POLITECNICO DI TORINO

Master degree course in Computer Networks

Master Degree Thesis

Enabling Job Aware Scheduling on Kubernetes Cluster

Supervisors
prof. Fulvio Giovanni Ottavio Risso

Candidates
Stefano Galantino
Student ID: 255314

Academic Year 2019-2020

Contents

Chapter 1

Introduction

At the beginning of Cloud the concept of "monolithic" application was the predominant development pattern for web services; applications were developed and distributed as a single entity, self-contained, and independent from other computing applications.

Over the last decade we have noticed a deep change in the development of web based application; developers now can benefit of the massive computational power of data-centers and cloud environment by leveraging on microservices and containerization.

Now if we consider the modern world of microservices we deal with two kind of users:

- end-users, which expect applications to be available 24/7
- developers, which deploy new versions of their applications several times a day in order to make their service as reliable as possible

In this scenario containerization helps to package software, enabling applications to run unmodified in a wide range of Linux distributions and to be released and updated in an easy and fast way without downtime. Kubernetes, as a microservice orchestrator, helps developers to make sure those containerized applications run in cloud environment, providing all the resources and tools they need to work properly.

With the advent of 5G and Edge Computing a lot of solutions are gaining more and more popularity every day by extending Kubernetes behaviour also to the edge of the network. In this "new" concept of datacenter a lot of new challenges need to be addressed in order to provide the best possible performance for the end user; this happens

because there are a lot of additional constraints to take in account.

1.1 Goal of the thesis

This new paradigm of containerization and microservices while reducing the effort for developers, it increased the complexity for Cloud providers and in particular for Cloud orchestration platforms.

Dealing with microservice life-cycle is very complex, this is why over the last decade a big effort has been put in research for solutions aiming to reduce the managing complexity of this development framework.

Within the context of the Computer Networks Group at Politecnico di Torino and the Liqo project [9] a big effort has been put in the development of a custom Kubernetes scheduler, capable of better performance in certain scenarios (compared to Kubernetes default one). The work of this thesis is meant to operate close to that custom scheduler by providing it a set of concise information, which describes the expected behaviour of a given microservice (this concept will be hereafter referred as Profiling). In this context profiling means infer the requirement and the communication pattern of a given microservice based on his previous executions in order to take the best scheduling decisions. The final goal of the profiling system is to improve as much as possible the Quality of Service perceived by the end user.

This thesis is structured as follows:

- **CH1-Introduction**. This chapter provides a brief introduction on both the current scenario for microservice orchestration and the reasons behind the decision to develop this thesis

- **CH2-Kubernetes**. This chapter is an introduction to Kubernetes as a container orchestrator, focusing mainly on the features that will be exploited in the work of this thesis, providing a general background for the concepts that will be presented afterwards

- **CH3-Profiling in cloud environment**. This chapter describes what is the state of the art for what concerns the profiling of microservices. It briefly introduce the most relevant papers, which already faced this research topic. This thesis is meant to be a part of the Liqo project, so the final section of this chapter will introduce it

- **CH4-Job profiling design**. This chapter describes the design choices behind the profiler developed in this thesis, analyzing first microservices common behaviour in production datacenter, moving then to the actual logical design of the system
- **CH5-Job profiling implementation**. This chapter starts from the logical implementation described in the previous chapter and provides a more practical description of the actual implementation of the algorithm
- **CH6-Experimental evaluation**. This chapter evaluates the performance of the profiling algorithm described previously in a real scenario
- **CH7-Conclusion and future work**. This final chapter starts from the results obtained in the previous chapter and evaluates them, providing a critical analysis of the algorithm and some possible improvements

Chapter 2

Kubernetes

In this chapter we provide a brief introduction of containerization and Kubernetes as the leading solution for container orchestration. In particular we will focus our attention both on:

- Kubernetes architecture, describing the main components of the control plane
- Kubernetes resources, explaining the abstractions that the orchestrator can provide to developers

The chapter ends with an introduction of the main Service Mesh solutions that will be exploited for the purpose of this thesis.

Kubernetes is a huge and complex framework, so this chapter is not meant to be a complete description of all its functionalities, but we will focus mainly on the features that we consider the most relevant to understand properly the work of the thesis. For further details please refer to the official documentation [8].

2.1 Introduction

The story of Kubernetes began at Google: like no other, the company needed a huge infrastructure to make its search engine and the associated advertising available to all people. The enormous growth was a challenge for which various ideas arose. Virtualization of hardware, site reliability engineering and container technologies represent three essential pillars, which are vital for the subsequent solution.

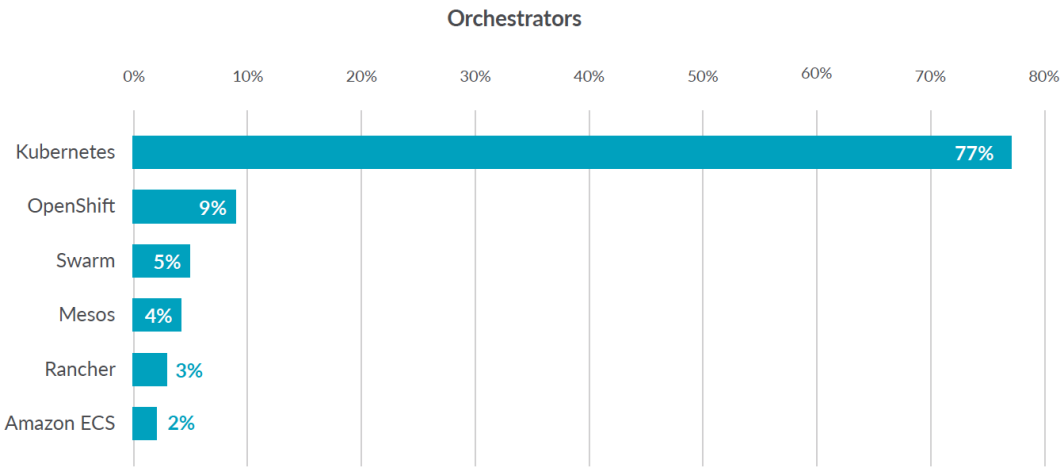

Figure 2.1. Kubernetes adoption (2019)

Google at the end relied on containers and their advantages in many ways. In order to manage (or orchestrate) containers sensibly, the employees developed the "Borg" project. Borg was an undisputed competitive advantage of Google because it utilized the machines much better than purely virtualized operations. Google strengthened its market position by being able to provide huge infrastructure landscapes at a much lower cost.

The big hit for the general public was the idea of making Borg available as an open-source solution. Kubernetes (Greek for helmsman; short: K8 or K8s) was born.

Containers are a good way to bundle and run applications. In a production environment, there is the need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start, replacing it.

That is how Kubernetes comes to the rescue, providing a framework to run distributed systems resiliently. It takes care of scaling and possible fail-over for applications, provides deployment patterns, and more.

More in detail the main features that made Kubernetes the leading solution for container orchestrator are:

- **Service discovery and load balancing**; Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.

- **Storage orchestration**; Kubernetes allows developers to automatically mount a storage system of many different kind, such as local storages, public cloud providers, and more.

- **Automated rollouts and rollbacks**; Kubernetes allows to describe the desired state for deployed containers, and the framework takes care of changing the actual state to the desired state at a controlled rate.

- **Automatic bin packing**; it is possible to provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. It is also possible to specify Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto nodes to make the best use of their resources.

- **Self-healing**; Kubernetes restarts containers that fail, replaces containers, kills containers that do not respond to user-defined health check, and does not advertise them to clients until they are ready to serve.

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

2.2 From Virtualization to Containerization

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Containerization is just the last step for servers resource management solutions. In the early days of what we now call Cloud, the main idea for application deployment was the "One application per server" rule, meaning that each server was meant to be used by a single application; this approach had indisputably some advantages but a the same time it showed a lot of disadvantages.

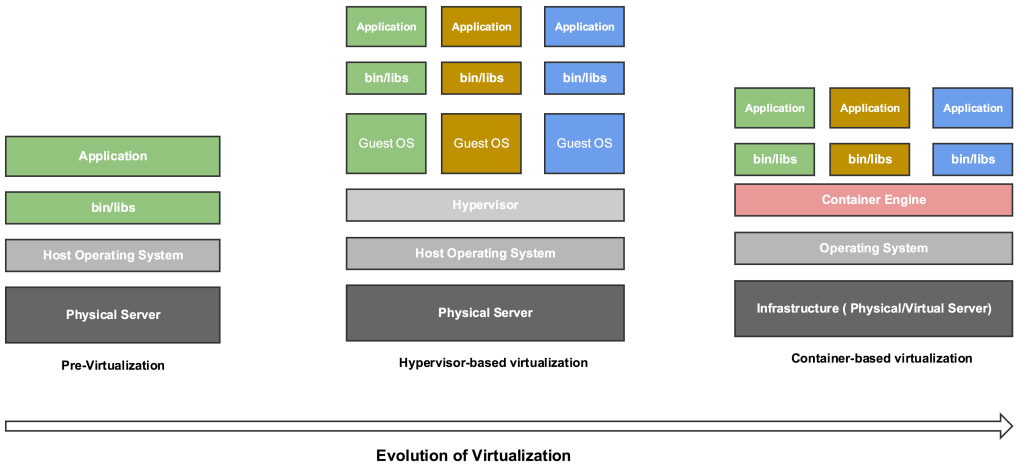

Figure 2.2. Application hosting technologies

This is why, over the years, many solutions were born and many other disappeared, trying to solve this problem of managing datacenters resources efficiently. In the following sections we will discuss briefly about the main families of solutions that over the years were introduced.

2.2.1 Pre-Virtualization Era

Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would under perform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

2.2.2 Hypervisor-Based Era

As a solution, virtualization was introduced. It allows to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed

by another application.

Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization it is possible to present a set of physical resources 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.

2.2.3 Container-Based Era

Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications; therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

2.3 Kubernetes Architecture

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

This section outlines the main components of Kubernetes control plane.

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied).

Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on

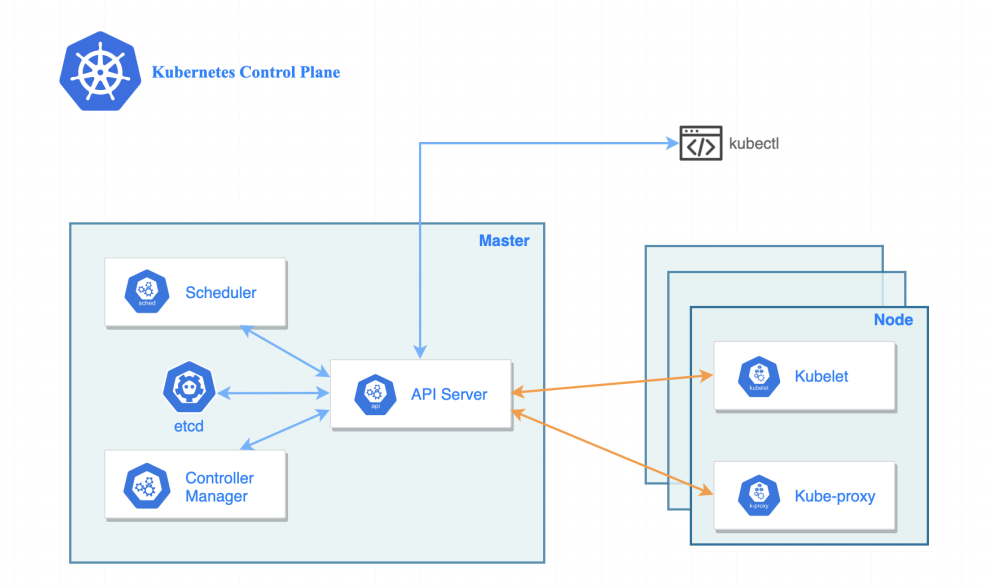

Figure 2.3. Kubernetes control plane

this machine.

2.3.1 Kube API-Server

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The Kubernetes API is the front end of the Kubernetes control plane, handling internal and external requests. The API server determines if a request is valid and, if it is, processes it. It is possible to access the API through REST calls, through the kubectl command-line interface (that internally translate the request into REST calls).

The main implementation of a Kubernetes API server is `kube-apiserver`. `kube-apiserver` is designed to scale horizontally—that is, it scales by deploying more instances. It is possible to run several instances of `kube-apiserver` and balance traffic between those instances.

2.3.2 Kube-scheduler

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

2.3.3 Kube-controller-manager

Controllers take care of actually running the cluster, and the Kubernetes `controller-manager` contains several controller functions in one. One controller consults the scheduler and makes sure the correct number of pods is running. If a pod goes down, another controller notices and responds. A controller connects services to pods, so requests go to the right endpoints.

These controllers include:

- Node controller: Responsible for noticing and responding when nodes go down.
- Replication controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
- Endpoints controller: Populates the Endpoints object (that is, joins Services and Pods).
- Service Account and Token controllers: Create default accounts and API access tokens for new namespaces.

2.3.4 Kubelet

An agent that runs on each node in the cluster. It is an application that communicates with Kubernetes control plane. The `kubelet` makes sure containers are running in a pod and when the control plane needs something to happen in a node, the kubelet executes the action.

The `kubelet` takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The `kubelet` does not manage containers that were not created by Kubernetes.

2.3.5 Kube-proxy

`kube-proxy` is a network proxy that runs on each node of the cluster, implementing part of the Kubernetes Service concept. `kube-proxy` maintains network rules on nodes, allowing network communication to Pods from network sessions inside or outside of the cluster.

`kube-proxy` uses the operating system packet filtering layer if there is one and it is available, otherwise forwards the traffic itself.

2.3.6 Container Runtime

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

2.4 Kubernetes Resources

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to 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 behave, such as restart policies, upgrades, and fault-tolerance

A Kubernetes object is a "record of intent", once an object is created, the Kubernetes system will constantly work to ensure that object exists. By creating an object, we're effectively telling the Kubernetes system what we want our cluster's workload to look like; this is the cluster's desired state.

Almost every Kubernetes object includes two nested object fields that govern the object's configuration: the object spec and the object status. For objects that have a spec, it is mandatory to set this at creation time, providing a description of the desired characteristics the resource should 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 supplied state.

For example: in Kubernetes, a Deployment is an object that can represent an application running on the cluster. When a Deployment is created, it is possible to set the Deployment spec to specify that three replicas of the application to be running are required. The Kubernetes system reads the Deployment spec and starts three instances of the desired application–updating the status to match the spec. If any of those instances should fail (a status change), the Kubernetes system responds to the difference between spec and status by making a correction–in this case, starting a replacement instance.

In the following sections we will present the most relevant Kubernetes objects.

2.4.1 Namespace

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. Namespaces provide a scope for names, because resources names need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.

Namespaces are a way to divide cluster resources between multiple users (via resource quota).

2.4.2 Pod

Pods are the smallest deployable units of computing that is possible to create and manage in Kubernetes.

A Pod (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage/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.

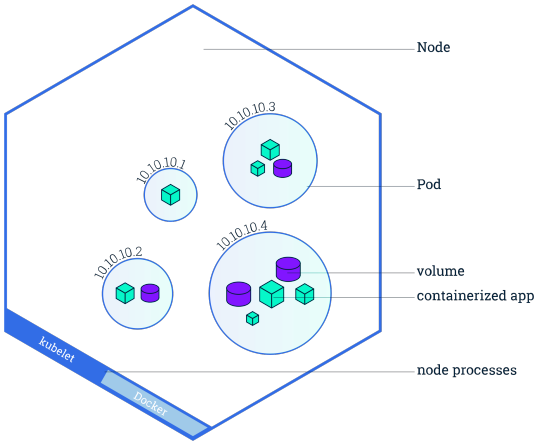

Figure 2.4. Kubernetes Pod

2.4.3 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.

A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.

A ReplicaSet is linked to its Pods via the Pods' metadata.ownerReferences field, which specifies what resource the current object is owned by. All Pods acquired by a ReplicaSet have their owning ReplicaSet's identifying information within their ownerReferences field. It is through this link that the ReplicaSet knows of the state of the Pods it is maintaining and plans accordingly.

2.4.4 Deployment

A Kubernetes deployment is a resource object in Kubernetes that provides declarative updates to applications. A deployment allows 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.

The process of manually updating containerized applications can be time consuming and tedious. Upgrading a service to the next version requires starting the new version of the pod, stopping the old version of a pod, waiting and verifying that the new version has launched successfully, and sometimes rolling it all back to a previous version in the case of failure. Performing these steps manually can lead to human errors, and scripting properly can require a significant amount of effort, both of which can turn the release process into a bottleneck.

A Kubernetes deployment makes this process automated and repeatable. Deployments are entirely managed by the Kubernetes backend, and the whole update process is performed on the server side without client interaction. A deployment ensures the desired number of pods are running and available at all times. The update process is also wholly recorded, and versioned with options to pause, continue, and roll back to previous versions.

2.4.5 Service

An abstract way to expose an application running on a set of Pods as a network service. With Kubernetes there is no need to modify the application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

Kubernetes Pods are mortal, they are born and when they die, they are not resurrected. If a Deployment is used to run an app, it can create and destroy Pods dynamically.

Each Pod gets its own IP address, however in a Deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.

This leads to a problem: if some set of Pods (call them “backends”) provides functionality to other Pods (call them “frontends”) inside the cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

Enter Services.

There are four types of Kubernetes services, each one designed for different purposes:

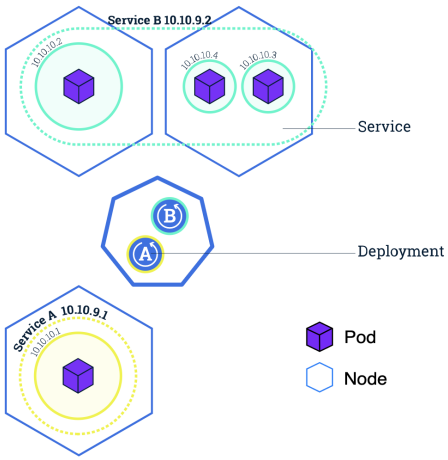

Figure 2.5. Kubernetes service

- ClusterIP. This default type exposes the service on a cluster-internal IP. It is possible to reach the service only from within the cluster.
- NodePort. This type of service exposes the service on each node's IP at a static port. A ClusterIP service is created automatically, and the NodePort service will route to it. From outside the cluster, it is possible to contact the NodePort service by using "<NodeIP>:<NodePort>".
- LoadBalancer. This service type exposes the service externally using the load balancer of the cloud provider. The external load balancer routes to NodePort and ClusterIP services, which are created automatically.
- ExternalName. This type maps the service to the contents of the externalName field (e.g., foo.bar.example.com). It does this by returning a value for the CNAME record.

2.5 Service Mesh

A service mesh is a configurable, low-latency infrastructure layer designed to handle a high volume of network-based interprocess communication among application infrastructure services using application programming interfaces (APIs). A service mesh ensures that communication among containerized and often ephemeral application infrastructure

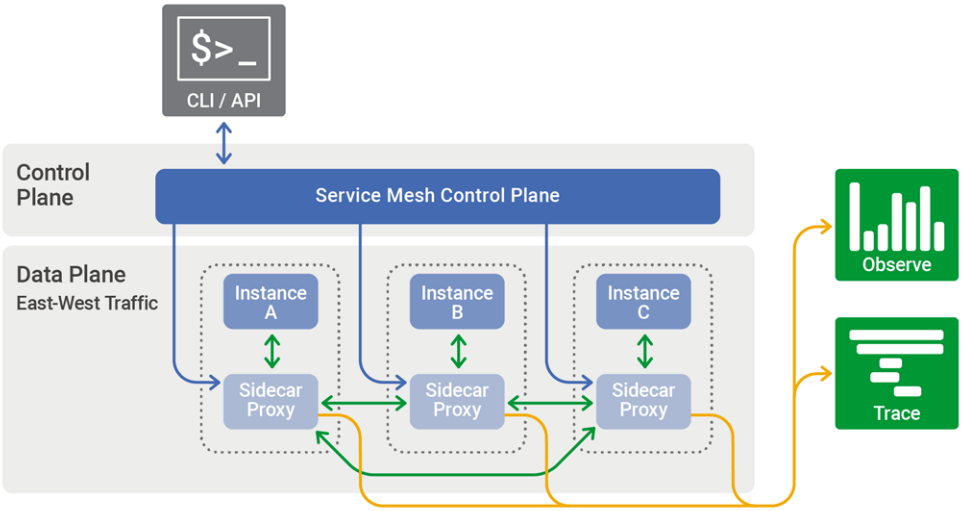

Figure 2.6. Service mesh generic topology

services is fast, reliable, and secure. The mesh provides critical capabilities including service discovery, load balancing, encryption, observability, traceability, authentication and authorization, and support for the circuit breaker pattern.

The service mesh is usually implemented by providing a proxy instance, called a sidecar, for each service instance. Sidecars handle inter-service communications, monitoring, and security-related concerns – indeed, anything that can be abstracted away from the individual services. This way, developers can handle development, support, and maintenance for the application code in the services; operations teams can maintain the service mesh and run the app.

Istio [6], backed by Google, IBM, and Lyft, is currently the best-known service mesh architecture, but is not the only option, and other service mesh implementations are also in development. The sidecar proxy pattern is most popular, as illustrated by projects from Buoyant, HashiCorp, Solo.io, and others. Alternative architectures exist as well: Netflix's technology suite is one such approach where service mesh functionality is provided by application libraries (Ribbon, Hysterix, Eureka, Archaius), and platforms such as Azure Service Fabric embed service mesh-like functionality into the application framework.

Service mesh comes with its own terminology for component services and functions:

- **Container orchestration framework**. As more and more containers are added to an application's infrastructure, a separate tool for monitoring and managing the set of containers – a container orchestration framework – becomes essential. Kubernetes seems to have cornered this market, with even its main competitors, Docker Swarm and Mesosphere DC/OS, offering integration with Kubernetes as an alternative.

- **Sidecar proxy**. A sidecar proxy runs alongside a single instance or pod. The purpose of the sidecar proxy is to route, or proxy, traffic to and from the container it runs alongside. The sidecar communicates with other sidecar proxies and is managed by the orchestration framework. Many service mesh implementations use a sidecar proxy to intercept and manage all ingress and egress traffic to the instance or pod.

- **Load balancing**. Most orchestration frameworks already provide Layer 4 (transport layer) load balancing. A service mesh implements more sophisticated Layer 7 (application layer) load balancing, with richer algorithms and more powerful traffic management. Load-balancing parameters can be modified via API, making it possible to orchestrate blue-green or canary deployments.

- **Support for the circuit breaker pattern**. The service mesh can support the circuit breaker pattern, which isolates unhealthy instances, then gradually brings them back into the healthy instance pool if warranted.

Chapter 3

Profiling in Cloud Environments: state of the art

With the advent of containerization and Cloud orchestration platforms the focus on the profiling problem has increasingly grown over the years. The microservices approach provides a lot of advantages for developers such as:

- ease of code build and maintenance
- flexibility and scalability in using technologies
- fault isolation
- faster deployment process

At the mean time, while reducing the effort for developers, it increased the complexity for Cloud providers and in particular for Cloud orchestration platforms. This is why over the last decade a big effort has been put in research for solutions aiming to reduce the complexity of this development framework (profiling is indeed one of these previously mentioned research topic).

In this chapter we will first introduce the concept of profiling, moving then to explore the most important papers that already faced this research topic.

3.1 The Concept of Profiling

In software engineering, profiling ("Program Profiling", "Software Profiling") is a form of dynamic program analysis that measure, for example:

- the space (memory) or time complexity of a program
- the usage of particular instructions
- the frequency and duration of function calls

Most commonly, profiling information serves to aid program optimization.

Program analysis tools are extremely important for understanding program behaviour. Computer architects need such tool to evaluate how well program will perform on new architecture, but also the same information can be used by some automatic tool to take the best decisions on the execution of the same application.

Profilers, which are also programs themselves, analyze target programs by collecting information on their execution. Based on their data granularity, on how profilers collect information they are classified in:

- **event based**, the execution of the profiling algorithm is triggered by some external event (i.e. system call, thread creation, etc)
- **statistical**, a sampling profiler probes the target program's at regular intervals using operating system interrupts; are typically less numerically accurate and specific, but allow the target program to run at near full speed

Profilers not only differ by their data granularity, but also by their degree of interaction with the code of the application they're collecting information from.

In some case the profiling process may require code **instrumentation** (see Figure 3.1), this technique effectively adds instructions to the target program to collect the required information; to implement such a behaviour it requires the use of specific libraries. Note that instrumenting a program can cause performance changes, and may in some cases lead to inaccurate results and/or heisenbugs (software bug that seems to disappear or alter its behavior when one attempts to study it);

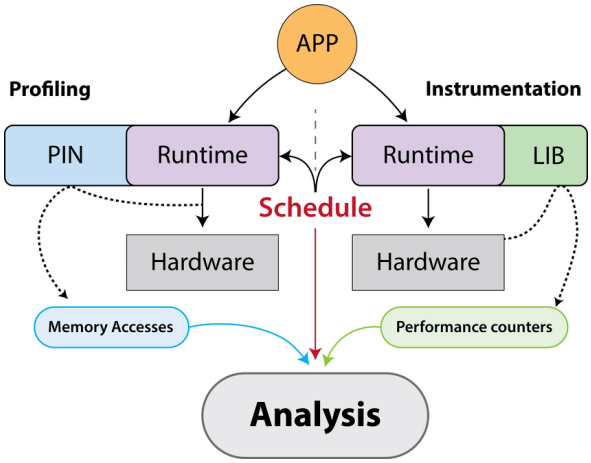

Figure 3.1. Transparent profiling and instrumentation

the effect will depend on what information is being collected and on the level of timing details reported. For example, adding code to count every procedure/routine call will probably have less effect than counting how many times each statement is obeyed. A few computers have special hardware to collect information; in this case the impact on the program is minimal.

Code instrumentation can definitely provide a wide range of information but at the mean time it requires the developer to become confident with these libraries and to use them in the proper way; this is probably the main drawback of these kind of solutions.

For this reason over the years many other solutions has been developed to overcome this limitation: the idea is to create profiling environment able to work seamlessly with any kind of application and most importantly without requiring any change in the code. These kind of solutions may seem better from the logical point of view, but actually they have disadvantages too: the most relevant one is probably the fact that the amount of information these solutions can extract from the execution of an application is limited, so in this case the challenge is to being able to generate some reliable profiling out of the limited information available.

3.2 Autopilot

In many public and private Cloud systems, users need to specify a limit for the amount of resources (CPU and Memory) to provision for their

workload. A job that exceed its limits might be throttled or killed, resulting in delaying or dropping end-user requests, so human operators naturally err on the side of caution and request a larger limit than the job needs.

This behaviour from the developer point of view helps to improve the reliability of the application, being sure that the application will work in all possible scenarios, also the ones which have not been considered. But at the same time this conservative approach, at a scale, results in massive aggregate resource wastage for cloud providers.

Further studies from Google have shown that for these manually-managed jobs the slack - the difference between the limit and the actual resource usage - is about 46%.

This is why at Google - as a public Cloud provider - they had the need to develop an automatic profiling system able to reduce this slack while minimizing the risk that a task is killed with an out-of-memory (OOM) error or its performance degraded because of CPU throttling.

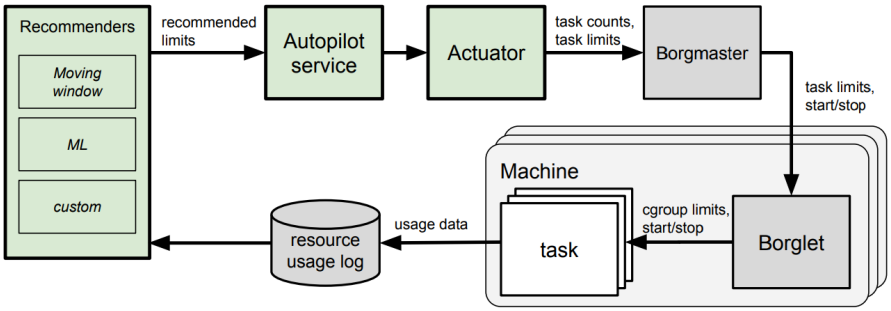

Figure 3.2. Autopilot dataflow diagram

For this reason they developed Autopilot [7], which is a combination of two different components:

- a **recommendation** component, in charge of computing the profiling for each microservice
- an **actuator**, which actually takes these recommendations and reflect them in amount of resource assigned to the microservice

This technique of constantly compute and change resources associated to microservice execution is not new and goes under the name of **Vertical Scaling**.

In particular for the recommendation component they developed two different algorithms providing the same output. They tested both of them with their datacenter workloads in order to understand their behaviour in a wide range of scenarios:

- the first algorithm computes the prediction by weighting the historical data of resource usage, extracting then different statistics out of them to compute the profiling
- with the second algorithm they followed a slightly different approach by implementing a profiling system based on Machine Learning techniques able to optimize some custom defined cost functions

Both the algorithms perform similarly and they can reduce the difference between the limit and the actual resource usage of microservices while minimizing the risk that a task is killed with an out-of-memory (OOM) error or its performance degraded because of CPU throttling.

3.3 Network Profiling

Over the past decade, cloud computing adoption has seen explosive growth, at both consumer and enterprise levels. Legacy software providers such as Microsoft, Oracle and Adobe have all made huge, concerted efforts to encourage users of their on-premises software offerings to upgrade to their cloud equivalents, which are usually offered on a subscription pay-as-you-go basis.

Over the course of the last ten years or so, cloud computing has evolved from being something that service providers told companies they should be adopting, to the very lifeblood that runs through most modern enterprises [3]. As a consequence of this explosive growth the scalability of modern data centers has become a practical concern and has attracted significant attention in recent years.

The predominant architectural pattern for datacenter design is the so called **three-tier-architecture** (see Figure 3.3): while providing some interesting features like scalability and extensibility, it can experience performance issues in the core of the network.

In this scenario the placement of VMs become crucial. Let's suppose, as a Cloud provider, we are asked to create two VMs, which need to communicate a lot one another: if we place them in two server in the

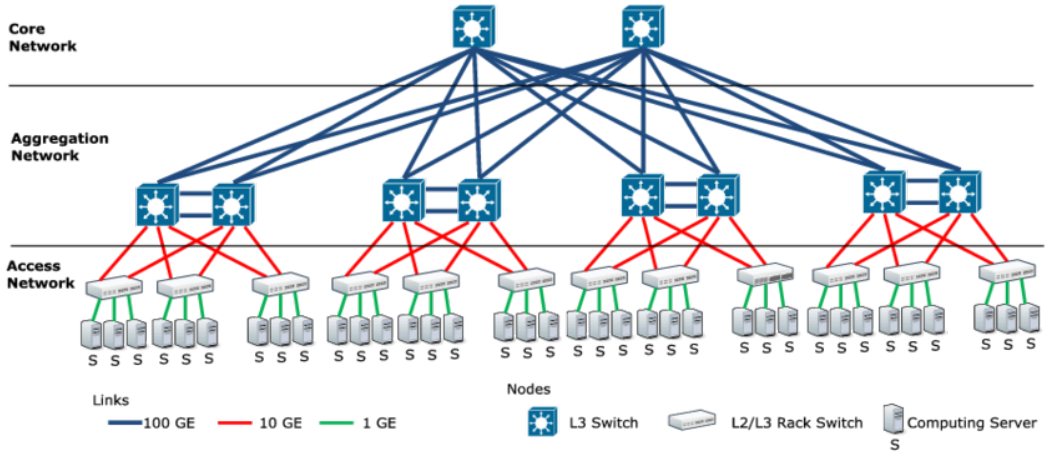

Figure 3.3. Three tier architecture

same rack we will experience that the traffic between them will never cross the backbone; on the contrary if we place them "far apart" we will experience that their traffic will always cross the backbone, reducing the available bandwidth of the connection.

For this reason is crucial to create a representation of the network communications between VMs (the same concept can be applied to containers) [12]. The paper proposes a simple approach to solve this problem by defining a suitable traffic matrix to represent the traffic exchanged between each pair of virtual machines where the entries are the sum of all the traffic, the uplink plus the downlink of each virtual machine.

3.4 Vertical Pod Autoscaler

The idea of using Kubernetes is to pack as many containers as possible in the least infrastructure. Developers and system administrators struggled to find the optimum value of resource requests and limits; tuning them required a fair amount of monitoring and understanding the utilisation of both, through a benchmark testing or through general observation of production utilisation and traffic.

With the growth of containers in a microservices architecture, it became more and more challenging to understand the resource utilisation patterns as system admins focused more on stability and ended up using a resource request far above what was actually needed. Kubernetes

came up with a solution to tackle this problem with the Vertical Pod Autoscaler.

The Vertical Pod Autoscaler uses three main components to make the autoscaling work. Here's the algorithm:

- **VPA admission hook**. Every pod submitted to the cluster goes through this webhook automatically, which checks whether a VerticalPodAutoscaler object is referencing this pod or one of its parents (a ReplicaSet, a Deployment, etc.)
- **VPA recommender**. Connects to the metrics-server application in the cluster, fetches historical data (CPU and memory) for each VPA-enabled pod and generates recommendations for scaling up or down the requests and limits of these pods.
- **VPA updater**. Runs every 1 minute. If a pod is not running in the calculated recommendation range, it evicts the currently running version of this pod, so it can restart and go through the VPA admission webhook, which will change the CPU and memory settings for it, before it can start.

Vertical Pod Autoscaler comes with a lot of nice feature for developers but also with quite a long list of limitations:

- the development is still in beta stage. This means that the use of the Vertical Pod Autoscaler is strongly discouraged in production datacenters
- it requires at least two healthy pod replicas to work. This kind of defeats its purpose in the first place and is the reason why it isn't used extensively. As a VPA destroys a pod and recreates it to vertically autoscale it, it requires at least two healthy pod replicas to ensure there's no service outage.

3.5 Liqo

Liqo is an open source project started at Politecnico of Turin that allows Kubernetes to seamlessly and securely share resources and services, enabling to run tasks on any other cluster available nearby.

Thanks to the support for K3s, also single machines can join a Liqo domain, creating dynamic, opportunistic data centers that include also commodity desktop computers and laptops.

Differently from existing federation mechanisms, Liqo leverages the same highly successful "peering" model of the Internet, without any central point of control, nor any "master" cluster. New peering relationships can be established dynamically, whenever needed, even automatically. In this respect, Liqo supports automatic discovery of local and remote clusters, to further simplify the peering process.

Sharing and peering operations are strictly enforced by policies: each cluster retains full control of its infrastructure, deciding what to share, how much, with whom. Each cluster can advertise the resources allocated for sharing to other peers, which may accept that offer. In that case, a contract is signed and both parties are requested to fulfill their obligations. Each cluster can control exactly the amount of shared resources, which can be differentiated for each peer.

Security is very important in Liqo. In this respect, Liqo leverages all the features available in Kubernetes, such as Role-Based Access Control (RBAC), Pod Security Policies (PSP), hardened Container Runtimes Interfaces (CRI) implementations.

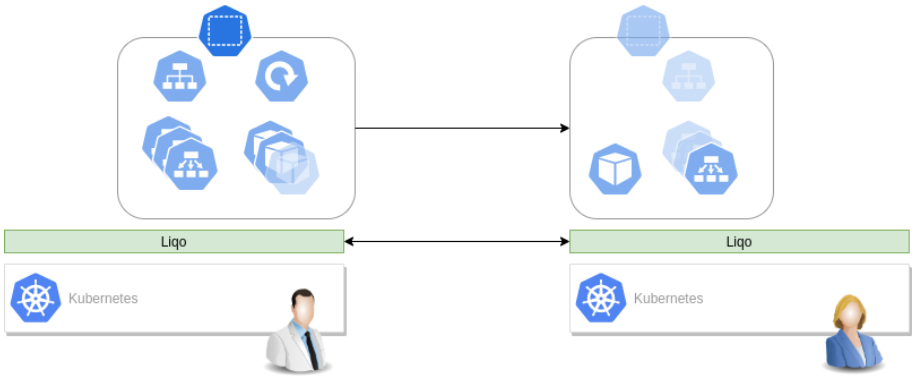

Figure 3.4. Liqo architecture

Kubernetes users will experience the usual environment also after starting Liqo: all administrative tasks are the same, performed in the usual way and with the well-known tools (e.g. kubectl). The only difference is that the cluster can become more powerful, as resources and services can be borrowed from the other clusters.

With Liqo, it is possible to leverage an unlimited amount of resources

by simply peering with other clusters. Similarly, resource providers can leverage their infrastructure by selling their resources to many different peers, in a highly dynamic way.

Chapter 4

Job Profiling Design

This thesis aims to design a profiling algorithm, able to work in combination with the scheduler in order to improve scheduling decision not only on standard datacenter scenario, bu also in the Liqo resource sharing scenario.

The designing phase of the profiling algorithm cannot start from scratch: we first need to understand how microservices usually behaves in production datacenters; in the fist section of this chapter we will show two reports analyzing microservices common behaviour.

After this introduction part we will evaluate the design choices for the profiling algorithm.

4.1 Introduction

Before starting to develop any kind of profiling algorithm it is interesting to understand what are the kind of microservices usually hosted by cloud infrastructure and also what are their patterns for what concerns resource usage.

Many papers and reports already analyzed this concept [1] [5] [2]. In this wide scenario Google is probably one of the most interesting source of information; periodically they release reports about the behaviour of their datacenters and the workload they deal with, enabling researchers to explore how scheduling works in large-scale production compute clusters.

According to their last report "*the compute and memory consumption of jobs are extremely variable, with squared coefficients of variation over*

23000. Both compute and memory consumption follow power-law Pareto distributions with very heavy tails: the top 1% of jobs ("resource hogs") consume over 99% of all resources, which has major implications on scheduler design: care is needed to insulate the bottom 99% of jobs (the "mice") from the hogs to keep queueing times under control" [10].

It worth noticing that there is not only a difference in resource consumption between different jobs but there are also differences considering the same job: the resource consumption pattern may vary a lot during the time.

When we deal with microservices having a knowledge of what will be their resource consumption in most of the case is not enough to have a complete overview of what will be the general behaviour of the system; in fact if we split an application into microservices, we replace function calls with network calls. So, if we want to improve our profiling choices (and the scheduling decision accordingly), we must also analyze what is the communication pattern between microservices.

In this case the paper [12] helps us to have a better understanding of what is the common scenario in production datacenters (the paper focuses on VMs instead of containers but the results are still very interesting).

"*While 80% of VMs have average rate less than 800 KBytes/min, 4% of them have a rate ten times higher. The inter-VM traffic rate indeed varies significantly. [...] Although the average traffic rates are divergent among VMs, the rate for a large proportion of VMs are found to be relatively stable when the rate is computed at large time intervals (an interval is labeled as stable if the rate within it is no more than one standard deviation away from the mean in the entire measurement period). It shows that for the majority of VMs (82%), the standard deviation of their traffic rates is no more than two times of the mean. Finally we experience that for 82%-92% of VMs, no less than 80% of the entire set of time intervals are stable*" [12].

After having analyzed these results we started to develop logically the profiler trying to address all these requirements.

4.2 Connection Profiling

The Liqo project is trying to define the concept of cluster resource sharing; each cluster shares some computational resources (servers) with

other clusters. The idea is that a cluster can decide whether to schedule a given job in his infrastructure or offload it to another one. The project has been developed leveraging on functionalities offered by an already existing container orchestrator, Kubernetes, extending it with additional features.

Extending an existing orchestrator solution means that we can focus mainly on additional features but at the mean time we need to deal with it's control plane components, originally developed with different mindsets and objective.

This is the case, for instance, of the component in charge of making scheduling decisions for new incoming microservices; originally default Kubernetes scheduler was meant to make decisions considering servers hosted in the same datacenter. The Liqo project tries to overcome this kind of limitation by providing the resource sharing feature previously mentioned.

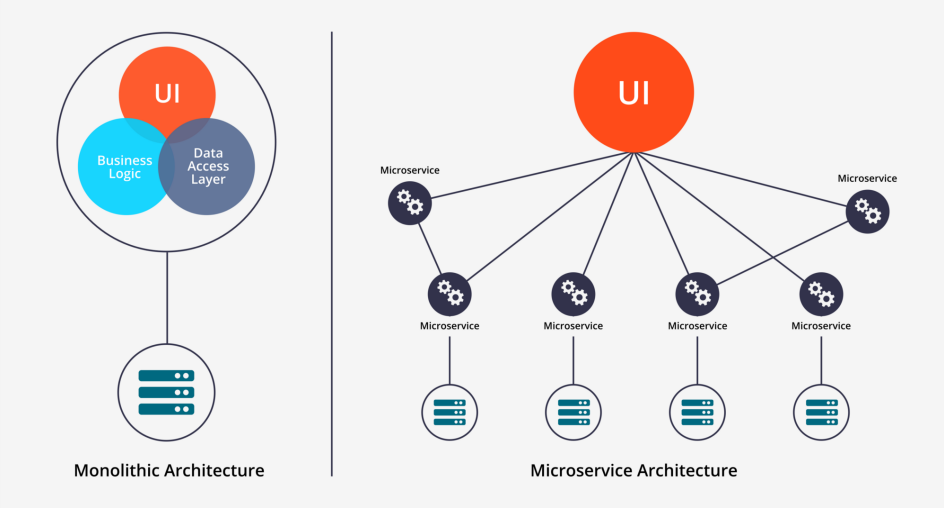

Figure 4.1. Microservice application example

When we deal with applications developed as a set of microservices in most of the cases we have a set of logically connected components (like the ones depicted in figure 4.1), which communicate through the network; in the Liqo scenario we are dealing with a more flexible datacenter architecture, where servers can be distributed geographically apart one another; so if we consider microservices we have a set of jobs that need to communicate through the network and this is why having a detailed knowledge of what is the communication pattern between them is crucial to handle their execution properly. The combination of Profiler +

Scheduler aims to overcome these series of limitation that we previously discussed.

4.3 Resource Profiling

The knowledge of the communication pattern between microservices provides to the scheduler a lot of useful information in the decision process of finding the "best" server to host a given job; but there are also some other information, such as job expected RAM and CPU consumption, which can enlarge the amount of data we have about the system of microservices, allowing us to improve their execution policies.

When we deal with production datacenters usually it is not a good practice to have microservices that can require as many resource as they want (in term of CPU and RAM); this happens because since the infrastructure is shared among a wide range of other microservices we do not want that the misbehaviour of one job can affect the execution of the others (i.e. if a job for some reason starts to request all the available core of the CPU, the other jobs hosted by the same server will starve because of lack of resources). And this problem is amplified in case of public cloud providers, which "rent" their resources to multiple customers.

This is why it is always a good practice to specify the execution boundaries of any microservice in order to avoid these kind of situations. The problem is that developers in many cases do not know how to quantify properly these boundaries in advance, ending up in two common errors:

- over-commitment, defining boundaries that are way higher than the actual needs of the microservice, in order to be sure that it will always work
- under-commitment, defining boundaries that are lower than the actual needs, because of some consideration error

Both these errors leads to different problems:

- in the first case for the cloud provider it will result in poor physical resources utilization: in Google datacenter they estimate an average over-commitment for microservice requests of almost 50%
- in the second case it will end up in sub-optimal executions for the microservice

4.4 Architecture

Given the complexity and the heterogeneity of microservices behaviour we designed the profiling algorithm with two different components, in order to adapt to a wide range of scenarios:

- **OTP (One Time Profiler)**, performed one time at the beginning of job execution
- **CP (Continuous Profiler)**, performed periodically on any running job, in order to fit the output of the algorithm to the actual behaviour of the microservice

Exploiting this two layer profiling algorithm we have a **proactive** component (OTP), which tries to predict the future behaviour of a microservice based on historical data about its previous executions; and a **reactive** component (CP), which constantly watches the microservice and tries to refine the profiling decision based on information about its execution (detailed information about the two components will be provided in sections 4.5 and 4.6).

As we saw in the introduction section of this chapter, in order to have a complete overview of microservices behaviour we need to consider many different information about their execution. The output of both profiling components provide a knowledge about:

- job connections, what is the communication pattern of a microservice towards other microservices
- job resources, what is the resource usage of a microservice in term of:
  - Memory usage
  - CPU usage

As we saw in the introduction of this chapter there is not only a difference in resource consumption between different jobs but there are also differences considering the same job: the resource consumption pattern may vary a lot during the time. Let's think for example of a web server scenario: during the daytime we can assume that the number of potential clients is higher than the one during the night hours; this will result in a much higher pressure for the system during the day. In order

to adapt the profiling algorithm also to these kind of applications we introduced the concept of **timeslot**: we logically divided the daily 24 hours into smaller chunks (called timeslot), exploiting then not all the possible information about job execution, but only the ones belonging to the same timeslots in the past.

4.5 OTP - One Time Profiler

As we can see from the reports of the previously cited papers in most cases resource consumption pattern of microservices can be considered stable over the time. This means that we can expect that in the future the behaviour of a given microservice will be the same of the one in the past; the **OTP (One Time Profiler)** has been designed having this key concept in mind.

The OTP (see figure 4.2 for a visual representation):

- collects information about microservice previous executions
- extract some key features from the historical data
- uses these key features to compute the profiling

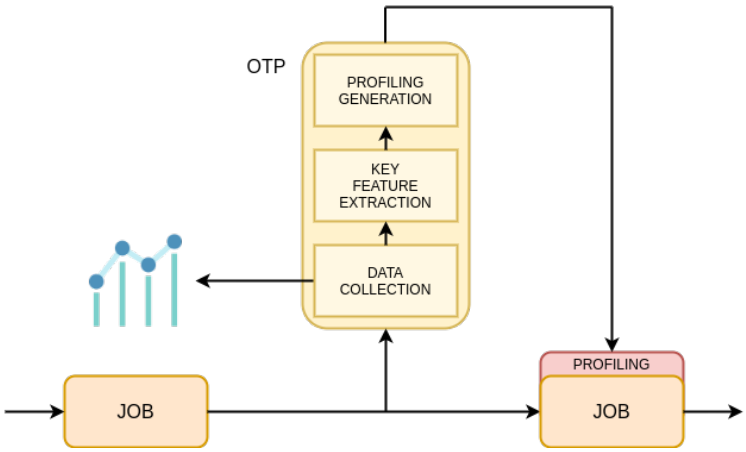

Figure 4.2. OTP architecture

The default behaviour of an orchestrator (and Kubernetes as such) whenever a request for the deployment of new job arises is to intercept it with the scheduler, which is the component in charge of finding a server

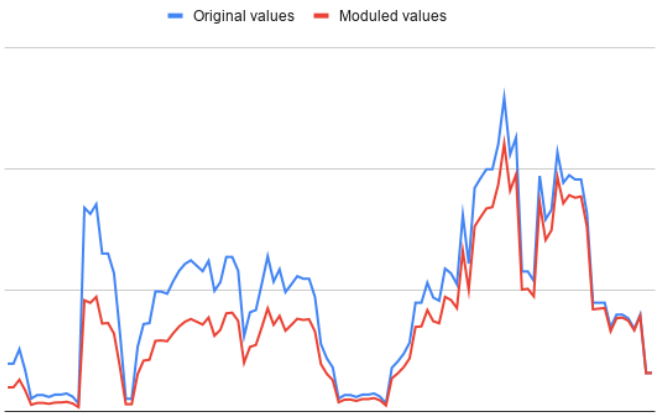

Figure 4.3. Visualization of moduled data

with enough free resources to host it. In this scenario the scheduler takes a decision only based on the boundaries defined by the developer for that job and the amount of available resources in the cluster (as we saw in section 4.3 these boundaries may not be reliable).

Within the context of the Liqo project a custom scheduler has been developed, capable of receiving as input not only the job but also a set of additional information in the form of profiling.

So whenever a request for a job deployment arises, it is intercepted in this case by the profiling system. The first task is to collect information about the job previous executions: if we are about to perform the profiling of the memory usage for that particular job, we will collect historical data about its memory consumption (the same concept is applied also to CPU and connections).

After having collected these information, the historical data are processed; in particular the historical data are weighted by the function w[t]:

$$w[t] = 2^{-t/\tau}$$

where τ is the half life: the time after which the weight drops by half. The historical data are weighted to smooth the response to a load spike and to give more relevance to samples closer in time (Figure 4.3 helps to visualize the difference between the original signal and the weighed one).

Starting from the weighted signal, some key features are extracted depending on the resource to profile an then they are used to compute

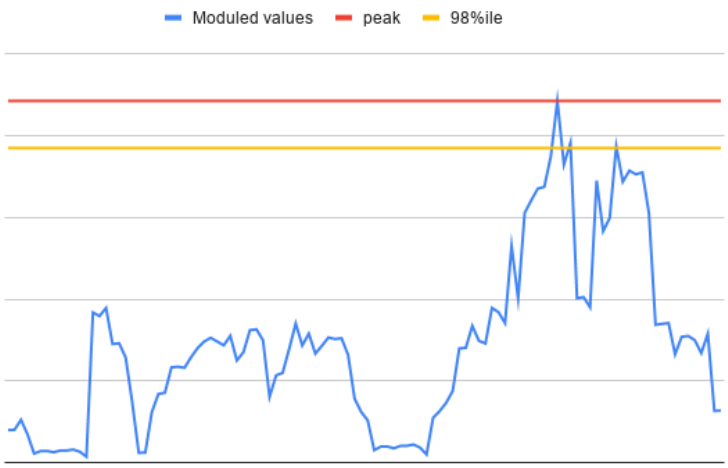

Figure 4.4. Historical data feature extraction

the profiling (see subsections 4.5.1 and 4.5.2 for the description of these features). Finally the profiling is linked to the job so that the scheduler can use it in its decision making process.

4.5.1 Resources (RAM and CPU)

The output of the profiler is the definition of the boundaries in term of RAM and CPU usage for a job. This section aims to describe the features to extract from weighted historical data to compute such a prediction, but first it is interesting to understand how too low boundaries can affect the execution of a given job.

Let's consider the two resources:

- CPU: in case a microservice exceeds its CPU limits the orchestrator reacts by imposing some throttling cycles
- RAM: in case a microservice exceeds its Memory limits the orchestrator reacts by killing the microservice, sending an out-of-memory (OOM) error

This is why RAM profiling is far more critical than CPU and hence the profiling of the first one must be much more conservative than the second one. For this reason, following the work started by Google [7], starting from the weighted historical data we extract the 98TH percentile

to compute the profiling of the CPU, while in case of RAM we consider the peaks of the weighed samples (see Figure 4.4).

This section and also the following one about connection profiling are meant to be just an introduction, more details will be presented in the next chapter about the implementation of the algorithm.

4.5.2 Connections

The profiling of the connections is less critical than resource one because it does not reflect on any particular constraint for the microservice execution. In fact the main purpose for the connection profiling is to provide to the scheduler an high level abstraction of the web application and the degree of interaction between the microservices.

For this reason starting from the weighted historical data we compute the average of bytes sent/received between each couple of microservice, using this value to compute the profiling.

4.6 CP - Continuous Profiler

The other concept that emerges from the previously cited papers is that for some microservices is not possible to profile their future behaviour in term of resource consumption because of their unpredictability. The OTP in these cases can't fit our requirements, because it is not possible to identify reliable patterns on historical data.

As the name suggests the **CP (Continuous Profiling)** aims to:

- constantly monitor different metrics, which describe microservice execution
- tune the profiling created by the OTP to the actual current needs of the microservice
- updates the profiling, making it available to the scheduler

The metrics used by the CP, which provide information about jobs current executions, are processed in a different way; the CP does not work on a single microservice level, but with the group of microservices that compose the whole application.

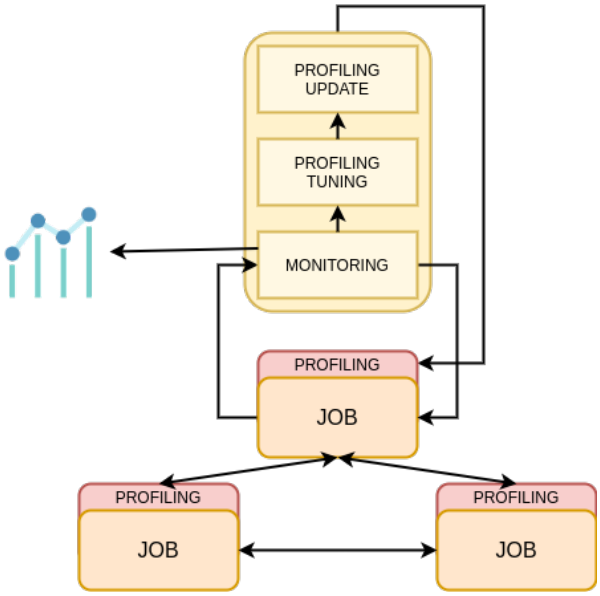

Figure 4.5. CP Architecture

We decided to follow this approach because updating the profiling for each microservice independently leads to two different problems:

- by simply watching the execution of a single microservice is difficult to estimate the quality of its execution unless a certain threshold is defined. Let's say if a given parameter of the job execution exceed this threshold we can say that it is behaving correctly or not. The problem is that this threshold is difficult to define a priori or at execution time (this value is no longer needed if we consider the whole group of microservices)
- considering one microservice at a time we lose the big picture of the entire application. The main goal for the CP is to identify the microservices whose execution can affect the overall quality of service offered by the application, the bottleneck of the system, and improve the profiling only for them

Chapter 5

Job Profiling Implementation

The code of the profiler has been developed in Golang [4] mainly for two reasons:

- Kubernetes framework has been developed in Golang, so it is easier to integrate the code with the orchestrator in such a way
- Golang has very good performance in term of execution, very important feature if we deal with critical services like the scheduler in our case

The profiler leverages on Prometheus (5.1.1) and its query language PromQl to obtain information about job execution, creating an internal representation of the requirements for that particular job.

In this chapter we will first discuss the way we obtain information about job execution moving then to a second part showing how to translate these information into profiling suggestions for the scheduler.

5.1 Metrics

A software metric is a standard of measure of a degree to which a software system or process possesses some property. Even if a metric is not a measurement (metrics are functions, while measurements are the numbers obtained by the application of metrics), often the two terms are used as synonyms.

Metrics in production clusters are essentials because they provide feedback about:

- containers and cluster health
- cluster available resources
- network behaviour
- I/O and disk pressure on servers
- ... and many other information

Because of all these interesting features over the last few years a lot of monitoring tools have been released and adopted in production datacenters (see 5.1.1).

Metrics are deeply exploited by the profiling system in order to retrieve information about jobs execution; in details the most interesting ones are:

- for the **OTP**:
  - `container_cpu_usage_seconds_total`, provides information about cpu usage of a container
  - `container_memory_usage_bytes`, provides information about memory usage of a container
  - `istio_request_bytes_sum`, provides information about the amount of bytes sent from a container to others
- for for the **CP**:
  - `container_cpu_cfs_throttled_seconds_total`, provides information CPU throttling periods of a container
  - `container_memory_failures_total`, provides information about the amount of memory failure events for a container

All the previously listed metrics are collected using the PromQl query language, that lets us select and aggregate time series data in real time.

5.1.1 Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company.

Prometheus gathers metrics from instrumented jobs either directly or through an intermediary gateway designed for temporary jobs. The samples are stored locally and scanned by rules in order to either collect and record a new time series from the existing information or create alerts.

Prometheus's main features are:

- a multi-dimensional data model with time series data identified by metric name and key/value pairs
- PromQL, a flexible query language to leverage this dimensionality
- multiple modes of graphing and dashboarding support (useful for datacenter maintainer)

5.1.2 Istio

Istio is an open source service mesh platform that provides a way to control how microservices share data with one another. It includes APIs that let Istio integrate into any logging platform, telemetry, or policy system. Istio is designed to run in a variety of environments: on-premise, cloud-hosted, in Kubernetes containers, in services running on virtual machines, and more.

Istio's architecture is divided into the data plane and the control plane. In the data plane, Istio support is added to a service by deploying a sidecar proxy within the container environment. This sidecar proxy sits alongside a microservice and routes requests to and from other proxies. Together, these proxies form a mesh network that intercepts network communication between microservices. The control plane manages and configures proxies to route traffic. The control plane also configures components to enforce policies and collect telemetry.

With a service mesh like Istio, dev and ops are better equipped to handle the change from monolithic applications to cloud-native apps—collections

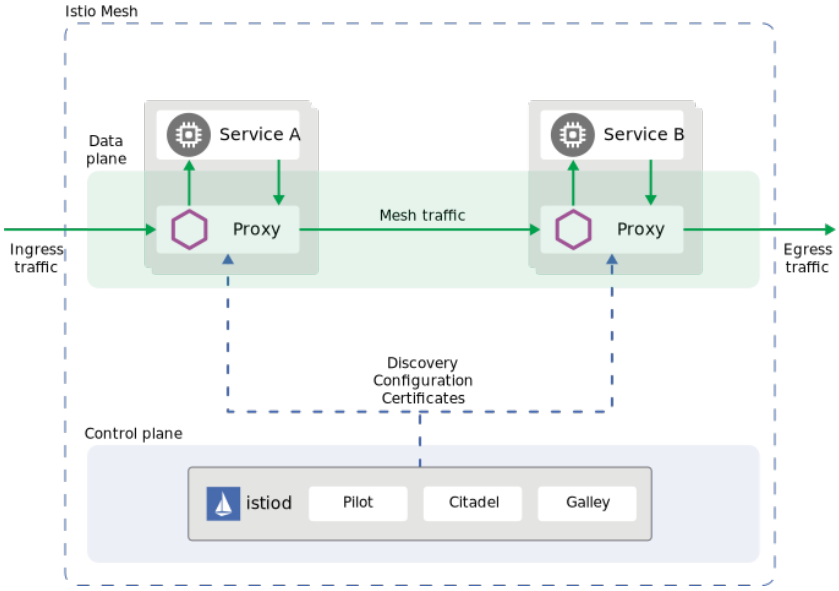

Figure 5.1. Istio architecture

of small, independent, and loosely coupled microservice applications. Istio provides behavioral insights and operational control over the service mesh and the microservices it supports. Using a service mesh reduces the complexity of deployments, and takes some of the burden off of any development teams.

Istio's main features include:

- Traffic management - Traffic routing and rules configuration in Istio allow to control the flow of traffic and API calls between services.
- Security - Istio provides the underlying communication channel and manages authentication, authorization, and encryption of service communication at scale. With Istio, it is possible to enforce policies consistently across multiple protocols and runtimes with minimal application changes.
- Observability - Get insights into service mesh deployment with Istio's tracing, monitoring, and logging features. Monitoring lets us see how service activity impacts performance upstream and downstream.

The observability feature is probably the most interesting in our case because it provides us additional information about job execution and in particular metrics related to microservice network behaviour.

5.2 Profiling Implementation

In the implementation process of the logical architecture described in chapter 4 we decided to keep the OTP and the CP as separate components, working on the same data model; for this reason the code has been structured exploiting multi threading.

In particular for each resource we have two working threads:

- one thread that implements the behaviour of the OTP; it watches for new job deployment requests and compute the prediction based on historical data
- one thread that implements the behaviour of the CP; it constantly refreshes the information in the model

Let's spend some more words to describe more in depth these two threads, but first we need to introduce the Kubernetes library we use to integrate the code to the orchestrator platform.

The Kubernetes programming interface in Go mainly consists of the `k8s.io/client-go` library (for brevity we will just call it `client-go` going forward). `client-go` is a typical web service client library that supports all API types that are officially part of Kubernetes. It can be used to execute the usual REST verbs: Create, Get, List, Update, Delete, Patch; furthermore, the verb Watch is supported.

```
watch, err := client.CoreV1().Pods(namespace).Watch(...)

for event := range watch.ResultChan() {
    //...
    go connection.ComputePrediction(event)
    go memory.ComputePrediction(event)
    go cpu.ComputePrediction(event)
    //...
}
```

Listing 5.1. OTP thread, simplified code

As we can see from the snippet 5.1 the fist performed task is to create a `watch.Interface` via the chain of method calls `client.CoreV1().Pods(namespace).Watch(...)`; this interface allow us to be notified whenever something changes in Pods representation in a given namespace.

The method `watch.ResultChan()` implements the previously described behaviour by returning a channel, managed by the client-go library, which will notify us whenever Pod scheduling requests arises in a form of an `event`. The information contained in the event are finally used to compute the prediction; in particular three threads are created, one for each resource (the keyword `go` executes the function in a separate thread).

The profiler is meant to work close to the scheduler, so performance are crucial in this particular scenario; for this reason the function `ComputePrediction(event)` on each resource has been implemented with the following key concepts:

- if a model (CPU, Memory and Connections) has already the information about the Pod the profiling is computed
- if these information are missing in the model the function takes care of updating it, connecting to Prometheus and collecting historical data. In this case no profiling is computed because the completion time for these tasks can be huge (compared to scheduling times) and we do not want the scheduler to wait such a long time

Moving then to the implementation of the CP we can see that the `client-go` is no longer used (see snippet 5.2), because the update process is triggered internally.

```
for {
    job := memory.data.GetLastUpdatedJob()
    //job := cpu.data.GetLastUpdatedJob()

    jobConnections := connection.GetJobConnections(job)

    //...
    go cpu.UpdatePrediction(job)
    go memory.UpdatePrediction(job)
    //...

    time.Sleep(...)
}
```

Listing 5.2. CP thread, simplified code

The CP module has been implemented as a never ending thread. In each iteration of the `for` loop the code checks what is the last updated

job (the job whose profiling information will be updated in this iteration) with the function `memory.data.GetLastUpdatedJob()` (in this case we call the memory model to retrieve this information, using the cpu model would provide the same result).

The CP module is meant to work at a higher abstraction level compared to the OTP; in fact it does not consider each job individually, but it carries out the following tasks on the complete set of microservices that compose the application. The connection model has a complete knowledge about the communication pattern between microservices, so the function `connection.GetJobConnections(job)` gets as input a job and returns the list of all the other jobs connected to it.

This list of jobs is passed to the `UpdatePrediction(job)` function that actually connects to Prometheus, collects all the metrics it needs to the profiling process and then stores the final result in the model. The `UpdatePrediction(job)` for the connection model is not yet implemented.

Finally sleeps until the next iteration.

5.3 Resource Profiling Implementation

In the previous section we presented the general behaviour of both the Continuous Profiling and the One Time Profiling; this section aims to provide some information about the metrics, which are collected and processed, the data structures developed in the system and the expected output.

We decided to split the problem of resource profiling (RAM and CPU) from the problem of connection profiling; in this section we will present the former, in section 5.4 the latter.

5.3.1 Metrics

The profiling system leverages on Prometheus as metrics collector. We already discussed briefly the features of this open source project, so what we would like to add in this section are some more practical information about how the information collected by Prometheus can be accessed from third party components.

Prometheus exposes an endpoint to access those data and provides

its own query language PromQL to collect and aggregate them according to our needs. PromQL is designed for building powerful yet simple queries for graphs, alerts or derived time series (aka recording rules). It is designed from scratch and has zero common grounds with other query languages used in time series databases; this allowed creating a clear language for typical TSDB queries.

We provided the profiling system with a custom client to interact with Prometheus:

- according to the type of data we need it creates the corresponding query in PromQL language
- it connects to Prometheus, asking to perform the previously created query
- it handles the response from Prometheus (the response in in JSON format), un-marshalling it into an internal representation of data
- returns the un-marshalled data

As we previously mentioned the metrics involved in the profiling of RAM and CPU required by the OTP and the CP are different because they're used for different purposes; in the case of the OTP two examples of PromQL queries for collecting historical data are:

- avg by (pod, namespace) (
  container_memory_usage_bytes
  {namespace="namespaceName", name!="", container!="", pod= "podName.*"})
- sum by (pod, namespace) (
  rate (
  container_cpu_usage_seconds_total
  {image!="", pod= "podName.*"}[1m]))

While in the case of CP for runtime data:

- sum by (pod, namespace) (
  label_replace (
  rate (
  container_memory_failures_total
  {namespace= "namespaceName", pod= "podName.*", container!=""}[1m]),
  "pod", "$1", "pod", "(.*)-.5"))

- sum by (pod, namespace) (
  label_replace (
  rate (
  container_cpu_cfs_throttled_seconds_total
  {namespace= "namespaceName", pod= "podName.*"}[1m]),
  "pod", "$1", "pod", "(.*)-.{5}"))

5.3.2 Metrics Processing

Before starting to analyze how we process the metrics shown in the previous section, we first need to understand what happens to microservices when the exceed their resource limits, because this has an impact on the actual processing.

In production datacenter, as we saw in one of the previous chapter, it is always a good practice to define some limits in resources consumption for microservices to avoid a series of misbehaviour; these limits can be set by the developer or by some automatic tool like our profiler, but the concept is still the same.

Let's start considering CPU limits: if there is little contention (as measured by the overall CPU utilization), tasks are allowed to use CPU beyond their limits. However, once there is contention, limits are enforced and some tasks may be throttled to operate within their limits. So in case a microservice exceeds its CPU limits the orchestrator reacts by imposing some throttling cycles.

In case of memory limits instead the situation is much more complex: a task is killed with an out-of-memory (OOM) error as soon as the task exceeds its limit, and the failure is handled by the orchestrator.

As we can see an error in the definition of memory limits has a much bigger impact in job executions, compared to one in the definition of CPU limits. This is why some parts of the metrics processing are similar for RAM and CPU but other ones are not because of the previously shown differences.

For what concerns the common part of the metrics processing, following the work began by Google [7] that we already discussed in the design section 4.5, the historical data for the OTP are first collected and then weighted by the function w[t]:

$$w[t] = 2^{-t/\tau}$$

to give more relevance to samples closer in time. Once the samples are weighted we summarize them by extracting some features depending on the resource; in particular:

- for the RAM we extract the peaks from the weighted data, because it is the most conservative value we can consider
- for the CPU we extract the $P_{98\%}$ from the weighted data

Finally both these raw recommendations of peak value and $P_{98\%}$ are post-processed before being applied by increasing them by a 15% for safety margin.

The metrics used by the CP, which provide information about jobs current executions, are processed in a different way. As we can see from Listing 5.2 it does not work on a single microservice level, but with the group of microservices, which compose the whole application.

For this reason these kind of metrics are processed in a different way:

- an average is computed for each microservice
- using the average computed for each single microservice we compute the average for the system of microservices
- based on the final value of average we compute some boundaries defined as $\pm 20\% application\_avg$
- finally we iterate on all the values of average defined at point 2; for all the microservices that have an average value for that particular metric below the boundaries we decrease the profiling, for the ones above the boundaries we increase the profiling

5.3.3 Data Structure

The resource profiling data structure has been developed as an interface to make the code extensible (now only RAM and CPU are considered but the interface approach makes it easier to implement other resources in the future).

```
type ResourceModel interface {
  InsertJob(...)
  UpdateJob(...)
  GetJobUpdateTime(...)
  GetLastUpdatedJob(...)
  GetJobPrediction(...)
  PrintModel(...)
}
```

Listing 5.3. Resource model interface

Each resource (Memory and CPU) implements this interface, defining its own implementation of these functions: it declares methods to insert new jobs in the datastructure `InsertJob`, invoked by the OTP, to update the model for an existing job `UpdateJob`, invoked by the CP. It is possible to retrieve the profiling for a given job via the method `GetJobPrediction`.

Given the multi-threading approach all the actual implementations of this interface (RAM and CPU) are designed to be thread safe and resilient to concurrent access.

5.3.4 Output

The output of the profiling system is a Custom Resource (CR), but to understand what CR is, we must go over a couple of concepts in Kubernetes:

- A resource is an endpoint in k8s API that allows to store an API object of any kind.
- A custom resource allows to create arbitrary API objects and defining their own kind just like Pod, Deployment, ReplicaSet, etc.

They allows to extend Kubernetes capabilities by adding any kind of API object useful for any application. Custom Resource Definition is what we use to define a Custom Resource. This is a powerful way to extend Kubernetes capabilities beyond the default installation.

Custom resource creation and management can be very complex because they require a lot of additional components. Because of their versatility they are employed in a wide range of situations; here in this

thesis are simply used as a way to store data in a distributed way. In particular we designed two different CRDs to save profiling information about RAM and CPU:

- `memoryprofiles.webapp.liqo.io.profiling`
- `cpuprofiles.webapp.liqo.io.profiling`

```
apiVersion: webapp.liqo.io.profiling/v1
kind: MemoryProfile
metadata:
  # ...
spec:
  memoryProfiling:
    updateTime: 2020-09-22 12:07:38
    value: 112654345
```

Listing 5.4. Memory profile output example

```
apiVersion: webapp.liqo.io.profiling/v1
kind: CPUProfile
metadata:
  # ...
spec:
  cpuProfiling:
    updateTime: 2020-09-22 12:07:38
    value: 0,867
```

Listing 5.5. CPU profile output example

As we can see from listings 5.4 and 5.5 both the resource profiling have two values:

- the update time for that profiling
- the value of the profiling expressed in bytes for RAM and in millicore for the CPU

5.4 Connection Profiling Implementation

In the previous section we presented the resource profiling system on both the Continuous Profiling and the One Time Profiling; this section

aims to provide some information about the orthogonal problem of connection profiling, the metrics that are collected and processed, the data structures developed in the system and the expected output.

5.4.1 Metrics

The metrics collection process in connection profiling relies on the same Prometheus client already presented in the section 5.3.1. The main difference in this case, compared to resource profiling, is that there is no implementation for CP profiling hence only historical data are collected and processed; in particular two examples of metrics collection query are:

- sum by (
  namespace, source_workload,
  destination_workload, destination_workload_namespace) (
  increase(
  istio_request_bytes_sum
  {namespace="namespaceName", source_workload="podName"}[1m]))

- sum by (
  namespace, source_workload,
  destination_workload, destination_workload_namespace)(
  increase(
  istio_response_bytes_sum
  {namespace="namespaceName", source_workload="podName"}[1m]))

5.4.2 Metrics Processing

The profiling of the connections is less critical than resource one because it does not reflect on any particular constraint for the microservice execution.

For this reason the historical data for the OTP are first collected and then weighted by the same function w[t] used for resource profiling to give more relevance to samples closer in time. Then starting from the weighted historical data we compute the average of bytes sent/received between each couple of microservice over the time.

5.4.3 Data Structure

A suitable data structure has been developed to create an in memory representation of the connection between different microservice.

```
type ConnectionGraph struct {
  jobs  map[string]*connectionJob
  ...
}

func (cg *ConnectionGraph) InsertNewJob(...)
func (cg *ConnectionGraph) GetJobUpdateTime(...)
func (cg *ConnectionGraph) GetLastUpdatedJob(...)
func (cg *ConnectionGraph) GetJobConnections(...)
func (cg *ConnectionGraph) FindSCC(...)
```

Listing 5.6. Connection graph data structure

The connection model stores information of jobs connections in a map. Additionally it provides methods to insert new job in the datastructure via the function `InsertNewJob()`, to get information about a job via the methods `GetJobUpdateTime()`, `GetLastUpdatedJob()` and `GetJobConnections()`.

The last function `FindSSC()` computes the strongly connected components of the connection graph: a strongly connected component is a sub-graph where every vertex is reachable from every other vertex. In this context we exploited this property of the graph theory to compute the list of all the connected job that compose the application.

5.4.4 Output

The output of the connection profiling process is, as in the previous case, a CR object.

The connection profiling system produces a CR for each connection between microservices; in particular the most interesting fields are:

- the value bandwidth_requirements represents the connection profiling expressed in Bytes/sec
- the combination <destination_job, destination_namespace> uniquely identifies the destination job of the communication

- the combination <source_job, source_namespace> uniquely identifies the source job of the communication

```yaml
apiVersion: webapp.liqo.io.profiling/v1
kind: ConnectionProfile
metadata:
  # ...
spec:
  bandwidth_requirement: "89219.00"
  destination_job: reviews-v2
  destination_namespace: default
  source_job: productpage-v1
  source_namespace: default
  update_time: 2020-09-22 15:43:18
```

Listing 5.7. Connection profile output example

Chapter 6

Experimental Evaluation

The profiling system aims to operate in combination with the scheduler to provide the the best execution policies for microservices. Unfortunately the development of this custom scheduler is not finished yet, so we defined two different testing scenario to overcome such a limitation and to understand the behaviour of the profiler in different situations.

The profiling system is composed of two components:

- **OTP (One Time Profiler)**, performed one time at the beginning of job execution
- **CP (Continuous Profiler)**, performed periodically on any running job, in order to fit the output of the algorithm to the actual behaviour of the microservice

The first test has been designed to evaluate the performance of the OTP, exploiting traces from microservices of a real datacenter, hosted in Ladispe in Politecnico (this one will be referred as Historical Data Test, see 6.2), while the second one's goal is to analyze the behaviour of both the components in a real case scenario (hereinafter referred as Application Profiling Test, see 6.3).

6.1 Microservice Analysis

If we consider the modern world of microservices we deal with two kind of systems:

- microservices hosted by the cloud, providing a service
- end-users (clients), which connect to the microservices to get the service

So if we want to create a framework that manages microservices (like Liqo does) we must create an infrastructure both able to host them properly and capable to provide to the end user the best possible performance for the service. The second feature is not something new, it struggled researcher for decades and goes under the name of **Quality of Service (QoS)**.

The final goal of the profiling system is indeed to improve as much as possible the QoS experienced by the client so that no differences are perceived between a microservice application hosted in a standard cloud scenario and the same application handled by the Liqo framework.

The connection profiling definitely works in that direction, by providing to the scheduler information about the communication pattern between components; since in a microservice environment jobs communicate with each other through the network, if they are placed properly it is possible to improve their communications and so the final perceived quality. Apparently the resource profiling improves only the execution of each microservice, but it does not affect the overall QoS of the system. This is why we decided to perform some tests to find out if our ideas was actually correct or it is possible to exploit the resource profiling in this improvement of the QoS.

For the purpose of the test we picked an existing web application developed by Google called Online Boutique [11], which consists of a 10-tier microservices application (see figure 6.1). The application is a web-based e-commerce app where users can browse items, add them to the cart, and purchase them. In such a scenario we identified the measure that better represents the QoS perceived by the customer in the latency.

A suitable testing tool has been developed to replicate end-user interaction with the system, but allowing us to scale horizontally, increasing the number of contemporary requests, and collecting information about the perceived latency.

For the testing phase we defined four different scenarios:

1. the web application is deployed removing all the limits for RAM and CPU usage; this scenario is useful to define the lowest possible value of latency perceived

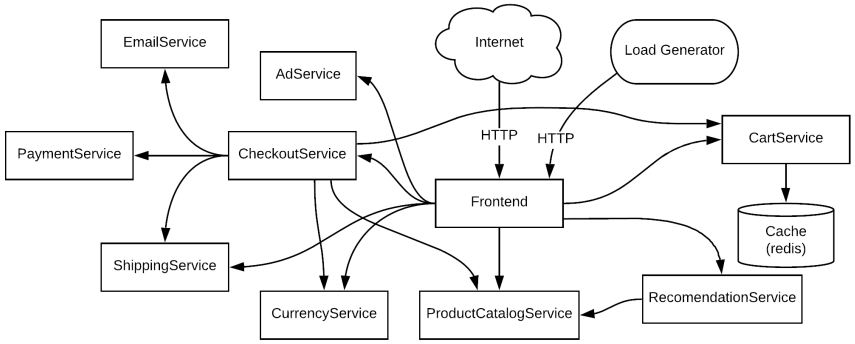

Figure 6.1. Online Boutique architecture

2. the web application is deployed with the defaults limits set by the developers; this scenario allow us to identify a normal value of latency perceived

3. the web application is deployed with the defaults limits decreased by 1/3; with this scenario we would like to see if there is any degradation in latency with under estimated limits

4. the web application is deployed with the defaults limits decreased by 1/3 only for those microservices we think are not suffering for this limits reduction, while all the others are deployed with default limits

	#1	#2	#3	#4
avg	61,89ms	68,47ms	83,47ms	69,15ms
95%ILE	64,05ms	180,61ms	190,54ms	165,23ms
98%ILE	78,10ms	281,64ms	343,67ms	276,76ms
99%ILE	105,83ms	362,74ms	740,53ms	384,39ms

Table 6.1. Microservices analysis results

The results of the tests, as shown in table 6.2, are surprising: if we consider scenario #2 and #4 we can see that while in both cases 99% of latency values are almost comparable and below 390ms, the resource consumption in case #4 are much lower (almost reduces of a 1/3) compared to case #2.

These results let us believe that the overall Quality of Service can benefits not only by the connection profiling, but also by an accurate resource profiling.

6.2 Historical Data Test

The main purpose of this test is to evaluate the performance of the **OTP** using data coming from real datacenter workload. We specifically designed this test to obtain a feedback of what would have been the profiling output of the OTP when dealing with such data.

As we mentioned in the implementation chapter of this thesis the OTP constantly watches for incoming job scheduling requests and based on those requests it computes the profiling according to some historical data. This behaviour cannot be applied in this test case, because no scheduling is involved, so we slightly modified the code to implement the behaviour described by the pseudo code in the following snippet 6.1.

```
for i := startTime ; i < endTime ; i += offset {

    // collect historical data at time i
    data := collectData(i)

    // compute prediction with the collected data
    prediction := computePrediction(data)

    // generate a suitable output for the prediction
    generateOutput(prediction)
}
```

Listing 6.1. OTP pseudo code

The previously shown loop at each time instant i generates a fake scheduling request for a given job (the target job is always the same in all the iterations) and then triggers the profiling system to compute the prediction, collecting the results to an output file.

6.2.1 Expectations

In this section we would like to describe what is the output we expect to have from the profiling system and most importantly what are the values we will use to evaluate the quality of the profiling.

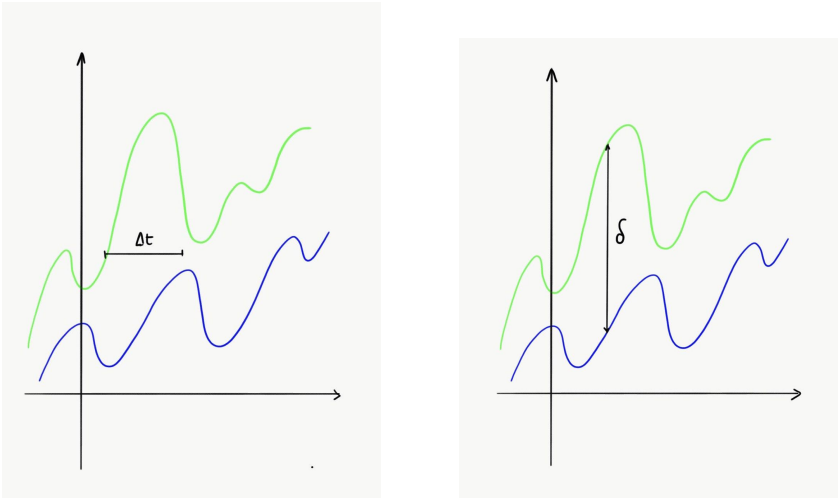

Figure 6.2. Quality and overhead of prediction visualization

In each time instant ideally the profiling output should always be $\geq$ than the value of resource usage. This characteristic can be easily analyzed by looking at a graphical representation of both the profiling output and the resource usage graph; what we would like to define here are some values able to characterize objectively the output. In particular we are focusing on:

- the quality of the prediction
- the overhead in term of resource wastage introduced by the profiling

Now let's spend some more words about the previously listed items.

The "quality of the prediction", as the name suggests, aims to evaluate if the prediction at a given time instant has been good or not. To do so we define the value Δt (see figure 6.2) as:

$$\Delta t = prediction\_time + 30 minutes$$

using this simple formula we can label each profiling sample as:

- GOOD, if the value of the resource usage never exceeds the profiling value within these 30 minutes
- BAD, if the value of the resource usage exceeds the profiling value at least once within these 30 minutes

In the end the ratio between GOOD/BAD profiling samples and the total amount of profiling sample can provide an interesting feedback of the quality of the prediction.

The metric we've just described is definitely important but it cannot provide a complete overview of the behaviour of the profiling; for this reason we introduced another value called "overhead of prediction" to evaluate the difference between the output of the profiling and the actual resource usage.

In this case we defined for each profiling sample the value δ (see figure 6.2) as:

$$\delta = profiling\_value - resource\_value$$

where $resource\_value$ is the average of the resource values within the next 30 minutes starting from the profiling time.

Averaging each computed δ we can have a value that describes the overall amount of resource wastage introduced by the profiling.

6.2.2 Input

As a sample data input we selected an application called Kubevirt, a project that enable VMs management inside a Kubernetes cluster; in particular we mainly focused one one single component called `virt-handler`.

Unfortunately the production datacenter that provide us input data does not collect metrics about microservices network behaviour; for this reason in this test case we decided to analyze the profiling output only for resource usage (RAM and CPU).

The `virt-handler` component is a deployment in the Kubernetes cluster with 4 different replicas; each replica shares with the others the same image, but each of them can require a different amount of resources to the system according to the actual workload. Figures 6.3 and 6.5 show a visual representation each replica resource usage (respectively RAM and CPU), while in figures 6.4 and 6.6 we extracted for each time instant the maximum value of resource usage among the replicas for a better visualization (again respectively RAM and CPU)

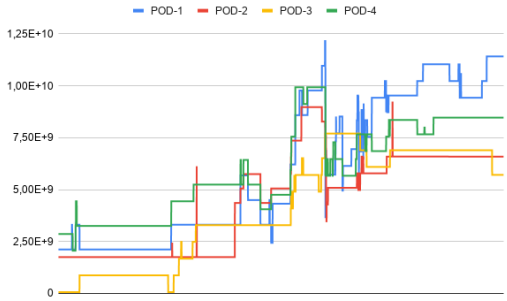

Figure 6.3. Pod RAM usage

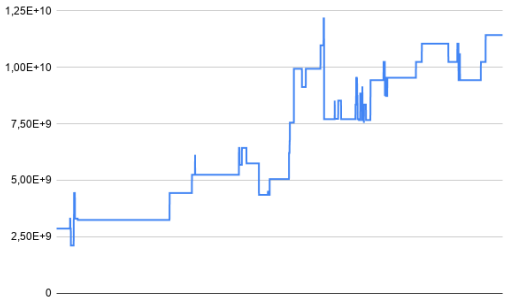

Figure 6.4. Pod max RAM usage

Figure 6.5. Pod CPU usage

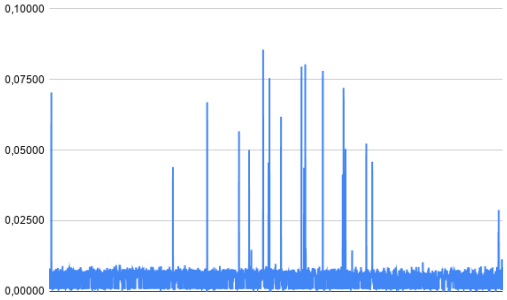

Figure 6.6. Pod CPU max usage

6.2.3 Output

In table 6.2 we summarized the values we previously discussed to evaluate the performance of the profiling system.

	Profiling quality	Profiling overhead
RAM	98,64%	16,73%
CPU	96,12%	29,73%

Table 6.2. OTP analysis results

Some of the results may require some additional consideration, in particular:

- both for RAM and CPU we have a very high quality of profiling (>96%); it worth noticing that the value for RAM profiling quality

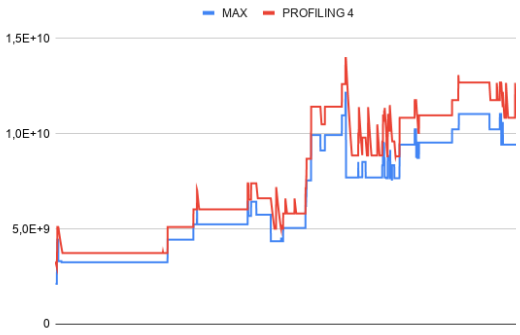

Figure 6.7. RAM profiling

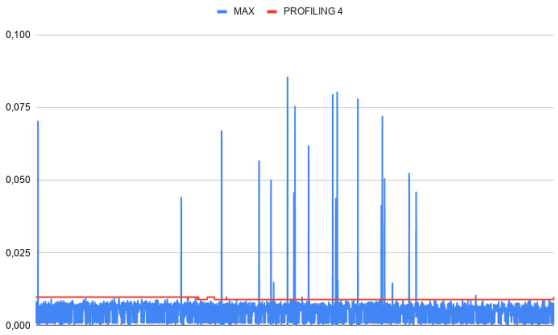

Figure 6.8. CPU profiling

is slightly higher compared to CPU's one and this is because RAM profiling is computed following some stricter constraints

- the profiling overhead introduced by both is below 30%, which is a very good result (according to Google reports developers usually overestimate microservices request by at least 50%); the fact that CPU profiling overhead is higher than RAM's one may seem strange but this is due to the pattern of CPU usage of the microservice, which is very "spiky"

Figures 6.7 and 6.8 shows a visual representation of the output of the profiling and the resource usage.

6.3 Application Profiling Test

This test has been designed to evaluate the performance of the complete profiling system in a real time scenario.

The profiling system has been designed to work with the scheduler to determine the best execution policies for any given job; an high level overview of the behaviour of the complete system can be summarized in the following:

- a job scheduling request arises
- the profiling system intercept the scheduling request and compute a suitable prediction for that given job

- finally the scheduler, among other things, enforces these predictions by changing the resource allocation for the job execution

Unfortunately the development of the scheduler has not finished yet, so for the purpose of this test we slightly modified the code of the profiler by providing it the capability not only to compute the profiling, but also to enforce it in the pod execution policies.

With this code modification it is possible to run seamlessly a set of microservices, while the profiling system constantly adjusts the resources allocated to them in order to meet their actual needs.

6.3.1 Input

Differently from the previous test in this case we will first analyze what is the kind of application we will use, because this choice will affect not only the testing tool, but also the values to extract in order to have reliable information of the quality of the profiling system.

For the purpose of the test we picked the same web application developed by Google that we already presented in section 6.1.

6.3.2 Expectations

This test required the development of two different components:

- the code addition in the profiler previously discussed
- the development of suitable testing tool able to extract some useful data of the behaviour and the quality of the profiling system

In such a scenario we identified the measure that better represents the QoS perceived by the customer in the latency. A suitable testing tool has been developed to replicate end-user interaction with the system, but allowing us to scale horizontally, increasing the number of contemporary requests, and collecting information about the perceived latency.

During the test we started the execution of the microservices and the profiling system at the same time in order to replicate what could be the worst case scenario: no information are accessible from the profiling system about microservices previous executions.

First of all we started by testing the behaviour of the same microservices disabling the profiler and collecting information about the latency perceived by the end user; these values will be used as a benchmark, by comparing them with the ones obtained with the profiling system enabled.

Starting from this benchmark values we decided to implement such a test scenario mainly for three reasons:

- to understand if the system is able to converge, improving the Quality of Service perceived by the end user
- to quantify the convergence time of the algorithm, comparing the results with the execution without the profiling system
- to evaluate the final suggestions provided for RAM and CPU limits, comparing them again with the execution without the profiling system

6.3.3 Output

We run the test for approximately 60 minutes, constantly collecting both information about job executions and latency values perceived by the end user, which will be used later to evaluate the overall quality of the profiling system.

Figures 6.9-6.12 show the resource consumption of each microservice (beware that even if sometimes some time series change color over the time, it does not mean that they are different microservices; this behaviour is due to the fact that sometimes microservices are rescheduled hence they are considered different instances by the metrics system even if they are not); in particular:

- as we can see from figure 6.9 over the time some microservices increase their CPU usage thanks to the profiler. Not all the microservices experience such a behaviour because the profiling system increases the available resources only for a subset of them; in particular it selects only the ones which are suffering for lack of resources. As a consequence of these choices we can see from figure 6.10 that the throttling periods of the system of microservices keeps reducing over the time

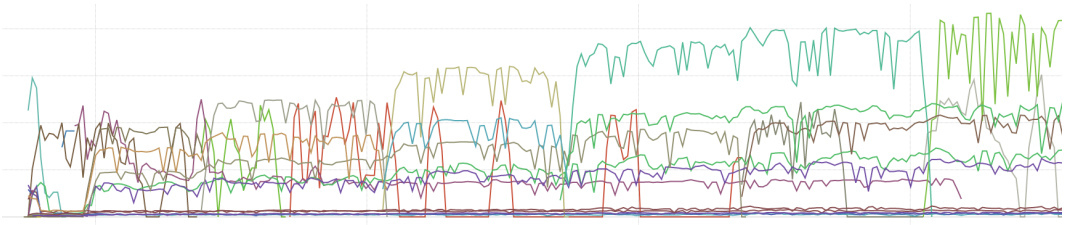

Figure 6.9. Microservices CPU consumption

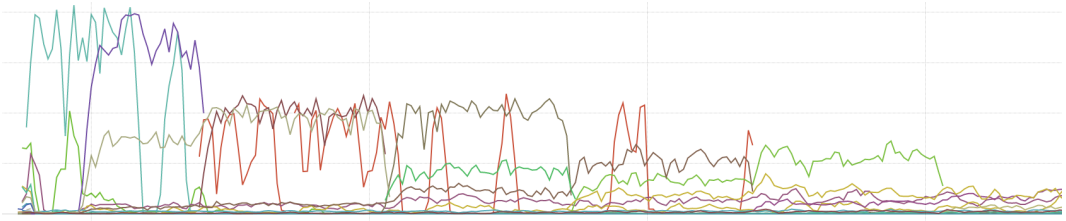

Figure 6.10. Microservices CPU throttling

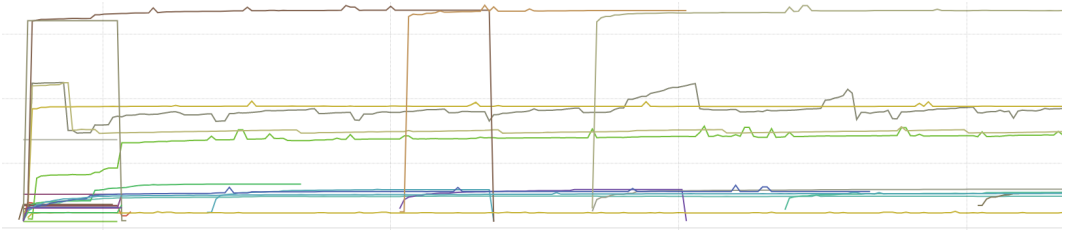

Figure 6.11. Microservice Memory consumption

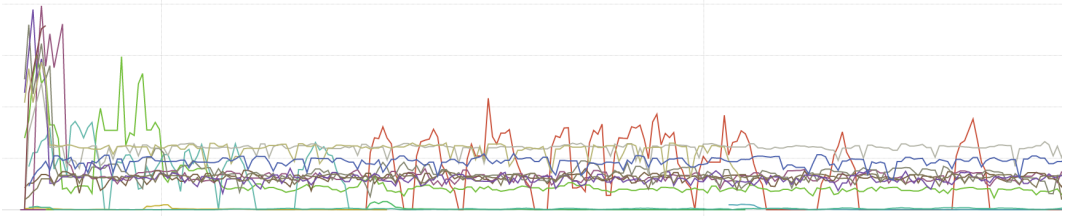

Figure 6.12. Microservice Memory errors

- by executing the microservices without the profiling system we realized that the microservices do not start to require more memory as the number of external requests grows. What is interesting to notice is that the profiling system is aware of that and so it does not find it necessary to increase them at runtime as we can see from figure 6.11, because even if there are some memory failure events (see figure 6.12) their number is very low so they do not affect the execution of the microservices

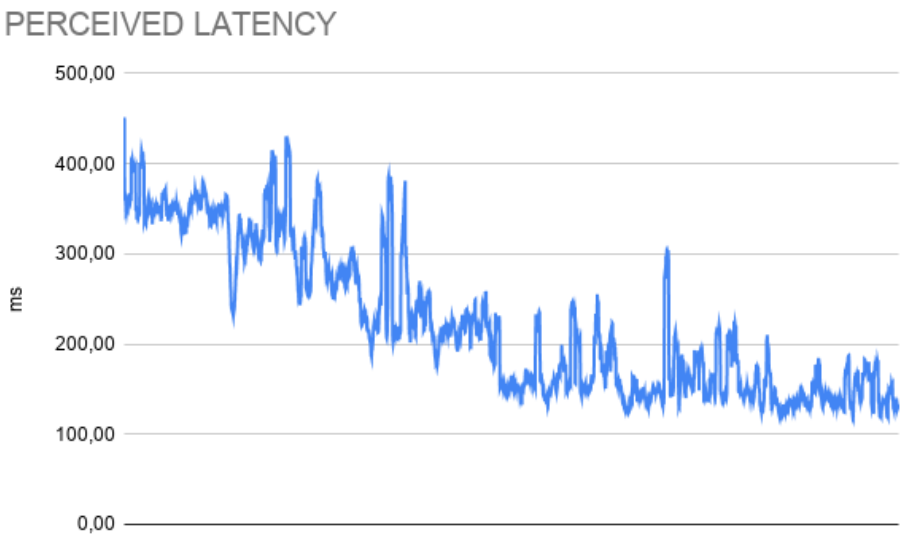

Figure 6.13. Perceived latency

After approximately 60 minutes of test we collected the values of latency perceived by the end user and we started to analyze them in order to understand the behaviour of the profiling system by comparing them to the ones obtained without our profiling system.

Looking at figure 6.13 we can see that over the time the latency perceived keeps reducing; in particular we defined 2 values in order to evaluate more in detail the overall performance of the system:

- settling time, defined as the time elapsed from the beginning of the test to the time at which the latency output has entered and remained within a specified band. In this case we estimated this settling time in approximately 50 minutes, which is quite impressive if we consider that the profiling system did not have past information about the microservices at the beginning of its execution. It worth noticing that the profiling system updates the resource associated to each microservice every 10 minutes meaning that this settling time has been reached after 5 iterations of the profiling algorithm
- the final latency perceived in the last 10 minutes is about 140ms; this result is quite impressive if we consider that the average latency perceived without the profiling system is approximately 120ms

In the end this test can provide us a very interesting feedback: within

50 minutes the profiling system is able to converge, adapting the amount of resources associated to each microservice to the actual workload of the system. What is even more interesting actually is that even if after 50 minutes the final latency perceived with/without the profiling system is comparable, the amount of resources allocated to the microservices is very different in the two cases (see Table 6.3).

		Profiling Disabled	Profiling Enabled
CPU	Requests	1625m	599m
	Limits	2825m	1198m
RAM	Requests	1368MB	550MB
	Limits	2542MB	1100MB

Table 6.3. Allocated resources

Chapter 7

Conclusions and Future Works

This work aims to propose a profiling system capable of enabling a job aware scheduling on Kubernetes cluster. In this first implementation we succeeded to:

- create a profiling system capable of providing information about microservice expected resource usage and communication patterns
- exploit a two level profiling able to predict the expected behaviour of any given microservice, adjusting that prediction at runtime by constantly looking at job performance

In Chapter 6 we evaluated the overall performance of the profiling system and we obtained the following feedback:

- the OTP has an average quality of prediction of 97% within the next 30 minutes from the profiling time
- the profiling system is able to converge, adjusting the resources associated to each single microservice to the actual workload of the system and providing an overall Quality of Service for the profiled application similar to the one experienced without any profiling tool
- the profiling system is not only able to meet applications Quality of Service requirements, but also can reduce significantly the difference between the limit and the actual resource usage for microservices

These results are very promising but analyzing these numbers there are a lot of possible improvements for the profiling system:

- the 97% average quality of prediction within 30 minutes is a good result but we would like to understand if it is possible to achieve the same quality in the next 60 minutes by correlating different metrics about microservices previous executions
- the 50 minutes (5 iterations) convergence time can be improved by balancing carefully the resources assigned to each microservice

Finally we would like to combine the profiling system and the custom scheduler developed within the Liqo project in order to test both the components in a real case scenario and get additional feedbacks useful for even further improvements.

Bibliography

[1] G. Xu C. Lu K. Ye. "Inbalance in the cloud: an analysis on Alibaba cluster trace". In: *BigData'17* (2017).

[2] Kostantinos Karanasos Carlo Curino Subru Krishnan. "Hydra: a federated resource manager for data-center scale analytics". In: *NSDI'19* (2019).

[3] *Evolution of Cloud*. URL: `https://the-report.cloud/the-evolution-of-cloud-computing-wheres-it-going-next`.

[4] *Golang Official Website*. URL: `https://golang.org/`.

[5] Y.Zheng H. Tian Y .Zheng. "Characterizing and synthesizing task dependencies of data-parallel jobs in Alibaba Cloud". In: *SoCC'19* (2019).

[6] *Istio Official Website*. URL: `https://istio.io/`.

[7] Jacek Swiderski Krzysztof Rzadca Pawel Findeisen. "Autopilot: workload autoscaling at Google". In: *EuroSys20* (2020).

[8] *Kubernetes Official Website*. URL: `https://kubernetes.io/`.

[9] *Liqo Official Website*. URL: `https://liqo.io/`.

[10] Nan Deng Muhammad Tirmazi Adam Barker. "Borg: the Next Generation". In: *EuroSys20* (2020).

[11] *Online Boutique Demo*. URL: `https://github.com/GoogleCloudPlatform/microservices-demo`.

[12] Li Zhang Xiaoqiao Meng Vasileios Pappas. "Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement". In: *2010 Proceedings IEEE INFOCOM* (2010).