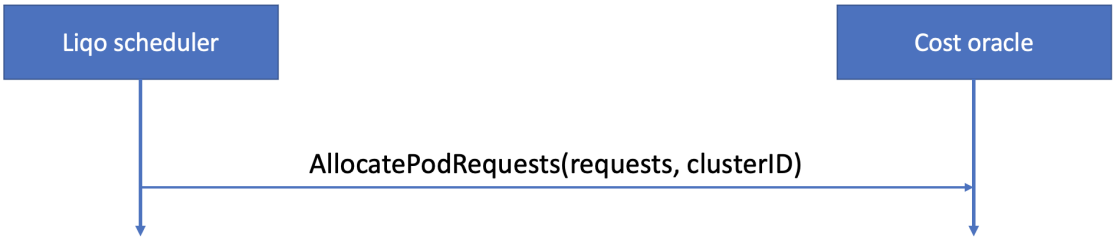

Figure 5.8. AllocatePodRequests interaction Liqo Scheduler-Cost Oracle in single-owned "big cluster"

Latency considerations

From the performance point of view we can distinguish the interactions for their latency: it depends on the location of the actors. In the multi-owned "big cluster" almost all the Liqo scheduler-Cost oracle interactions happen remotely except when their instances are on the same cluster (e.g. the Liqo scheduler asks to its cost oracle a Pod's cost). In the single-owned "big cluster" instead all these interaction are local and happen in the master cluster. The figure 5.6 shows the sequence diagram of the interaction in the multi-owned scenario, while the sequence diagrams of the interactions in the other case are shown in figures 5.7 and 5.8.

5.2.2 Oracle - Oracle

The second main kind of interactions is the one between different clusters' oracles and is depicted in figure 5.9. These interactions do not exist in the multi-owned "big cluster" scenario since each oracle does not share nothing with others. In a single-owned multi-cloud "big cluster" managed from a central point as the case of figure 5.1, at the peering moment between two clusters an interaction happens between the respective oracle in order to synchronize them. They exchange their cost model and their cost coefficients as well as their topology with the Cluster Autoscaler configuration. The cost model is the formula that will be used by the oracle to compute Pod costs at that cluster, the cost coefficients are the actual tariff values that have to be used in the model for the computation. The topology information contains the number and the capacity of the nodes of the cluster as well as the Cluster Autoscaler configuration if used. The CA configuration is composed of two parameters: the minimum and maximum number of nodes on in the cluster.

The information exchanged by the oracle can change over the time, e.g. the resources tariffs changes, so have to be refreshed. In order to keep them updated this interaction must be repeated periodically.

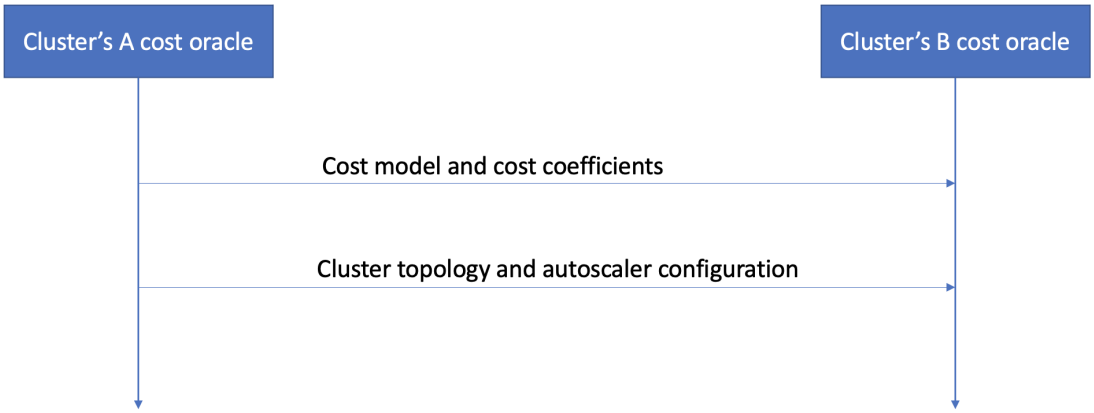

Figure 5.9. Interaction between two clusters' cost oracles

5.3 Cost oracle's cluster abstraction

A cost oracle abstracts the cluster by means of its resources model and its cost model. The first model tracks its nodes' resources capacity and their actual allocation. In order to do it, the oracle knows how many nodes the cluster is composed of and what is their hardware profile (e.g. CPU cores and amount of memory). Furthermore if the Kubernetes Cluster Autoscaler is enabled to handle nodes allocation, the oracle will know its configuration, that is the minimum and maximum number of nodes on in the cluster. Depending on the scenario, the oracle will track a cluster's resources allocation or via its Prometheus server (multi-owned case) or collecting the information from the scheduler instance (single-owned case). The cluster's cost model is a formula that can be applied to some resources requirements, such as Pod's requests, to compute the cost of allocating them at the specific cluster. In general each cluster has its own cost model: it will differ mainly depending it is hosted at a public provider or at an on-prem infrastructure. In order to make the formula applicable, there are some resources tariffs that can be retrieved and kept update over the time. Such tariffs are the resource's price in a conventional time unit.

Chapter 6

Implementation of the cost oracle

A prototype of the cost oracle has been implemented in a simulation scenario of the single-owned ”big cluster” case. This is a simple scenario in which the multi-cluster infrastructure is handled from a central point, the so called master cluster. The master cost oracle knows the cost models of each of the clusters of the topology. This prototype has been integrated within the Liqo multi-cluster scheduler and has been wrote in its same programming language, Go. In order to simulate the multi-owned ”big cluster” scenario, this prototype has been extended to support batch processing of the requests coming from the Liqo scheduler.

6.1 The cost model tree

The prototype handles the cluster's cost model as a tree structure, an example is given in figure 6.1. In the cost model tree:

- non-leaf nodes are aggregating components, such as ”Sum” or ”Product”;

- leaf nodes are the conventional name of the formula coefficient, described in the table below.

Coefficient name	Meaning	Unit of measure
c_{VM}	Virtual Machine instance tariff	USD/h
FPN	First Pod on Node	adimensional
$c_{storageclass}$	Storage class tariff	USD·GB/h
vol_{size}	Storage volume size	GB
networkTrafficCost	Outgoing network traffic cost	USD

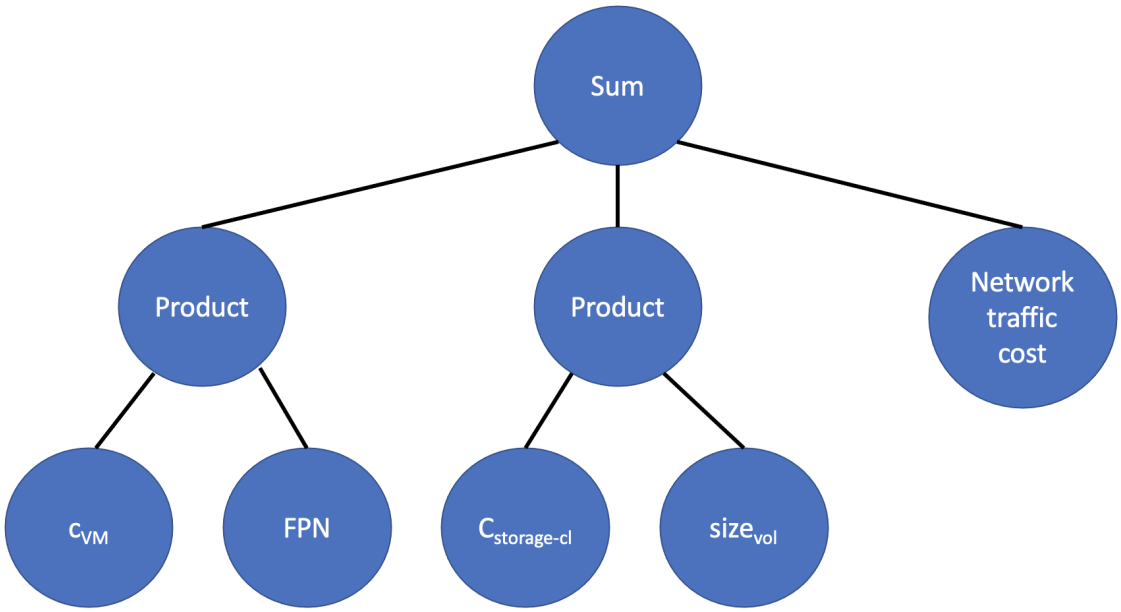

Figure 6.1. Cost model tree in the cost oracle implementation

FPN (First Pod on Node) is a special cost model element: it can have as value 1 or 0 depending on whether the Pod of the computation will cause the powering on of a new node or not. There is another special element: the NetworkTrafficCost. This is interpreted as the scalar product of the cost model between the outgoing network traffic vector and the outgoing network traffic cost. The cost coefficients starting with "c" are substituted in the computation with their provided value in the coefficients data section. Coefficients can be of two kind: scalar (as the VM hourly cost in the example) or vectorial (e.g. the storage volume cost are given for each different storage class).

6.2 Node powering on prediction

The cost oracle tracks each cluster's resource allocation thanks to the "AllocatePodCost" interaction with the scheduler. When a Pod cost computation is requested, it must understand if the FPN coefficient is one or zero to charge or not the VM instance cost. Let's have for each cluster some vectors with as many components as the number of kind of resources is; in the following two: one for the CPU and one for the memory.

$$\boldsymbol{A} = \begin{bmatrix} a_{CPU} & a_{memory} \end{bmatrix}$$

$$\boldsymbol{C_{node}} = \begin{bmatrix} c_{CPU} & c_{memory} \end{bmatrix}$$

$\boldsymbol{A}$ is the cluster allocation vector: it contains for each component the overall amount of the corresponding resource allocated in the cluster.

$\boldsymbol{C_{node}}$ is the node capacity vector: each component contains the maximum allocatable amount of the corresponding resource in a cluster's node.

$$\boldsymbol{E} = \begin{bmatrix} e_{CPU} & e_{memory} \end{bmatrix}$$

$\boldsymbol{E}$ is the vector of the nodes number estimates, where:

$$e_i = \left\lfloor \frac{a_i}{c_i} \right\rfloor$$

The estimate of the number of powered on nodes in the cluster is computed as the greater component among those in $\boldsymbol{E}$.

When the scheduler asks a Pod cost computation, it provides its resource requests (the $\boldsymbol{R}$ vector):

$$\boldsymbol{R} = \begin{bmatrix} r_{CPU} & r_{memory} \end{bmatrix}$$

The oracle performs first the previously described computation to estimate the number of nodes on in the cluster. Then it performs a second computation with a different nodes number estimates vector $\boldsymbol{E'}$:

$$\boldsymbol{e'} = \left\lfloor \frac{a_i + r_i}{c_i} \right\rfloor$$

The last computation yields the nodes number estimate considering also Pod's requests: if it is greater than the previously computed value then FPN will be 1, otherwise the Pod will not be charged for the node's cost.

Chapter 7

Evaluation

The models proposed has been validated by using the instruments given by cloud providers to compute the cost of a cluster and by means of some simulations. The cost oracle prototype has been evaluated to understand what is its impact over the Liqo scheduler performance and how the Pods distribution across the clusters change with it.

7.1 Cost models validation and considerations

The cluster cost model proposed actually fits how cloud providers charge the virtual infrastructure: a comparison has been done by using the formula proposed and the cloud providers' price calculators[15][16][19]. The provisioning mode of the comparison is VM-as-node, since it is the case addressed in this thesis.

A simulation has been done in order to understand what is the weight of each of the cluster's components on the overall cost in a realistic scenario. The application considered is the Google's Online Boutique, which is an example of micro-services application architecture. It is composed of twelve Pods, each with a CPU requests between 0.07 and 0.3. A matrix contains the network traffic estimates for each pair of Pods. In the simulation these Pods have been distributed over two clusters in four different scenarios:

1. both the clusters at AWS in region Europe/Paris;
2. both the clusters at AWS, one in Europe/Paris, one in Europe/London;
3. both the clusters at AWS, one in Europe/Paris, one in US/West(Oregon);
4. a cluster at AWS in Europe/Paris, a cluster at GCP in Asia East 2.

Having more than one cluster for the simulation was mandatory since otherwise network traffic costs would be always null. Each cluster is composed of three nodes with an hardware profile of 2 CPU cores and 8 GB of memory. The simulation involves executing for one week the application with Pods equally distributed between the two clusters in each scenario. The figure 7.1 shows the overall cost of the two clusters in the four scenarios. The most interesting result is that network traffic costs are smaller than nodes' costs by three to four orders of magnitude. Slightly more impacting are the control plane costs: since they are constant become less important by increasing the number of nodes in the clusters.

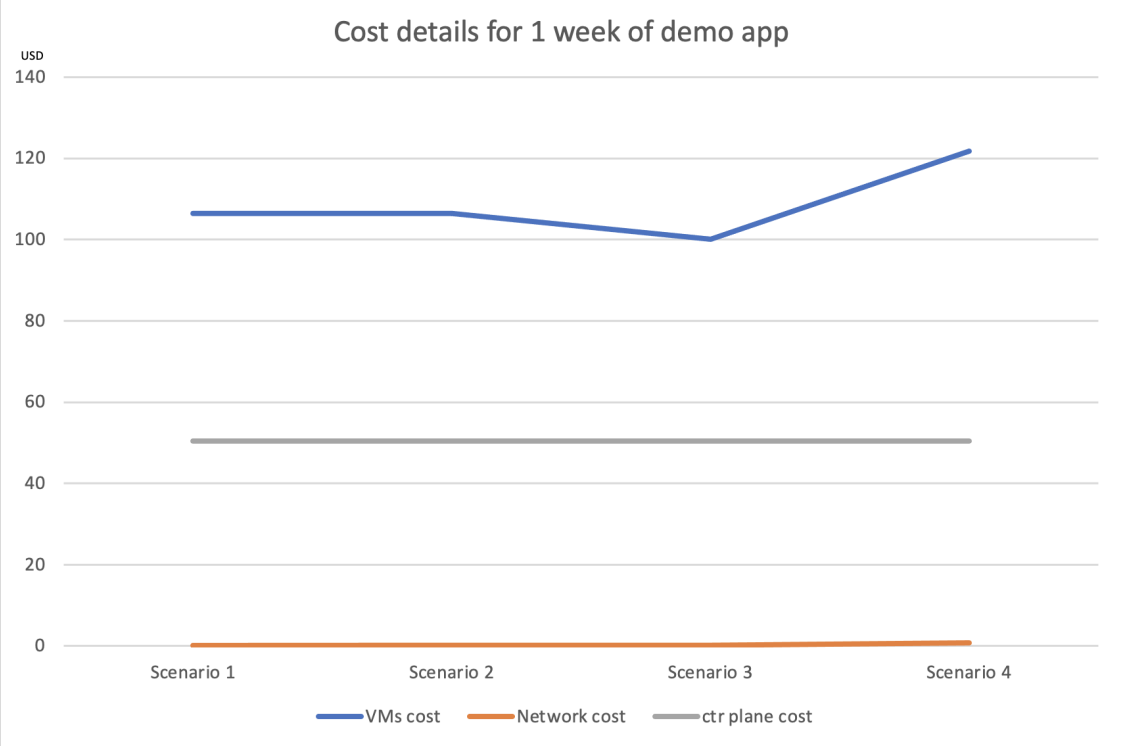

Figure 7.1. Cost of two clusters executing the demo application

Another simulation has been performed to evaluate the aggregate cost allocation with the passenger model, by progressively allocating Pods over a cluster. Such simulation rely on the usage of the cost oracle prototype as passenger model implementation. The cluster is composed of 20 nodes each with 8 CPU cores and 8 GB of memory. The Cluster Autoscaler is enabled and configured to scale nodes from a minimum number of 1 to a maximum

number of 20. The application is composed of a single Pod with as requests 1 CPU core and between 400 MB and 800 MB of RAM. In order to saturate the cluster's resources (actually only the CPU), 100 applications have been allocated. The figure 7.2 shows the aggregated cluster cost expressed in USD. On the x-axis there is the number of allocated applications. The result is exactly the one expected: the cluster cost increases with some steps. There is a step increment each time a new node is powered on. In this test there are no network costs because the Pods are located on the same cluster, in this situation the outgoing traffic is free.

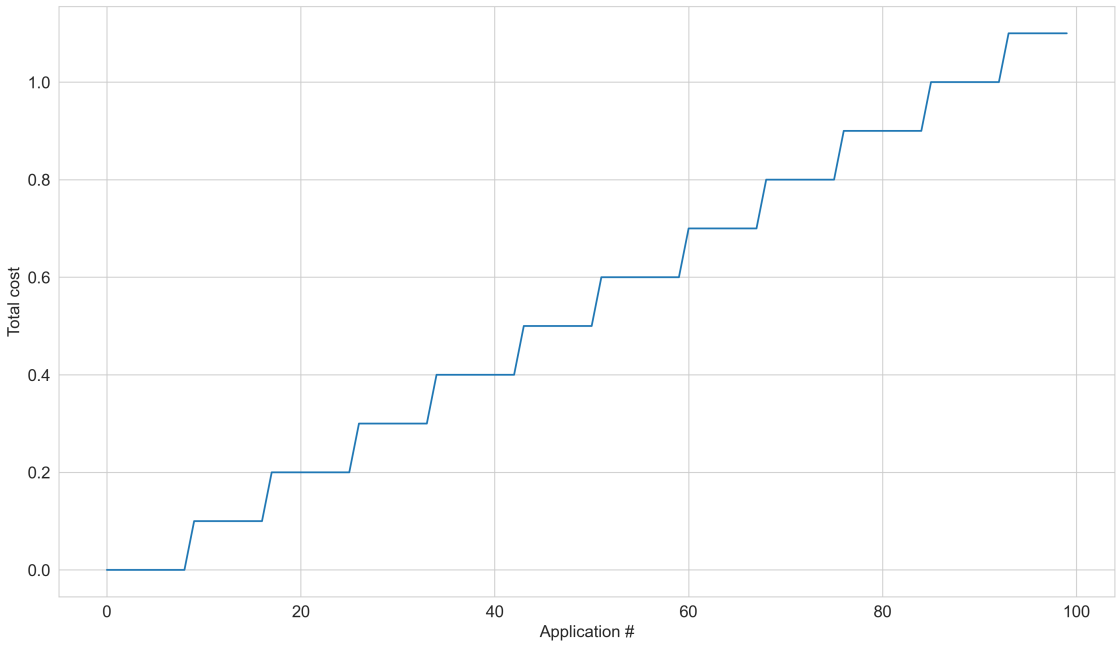

Figure 7.2. Passenger model's allocation cost trend

7.2 Cost oracle prototype evaluation

The cost oracle prototype integrated within the Liqo scheduler has been evaluated in a simulation scenario, where the multi-cluster topology and the applications to be scheduled are abstracted by some data. In the following tests an instance of the scheduler has been executed within a Docker container on a physical machine with the following hardware capabilities:

- Intel Xeon CPU E3-1245 v5 @ 3.50GHz with 8 logical cores and Intel

Hyper-Threading technology, x86_64 architecture;

- 64 GiB of DIMM DDR4 memory: two 16 GiB banks @ 2400 MHz, two 16 GiB banks @ 2667 MHz.

From the Liqo scheduler point of view, the multi-cluster topology is modeled as a graph where:

- the node represents a cluster, seen as an aggregate of hardware resources such as CPU cores and amount of RAM;
- the edge represents a network link between two clusters with a fixed bandwidth capacity.

The cost oracle knows more details about the cluster:

- the provider, the region and the availability zone it is in;
- its topology, that is the number of nodes it is composed of and their hardware profile;
- its cost model, that is the formula to be used to compute costs;
- the cost coefficients that represents the resource tariffs (e.g. the node hourly cost).

The provider, region and zone are needed to retrieve from the network traffic cost matrix the actual outgoing traffic cost. The topology includes if the Kubernetes' Cluster Autoscaler is enabled for the cluster: if so the number of nodes will change to fit the actual workload.

The application is modeled as a graph as well where:

- the node is the micro-service's Pod with its hardware resources requests that will be allocated for it;
- the edge represents the interconnection between two Pods, it is the estimate of the network traffic they exchange.

7.2.1 Multi-owned multi-cluster infrastructure test

This test aims at simulating the scenario described in the design chapter where the multi-cluster infrastructure is made up of clusters owned by different entities. Such entities want to share their resources but do not want

to share detailed information such as the topology (the number and profile of nodes in the cluster) and the cost model. Each cluster has its own cost oracle instance responsible of building and maintaining the cluster's resource model and cost model. When a Liqo scheduler instance is asked to schedule an application, it must interact with all the oracles of the clusters in the topology in order to collect the Pods price quotations.

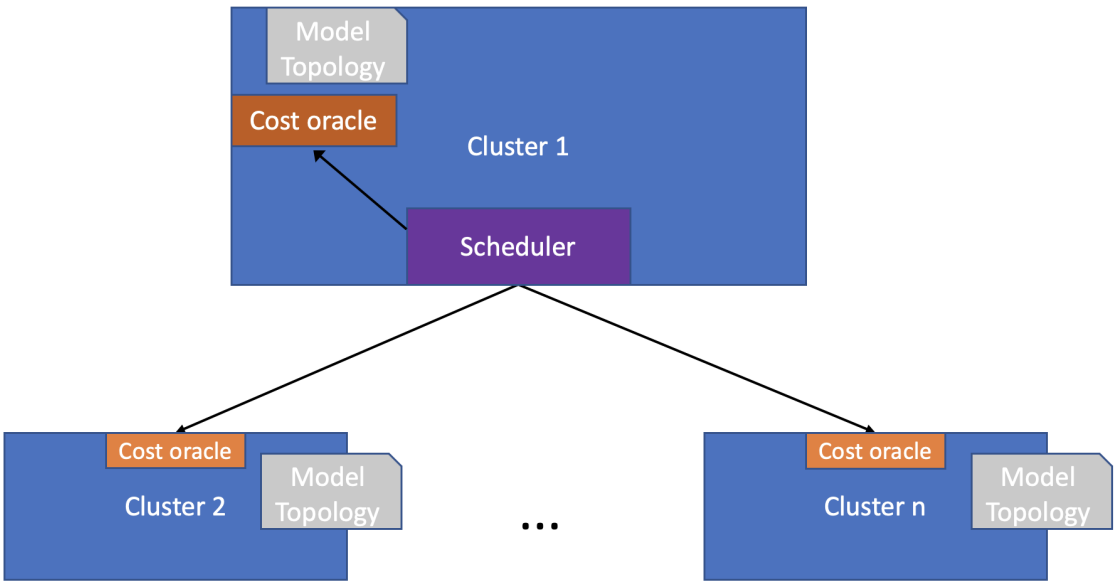

Figure 7.3. Architecture of the multi-owned multi-cluster test

The figure 7.3 shows the architecture of the test: the perspective is that of the Cluster 1's Scheduler instance that receives the application scheduling requests. It will distribute applications' components (the Pods) in the multi-cluster topology. It will interact with each cluster's cost oracle instance to retrieve Pods price quotations: these interactions are remote so they involve some latency time. When the scheduler interacts with the cost oracle instance of its own cluster (Cluster 1 in the figure) the latency time is neglected. As the figure shows each Cost oracle instance keeps its cluster's cost model and topology. The test aims at providing Cluster 1's scheduler instance some application scheduling requests in five different topologies measuring the time spent in interacting with the Cost oracles in the topology and the overall scheduling time. A cluster is modeled as the following:

- it aggregates 320 CPU cores and 640 GB of memory;

- it is hosted at AWS in a randomically chosen European region;
- it is composed of 5 nodes with hardware profile of 64 CPU cores and 128 GB of RAM;
- the Cluster Autoscaler is enable and configure to scale the nodes number from a minimum of 1 to a maximum of 5.

An application is modeled as:

- made up of 10 Pods;
- each Pod requests between 0.9 and 1 CPU core and between 200 MB and 400 MB of memory;
- the probability of two Pods interacting each other is 0.3 with a bandwidth request between 160 KB/h and 8000 KB/h.

The test is done by scheduling 100 applications in five different topologies. The topologies are composed of 5, 10, 25, 50 and 100 clusters each composed as described previously. The latency time between the Scheduler and the oracle instance is randomically computed with values between 20 ms and 50 ms. Latency times are in this range according to some measurements done in a technical paper[18] between a domestic client and different servers hosted on European public data-center locations, with 1 kB as request size. The interaction with the oracle on the same cluster implies no latency. The goal of this test is to compare the scheduling and the scheduler-oracle interaction times as the number of clusters in the topology grows. The figure 7.4 show the results of this test.

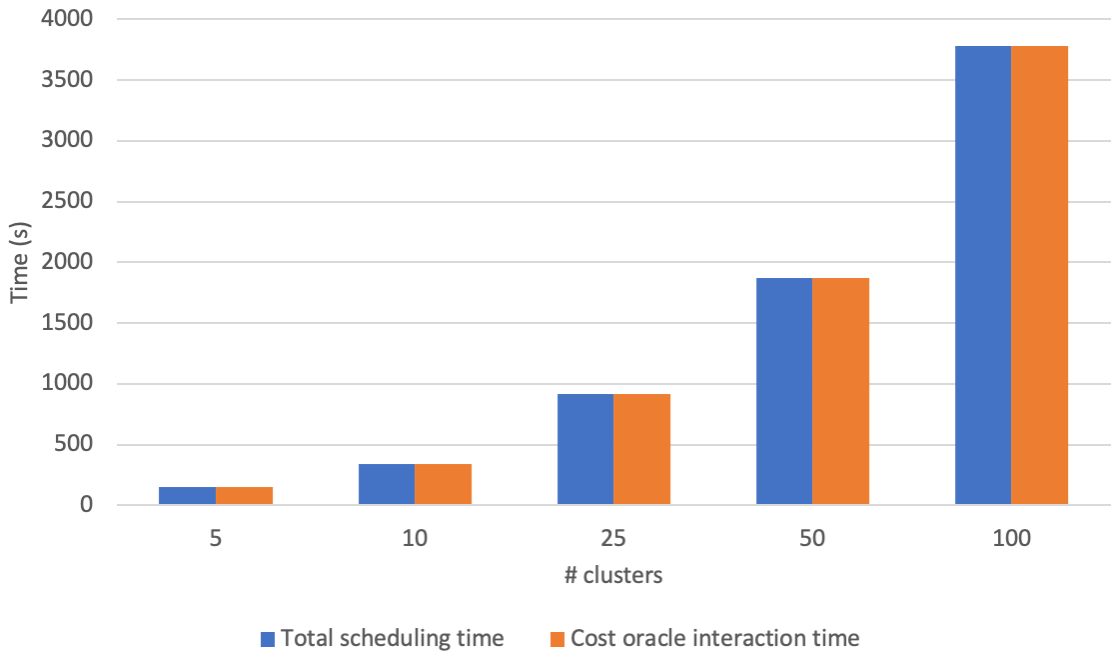

Figure 7.4. Scheduling and Scheduler-Oracle interaction times in the multi-owned scenario

In all the infrastructures we can see that the total scheduling time is in the order of minutes. The measures grows linearly with the number of clusters. This results show that having remote interactions during the scheduling process is very limiting from the performance perspective. Waiting almost 60 minutes to schedule 100 applications in a topology of 100 clusters is unacceptable. So some considerations are needed on how the efficiency can be improved. For example the precision of the price quotations can be degradated by caching a result and reusing it the following times building some simple local model. The key of this modification is finding the right balance between performances and costs precision. This test highlighted that implementing the design proposed in this particular scenario is not a good idea without some extra modifications. So from now on the focus will be on the evaluation of the cost oracle's design in the other described architecture.

7.2.2 Single-owned multi-cluster infrastructure tests

The single owned multi-cluster scenario has been tested in the simulation scenario. In this case the multi-cluster Scheduler instance interacts just with the

master cost oracle's instance that knows and maintains the resource and cost models of all the clusters in the topology. Such interaction happens locally to a cluster so no latency time is considered. The multi-cluster infrastructure used in the following is:

- there are 100 clusters connected each other through a 40 G bandwidth link;
- each cluster has 320 CPU cores and 640 GB of RAM;
- the clusters have the Cluster Autoscaler enabled that varies the number of nodes between 1 and 5;
- a cluster's node is composed of 64 CPU cores and 128 GB of RAM;
- the clusters' provider is AWS;
- the clusters differs only for their region and zone, assigned randomically to simulate the geographical distribution across all the available European locations;
- the cluster's node hourly tariff is the actual one in the region assigned.

The application used in the following is:

- an application is composed of ten Pods;
- each Pod requires a randomically quantity of CPU cores between 0.4 and 2 and of RAM between 512 MB and 1 GB;
- the probability of two Pods interacting each other is 0.3 with a bandwidth request between 16 KB/h and 800 KB/h.

These two test have been performed twice: in the case of the cost oracle prototype integrated within the Liqo scheduler and in the case of not.

Half filled infrastructure tests

This test aims at scheduling 1200 applications over a multi-cluster infrastructure to fill about half of the available resources. These conditions allow to understand how the cost oracle influences the scheduling process.

The figure 7.5 shows the aggregated allocation cost (expressed in USD/h) for each scheduled application in the test with the cost oracle integrated in the scheduler and without it. An important note: the absolute value of

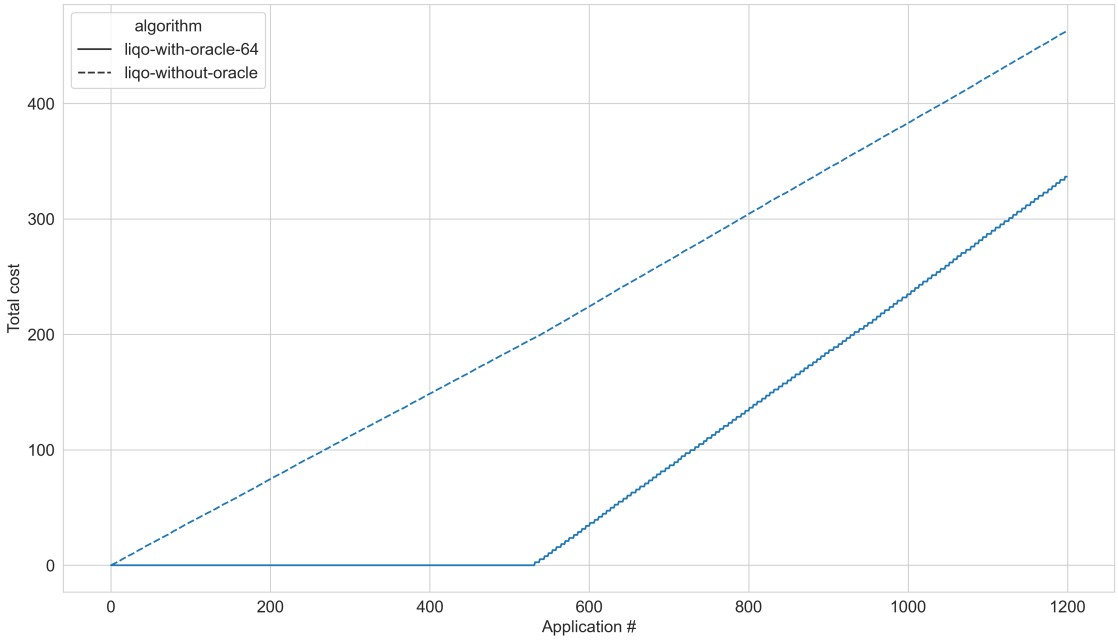

Figure 7.5. Allocation cost trend of the half filled test with and without the cost oracle

the costs is not comparable between the two cases. The reason is that the scheduler without the oracle uses a simple slice cost model, that is it applies some per-resource tariffs to the Pods requests. That cost is not real since VM are charged entirely for their allocation time and not for their actual usage. Indeed the cost oracle has been designed to compute precisely the real cost of Pods taking into consideration the real cost model. So the only thing that can be compared between the two curves is the trend. Without the oracle, the allocation cost grows linearly from the first to the last scheduled application, its slope is due to the allocated resources tariffs. With the oracle, the allocation cost has a very different trend: we can observe that the cost is almost null for the first 520 applications, then it grows with a step trend. Initially the cost is zero for clusters Autoscaler configuration: it is set to power the nodes between a minimum of 1 to a maximum of 5. This means that at the time zero, each cluster has preallocated exactly one node whose cost is considered already paid. After about 520 applications have been scheduled, the preallocated resources have run out: from that point on in order to schedule Pods new nodes have to be powered on. Every time a Pod has required the allocation of a node the cost increases of the node's cost

with a step trend. These two different cost curves have different impact on the scheduling process.

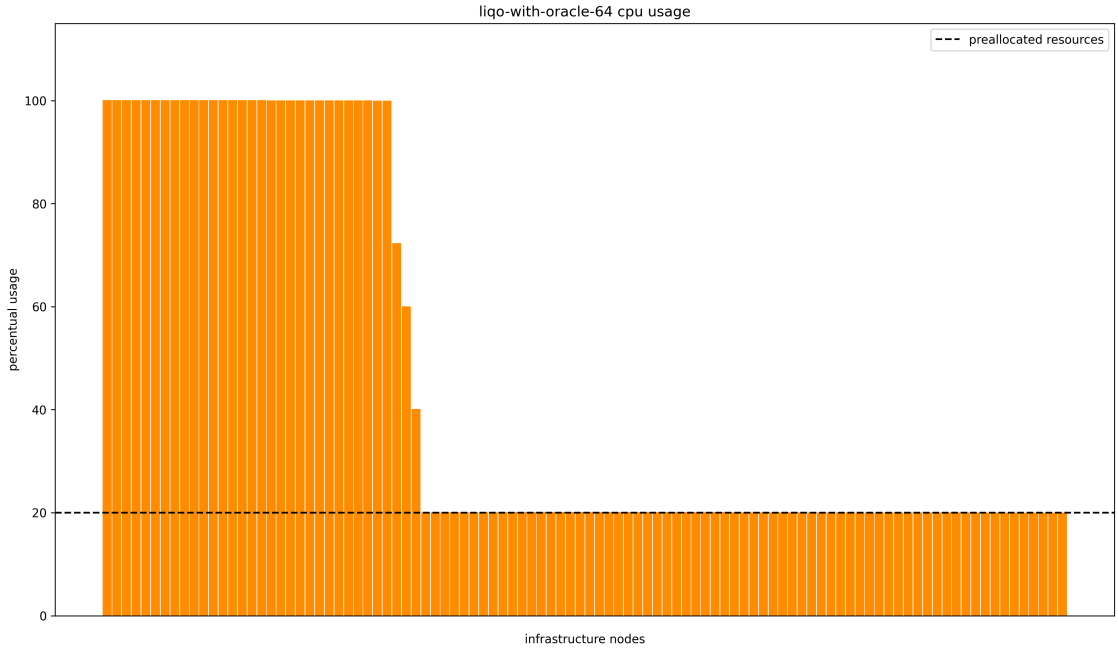

Figure 7.6. CPU allocation of the half filled test with the cost oracle

The figure 7.6 shows how the CPU allocation of all the clusters in the topology when the scheduling process is over. There is a bar of each cluster in the topology that represents the percentage of CPU allocated. The choice of showing the CPU allocation is because it is the most loaded resource with the applications requests. The black dashed line indicates the amount of preallocated CPU percentage in each cluster due to the Cluster Autoscaler configuration. We can see that the scheduling process has actually exploited these ”free” resources in the initial phase: Pods have been distributed uniformly amount all the clusters in the topology. When preallocated resources are full of load, new nodes have been allocated starting from the cluster where the hourly tariff is more convenient on.

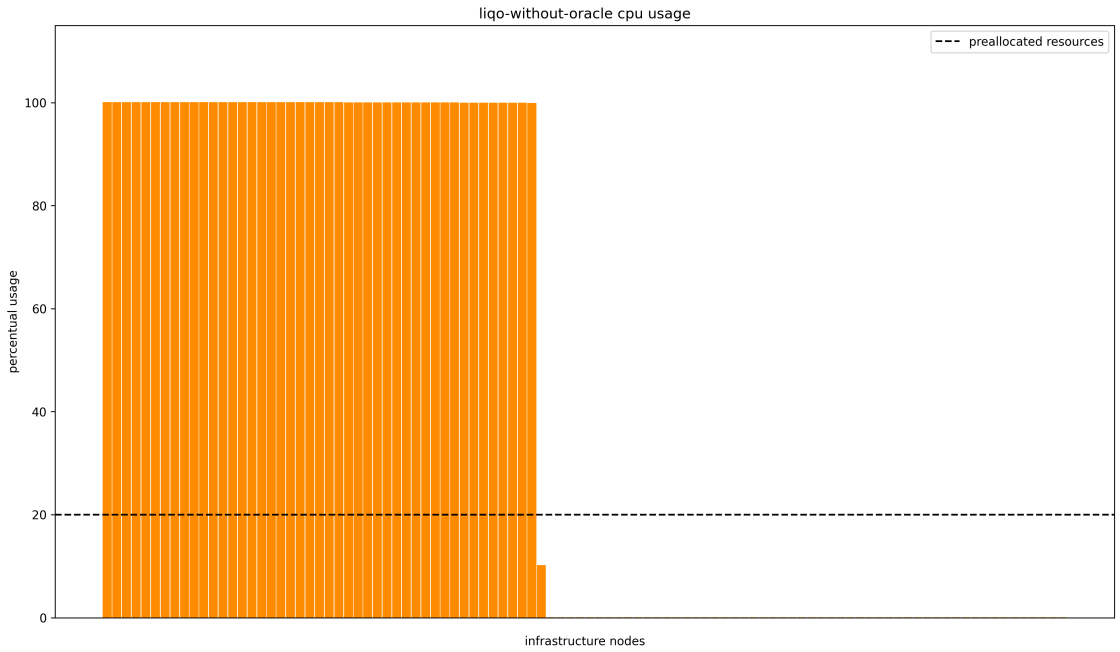

Figure 7.7. CPU allocation of the half filled test without the cost oracle

Figure 7.7 shows the CPU allocation distribution in the test involving the Liqo scheduler without the cost oracle. In this case we can see that the distribution is more consolidated. Pods are scheduled starting from the cluster with lowest resource tariffs on, regardless of the fact that some resources are already preallocated in the clusters. The reason is that in this case the cost model is stateless.

Fully filled infrastructure tests

This test differs from the previous one for the number of applications scheduled over the multi-cluster topology. 3000 applications as the one described previously have been scheduled over the 100 clusters infrastructure in order to fully fill all the available resources. This tests show the performance impact of the cost oracle over the scheduling process. Each test has been repeated 10 times in order to have more realistic time values.

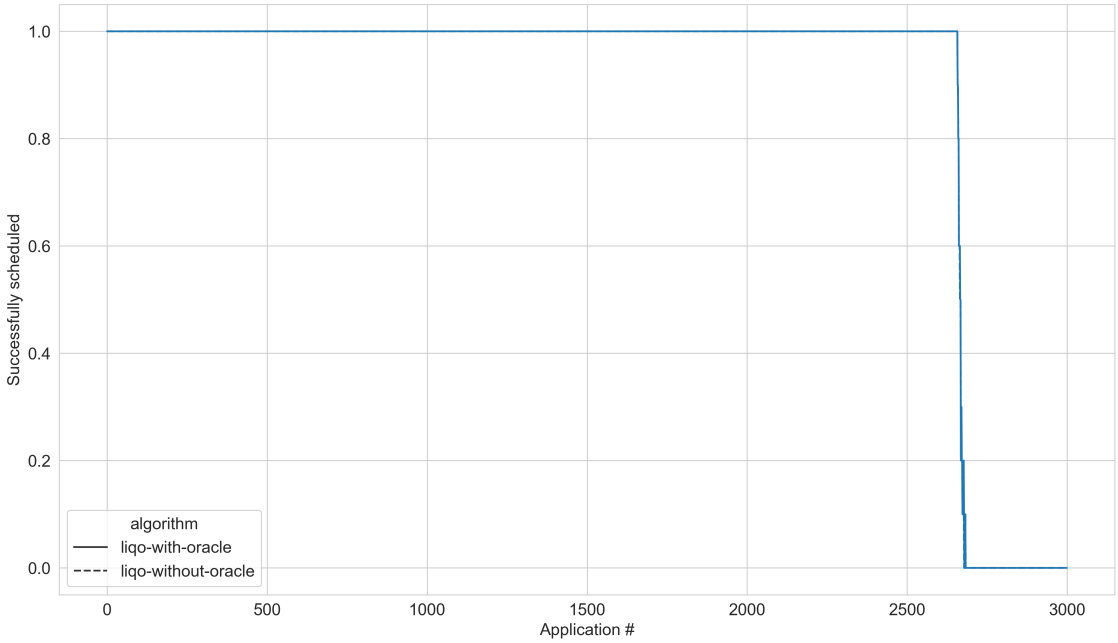

Figure 7.8. Scheduling success percentage of the fully filled test with and without the cost oracle

The figure 7.8 shows the scheduling success rate for the provided 3000 applications in the case of the cost oracle integrated in the scheduler and in the case of not. An application is considered successfully scheduled if and only if all the Pods it is composed of have been successfully scheduled. In the figure the value on the y axis is the count of runs in which the application on the x axis has been successfully scheduled divided for the times the test has been repeated. The scheduling fails when the scheduler does not find a solution for a Pod: this happens when the available resources in the clusters are not enough to allocate its requests. The figure shows that the applications success rate in the two cases are almost identical. This tells us that there is no difference on the scheduling success rate in presence of the cost oracle or not.

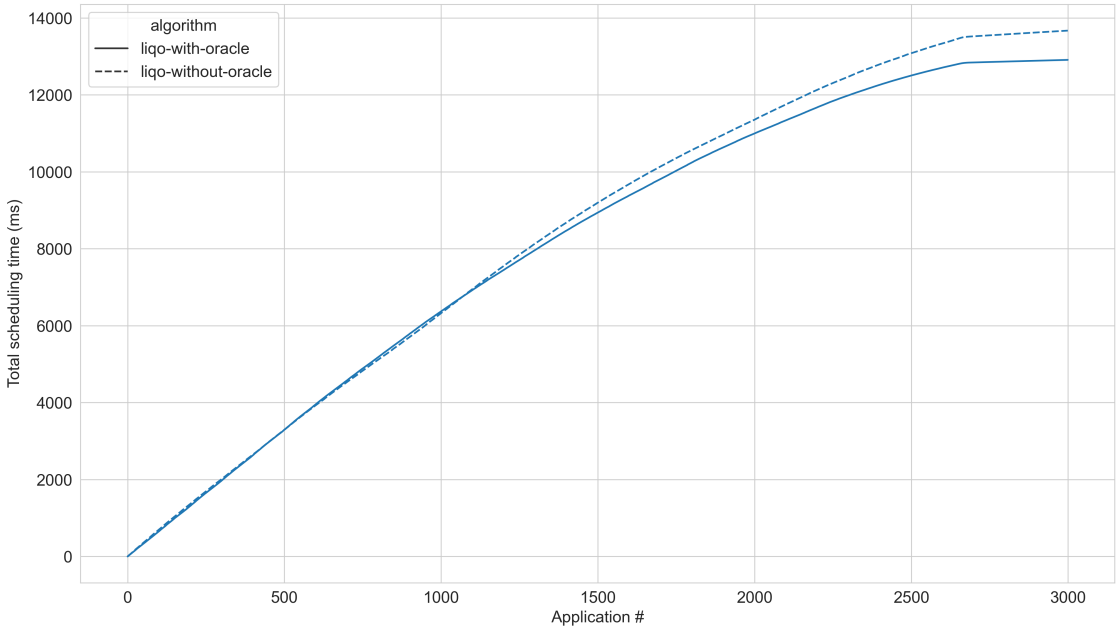

Figure 7.9. Aggregated scheduling time of the fully filled test with and without the cost oracle

Figure 7.9 shows that the aggregated scheduling time spent by the Liqo scheduler with and without the cost oracle. The curve trends are very close in the first 1100 scheduled applications. From that point on a slight gap appears between the two times: the scheduler with the cost oracle integrated performs slightly better than the other. This tells us that the overhead of having such component is negligible (actually null) with respect to relying on the scheduler's slice cost model. The difference can be explained by the fact that the applications components are distributed differently over the infrastructure. The distribution obtained with the cost oracle leads to situations in which scheduler's used heuristics more frequently allow to optimize scheduling times.

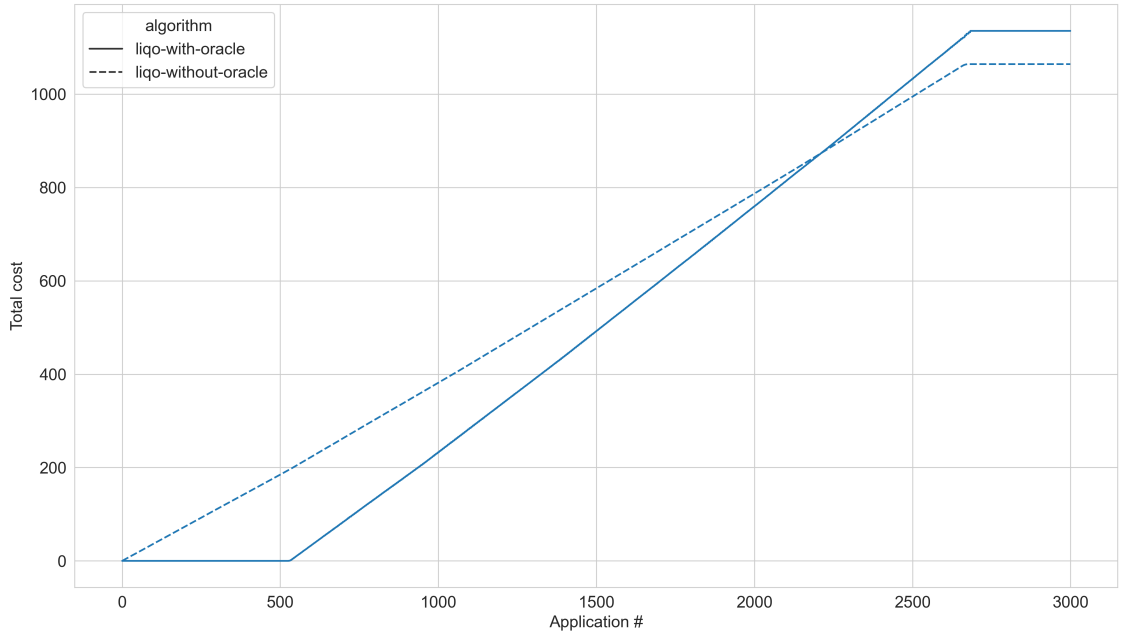

Figure 7.10. Aggregated allocation costs of the fully filled test with and without the cost oracle

The figure 7.10 compares the aggregated cost of the scheduled applications expressed in USD/h. The absolute values on the two curves are not comparable because are computed with two completely different cost models. What can be compared are the different trends in the two cases and the consequences on the applications components distribution. Until about the 520th application the allocation cost with the cost oracle is about null because the scheduler is exploiting the cluster's preallocated nodes, as defined in the Cluster Autoscaler configuration. So in this first phase Pods are uniformly distributed across all the clusters in the test with the cost oracle integrated. This is the main difference in the load distribution that then impacts over the scheduler's heuristics different success rate. From the 520th application on the two trends become similar: they are linear (actually with some small steps growth trend in the case with the cost oracle).

Chapter 8

Conclusions and future works

Companies are increasingly investing in hybrid cloud and multi cloud. These solutions allow them to deploy services in a much more resilient way and at the same time can increase their geographical presence, getting closer to the customers. At the same time, diversifying the infrastructure can have room for cost optimizations.

The thesis work is started analyzing the costs associated to having a cluster at a public provider and at an on-premise data center. Public cloud providers charge for cluster resources allocation, in particular for the Kubernetes control plane deployment, for the VMs acting as nodes and for persistent storage volumes plus some networking related costs. On-premises cloud costs have been identified in all the phases of the infrastructure, from the design to the upgrade, passing through the build, the deploy and the maintain phases. Some formulas have been proposed to compute the overall Kubernetes cluster cost by summing up all the partial cost components in the two cases.

A focus has been made on modeling how Pod's cost can be computed. Two approaches have been proposed: a simpler stateless slice cost model, charging the Pod only for the slice of the resources allocated to him; and a stateful passenger cost model, charging the Pod for the whole resource it causes the allocation according to the real providers' pricing model.

Then the design of a solution for making the multi-cluster scheduling process costs aware, has been described with reference to the Liqo platform. The solution is based on the introduction of a component, the cost oracle, implementing the passenger Pod cost model. Each cluster has its own cost oracle,

configured with its detailed topology, cost components and tariffs. The cost oracle interacts with the multi-cluster Liqo scheduler to provide it the price quotation of scheduling the Pod described in the request, in the cluster it manages the cost model.

The architecture is slightly different for two analyzed scenarios:

- a first one in which all the clusters share their cost model and topology with a special cost oracle, located in a cluster called "master" that acts as the centralized point of usage of the multi-cluster infrastructure;
- a second one in which the clusters does not share their cost model and topology, so the Liqo scheduler must interact with all of them to perform its job.

The two designs have been implemented in a simulation scenario, where the clusters and the applications are abstracted by means of the data needed by the scheduler and the cost oracle.

Several simulation have been made in order to evaluate the cost oracle prototype performance and its impact on the scheduling decisions.

The simulation of the design that involves remote interactions between the scheduler and the cost oracles has shown that the overhead caused by the latency is too penalizing for a scheduling context. This result suggests that the design proposed must be revised to be applied effectively in that particular scenario.

On the other hand, the simulation of the design with only local interactions scheduler-cost oracle has shown good results. The cost oracle has changed the Pods distribution, allowing the exploitation of all the clusters' preallocated resources before starting allocating new nodes from the clusters with more convenient tariffs on. The implementation improves the cost efficiency of the application distribution and at the same time does not introduce a performance overhead: on the contrary scheduling performances can even improve thanks to a better heuristics success rate.

Clusters and container cost models can be applied to wide scenarios and can lead to money savings.

Future works can involve the integration in the Liqo project of the cost oracle, implementing the design that gave better results. The Kubernetes cluster cost models proposed can be used to implement a simulator, capable of estimating the infrastructure cost and comparing it with other alternatives, such as hosting it on-premises or at a public cloud provider.

Bibliography

[1] Amazon Web Services, *Amazon EKS Pricing*
URL: https://aws.amazon.com/eks/pricing/

[2] Amazon Web Services, *Amazon EC2 Pricing*
URL: https://aws.amazon.com/ec2/pricing/

[3] Criticalcase, *Calculating the TCO: cloud vs on premise infrastructure*
URL: https://www.criticalcase.com/blog/calculating-the-tco-cloud-vs-on-premise-infrastructure.html

[4] Kenny & Company, *Evaluating the Total Cost of Ownership for an On-Premise Application System*
URL: https://michaelskenny.com/points-of-view/evaluating-the-total-cost-of-ownership-for-an-on-premise-application-system/

[5] Kubernetes, *Kubernetes Documentation*
URL: https://kubernetes.io/docs/home/

[6] Github, *Liqo*
URL: https://github.com/liqotech/liqo

[7] Github, *cAdvisor*
URL: https://github.com/google/cadvisor

[8] Github, *kube-state-metrics*
URL: https://github.com/kubernetes/kube-state-metrics

[9] Github, *Kubecost cost model*
URL: https://github.com/kubecost/cost-model

[10] Github, *Cloudinfo*
URL: https://github.com/banzaicloud/cloudinfo

[11] Github, *Kubernetes Cluster Autoscaler*
URL: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler

[12] Google Cloud, *Preemptible VM instances*
URL: https://cloud.google.com/compute/docs/instances/preemp

tible
[13] Google Cloud, *Google Kubernetes Engine Pricing*
URL: https://cloud.google.com/kubernetes-engine/pricing
[14] Google Cloud, *Google Compute Engine Pricing*
URL: https://cloud.google.com/compute/all-pricing
[15] Amazon Web Services, *AWS Pricing Calculator*
URL: https://calculator.aws/#/createCalculator/EKS
[16] Google Cloud Platform, *GCP Pricing Calculator*
URL: https://cloud.google.com/products/calculator
[17] Liqo, *Liqo Docs*
URL: https://doc.liqo.io/
[18] Marco Iorio, Fulvio Risso, Claudio Casetti, *When Latency Matters: Measurements and Lessons Learned*
URL: https://giorio94.github.io/papers/05-WhenLatencyMatters.pdf
[19] Microsoft Azure, *Azure Pricing Calculator*
URL: https://azure.microsoft.com/en-us/pricing/calculator/?service=kubernetes-service
[20] Microsoft Azure, *Azure Kubernetes Service Pricing*
URL: https://azure.microsoft.com/en-us/pricing/details/kubernetes-service/
[21] Microsoft Azure, *Azure Linux Virtual Machines Pricing*
URL: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/