Energy4Cloud - Energy-aware Scheduling with Docker Containers

Sérgio da Silva Mendes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Luís Manuel Antunes Veiga
Co advisor: José Simão

Examination Committee

Chairperson: Prof. Ana Teresa Correia de Freitas
Supervisor: Prof. Luís Manuel Antunes Veiga
Members of the Committee: Prof. Nuno Filipe Valentim Roma

November 2017

Acknowledgments

I would like to say a special appreciation note professor Luís Veiga for all the enthusiasm, dedication and help provided throughout the thesis development. This thesis would not be the same without his advisory. A special thanks to Sérgio Esteves for the help and comprehension provided throughout the implementation of my thesis. Another special thanks to José Simão for all the support and availability to provide assistance throughout the thesis. A final note to thank my family and friends for all the support provided.

Abstract

The ever increasing size of data centers and their energy demands brought the attention of the academia and a panoply of research exists regarding this area, however the problem persists. The emergence of containers brought new opportunities and the advantages they provide, can, and should, also be extended with energy concerns. Surprisingly, there is still not much work with containers where energy is concerned. To this end, in this thesis, a thorough analysis is performed on the state-of-art regarding the different types of containers, OS and application containers, their orchestrators, as well as the approaches that have been proposed by the literature to improve the energy efficiency on cloud environments, energy-aware mechanisms and strategies.

Having analyzed the different container platforms, the different orchestrators and the different strategies to optimize energy efficiency, this thesis proposes an extension to Docker's orchestrator, Docker Swarm, with an energy-efficient scheduling algorithm, based on maximizing resources utilization to levels where the energy efficiency is maximized. This solution improved CPU utilization by 5.6 *p.p* and 8.2 *p.p* over Spread and Binpack (Docker Swarm strategies) respectively, and improved memory utilization by 15.8 *p.p* and 18.9 *p.p* over the same strategies, during an one hour evaluation. Despite the comparatively longer scheduling times w.r.t other approaches, this is largely compensated due to the fact that our solution manages to allocate more requests, having obtained a successful allocation rate of 83.7% against 57.7% and 56.5% of Spread and Binpack respectively.

Resumo

O tamanho cada vez maior dos centros de dados e as suas necessidades energéticas chamou à atenção do mundo académico e existe uma panóplia de pesquisa sobre essa área, no entanto, o problema persiste. O aparecimento de containers trouxe novas oportunidades e as vantagens que eles fornecem, pode, e deve, também ser extendido com preocupações energéticas. Surpreendentemente, ainda não existe muito trabalho relativamente a preocupações energéticas com containers. Para este fim, nesta tese, é feita uma análise minuciosa ao estado da arte em relação aos diferentes tipos de containers, OS e application containers, e os seus orquestradores, bem como as abordagens que foram propostas pela literatura para melhorar a eficiência energética em ambientes em núvem, mecanismos e estratégias que melhorem a eficiência energética.

Tendo analisado as diferentes plataformas de containers, os diferentes orquestradores e as diferentes estratégias para otimizar a eficiência energética, esta tese propõe uma extensão ao orquestrador do Docker, Docker Swarm, com um algoritmo de escalonamento eficiente energéticamente, com base na maximização da utilização de recursos em níveis onde a eficiência energética é maximizada. Esta solução melhorou a utilização de CPU em 5.6 *p.p* e *8.2 p.p* sobre o Spread e Binpack (estratégias de escalonamento do Docker Swarm) respetivamente, e melhorou a utilização da memória por 15.8 *p.p* e 18.9 *p.p* sobre as mesmas estratégias, durante uma avaliação de uma hora. Apesar dos tempos de escalonamento serem comparativamente superiores às outras abordagens, isso é amplamente compensado pelo facto de que a nossa solução consegue alocar mais pedidos tendo obtido uma taxa de sucesso de 83.7% contra 55.7% e 56.5% do Spread e Binpack respetivamente.

Keywords

Cloud; Energy efficiency; Scheduling; Docker; Containers; Resource utilization; Resource monitoring;

Palavras Chave

Cloud; Eficiência energética; Escalonamento; Docker; Containers; Utilização de recursos; Monitorização de recursos.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

API	Application Program Interface
IaaS	Infrastructure as a Service
IP	Internet Protocol
OS	Operating System
PaaS	Platform as a Service
IoT	Internet of Things
CSPs	Cloud Service Providers
VMs	Virtual Machines
SLAs	Service Level Agreements
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
COM	Component Object Model
OSGi	Open Services Gateway Initiative
BSD	Berkeley Software Distributed
LXC	Linux Containers
ACR	App Container Runtime
ACI	App Container Image
ACD	App Container Discovery
IPC	Inter Process Communication

DNS	Domain Name System
AWS	Amazon Web Services
DVFS	Dynamic Voltage Frequency Scaling
PDUs	Power Distribution Units
PMCs	Performance Monitor Counters
MIPS	Millions Instructions Per Second
PM	Physical Machine
LLC	Last Level Cache
OQI	Oracle Query Interface
NAS	Network Attached Storage
AC	Admission Control
KOB	Knowledge DB
RA	Risk Assessment
CaaS	Container as a Service
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
LEE	Low Energy Efficiency
DEE	Desired Energy Efficiency
EED	Energy Efficiency Degradation
REST	Representational State Transfer
Sigar	System Information Gatherer And Reporter

1 Introduction

Hardware equipment throughout the years has been improving and we can expect that trend to continue. Despite this continuous improvement, the current hardware resources cannot deal with the ever increasing data processed, which consumes more and more hardware resources (e.g. Big Data applications [1]) and with the emergence of Internet of Things (IoT) [2] we can expect that even more resources will be required.

A solution to the insufficient hardware resources was the adoption of Cloud [3], which led to the creation of massive data centers with tens of thousands or even more, servers. This solution is appealing for businesses and people, who, instead of buying the hardware infrastructure, can rent it and continuously adapt it to their needs, with elasticity. Without Cloud, businesses would have to buy the hardware infrastructure and if they had a workload peak, they had two options, buy hardware that would only be used for a short period of time or do not buy the hardware and provide a poor user experience. However, as mentioned in the previous paragraph, due to emerging trends, more hardware resources are going to be required, consequentially increasing the size of data centers.

Besides operations costs, this increase will also reflect on the energy consumed by these massive infrastructures, which already consume a significant amount of energy, incurring high costs for Cloud Service Providers (CSPs) [4]. This recent report [5] shows that data centers consumed almost 416.2 terawatt hours of energy in 2015. To understand how big this value is, this is higher than the yearly United Kingdom energy consumption. Besides the costs for CSPs, this amount of energy required has significant environmental consequences [6]. These issues bring the urgent need for energy-aware policies for cloud environments [7].

Cooling accounts up to 50% of the energy costs, servers (and storage) for 26%, 11% for power transformation, 10% for network network equipment and 3% for lighting [8]. Cooling costs are highly correlated with machine energy costs. If we can reduce the energy consumption or maximize the energy efficiency of a machine, besides reducing energy costs, less heat will be dissipated, requiring less cooling, thus reducing cooling costs.

1.1 Current solutions

On traditional cloud environments (e.g. data centers), virtualization using Virtual Machines (VMs) has been extensively used to enhance resource utilization [9–11]. This enhancement of resources provided by VMs, rose as an opportunity for many different solutions for improving the energy efficiency and/or reducing energy consumption (e.g. VM consolidation) to be proposed, as will be seen in more detail on Section 2.2.

However, containers [12] have been proposed as an alternative to VMs to virtualize resources. Containers are more lightweight than VMs, containing only the required application binaries to run a specific process and nothing more, not requiring a full guest Operating System (OS) instance. Since they are significantly more lightweight than VMs, a better resource utilization can be achieved using containers. Achieving an even better resource utilization than VMs and considering that VM energy-aware strategies already provide a significant reduction on energy costs, containers are an excellent opportunity to further increase this reduction.

1.2 Challenges

Having energy concerns also has drawbacks that need to be considered. There is always, at least, one tradeoff that is inevitable. It can either be, longer time to schedule requests or reduced Quality of Service (QoS), regarding response times of the workloads that were scheduled. These tradeoffs have to be mitigated, even if the solution improves energy efficiency/consumption significantly, because of the clients. Clients don't necessarily care about energy savings for the CSP, if it does not benefit them. In order to enforce these policies, the client must either be compensated (e.g. reduced price) or the tradeoffs must be completely mitigated or transparent, which is hard due to the complexity of making such decisions as will be seen.

Choosing the energy strategy must also be carefully deliberated, being dependent on many factors that need to be considered. The first and most important is, what is aimed to improve, maximizing energy efficiency or reducing energy consumption. These might seem the same but, there are completely different strategies for either of these two types of improvements in the literature, so it is important to distinguish them. In the literature, most of the works focus on improving the energy efficiency of the machines, which is mostly achieved through maximizing resource utilization, making better use of the energy being spent. If the goal is to reduce energy consumption, some different approaches are required such as turning off application components that are not being used or turning off machines to save energy. However, both approaches could be combined. For example, if resource utilization is maximized, thus increasing energy efficiency, fewer machines are required to run the same requests,

therefore the energy consumption also lowers.

Another important factor that needs to be taken into consideration is the size of the environment. While public clouds have a very dynamic environment, where the workloads executed and resources used vary arbitrarily, there are private clouds that have a static environment. Therefore, the strategy is highly dependent on the environment and must adapt to it. But not only on the environment, on the type of workload as well. Approaches where the strategy relies on estimations on environments where the workloads are constantly changing [13, 14], might result in slow scheduling decisions since it is always resorting to estimations and it may also not produce the expected results. If the estimations are wrong by a simple value as 1%, in a big environment, this could have significant consequences.

The last major factor that needs to be considered are the monitoring tools to be used. As will be seen on Section 2.2.1, there are different solutions to choose from. The approach to choose, again, depends on some factors. For example, there are approaches that are suitable for private clouds but not for public clouds, since they are intrusive, requiring changes to the internal of VMs. Another aspect to be considered is what should be monitored, yet again depending on many factors. As an example, if the workloads are only CPU intensive, there is no point in monitoring memory consumption.

The aforementioned challenges are for VMs but all of them apply to containers since they are used for a similar purpose. Despite already existing many solutions for all these different challenges, the state-of-the-art regarding energy-aware strategies for cloud environments mostly focus on using VMs and not containers. In our research, we only managed to find two works that takes both energy and containers into consideration [13, 15]. The first has some limitations due to the of usage computationally intensive computations (through the use of X-means) which can be an overkill on real cloud environments. The second work approach can lead to hosts not serving any requests due to using a static amount of hosts for profiling and others for long duration requests. Therefore, if there are no requests to be profiled or there are only short duration requests, those hosts will not be used, wasting energy. Both these works will be addressed in detail on Section 2.3.

Energy is also not considered on the current platforms for managing containers (e.g. Docker, Rocket). Their decisions, e.g. scheduling a container, do not use any energy-aware strategy.

1.3 Objectives

The lack of state-of-the-art approaches to schedule containers, taking into consideration such an important issue as is energy, provides a good opportunity to contribute to the literature with a solution that provides energy-aware scheduling for containers on cloud environments. Thus, we propose a schedul-

ing algorithm that promotes energy efficiency in the context of cloud environments, managed by Docker containers, based on maximizing resource utilization according to levels of energy efficiency, without violating Service Level Agreements (SLAs). We have developed a prototype of the solution in order to evaluate it in a realistic environment. The evaluation was performed according to a set of relevant metrics drawn from related work such as CPU and Memory utilization over time, comparing with relevant related systems.

As could be seen on previous section, to improve energy efficiency, many factors have to be considered, depending on the case at hand. Therefore, producing a solution that works well on all scenarios is impossible. However, we established a middle-ground and proposed a solution to works wells on most cases, independent of the environment and the type of workload. The strategy proposed on the previous paragraph, was also selected with this concern in mind.

1.4 Contributions

With the challenges and objectives identified, this thesis makes the following main contributions:

- An analysis about containers. How they came to be, which platforms exist to deploy them and which platforms exist to orchestrate them;
- A thorough analysis of the different energy-aware strategies and the mechanisms that complement these strategies, and how they are applied in works performed by the academia;
- Functional prototype of the proposed solution;
- The cut concept, increasing even further the resources utilization levels provided by an overbooking strategy. The kill concept preventing machine resources from being exaggeratedly used and maintaining QoS levels;
- An evaluation that highlights the urgent need to deal with the resource inefficiency that currently exists in cloud environments.

1.5 Document roadmap

This thesis is organized as follows: Our research on the related work about our proposal is described thoroughly on Chapter 2. Chapter 3 presents our proposed solution to accomplish the objectives proposed on this Section. Chapter 4 details how our solution is implemented. Chapter 5 presents how we evaluated our proposed solution. Chapter 6 concludes and addresses some system limitations and future work.

2 Related work

In this section we present our research on the most relevant topics regarding our work. First, on Section 2.1, we provide an overview about components and containers. Next, on Section 2.2, we present the relevant topics to energy-awareness, such as energy monitoring and mechanisms, finishing with addressing energy-aware optimization strategies. Finally, on Section 2.3, we present the relevant systems to our work based on what was described earlier.

2.1 From Components to Containers

To understand components and containers, we need to go all the way back to the 1970s where the concept of modular programming started to appear, with extensions to the ALGOL language. In the late 1970s, the first modular programming languages were developed, Mesa [16] and Modula [17]. The concepts of modular programming of that time are still valid today. Modular programming is the process of dividing a computer program into separate programs (called modules). These modules are independent from each other and can be reused to serve other applications besides the one that it was originally designed to.

It's interesting to have modular applications since they are simpler to design, develop, load and share. Their main purposes are to improve flexibility, comprehensibility and reusability [18]. These characteristics of modules also characterize components and containers and are their basic building blocks, however, the former serve different purposes than the latter. While modules are only used to add functionality to an application, components and containers can also be used to directly deploy/launch an application, as we will see next.

2.1.1 Components

When developing an application, if we want to implement a certain functionality, in many cases, someone else already implemented it. Given this, there is no need to waste time reinventing the wheel, we just have to search for that functionality. This *functionality* is usually referred to as a component. Components motivated the creation of component models. Their purpose was integrating different com-

ponents and managing them. They define how components are constructed, specified, deployed and connected among each other. Using these models, components are more resource efficient by having a simplified resource management that enables control over the application life-cycle. In these component models, components interact with each other by calling methods through a standard Application Program Interface (API), therefore allowing different components to interact with each other.

Common Object Request Broker Architecture (CORBA) [1] was launched in 1991 by the OMG (Object Management Group). It was the first component model. CORBA's purpose was to enable software components written in any programming language, to be able to work together, regardless if they are running on a single machine or if it is distributed (also regardless of the underlying operating system). Their main goal was to make everything compatible with each other.

Microsoft's response to CORBA was **Distributed Component Object Model (DCOM)**, a distributed version of Component Object Model (COM) [19]. Its purpose was the same as CORBA, communication between software components, but less ambitious than CORBA since DCOM only worked in Windows systems. However both, CORBA and DCOM, and other versions of component models created at that time, Koala [20] and SOFA [21], had significant limitations mainly due to their implementation complexity and scalability problems [22]. Despite these partially failed approaches, they inspired other component models to be implemented [23]. The one presented next, OSGi, is the most successful to date.

Open Services Gateway Initiative (OSGi)[2] was created by the OSGi Alliance in 1999. OSGi provides a Java framework for integrating Java-based components and is the only component model that provides a dynamic component system [23].

Bundles is the OSGi definition for components that provide services that can be used by other bundles. Bundles are deployed on an OSGi framework, which is the bundle runtime environment. Each bundle has its context isolated from other bundles. Services are Java objects and bundles can register them in the Service Registry, which keeps track of the services registered within the framework [24], allowing other bundles to use that service.

The framework is responsible for controlling the life-cycle of bundles and here is where they achieve the dynamic properties that no other component model has. In OSGi[3], bundles can be installed, started, stopped, updated and uninstalled without restarting the application, while preserving the dependencies with others bundles. They refer to the Execution (runtime) Environment as a *collaborative* environment since bundles run in the same Java VM and can share code among each other.

[1] http://www.corba.org/
[2] https://www.osgi.org/
[3] https://www.osgi.org/developer/benefits-of-using-osgi/

OSGi has the limitation that it only allows Java applications to be deployed. Containers are much more flexible and allow a much broader range of applications to be deployed only depending on the underlying OS.

2.1.2 Containers

It all started in 1982 when the *chroot* command was added to Berkeley Software Distributed (BSD) systems and later to the other Unix-based OSs. The purpose of this command was to create an isolated environment for processes, still sharing the same kernel. This was achieved by using chroot to change the root directory to the base directory that contains all the system files required for that process to run. This new root directory is also called *chroot jail*.

In 2000, FreeBSD explored this idea with *jails*, having more security features. Using chroot, processes are limited only to the part of the file system that they can access, everything else is still shared between processes. Jails virtualizes access to the file system, set of users and the networking subsystem [25]. The virtualization achieved by FreeBSD was very promising and new technologies quickly started to appear, e.g. Solaris zones [26].

The concept of *zones* was introduced by Solaris in 2004 [27]. A zone is a virtualized OS environment created within the Solaris OS. In zones, applications are completely isolated from each other and access to OS resources are centrally managed and administered. While jails has limitation on what it virtualizes as mentioned before, zones provide full virtualization of an instance of the Solaris OS.

Jails and zones were the basic building blocks for the containerization-based technologies we know today. The state-of-the-art container technologies can be categorized in three ways: **mechanisms**, **platforms** and **orchestrators**. We will start by describing mechanisms (which we will refer from now on as **OS containers**), then we will go into the platforms (which we will refer from now on as **application containers**) which are the most widely known type of containers (e.g., Docker). Finally we will cover the **orchestrators** (also known as **schedulers**) that allows application containers to be efficiently deployed.

OS containers: OS containers (also known as system containers) are very similar, in principal, to VMs. Like VMs, one can install and run different applications and also as VMs, everything is isolated from other VMs that run on the host OS. The main difference from VMs, is that instead of having its own kernel (i.e., a guest OS), each OS container shares the same kernel (the host kernel) with other OS containers, making them more lightweight than VMs. This isolation is provided through cgroups (control groups) and namespaces [28]. cgroups provides resource management mechanisms where it is possible, for example, to limit the amount of memory used by an OS container. Namespaces provides

sandboxing, limiting system resources access, i.e., not allowing a certain OS container to interfere with another OS container's system resources running on the same host. FreeBSD Jails and Solaris Zones mentioned before, are examples of OS containers. Other examples include Linux Containers (LXC), OpenVZ[4] and Linux VServer[5].

The most popular OS container is **LXC**[6] and it served as a basis for the applications containers that we will explain in the next. IBM launched LXC at 2008. Its purpose is to create and manage Linux containers which share the underlying kernel, each one being isolated from each other, which is achieved through the use of cgroups and namespaces, as explained before.

Application containers As mentioned before, in OS containers, multiple processes can be launched. Using application containers, only one process is launched per container. This is the main difference between these two types of containers.

In the previous subsection, we mentioned that applications could be built through the combination of different components. This can also be achieved with application containers and it actually outperforms component-models. This *component-based containerization* can be achieved by creating different containers for each component and when that component is no longer required, the container terminates making the application more lightweight, and in contrast, using OSGi for example, all components must always stick with the application until it ends, even though they may no longer be required.

Like OS containers, application containers are built from images. Each application will have its own image, containing only the required application binaries for it to run and nothing else, making it lightweight. If it is required to run the same application multiple times, all the containers used to launch that application can use the same image.

Next, we will describe in detail, the state-of-the-art in application container technologies: Rocket and Docker. We describe their architectures, their main features and what is a container in each approach's perspective.

Rocket: CoreOS launched in 2014, an alternative to the already popular Docker, called Rocket (or rkt) providing, essentially, more security guarantees than Docker, thanks to their daemon-less approach. In Rocket, a container is specified through an *App Container*[7] with the following three modules: **App Container Image (ACI)** , **App Container Runtime (ACR)** and **App Container Discovery (ACD)**. ACI specifies the image that the container will use to run its application. ACI can be encrypted providing further security and allowing them be shared in a secure fashion. ACR defines the environment and facilities that a container runtime should provide, like, devices, environmental variables and privileges.

[4] https://github.com/OpenVZ
[5] http://www.linux-vserver.org/
[6] https://linuxcontainers.org/
[7] https://coreos.com/rkt/docs/latest/app-container.html

ACD is the protocol to find and download an application container image (e.g. a NGINX server).

The deployable and executable unit in the App Container specification explained above are pods[8]. A pod is a list of applications that are going to be launched together sharing an execution context. Pods allows applications to perform Inter Process Communication (IPC), they can use the same Internet Protocol (IP) and port space, applications are aware of each other and they share a hostname.

Running a container in Rocket involves three stages. In stage 1, the container is prepared, creating the filesystem for the container and downloading the ACI into the directory that was just created for the container. Stage 2 is responsible for adding isolation (different levels of isolation can be configured) to the container and other configurations, e.g., the maximum amount of RAM it can use. This is achieved by configuring cgroups, namepsaces and mount points. Now everything is set up and the application inside the container, is ready to be launched which is done in stage 3.

Docker: When it was launched in 2013, Docker used LXC as its execution environment; however two years later, at version 0.9, they created libcontainer and made it the default execution environment, LXC is still supported though. Libconrtainer[9] provides more security (enabling the use of AppArmor[10]) than LXC and made the execution environment more stable since Docker can now manipulate cgroups, namespace and other configurations without depending on LXC. This also allowed to avoid problems concerning different versions and distributions of LXC.

Docker has two major components: the Docker Engine, which is the containerization platform, which we describe next explaining its architecture, and the Docker Hub.

Docker architecture is depicted in Fig. 2.1. Rocket is daemon-less, that is, when a command is issued in Rocket, it executes directly under the process that started it. Docker does things differently and uses a client-server architecture where the Docker Client issues commands to a Docker Host, where there is a long-running process running called daemon. This daemon is responsible for performing all the configurations required to launch a container.

Docker images are a read-only template from which Docker containers are instantiated (e.g., Nginx image). When a container is created, a new layer is added to the base image. All changes made to the running container only affect the added layer. This allows multiple containers to share the same base image because any changes they make only affect the layer added when that container was created, therefore not effecting the base image. When the container ends its job, this added layer is also deleted.

[8] https://github.com/appc/spec/blob/master/spec/pods.md
[9] https://github.com/opencontainers/runc/tree/master/libcontainer
[10] https://help.ubuntu.com/lts/serverguide/apparmor.html
[11] https://docs.docker.com/engine/understanding-docker/

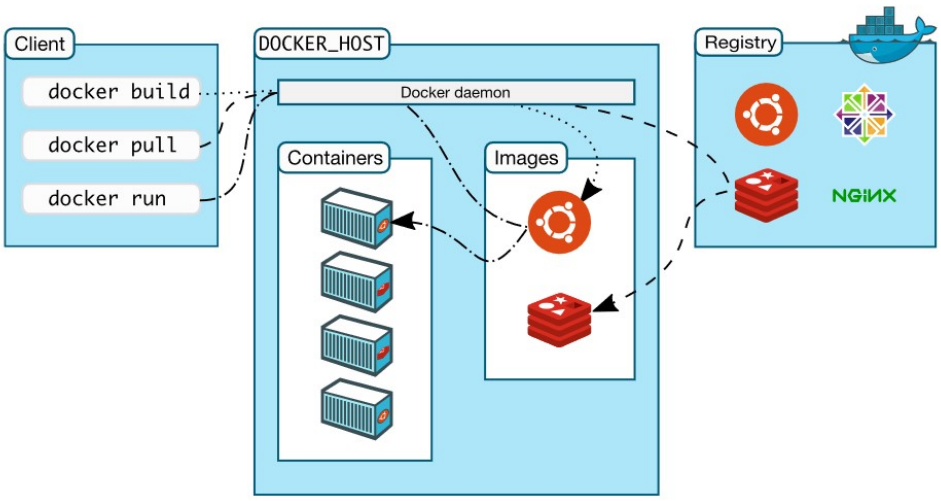

Figure 2.1: Docker Architecture[11]

An image is automatically built by reading instructions from the Dockerfile[12]. We can edit the Dockerfile to configure the image, like adding file/directories or adding environmental variables.

As mentioned previously, when a container finishes, the layer corresponding to that container is deleted. Volumes can be used to store data persistently on the Docker host, even after a container finishes. A data volume is a directory or file in the Docker host's filesystem that is mounted directly into a container. It is possible for multiple volumes to be mounted for a container and multiple containers can share one or more volumes. Docker registries are libraries of images and can have private or public access. An example of a public registry is Docker Hub.

Other important components of Docker are Docker Machine[13] and Docker Compose[14]. Using Docker Machine it is possible to install and manage Docker Engines (or Docker Hosts) on virtual hosts and manage them, whether it is locally, at a data center or a CSP. Docker Compose is a tool for defining and running multi-container Docker applications, configured through the Compose File. This is Docker approach for implementing the component-based containerization as mentioned when we introduced Application containers. Docker Compose therefore allows a single application to be built from multiple containers, defining how they work together and how are they linked.

Analysis: Due to being daemon-less and not executing as root (as opposed to Docker which the daemon runs as root), Rocket provides more security guarantees than Docker. It is also simpler than Docker as could be seen above by the number of the different features Docker has in comparison with

[12] https://docs.docker.com/engine/reference/builder/
[13] https://docs.docker.com/machine/overview/
[14] https://docs.docker.com/compose/overview/

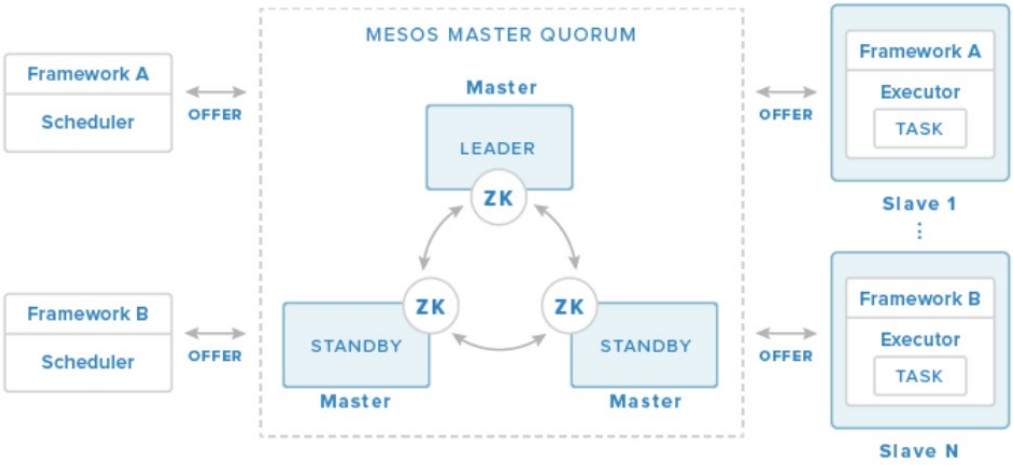

Figure 2.2: Mesos Architecture[16]

Rocket. However this simplicity is also one of Rocket disadvantages, since the extra features in Docker can be very useful. As an example, Docker layering although it introduces a (small) overhead, its benefits (e.g. reducing disk usage) in allowing the reuse of images, compensate for that overhead. Also, Rocket is still in the process of maturation while Docker is already a stable solution.

2.1.3 Orchestration platforms

Now that we know what are the current mechanisms and platforms for deploying containers, next, we will look into the state-of-the-art orchestration platforms that allow the creation and management of clusters of containers: Mesos, Kubernetes and Docker Swarm. We will categorize them in terms of the most important features of an orchestrator: architecture, fault-tolerance capabilities, scheduling algorithm(s), and service discovery.

Mesos: Apache Mesos [29] is a cluster manager and was released in 2011. We will also present Marathon[15] which is the most widely used framework for managing container orchestration for Mesos. Fig.2.2 presents Mesos architecture.

The Mesos Master is responsible for managing Mesos Agents running on each cluster node and Mesos frameworks (which in this case is going to be Marathon, other examples include Hadoop) that run tasks on the agents. The frameworks are composed of schedulers and executors. The former registers with the master to be offered resources (e.g., amount of RAM available) while the latter is a process that is launched on agent nodes to run the framework's tasks. The master decides how many resources to be offered to each scheduler according to policies, such as fair sharing or strict priority [30].

[15] https://mesosphere.github.io/marathon/
[16] https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere

Once the framework schedulers receives the offers, taking into account the policy chosen, the scheduler selects which of the offered resources to use. Once they're chosen it passes to the master a description of the tasks it wants to run on them and finally, the master, launches the tasks on the appropriate agents. Mesos uses Apache Zookeeper [31] for providing fault-tolerance guarantees.

Marathon, like Kubernetes and Docker Swarm, provides scalability to the cluster by automating most of the monitoring and management tasks. It provides the following features: **Constraints**[17] control where the container is going to run. The constraints consists of three parts: a field name, an operator and optional value field. The field name can consist of the agent hostname or an agent attribute (a tag on the agent node). An operator can be of several types such as: **UNIQUE** (forces uniqueness, e.g., through this, we can ensure that there is only one application instance running globally); **MAX PER** (can be used to limit tasks across nodes).

Health checks[18] is another fault-tolerant implementation. It provides detailed information about the status of applications and allows the developers to specify what should happen if an application fails a health check (e.g.,terminate it and launch a new one on another node).

Service discovery[19] is required in order to send data to containers, from containers of the same cluster or from external sources. Mesos already offers a service discovery mechanism called Mesos-DNS which like the name indicates, uses Domain Name System (DNS). Marathon also provides a service discovery which implements a TCP/HTTP proxy on each host, transparently forwarding connections to the static service port on localhost, to the dynamically assigned port of the tasks.

Kubernetes: Google released Kubernetes in 2014 as another solution for orchestrating containers. A year later, Google donated the project to the Cloud Native Computing Foundation, which is a partnership from Google and the Linux Foundation.

The basic working unit of Kubernetes are pods[20]. A pod is a group of one or more containers that can be, one or more applications, which are tightly coupled. This leads to containers inside a pod to be scheduled to the same host and share the same context, which means they share volumes and an IP address space, therefore being able to connect with each other via localhost.

Kubernetes architecture is composed of Master Components[21]. Master Components provide cluster management, being responsible for making global decisions about the cluster (e.g., scheduling) and, detecting and responding to cluster events (e.g., when a failure occurs, deal with that failure). It consists of the following main components:

- Etcd;

[17] https://mesosphere.github.io/marathon/docs/constraints.html
[18] https://mesosphere.github.io/marathon/docs/health-checks.html
[19] https://mesosphere.github.io/marathon/docs/service-discovery-load-balancing.html
[20] http://kubernetes.io/docs/user-guide/pods/
[21] http://kubernetes.io/docs/admin/cluster-components/

- API Server;
- Controller Manager;
- Scheduler.

Etcd is a distributed key-value store. Kubernetes uses it to store configuration data that can be used by nodes. It can also be used for service discovery. *API Server* is one of the core components of Kubernetes. Through it are performed pods and workloads configurations. The *Controller manager* is responsible for running and managing controllers, which are background threads that regulate the state of the cluster and adjust to it. It also handles routine tasks in the cluster, such as: *Node Controller* - responsible for detecting nodes that fail and deal with that failure; *Replication Controller* - ensures that the correct number of replicas of pods are up.

The process of assigning pods to nodes is performed by the *Scheduler*. To perform this assignment, two steps are required. First, all nodes are filtered according to predicates. Kubernetes has predicates based on volumes (e.g. Amazon Web Services (AWS) volumes), resources (e.g. if memory available meets pod's requirements) and host information (e.g. host with a certain port). Predicates are similar to Marathon's constraints but different rules are used by both. The purpose of this filtering is to filter out nodes that do not meet the requirements of the pods, therefore making the scheduling algorithm simpler since it has less alternatives to schedule the pod to. The next and final step after filtering, is applying priorities to the filtered nodes. The priority function will rank the node from 0-10, 0 being least preferred and 10 being most preferred. The node with the highest score is the one the pod is assigned to. Kubernetes provides multiple scheduling functions to be used. As an example, *BalancedResourceAllocation* purpose is to put the pod into a node such that the CPU and Memory utilization is balanced after the pod is deployed.

Docker Swarm: Developed by Docker, it aims to be the standard orchestrator for Docker containers. Its architecture comprises of manager nodes and worker nodes. Like Mesos and Kubernetes, Docker Swarm[22] is also a centralized orchestrator and that role is played by the manager nodes, which are responsible for scheduling containers to worker nodes, and managing them.

Like any other centralized approach, fault-tolerant mechanisms are required for the manager nodes otherwise it would be a single point of failure. Docker Swarm uses the Raft consensus algorithm [32] for fault-tolerance.

Docker Swarm uses a distributed key/value store for service discovery, which they recommend to use their own implementation, libkv[23]. However, others are also supported, e.g., Consul. They also support DNS for service discovery.

[22] https://docs.docker.com/swarm/overview/
[23] https://github.com/docker/libkv

As mentioned earlier, the manager node is responsible for scheduling containers to the workers nodes. The scheduling algorithm used by Docker Swarm is similar to Kubernetes's algorithm, it also has two steps. It starts by filtering the worker nodes before applying the scheduling algorithm. For this purpose, it has two types of filters[24], (similar to Marathon's constraints and Kuberneter's predicates) node filters, which filters based on characteristics of the worker nodes, and container configuration filters, which filters based on characteristics of the containers. There's currently three types of nodes filters:

- Constraint
- Health
- Containerslots

Worker nodes can be tagged, having an associated key/value pair to it. The constraint filter can be used to select worker nodes with a certain tag (e.g. type of OS). The health filter prevents running containers on unhealthy nodes, that is, if the node is down or can't communicate with the cluster store. Containerslots filter can be used to limit the amount of containers on a worker node. The other type of filters, container configuration filter, also have three types:

- Affinity
- Dependency
- Port

The affinity filter is used to schedule containers next to containers that can fill the following criteria: container name or id; an image; a custom label applied to the container. The dependency filter is used to schedule containers that depend on other containers. This dependency is related to having a shared volume, or a certain container is dependent on another to run or being on the same network. A port filter is used if we want to run a container on a specific port.

After the worker nodes are filtered, they go through one of the three scheduling algorithms[25]. **Random** is one of the algorithms, and as the name indicates, the worker node is chosen randomly. **Spread** strategy chooses the worker node dealing with fewest containers. The last strategy is **Binpack**, which causes Docker Swarm to choose the node that is most packed (i.e., has the minimum amount of CPU/RAM available).

Analysis: Table 1 presents the summary on the orchestrators described above. From our study we can conclude that Docker Swarm has the simplest architecture with just two entities, manager nodes and worker nodes, while Kubernetes has the more complex architecture having at least four separate

[24] https://docs.docker.com/swarm/scheduler/filter/
[25] https://docs.docker.com/swarm/scheduler/strategy/

Orchestrator	Architecture	Scheduling algorithm	Fault-Tolerance	Service Discovery	Docker API
Docker Swarm	Manager and worker nodes	Two step algorithm. Also uses filters to couple similar containers	Manager replication using Raft consensus algorithm	Distributed key-value store or DNS	Yes
Mesos (using Marathon framework)	Mesos Masters and Agents	Resource offering-based (Five step algorithm). Also uses constraints to couple similar containers	Apache Zookeeper and health-checks	DNS-based or TCP/HTTP proxy	No
Kubernetes	Master components	Two step algorithm	Controller Manager	Distributed key-value store	No

Table 2.1: Orchestrators summary

entities. Regarding scheduling, Kubernetes has the simplest algorithm thanks to pods, which avoids the usage of filters (by Docker Swarm) and constraints (By Mesos) to co-relate similar containers. Docker Swarm is the less robust only replicating manager nodes while Mesos with Zookeeper and with health-checks provide a good reliability. Kubernetes and Docker Swarm use a similar approach for service discovery while Mesos uses DNS and TCP/HTTP proxy which can provide a slightly bigger overhead than the distributed key-value store approach. Finally Docker Swarm uses the standard Docker API which simplifies the development.

2.2 Energy-awareness

Energy consumption in Cloud environments (e.g., data centers) is a problem that persists throughout the years despite all the efforts done by companies and academia [33]. Servers have been identified has the main source of power consumption in data centers [34] and they also contribute to cooling costs due to the energy to the heat they generate and needs to be dissipated.

To try and deal with this issue, energy-aware mechanisms and energy-aware optimization strategies have been proposed. In this subsection, we will describe the state-of-the-art regarding those two areas for Cloud environments.

2.2.1 Energy Mechanisms

Before applying strategies to schedule workloads in an energy-efficient manner, it is necessary to have mechanisms that help us make decisions. Throughout this section we will describe these mechanisms, starting with mechanisms for monitoring energy consumption and finishing with Dynamic Voltage Frequency Scaling (DVFS), which is an intervention mechanism.

Energy-Aware Monitoring: A real and accurate monitoring can only be achieved through monitoring hardware. At the OS layer, we can also achieve an accurate monitoring since its reproducing metrics collected from the hardware. There are other approaches that measure energy consumption at the Middleware and Application layer. At these layers there are two possibilities, either they are reproducing

metrics collected from the OS and the hardware, or it involves estimating energy consumption. However, it is hard to measure energy consumption at the Middleware and Aplication layer, as an example consider an application using a library. There is nothing at the hardware layer that identifies the library therefore estimation is required. Before the library is called, we take a measurement of the current energy being consumed by the machine, then when the library is ran, we take another measurement and compare them both. This is a very primitive approach but is useful to highlight what could go wrong in estimations. Between measurements, other processes could be launched, there could be a workload peak, many situations can occur that will influence the measurement. Even harder is to estimate in a cloud environment where the machine is shared between different VMs, some of which, we may not have control and know what they are running.

There are three main challenges in monitoring the energy consumption of VMs [35]. Typical approaches for measuring service power accurately cannot be directly used for VMs and there is no hardware that can measure the energy consumption of each VM. Second, the power consumption of a VM can be seen as the sum of the energy costs of all the hardware resources consumed by the VM. However, the power consumption of hardware is not static and changes significantly from application to application, so, it is not easy to measure the energy consumption of hardware resources. As mentioned in the previous paragraph, a machine is shared among several VMs. The energy consumption of a VM is affected by co-located VMs and is very difficult to distinguish which VM is using which hardware resources. The final challenge, energy consumption of a server can be divided into two parts: static and dynamic power [36] and these must be taken into account when measuring energy consumption. Static power is consumed whenever a server is turned on (e.g., in an idle state), while Dynamic power is consumed when the server executes instructions or other operations.

Despite all these challenges, some research has been done on this area and several approaches have been proposed: white-box, black-box and other relevant approaches. All these ideas share these common steps [35]:

1 Collect resource information (usually CPU and memory) for calculating the energy consumption. Measuring physical server energy consumption can be done through an external meter, normally Power Distribution Units (PDUs), but there are others approaches [37]. An alternative to external meters, are internal meters such as power sensors or a special motherboard. External meters have the advantage that they are non intrusive, meaning that attaching or detaching them do not affect the system. However its unfeasible for cloud environments where thousands of servers are used. On the other hand, internal meters are intrusive, since they will influence the information measured. However contrasting with external meters, they are easily deployed and managed. Performance Monitor Counters (PMCs) [26] are

[26] https://www.codeproject.com/Articles/8590/An-Introduction-To-Performance-Counters

counters that are often used. They record the accumulation values of registers or events of the system.

2. After we got the raw information (resource usage), we must run this information through a power model which will convert this information into the energy consumption. There are numerous power models in the literature, including CPU [38], memory and CPU [39], and more [8].

3. The final step consists in estimating the energy consumption of each VM using the information collected in 1 and the model chosen in 2.

White-box approaches: These techniques use information collected inside the VM to create power models that indicate the energy consumption of each VM. To achieve this, a proxy program is inserted into each VM to collect resources utilization or PMC events. The proxy will send the information collected to an external module, which collects that information and information from the host. Once it has enough information from both the VM and the host, the external module will send both information to a different module that is responsible for using a power model and estimate the energy consumption.

This is a simple technique but has a problem. The proxy program must be inserted into VMs and that is not possible in public clouds, unless authorized by clients. Also, the accuracy of the information collected from the VMs is difficult to predict.

Black-box approaches: In this approach, information from each VM is collected at the host OS and/or at the hypervisor, without VM modification. The difficulty lies in distinguishing which information belongs to each VM. There are several works using this approach. They differ from the type of information used, some CPU, memory and I/O [40], in this case and most cases, the only I/O operations considered are disk since others are small and can be neglected. Other approaches use only CPU and memory, arguing that I/O operations are small and can be neglected [41], other techniques only use PMCs and many more different techniques as can be seen here [35].

Other approaches: The problem of the two approaches above is that they introduce additional overhead because of all the extra mechanisms required to use those models. Also, those models are chosen assuming that the VM always has the same characteristics. As an example, if a VM is only running CPU-intensive applications, then a power model taking only CPU into consideration is enough, but if the VM suddenly runs an application that is more memory-intensive than CPU-intensive, the energy consumption measurement for that VM would be completely wrong.

Using software instrumentation on source code, it is possible to know at runtime if it is a CPU-intensive application, memory-intensive, etc, and choose the appropriate power model based on that. The problem of instrumentation is that it cannot be done on cloud environments since CSPs cannot instrument applications running on VMs, unless authorized by the clients, but even so, instrumentation

itself would add more overhead to the application and would impact the energy consumption of the VM.

Analysis: At which layer to perform the monitoring depends on the situation. If, for example, we want to perform migration of applications depending on their resource utilization, then we need to monitor applications at the Middleware or Application layer. Using external meters to monitor energy consumption is not feasible in cloud environments due to the vast number of external meters that would be required. White-box and software instrumentation approaches are also not feasible for Cloud environments due to their intrusiveness. In our view, the only approach that is feasible in cloud environments, is the black-box approach since it is not intrusive and does not require additional hardware.

DVFS: It is incorporated into almost every processor and is a classical mechanism for decreasing energy consumption. However, it is still used in recent works and achieves promising results [42–44]. It is a mechanism that can dynamically change the frequency and the voltage of a CPU and/or memory. Although CPU is usually what consumes more energy, memory, also consumes an amount of energy that should not be neglected [45]. Being dynamic, allows it to adjust to the current demand, limiting clock frequency and voltage in periods of low demand or idle items, increasing it back when necessary.

This mechanism must be carefully used since reducing CPU frequency (for example) can have significant performance penalties and can violate SLAs, which can cause discontent among CSP clients. If used carelessly, it can even increase energy consumption since tasks would require much more time to complete, therefore potentially consuming more energy. The overall challenge is choosing the right setting in order to lower energy consumption and at the same time, sacrificing the less performance possible and not violating SLAs.

2.2.2 Energy-aware optimization strategies

There is a lot of work performed on this area and many approaches are proposed as can be seen here [46], where the main goal is generally optimizing resource utilization which consequently, improves energy efficiency and allows less hosts to be used, saving energy costs. Virtualized approaches are the main contributors for energy-aware optimization strategies as we will see next, however there also other approaches. For the remainder of this subsection we will describe these strategies.

VM Placement: Here we will address the initial placement of the VM which plays a critical role. If the VM is misplaced (e.g., to an overloaded host), we are just wasting resources, since that will cause that or another VM to be migrated (or another energy-aware strategy to be executed) to reduce the load at the overloaded server. The work in [46] identifies two types of main VM placement strategies: centralized and hierarchical. Centralized approaches assume the existence of a centralized structure that

has information about the whole infrastructure. They take advantage of this centralized information to make the decision on what is the Physical Machine (PM) to place the VM on. A typical Cloud architecture however, does not have this type of centralized structure and it has a hierarchical infrastructure. It consists of a cloud controller that controls several cloud sites (if it is a distributed data center), which the clients connect to, and clusters controllers (several at each cloud site), which control a certain amount of PMs (typically hundreds), which are controlled by a node controller [47]. Hierarchical approaches take advantage of this architecture since the node controller has information about each PM, it can pass that information to the cluster controller which can also pass the information to the cloud controller. Therefore placement decisions could be made at different levels, either at the cluster controller or at the cloud controller.

Consolidation: The hypervisor technology enables consolidation of VMs on PMs granting many advantages, being one of them, increasing energy efficiency. We can identify two types of VM consolidation: static which is not adaptable to the current resource usage and dynamic which is adaptable to current workload. In [46], they divide consolidation in three sub-problems, where different consolidation strategies emerge to solve these problems. They are the following:

- When to migrate;
- Which VM to migrate;
- Where to migrate.

One of the main problems of consolidation is, *when to migrate*. Migrating too early can lead to pointless migrations, wasting even more energy because migration also has its costs and they can't be ignored. Migrating too late, can lead to performance degradation due to overloaded servers, which also increases energy consumption since tasks are going to take more time to finish. Therefore, the right time to trigger the migration is very important. This "right time" can be found either by defining static [48, 49] or dynamic thresholds [50]. These thresholds are usually the % of CPU utilization but can also be other metrics, e.g., % of Memory utilization.

Which VM to migrate is easier in some situations and harder on others. When a PM is under-loaded, all VMs should be migrated and the PM can be shut down or put into a sleep mode to save energy. For overloaded PMs, only a few VMs should be migrated. In these cases, there are some solutions to pick which VMs should be migrated.

The simplest solution is randomly choosing VMs to be migrated. Correlated solutions pick VMs that have similar workloads to other VMs on different PMs. The VM to be migrated can also be selected according to the time it takes to migrate it. Another popular solution is called *Minimization of Migrations*

which selects the least number of VMs to migrate to achieve a low migration overhead. A final possibility for choosing what VM to migrate is called *Highest Potential Growth*. This technique chooses VMs with the lowest CPU usage compared to their requested amount. Its purpose is to ensure that SLAs are respected. According to [46], the Minimization of Migration solution is the one that provides best results.

Now that we know when and which VMs to migrate, the question that remains is, where do we migrate these VMs to. Care must be taken to avoid overloading servers or placing them in under-utilized hosts. VM placement strategies explained above could be, potentially, used at this step however, those approaches do not take into account the migration cost of a VM.

Co-located VMs "fight" for resources and besides from decreasing performance, it can also increase energy consumption. This is known as *performance interference* [51]. The first strategy takes this problem into account and it selects the PM where less performance interference will occur. Other strategies take into account the types of resources and workloads that are being used by each server and places VMs according to that. These strategies can have advantages such that this can reduce bandwidth utilization since resources can be shared among co-located VMs and it may not be necessary to access resources outside the PM.

To perform the migration of PMs these algorithms usually use one of these three types of migrations depending on the applications: **cold** migrations shuts down the VM before migrating it to the new host and restarts it on the new host; **warm** migration suspends the VM, copies RAM and CPU registers and continues on the new host; **live** migration copies across RAM while VM continues to run.

Overbooking: When using cloud services, cloud users tend to pick more resources than they actually need, either because CSPs only accept pre-defined sizes or because they want to have more resources to prevent overload situations. CSP can take advantage of this and use a strategy called Overbooking. **VM Sizing** is another strategy that on the literature is described in a very similar way to overbooking, however VM Sizing approaches have different goals than overbooking. These approaches normally rely on load predictions and adjust the resources provided to the user taking into account the predictions. As an example, if a user requests 2 GB of RAM for its application and the algorithm estimates that the application only consumes 1.5 GB of RAM, a little bit more of 1.5 GB RAM will be provided for that user. Besides being beneficial for the cloud provider which will have more resources to use, therefore able to put more VMs on that PM, it is also beneficial for the cloud users because, since they are using less resources, they will pay less (if its not pre-defined sizes).

However, this is a risky strategy [14]. It relies on prediction, which is prone to mistakes, even good prediction algorithms. These mistakes could result in SLA violations which are unacceptable. Another situation that can occur is overloads. If suddenly, in an overbooked host, the applications start to use

more resources than they were estimated to use, it will overload the host, violating SLAs and increasing energy consumption. The core of this strategy is therefore its estimation algorithm and in the literature it has been researched several ways of doing this estimation based on CPU utilization or a combination of CPU, Memory and I/O. To avoid SLAs violations, CSP and clients can, *a priori*, agree on parts of the application (or even the whole application) which can have degraded performance and in exchange, clients will pay significantly less.

Application strategies: In the literature we can also find energy-aware scheduling that do not focus on VMs, but rather on applications. These include application placement and consolidation. The work in [52] uses a bin-packing strategy to place applications and also performs application consolidation. Most strategies focus on consolidating applications in the minimum amount possible of PMs so that more PMs can be shut down, therefore saving energy. Such an approach is described in [53] but this work in particular, they try to consolidate the workloads based on their type of workload to try and avoid SLA violations.

When all the PMs are overloaded, consolidation and DVFS cannot be used because there is no place to consolidate to, and using DVFS would have severe performance penalties. To solve these issues some works use the concept of **brownout** [54]. Brownout is an intentional drop in voltage or complete shut down on power grids in case of emergencies to avoid short circuits, for example. This can be applied to achieve energy saving by shutting down components of applications that are not important, reducing fidelity but clients can be compensated financially.

Container strategies: In the literature, to our knowledge, we could only find one work regarding containers [13]. This work performs VM sizing for hosting containers and is going to be described later.

Analysis: On this analysis we are including DVFS, because despite being a mechanism, it is often used as a strategy to decrease the energy consumption. As was mentioned throughout this section, some strategies do not work on overloaded scenarios, which might be a limitation, although it is rare for data center to be completely overloaded. The only strategy that works on overload environments is brownout. Overbooking can be partially used to increase resource utilization, however care must be taken to avoid SLA violations. We say it can be partially used because at a certain point, there are no more resources that we can extend.

VM placement is the most limited strategy since it only considers the initial scheduling of VMs, which does not provide the opportunity to achieve significant reductions on energy consumption. VM consolidation is one of the most popular approaches but is also one of the most complex due to its three sub-problems. DVFS is another popular solution but it has the significant disadvantage that applying

DVFS on a host, it affects all the VMs on that host. For this reason DVFS is, usually, used together with another strategy. Overbooking has the big potential of solving the CSPs problems associated with the fixed sizes VMs which lead to significant resource wastage.

There is no single strategy better than all the other and what should be used, depends on the environment and the goals. Some might even be used together, e.g., DVFS and VM Placement [55].

2.3 Related Relevant Systems

In this section we will describe works performed on the topics described in the last section, which will be important to identify challenges and opportunities to our solution. We will follow the same structure as the previous subsection, starting with Energy-Aware Mechanisms and finishing with Energy-Aware Strategies.

2.3.1 Energy Mechanisms

As in Section 2.2, we will address first energy-aware monitoring and then DVFS with a representative system. As related work regarding energy monitoring, we've selected a work that measures energy consumption at the OS/hypervisor level and another that measures at the application layer.

Joulemeter: There are many possibilities in the literature to measure energy consumption at the OS/hypervisor layer, however most use linear models or a fixed non-linear model. Cloud environments require adaptive models since the workload is very susceptible to changes. Joulemeter [40] solves all the problems mentioned on Section 2.2. It does not require any external meter, additional hardware or software instrumentation. It uses a power model that adapts when application characteristics change. As mentioned above, Joulemeter does not require additional instrumentation leveraging existing hardware instrumentation (e.g. motherboard or power supply power sensors) to measure PM energy consumption. They track the resources used by each VM and convert into energy using power models. The hypervisor is responsible for scheduling hardware resources for each VM, they leverage this to associate each hardware resources being used to each VM.

In order to provide a good estimation, they use CPU, memory and disk to estimate the energy consumption of each VM. To estimate the CPU energy consumption, they track the CPU active and sleep times, using for example, Linux commands such as *top*. To know which VM is currently using the CPU, they track when a VM is active on a certain CPU core. Their approach for creating memory power models requires that Last Level Cache (LLC) miss counter is available. However if it is not available they deal with it as will be seen later. For the disk power model, they use the bytes written/read tracked by the hypervisor for each VM. To create their models they use an additional factor they called *unobserved states*.

These are states that can increase (for example) CPU utilization and are not being accounted on the power models. If for example, LLC is not available, it will be considered an unobserved state. To cope with this, since unobserved power states are highly correlated to the observed ones, the model based on a small number of observed states will capture the energy usage more accurately. Each unobserved state will be a separate variable to be used at the models.

As mentioned above, their power models are adaptive to the VM current workload. To achieve this, they introduce coefficient variables that change depending on the workload. This is achieved by continuously tracking the error between the sum of estimated power values for all VMs and the measured server power. If the error exceeds a threshold, then it means that the power model must be adapted to the new characteristics of the application.

Wattapp: An application-aware power meter [39] which has an architecture with the following main components: a **Model Builder** that reads system and application logs (energy and throughput values) and creates power models for each application based on the concepts that are going to be explained later; a **Configuration Orchestrator** which has the job to identify applications and PM types that do not have a power model at the required virtualization ratio (explained later) and performs calibrations to generate the required log data to create the power models; a **Oracle Query Interface (OQI)** which is used by **Power Managers** (provides a PMs list) which provide a PM and applications (along with their required throughput) as input, and the OQI returns an energy estimation, calculated using the models created by the Model Builder.

They start by making a power model for a PM running an application in a non-virtualized environment. Their power model is based on the intuition that the energy consumption by different resources have (as explained in Section 2.2) a static component independent of usage and a dynamic component. Since the static component is independent of usage, they only need to worry about the dynamic component which is explained next.

In their experiments they saw that energy consumption has a linear relationship with application throughput (application progression), hence they use application throughput as the basis for the power model. They also use two constant variables that are application dependent, since different applications may have different energy impacts. From their studies they confirm that both, memory and CPU, have a linear relationship with application throughput, therefore application throughput is an accurate source to create the power models and estimate the energy consumption of a PM. For virtualized environments, they introduce the VM overheads in their power models, which are mainly caused due to I/O operations and cache contention [56]. They observed that the impact of virtualization depends on the characteristics of the application and they conclude that applications with high I/O operations or low working set (cache contention issue) are impacted by virtualization while applications with low I/O operations and large

working set have a negligible impact from virtualization.

Since the impact of virtualization depends on the characteristics of the application, they add a virtualization ratio (from 1 to 7) to the power model they defined when virtualization was not considered. This model was created assuming a single type of application is run on the PM. Since the static component is independent of usage, they use only the largest static energy component among all co-located applications. For the dynamic power, they add the standalone dynamic energy of each co-located application.

Analysis: These two works provide some interesting insights. Joulemeter claims that using linear power models is not good for estimating energy precisely, while Wattapp says the opposite. The interesting note is that both achieve good energy estimations. The truth is that non-linear models strategies portray energy consumption in a more realistic way, however they are inherently much more complex than linear power models strategies. This complexity can itself cause estimation errors due to the overhead caused by the extra complexity.

CoScale: As a representative system of DVFS, we chose *CoScale* [57] because it takes CPU and memory into account therefore allowing to have a perspective on the implications of considering both CPU and memory when using DVFS. As they show in their evaluation, considering CPU and memory separately when using DVFS, have implications since they will conflict. If for example, we lower CPU frequency/voltage and do not have memory in consideration, the traffic to the memory would be reduced and therefore the memory manager could deduce that memory is being under-utilized, reducing memory frequency when it was not supposed to, causing a significant performance degradation.

Depending on the situation, they apply DVFS to one or more cores. Regarding memory, they apply DVFS on the memory bus, which will influence the Memory Controller and the Dual In-Line Memory Modules. Their approach is based on program slack: a target performance penalty to save energy. To avoid excessive performance degradation, they establish a limit slowdown.

CoScale uses fixed-size epochs to determine when to profile the system and then select cores and/or memory frequencies in order to minimize full system energy, while maintaining performance within the target. In the profiling phase, performance counters are read to make energy estimations.

Their algorithm starts by estimating performance assuming that cores and memory are at their highest frequencies. It then starts by decreasing frequencies, CPU or Memory depending which provides more benefits, until performance slack is reached. This algorithm keeps repeating epoch after epoch and, off course, provides an extra overhead to the system. This can be dealt by increasing the epoch time.

2.3.2 Energy-aware optimization strategies

In this section we will describe some relevant and state-of-the-art related works on the strategies explained in Section 2.2.

Dynamic consolidation algorithm: In [58] the authors propose an efficient adaptive algorithm for dynamic VM consolidation according to the current utilization of resources by VMs, leveraging live migrations. For their power model, they account for CPU, defined by Millions Instructions Per Second (MIPS), memory and network bandwidth. As was mentioned, disk can have a significant impact on energy consumption, however they do not consider because they assume that the PMs do not have physical disks and the storage is provided by a Network Attached Storage (NAS), which is what is normally used on cloud environments and facilitates VM live migrations.

The main components of their architecture are global and local managers. The global manager is on the master node and collects information from local managers which reside on nodes gathering resource utilization information. The local managers besides sending resource utilization to the global manager, they also have the job of resizing the VMs according to their resources needs and they decide when and which VMs should be migrated. They calculate that the cost of migrating a VM depends on the total amount of memory used by the VM and the network bandwidth. The image and data of the VM is stored at the NAS so it is not necessary to transfer it between VMs. Regarding their dynamic consolidation algorithm, they split it into three steps:

1. As mentioned on Section 2.2, the first problem regarding consolidation is when to migrate the VM. To address this issue, they define two adaptable thresholds: a lower utilization threshold that when its met, all the VMs from this host have to be migrated to another PM so this PM can be shut down or switched to sleep mode to conserve energy; an upper threshold which if exceeded, VM(s) have to be migrated to reduce resources utilization and avoid SLA violations. To have adaptable thresholds they use a statistical analysis of historical data collected during the lifetime of the VMs. Their proposal is to adapt the thresholds depending on the strength of the deviation of the CPU utilization. The higher the deviation, the lower the upper threshold is going to be, because if we have a high deviation and high upper threshold, a 100% CPU utilization could quickly occur and the algorithm would not react quickly enough, potentially causing SLA violations.

2. Next it is necessary to decide which VM(s) are going to be migrated. They have three policies which are applied iteratively. The first policy is **Minimum Migration Time** and selects the VM(s) that take the less time to be migrated. The migration time, as mentioned before, is estimated considering the amount of RAM and the network bandwidth. If several VMs are chosen from the first policy, the second policy selects one **randomly**. They called the final policy **Maximum Correlation**. It is based on the idea

that the higher the correlation between resource usage by applications, the higher the probability the server overloading due to competition for resources. So they select the ones with less correlation.

3. The last problem to solve is where to migrate the VM(s) to. They start by sorting VMs (that were selected to be migrated) in the decreasing order of their current CPU utilization. They take the first VM from the top of the list and allocate it to a host that provides the least increase of the energy consumption caused by the allocation. They do the until the list is empty, i.e. all the selected VMs were migrated.

Autonomic risk-aware overbooking: This work [59] uses an Overbooking approach with overloading risk concerns. Their system autonomously readjusts the risk threshold, allowing more or less VMs to be overbooked.

When a request arrives, the **Admission Control (AC)** module decides if this request should be accepted or not. To make this decision, it takes into account the current and predicted status of the system, and the long term impact this service is going to have on the overall data center. These assessments require further information about the data center status and, if available, the request prediction resource utilization. This information is available at the **Knowledge DB (KOB)** and is passed to the **Risk Assessment (RA)** module that, based on fuzzy logic programming [60], determines the risk associated in accepting this request. If this risk is bellow a given threshold, the AC will accept this request, otherwise the request is rejected.

The KOB is an important module since it holds vital information for the success of this algorithm. It has to measure and profile different requests behavior as well as keeping up-to-date the current data center resource status. They profile a request based on the following resources: CPU, memory and I/O. Therefore it holds information about each PM and VM resources utilization. To profile, they have a simulation and emulation module integrated in the KOB. To monitor they allow monitor tools (e.g., Nagios) to be integrated in KOB.

When a request is accepted, they now have to deal with the issue of, which PM to put this request. This is done by the **Smart Overbooking Scheduler** module. It selects the best PM to allocate the VM (used to serve the request). They use a worst-fit algorithm that schedules a VM to the least overbooked PM. This has the goal of improving overall utilization and reducing the overbooking impact.

To avoid overloading and SLA violation scenarios, AC decisions must be carefully considered. The RA plays a crucial role here. The risk calculation is based on three parameters: **Req** - CPU, memory and I/O required by the request; **UnReq** - the difference between total data center capacity and the capacity requested by all running requests; **Free** - the difference between the total data center capacity and the capacity used by all running services. If Req < UnReq, then there is no risk and if Req > Free then there is no space for this request and it must be rejected. If UnReq < Req < Free then the risk is calculated by $(Req - UnReq)/(Free - UnReq)$ and if its below the risk threshold it is accepted otherwise it is rejected.

The final decision to be made is to decide what is the threshold. Like was mentioned in the beginning, this is a dynamic threshold which depends on the system behavior, and the desired utilization levels, i.e., if more or less risk should be considered.

Energy efficient brownout enabled algorithm: This work proposes a brownout approach [54], that can reduce energy consumption on overloaded scenarios. To exploit brownout, they model an application as components, having mandatory and optional components. Mandatory components are crucial for the application and cannot be deactivated. Optional components can be dynamically deactivated/activated to achieve energy savings. The optional components are selected by developers/customers. These optional components are controlled by a *dimmer value*, which is used to determine the adjustment degree of power consumption. In addition, there is a brownout controller that controls the activation and deactivation of optional components. To note that prior to applying brownout, they require a VM placement algorithm to be used [61].

Components may have dependencies. To identify these dependencies they express them as *connections*. They consider if a certain optional component is deactivated, then all the optional components that connected to this component are also deactivated. Mandatory components even though they might connected, they will not be deactivated.

For the power model, they use static power, which is constant for all VMs, and dynamic power which they assume as being linear to the total utilization of VMs on a particular host. The utilization of each VM is modeled as summing up all the application utilization on a particular VM. Their algorithm consists of six steps:

1. First, the CSP has to define what is the power threshold, a value if surpassed, indicates that a host is overloaded, and the dimmer value.

2. The algorithm starts by checking all hosts and counts the ones that are above the defined power threshold.

3. Next, the dimmer value is adjusted according to the number of hosts that surpassed the power threshold. The dimmer value ranges from 0 to 1. It is 1 when all the hosts are overloaded, it is 0.5 if half the hosts are overloaded and it is 0 when no host is overloaded.

4. According to the dimmer value, the amount of reduced utilization of applications at the overloaded host is calculated. For example, if an application is executing at 100%, if the reduced utilization calculated is 40%, then optional components will be removed to achieve a utilization of 60%.

5. Now the optional components to be deactivated (and consequently their connections) need to be selected and they are chosen according to policies:

- **Nearest Utilization First Component Selection Policy** - it finds the nearest component to

the reduced utilization. If the reduced utilization is much larger than a single component, more components need selected to achieve the reduced utilization and since this policy only selects one, different policies have to used;

– **Lowest Utilization First Component Selection Policy** - selects the component with less utilization. This policy follows the assumption that the component with less utilization is less important;

– **Lowest Price First Component Policy Selection** - Selects the policy which provides the less discount, to benefit the CSP;

– **Highest Utilization and Price Ratio First Component Selection Policy** - considers both utilization and discount together. The object is to deactivate components with the higher utilization and smaller discounts.

6. If there is no host above the power threshold, optional components that were deactivated are reactivated.

VM sizing for hosting CaaS: Container as a Service (CaaS) is a recent trend which CSPs are leveraging. This new service type falls between Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). In this service type, instead of having a VM for each different application, only one VM is deployed and then, multiple containers (containing different applications) are deployed sharing the same underlying VM kernel.

The authors of [13] propose finding efficient VM sizes for hosting containers in such way that the workload is executed with minimum wastage of resources. The challenge is therefore finding an optimal size such that applications have enough resources to be executed. To achieve this, they propose a system model with multiple different components. They first require the user submitting the tasks to also indicate an estimation of the required resources for that task.

They identify two phases in their proposal: *pre-execution* and *execution*. In the pre-execution phase, some components need to be tuned before the system runtime. The *Task classifier* is the entry point to the system which receives task submissions by the users. It is responsible for categorizing the submitted tasks to classes using X-means. It also has the responsibility of identifying tasks with similar usage patterns (e.g., CPU utilization). The tasks are categorized based on: **Task length** - the time during which the task was running on a machine; **Submission rate** - the number of times a task was submitted to the data center; **Scheduling class** - how sensitive to latency is the task; **Priority** - how important a task is; **Resource usage** - based on CPU, Memory and disk utilization.

The *VM type definer* defines VM sizes based on the information provided by the Task classifier. The determination of optimal VM sizes requires analysis of the historical data about usage patterns of the

tasks. For each group of classes created, to determine the VM size, the average amount of resources (CPU and Memory) required per hour for serving the user requests during a 24 hours observation period is estimated. They do not account for disk because in their study the tasks required small amount of storage so they assume the disk size to be 10 GB. This component then outputs VM sizes to the *VM types repository*.

Now we are going to describe the components used by the system during the execution phase. The Task classifier also sends information to the *Container mapper*, which maps a task to a container and tries to assign them to an available VM. The main responsibility of this component is to estimate the number and type (looking at the VM types repository) of new VMs to be instantiated to support the arriving tasks. This component also has the responsibility of rescheduling tasks that are stored in the *Rejected task repository*. These are tasks that were rejected because the available VMs could not support them. *Virtual machine instantiator* instantiates a group of VMs according to the specifications sent by the Container mapper. The *Virtual machine provisioner* is responsible for the placement of new VMs on the available PMs or if no PMs is available, turn on PMs so the VMs can be placed. The *Host controller* runs on each PM and periodically monitors resource usage and identifies underutilized PMs and registers them in the *Available resource repository* which is is checked by the VM provisioner, when attempting to place a VM. The *Virtual machine controller* runs on each VM and monitors the VM usage and if the resource usage exceeds the VM limit, it kills some containers with low priority, to avoid starvation, the controller takes into account the number of times a task was killed. The killed tasks are sent to the rejected repository to be rescheduled by the Container mapper.

GenPack: In [15], the authors propose GenPack, a framework to schedule containers extending Docker Swarm. The general idea is to have the hosts divided into three distinct groups, which they refer to as generations. The containers will run on each generation depending on their resource profile.

Containers that have not been profiled before, i.e. whose workload is unknown, are placed into the *nursery* generation. During the time the containers spend in this generation, their resource requirements and power consumption are monitored. The hosts in this generation are static, that is, regardless if they are servicing containers or not, the hosts are always running.

Once those containers are fully profiled, the containers are migrated to a host on the *young* generation, if their execution as not completed yet. Hosts in this generation are in charge of containers whose lifetime is short and if the container lifetime surpasses a defined threshold, it advances to the next generation, the *old* generation. Their reasoning behind this solution is that a significant portion of containers are expected to have a short lifetime. Since this is the generation that will experience more load, it provides elasticity mechanisms, scaling up or down according to the current load. On the old

generation, the containers are consolidated in such a way that the minimum number of hosts are used. The amount of hosts in this generation is also static as in the nursery generation.

Periodically, containers running in the nursery are selected and hosts in the young generation are selected to host each of these containers. The containers are first converted into container envelopes according to its raw resources metrics. These envelopes categorize the type of workload of the containers (e.g. a CPU/Memory-intensive envelope contains CPU/Memory intensive containers). Within each envelope, containers are ordered per decreasing resource consumption score.

The algorithm used by their solution to select a host in the young generation, starts by homogeneously blending the content of the envelopes, into a single list, to increase the diversity of the containers per node. After this phase, it iterates over each blended container and picks the first host that matches the resources requirements of the container. This resource matching does not only check if the host as enough resources to couple with the container. It also considers the properties and requirements of the containers. For example, imagining that a host has full CPU and disk resources but a container is only memory intensive and uses little CPU and disk, then this container can be scheduled at this host. This why the containers are first sent to the nursery, to understand if they are CPU-intensive, Disk-intensive, etc, in order for these matches to be performed, achieving a more efficient resource utilizations since the containers do not compete for the resources. The hosts are ranked according to resource availability, where least available nodes are first. This allows to find the best fitting host for a container, minimizing the number of hosts required to be up.

After a host is selected for a container, its resource availability and rank are updated. The container is then asynchronously migrated from its host at the nursery generation to its new host at the young generation. If none of the available nodes fits for the container, a new host is provisioned in the young generation and the container is allocated to it.

Analysis: To better understand the differences and similarities between the different works we studied, we present Table 2.2. As we can see, CPU is considered on all works and memory in all but one. CPU and memory are indeed the dominant factors contributing for increases in energy consumption and both should be considered [45]. I/O impact is considerable when using disks, however in cloud environments, hosts normally do not have disk since they use NAS. Network bandwidth can also have some impact but should only be considered for workloads that produce intensive network communications.

As expected, a common tradeoff of applying energy-aware policies is performance. However some strategies could also have overload problems such as consolidation and overbooking. This particular overbooking work however, does not have overload issues, because they make some efforts in avoiding them, again at the expense of performance.

Work	Strategy	Resources considered	Implications	Limitations
CoScale [57]	DVFS	CPU and memory	Performance	- Does not on overload environments - Continuous estimations
Dynamic consolidation algorithm [58]	Consolidation	CPU, memory and network bandwidth	Possible overloads and performace	- Does not work on overload environments
Autonomic risk-aware overbooking [59]	Overbooking	CPU, memory and I/O	Performance	- Simulation and emulations required
Energy efficient brownout enabled algorithm [54]	VM Placement and brownout	CPU	Performance	- Limited functionality
VM Sizing for hosting CaaS [13]	VM Sizing	CPU, memory and I/O	Performance	- Heavy calculations
GenPack [15]	VM Consolidation and VM Placement	CPU, memory, I/O and network bandwidth	Performance	- Under utilized hosts

Table 2.2: Relevant related works summary

Summary

This chapter introduced the concept of components and how it evolved into today's containers, which are divided into two: OS and application containers. The most used type are application containers through Docker, however an alternative is starting to arise, Rocket, and we can expect more in the future due to the significant advantages over VMs as explained. There are multiple platforms to orchestrate containers. We introduced the three most relevant ones in the market, where none is significantly better than the other. However, none of them was designed with energy or resource utilization concerns, granting an opportunity to extend these platforms with these concerns in mind. The chapter proceeded with the study of the different strategies that could be used in this extension, complemented with energy mechanisms, vital to these strategies. This chapted finished with the study of related relevant systems, presenting how these strategies and mechanisms could be applied. The lack of work in the literature regarding energy optimization with containers lead us to proposing a solution, presented in the next chapter, to contribute to the literature in this scarce, but promising, area.

3 Proposed solution

Taking into consideration the analysis of the solutions presented at the end of Section 2.1.2, we choose Docker as the container technology for being more mature than Rocket. For orchestration, none of the three solutions is significantly better than the others, in fact, they only differ on small aspects as could be seen by our analysis at the end of Section 2.1.3. We choose Docker Swarm because it has the closest architecture to the one we propose next.

Our proposed solution consists of an extension to Docker Swarm to schedule Docker containers in an energy-efficient manner, based on resource utilization levels, also taking into consideration SLA agreements with the clients. Of the strategies presented on Section 2.2, we opted for an overbooking strategy, because the amount of resources that are wasted due to fixed size requests imposed by CSPs are a significant source of energy inefficiency, therefore creating an opportunity for increasing the energy efficiency by maximizing resource utilization. A report by Shehabi *et al.* [62] shows that approximately 30% of the servers on a data center are either idle or under-utilized, highlighting even further how an overbooking approach can be important to solve this important problem by being able to allocate beyond the machine nominal capacity.

On the following sections, we will explain in more detail how we will accomplish this but first, on Section 3.1 we present a use case to explain how the system works at a higher level. Then on Section 3.2 we detail our architecture that supports our data structures and algorithms that are presented next, on Section 3.3 and 3.4 respectively.

3.1 Use case

At high level our system consists of two components, a manager and hosts, similarly to other Cloud scheduling platforms like Docker Swarm. The process is depicted at Fig. 3.1, it starts by a client submitting a request, indicating the following requirements for the request:

- Number of CPU shares (e.g. 1024 shares represents 1 core);
- Memory required;
- Request image (e.g. NGINX);
- Request type;

– Request class;

The request type refers to a **service** (does not have a finite execution time, e.g. a web server) or a **job** (if it has a finite execution time, e.g. calculating a factorial). As for the request class, it refers to the maximum overbooking the client is willing to tolerate. We provide four classes for the client to choose:

– Class 1: No overbooking;

– Class 2: 120% overbooking;

– Class 3: 150% overbooking;

– Class 4: 200% overbooking.

Class 1 requests do not tolerate overbooking. These requests must run on hosts that are not experiencing overbooking. As for the other classes, they tolerate **1 - (100/requestClassValue)** overbooking. As an example, for a class 3 request: $1 - (100/150) = 0.33$, therefore these request classes can run on hosts that have up to 33% more resources allocated than its nominal capacity. The overbooking values will also have another use to help maximize resource utilization as will be seen on Section 3.4.2.

After this process, the Manager receives this information and according to it, among all hosts, it selects the one which maximizes overall resource utilization, allocating the request to it.

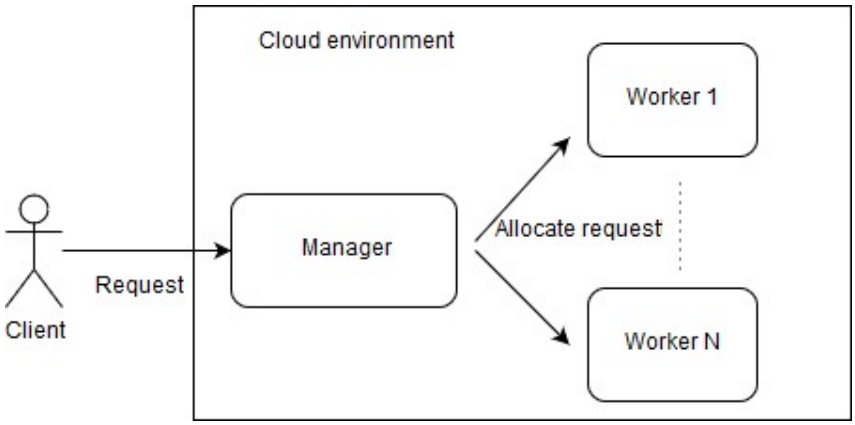

Figure 3.1: Use case

3.2 Architecture

Fig. 3.2 describes in more detail, the components inside the Manager and the hosts, and how they interact with each other. When a request arrives, it goes directly to the **Scheduler** (which runs our scheduling algorithm) and, if possible, it will schedule that request into one of the N hosts. Before

explaining in more detail how each of the components on Fig. 3.2 interact, we will start by explaining each component.

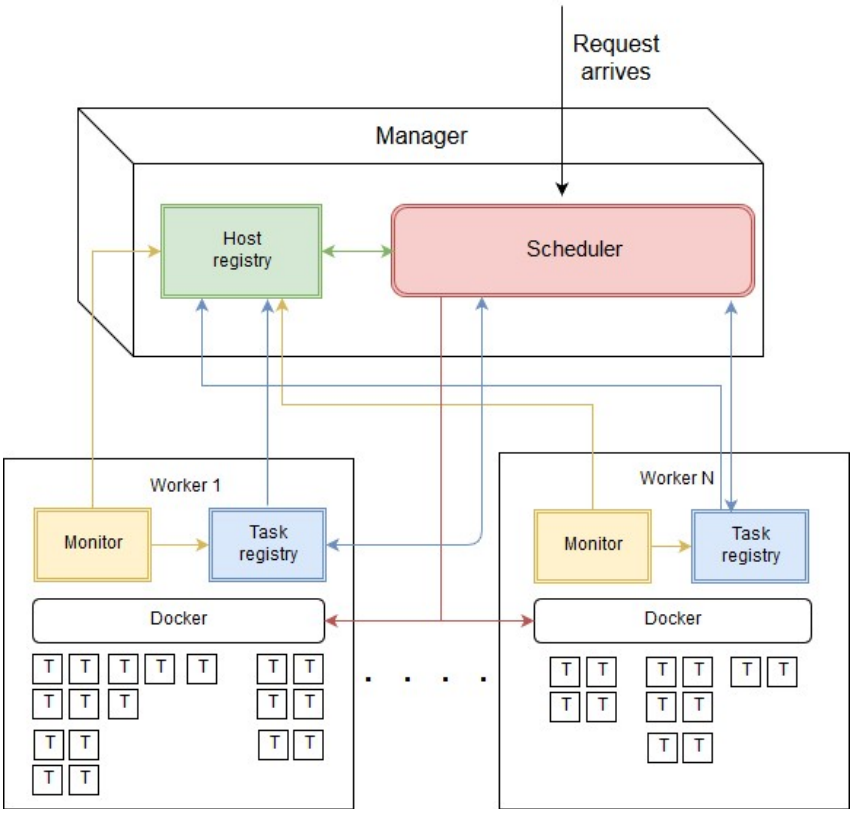

Figure 3.2: System Architecture

3.2.1 Components

Scheduler: The Scheduler is the first component the request interacts with. This component is Docker Swarm, which was extended to include our scheduling algorithm, the remaining components are new. Our algorithm (Section 3.4) is responsible for deciding which host is the best fit for the incoming request, with the goal of better resource utilization in mind, but also taking into consideration the over-booking limits presented on Section 3.1.

Host Registry: The Host Registry is a complex, but crucial, component responsible for performing several different tasks concurrently, in order to keep its data structures strictly organized, vital for the performance of the Scheduler. It keeps general information about the hosts (e.g. total resource utilization of each host) on these data structures, which are carefully explained on Section 3.3. Due to restrictions imposed by Docker, this component is also responsible for rescheduling tasks (sends them back

to the Scheduler) that are killed by the Scheduler and updating task information when a cut is performed.

The cut means that we are decreasing the resources assigned to that task. This is different from overbooking, because overbooking affects all the tasks on a host, while a cut affects a single task. This is useful for example, imagining that in a class 1 host there is 2GB RAM available and comes a class 4 request which requires 3 GB RAM, if we put the request there, it would increase the overbooking factor over 1 which is unacceptable on a class 1 host. But if we cut it (e.g. to 2GB RAM), then we can fit it there without bringing the overbooking factor over 1. The cut is equal to the overbooking that a class tolerates, so, for example, a class 2 task, would have its resources decreased by 16% (Section 3.1).

Kills refer to tasks that are killed (being rescheduled to another host) in order to allow lower level classes to be allocated to the host. The main purpose of resorting to kills is to avoid the hosts to reach extremely high resource utilization levels which would reflect in a degradation of energy efficiency. This approach will be more detailed on Section 3.4.3.

Task Registry: This component contains more specific information about each host (e.g. current tasks being served by the host), also with strictly organized data structures (Section 3.3). It is also responsible for killing the tasks chosen by our algorithm and for removing any disk space used by the task, which Docker does not remove by default and would occupy considerable disk space after many tasks were executed. The other Docker Swarm scheduling algorithms, which we will compare our solution with on Section 5, do not have this concern.

Monitor: In order to make the best scheduling decisions, the Host Registry and the Task Registry must be constantly updated. For this purpose, the Monitor is responsible for measuring resource utilization on each host and each task, and sending updated information to the Host Registry and Task Registry.

Next we will explain in detail what messages are exchanged between the components in order to keep the overall system functioning properly.

3.2.2 Components interaction

All the interactions are depicted on Fig. 3.2, but for more clarity and better understanding the following explanation of each interaction and why, we present a summary of all the interactions following typical task life-cycle:

1. Host Registry ⇒ Scheduler: **Send list of hosts**. Contains the following information about each host: IP, host class, region, total resources utilization, CPU utilization, memory utilization, allocated

CPU shares, allocated memory, overbooking factor, total memory, total CPUs;

2. Task Registry ⇒ Scheduler: **Sends list of tasks**. The tasks currently running on the host this Task Registry is running. The following information about each task is provided: task ID, task class, task image, CPU shares, memory, CPU utilization, memory utilization, task type, amount of cut resources;

3. Scheduler ⇒ Host Registry: **Update host information**. When a request is scheduled to the respective host. It sends all the request information provided by the client explained on Section 3.1.;

4. Scheduler ⇒ Task Registry: **Create Task**. When a task is created. Sends information about the task to the respective host;

5. Scheduler ⇒ Host Registry: **Cut tasks**. Sends tasks that will be cut due to the cut algorithm decision;

6. Scheduler ⇒ Task Registry: **Update Tasks**. When a task is cut. It sends the new CPU and memory of the task;

7. Scheduler ⇒ Task Registry: **Kill Tasks**. Sends tasks that will be killed;

8. Scheduler ⇒ Host Registry: **Reschedule tasks**. Sends tasks that will be rescheduled due to the kill algorithm decision;

9. Scheduler ⇒ Task Registry: **Terminated task**. Informs the Task Registry that a task has terminated so its data structures are updated;

10. Task Registry ⇒ Host Registry: **Terminated Task**. Informs the Host Registry that a task has terminated;

11. Monitor ⇒ Host Registry: **Create host**. When a host is added to the Cloud environment, the monitoring component sends the host information (IP, number of CPUs, memory amount) to the Host Registry;

12. Monitor ⇒ Host Registry: **Update resources utilization**. Update host resources utilization;

13. Monitor ⇒ Task Registry: **Update resources utilization:**. Update task resources utilization;

To make decisions, besides the information regarding the request, the Scheduler requires additional information about the hosts (message 1 and 2). This is provided by the Host Registry and the Task Registry. Information on the Host Registry is the first to be considered (explained in more detail in the next subsection), therefore being directly available at the Manager to avoid communication overheads.

However, more specific information might be needed about what is running on each host. When that is the case, the Scheduler will request that information from the Task Registry of the host it requires that additional information.

Besides requesting information, the Scheduler also sends information to both registries. When a request is scheduled, the Scheduler informs the Host Registry (message 3) to which host the request was scheduled and the corresponding request information (resources requirements, request type and request class). It also informs the Task Registry (message 4) that a task was just created sending the same information as message 3. Upon receiving this information, each Registry will update its data structures accordingly as will be seen on Section 4.2.

Messages 5, 6, 7 and 8 are sent when a decision is made either by the cut or kill algorithm. The reasoning behind these messages will be clarified on Section 3.4.3.

The Scheduler component is also responsible for detecting finished tasks. Upon detecting that a task has finished, it will send the task ID (message 9) to the Task Registry so it can update its data structures and remove any data left by the task.

The Monitor component monitors resource utilization of both host and the tasks running on that host. It updates them (message 12 and 13) when some conditions apply as will be seen on Section 4.2.3.

Upon receiving message 7 (due to a task being killed) or 9 (due to a task have finished) and after dealing with them accordingly, it will inform the Host Registry so the host allocated resources are updated.

One of the goals of our work was to extend Docker Swarm with our scheduling algorithm without interfering with the other scheduling algorithms, which we managed to achieve. In order to accomplish this, we had to make some adaptations to our algorithm, which affected the components interaction. For example, detecting that a task has finished (message 9) or that a host has joined the Cloud environment (message 11), could be performed by the Scheduler, however it would involve changes to the code that would affect the other algorithms. This is why these two messages are not sent by the Scheduler.

3.3 Data structures

As was mentioned on the previous section, the data structures play an important role in the performance of the algorithm, but also in its scalability. Since cloud environments tend to have tens of thousands of machines, it is important that our solution is scalable. We designed our data structures with this concern also in mind.

These two requirements bring tradeoffs in terms of complexity of the data structures as shall be seen next. This added complexity is the reason why we decided to build two extra components to keep the data structures organized, since it would stress the Scheduler too much if this computation was performed there. We will start by covering the internals of the Host Registry and finish this section with the Task Registry.

3.3.1 Host Registry

Our strategy for achieving better resource utilization and consequently, better energy efficiency, is based on the study performed by [63], which states that the energy consumed is proportional to the resource utilization and that energy efficiency starts degrading at high levels of resources utilization. Based on this, we decided to have three regions which map resource utilization (CPU and Memory) with energy efficiency:

- **Low Energy Efficiency (LEE)**: 0-50% total resources utilization;
- **Desired Energy Efficiency (DEE)**: 50-85% total resources utilization;
- **Energy Efficiency Degradation (EED)**: > 85% total resources utilization;

The LEE region refers to the region that has the lowest energy efficiency, due to under-utilized resources, having a significant energy inefficiency. We want to transit hosts on that region to the DEE region as quickly as possible, where an optimal energy efficiency is achieved. Our goal is to keep the hosts at region DEE, because heavily used resources (hosts at region EED) have a negative impact on the energy efficiency, increasing the energy consumption. How all this is achieved will be seen on Section 3.4, for the remainder of this section we will explore how these regions are designed in terms of data structures and how they are organized.

Organization: The Host Registry will maintain updated lists containing the hosts at each of these regions. For each region, we will have four lists, one for each overbooking class as illustrated by Fig. 3.3. What defines a host class is the lowest level class task currently running at that host. For example, if a host is serving tasks of classes 2, 3 and 4, then this host will be classified as a class 2 host.

Each host will have the following information associated with it: **IP; Class; Region; Total resources utilization; CPU utilization; Memory utilization; CPU shares allocated; Memory allocated; Overbooking factor; Total memory; Total CPU shares.**

The region a host belongs depends on the current total resources utilization of the host as was previously explained. The total resources utilization is represented as *max*{ *% of CPU utilization, % of memory utilization*}, since the highest of these two values is what is restraining more the utilization of

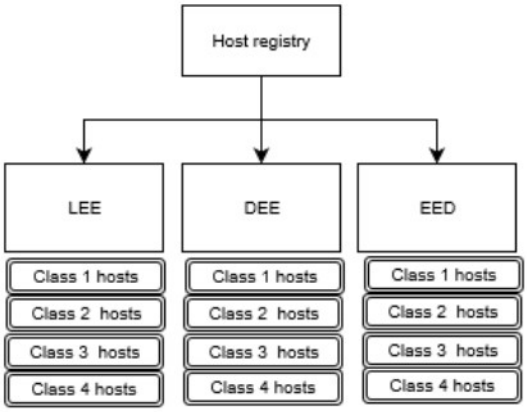

Figure 3.3: Host Registry data structures

the overall host resources. The overbooking factor is the *max{ CPU shares allocated/Total CPU shares, Memory allocated/Total memory}* . Again, we use the max because it is what is the most restraining. As an example, if the overbooking factor is 1.3, it means we have 30% more resources allocated on that host than the total amount of resources of that host.

Ordering: The lists on the regions LEE and DEE are ordered by descending order of total resources utilization and EED by ascending order. The hosts on the LEE region are ordered by descending order, because the goal is to make the hosts leave this region of energy inefficiency, bringing them up to the the DEE region as quickly as possible. Therefore the scheduling algorithm will try to schedule the requests on the first elements of the list since they are closest to the DEE region.

Since the DEE region is the desired region for hosts to be, we order its lists by descending order, to use a best fit approach, i.e. put as much requests on a host to maximize it but at same time avoid entering the EED region.

The EED list will only be used for kills (explained on Section 3.4). The hosts on that region are experiencing high resource utilization, therefore we don't want them to be receiving more requests which would only aggravate their energy efficiency. What we want is to bring them down to the DEE region, therefore we order the lists by ascending order so that the first on the list is the closest to the DEE region.

This a very dynamic environment since the hosts can switch the data structure they belong either due to a region change or a class change. This requires not only an efficient insertion algorithm but also a rigorous synchronization, in order to ensure the data structures are always consistent. The former will be explained on Section 3.4 and the latter on Section 4.2.2.

Updating: The data structures that compose the Host Registry can suffer changes if the following

events occur:

- **A request is scheduled to a host**. When this event occurs, the host to which the request was scheduled, will be updated accordingly at the Host Registry, updating its allocated resources and overbooking factor. This event can also result in a host class change in case the request class is lower than the current host class. If, for example, the host class is 3 and the request scheduled is a class 1, then the host class must be updated to 1;
- **Cuts are performed to tasks**. When cuts are performed to tasks, the allocated resources of the host those tasks belong to will be reduced, meaning they need to be updated. The overbooking factor will also decrease so it needs to be updated. This type of event does not trigger a class or region change since it does affects neither of them;
- **A task terminates**. As was covered on Section 3.2.2 a task terminates either because it finished or it was killed. In either case, the event is treated the same way. Since the task terminated, the allocated resources and overbooking factor of that host are reduced, so it needs to be updated. It can also imply a class change since the task terminated could be the last one restricting the host class. For example, if the host has one class 2 task and several class 4 tasks, if the task being terminated is the class 2 task. The host class must be updated to 4;
- **Utilization resources are updated**. When the CPU and/or memory utilization is updated, besides updating CPU and memory utilization, we must also update the total resources utilization of that host, by choosing the max between CPU and memory as was explained earlier. If the total resources utilization changes, then it might be required to make a region change. A check is performed to see what region the host falls in with the new total resources utilization value. If it is a different region than the current one, then we must update it. Otherwise there is no need to update. As will be seen on Section 5.2.2, these updates won't occur often since we managed to keep the hosts resources utilization stable over time, mainly the CPU, which is normally the max between CPU and memory utilization.

As depicted on Fig. 3.3, the host class is a sub-structure of the region structure. To update a host class, we first must remove it from the current host class structure and insert it, properly ordered (Section 4.2.2), on the new host class structure, under the same region structure. As an example, consider the case that a host X, currently on region LEE, with class 4, receives a class 2 request, the update that happens is as follows:

[LEE][class 4][host X] host X moves to [LEE][class 2][host X] , where [LEE][class 4] and [LEE][class 2] contains all hosts in region LEE and are class 2 and 4, respectively.

When a region update occurs, the procedure is the same. We just need to remove it from the host class sub-structure from the current region structure and insert it, again properly ordered, under the same host class sub-structure but under the new region structure. Picking the same example as before, but now, instead of a class update, with a region update, changing from LEE to DEE:

[LEE][class 4][host X] host X moves to [DEE][class 4][host X] , where [LEE][class 4] and [DEE][class 4] contains all hosts in region LEE and DEE respectively, with class 4.

3.3.2 Task Registry

As mentioned earlier, the Task Registry contains specific information about the tasks running on the host. Per host, there will exist four lists, one per overbooking class as shown in Fig. 3.4. Each task will have the following information associated with it: **ID, Class, Image, CPU shares, Memory, Total resources utilization (calculated the same way as for the Host Registry), CPU utilization, Memory utilization, Task type, cut received**.

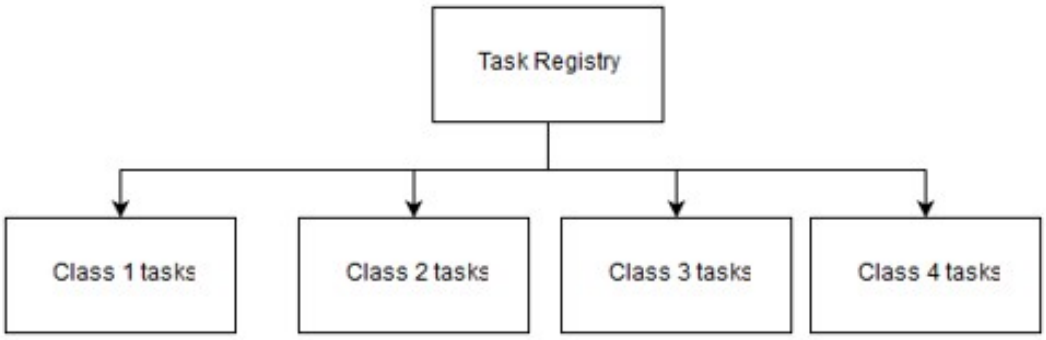

Figure 3.4: Task Registry data structures

Ordering: The information of the Task Registry will only used for the cut or kill algorithm (Section 3.4). Since the objective is to maximize resource utilization, priority is given to cutting or killing tasks that are using less resources. To achieve this, the data structures represented on Fig. 3.4 are ordered by ascending order of their total utilization resources.

Updating: The data structures that compose the Task Registry can suffer changes if the following events occur:

– **A request is scheduled to a host**. When a request is scheduled to a host, the Task Registry of that host, will create an entry for that request and insert it at the end of the data structure of the

request class. Since the task was just created and there is no information about its total resources utilization, it is inserted at the end;

- **Cuts are performed to tasks**. When cuts are performed to tasks, the allocated resources of those task will be reduced, therefore an update to those tasks information is required;
- **A task terminates**. As for the Host Registry, this type of event is treated the same way for a task that was killed and for a task that has finished. This task needs to be removed from the data structures;
- **Utilization resources are updated**. When the CPU and/or memory utilization is updated, besides updating CPU and memory utilization, we must also update the total resources utilization of the task. As with the hosts total resources utilization, it is the max between CPU and Memory utilization. If the total resources utilization changes, then we must resort the task position within its data structure;

Resorting the whole data structure is unpractical due to the amount of tasks it can hold, although, depending on the workload, these updates might not occur often. To avoid resorting the whole data structure because of a single change, when a change is required, we decided to remove the task from its current data structure and re-insert it again into its new position. As the Host Registry, this could lead to potential inconsistencies which need to be avoided through synchronization, as will be covered on Section 4.2.2.

On the following section we are going to approach the algorithm that uses the data structures explained on this section. In fact, our algorithm is divided in three sub-algorithms as will be seen next.

3.4 Algorithms

There are three core algorithms. The first, tries to schedule the request, taking some restrictions into consideration. However, if the request does not fit with the first algorithm, there are two options, either cut or kill tasks in order for the request to fit. We will also present the cut and kill algorithm but first, we start by presenting the overall algorithm which contains the first simple algorithm.

3.4.1 Overall algorithm

The goal of the overall Algorithm 3.1 is, first, to try and schedule the request either in the LEE or in the DEE region, without resorting to cuts or kills. It starts by getting the hosts that are in the LEE region, then the hosts on the DEE region are appended to that list (line 2). For example, if LEE has two class

1 hosts and one class 2 host, and DEE has three class 1 and one class 4, the list would be as follows: 1121114.

We prioritize scheduling in the LEE region so that those hosts can leave that region of energy inefficiency. Since the lists are ordered by descending order of total resources utilization, as we saw on Section 3.3.1, the first elements of the lists are always the best candidates in order to achieve the goals of the hosts on each region. In the case for hosts in LEE, to bring them to the DEE region, and hosts in DEE to use a best fit approach and keep them in that region.

The hosts retrieved must respect this condition: request.CLASS ≥ host Class and aggregate them by ascending order of the class. This is to try and aggregate class 1 requests so that they are not spread among the hosts, which would cause more energy inefficiency since no overbooking is allowed on class 1 hosts. This also benefits in avoiding pointless searches. For example, if a class 1 request arrives, there is no point in searching hosts whose class is greater than 1, because it would probably not fit there without resorting to cuts or kills. The next step is trying to schedule the request. **requestFits** function checks if the host has enough resources to couple with the resources the request demands. It also checks, if after the allocation, the overbooking allowed by the host is not violated (i.e. *overbooking factor* < *host class*).

Algorithm 3.1 Overall algorithm

```
1:  function SCHEDULEREQUEST(request)
2:      listHostsLEE_DEE = getHostsListsLEE_DEE()
3:      for listHostsLEE_DEE as selectedHost do
4:          if requestFits(selectedHost, request) then
5:              allocateRequest(selectedHost, request)
6:              return
7:      listHostsLEE_DEE = getHostsListsLEE_DEE()
8:      if cut(listHostsLEE_DEE, request) then
9:          allocateRequest(selectedHost, request)
10:         return
11:     listHostsEED_DEE = getHostsListsEED_DEE()
12:     if kill(listHostsEED_DEE, request) then
13:         allocateRequest(selectedHost, request)
14:         return
15:     warnClient()
```

If the request cannot be scheduled in any of those hosts, we must resort to cut or kill. We first try to cut. We do not cut tasks on the region EED. Cutting a task and putting a request there, it would increase the overbooking on that host, worsening the decrease of energy efficiency that is already felt by hosts on that region.

On line 7 (still on Algorithm 3.1) a lists of hosts is fetched again. This time, the hosts are aggregated differently than before. Here, the lists are fetched respecting this condition: request.CLASS ≤ host Class. We do not try to cut on class hosts that are bellow the request class, because there, it will be

unlikely that there is something we can cut (because we only cut tasks that are greater or equal than the incoming request). Although there could be tasks greater or equal than the request, we believe it is not worth the cost of searching all these hosts. Like line 2, they are also aggregated by classes, following an ascending order. Again, this is done to promote class 1 tasks aggregation and also because if a class 1 request is to be put at a class 3 host, for example (which might happen if it does not fit on class 1 and 2 hosts), it would have to cut everything there to put the overbooking factor from 1.5 to 1, to respect the overbooking limit that class 1 tolerates.

If we cannot cut anything to fit the request, our last chance is to try and kill tasks in order to fit the request (line 11). On line 10 the list of hosts on regions EED is obtained and we append the hosts on region DEE. Priority is given to killing tasks on region EED, because by killing tasks and assigning a new request to it, we could bring that host back to the DEE region. Since kill is our last resort to fit a request, all the hosts on that region are considered regardless of their class. They are also aggregated differently than before. Considering two tasks of each class in a host, this list would be aggregated as follows: if an incoming request is 1: 11223344; if it is 2: 22334411; if it is 3: 33442211; if it is 4: 44332211. This is done to avoid the problem that was mentioned at the end of the last paragraph. Next we will look into the cut algorithm and then finish with the kill algorithm.

3.4.2 Cut algorithm

Before explaining Algorithm 3.2, it is important to mention some restrictions to cutting. First, there are two choices, either cut the request being scheduled or cut tasks currently running on the host. We give priority to cutting the incoming request rather than the already running tasks, because cutting a task involves more overhead than cutting a request, due to the updates that have to be performed at the data structures as we saw on Section 3.3. Second, we have to be careful with the cuts. Since overbooking affects all the tasks on a host, imagine that a class 2 request receives a cut, then gets assigned to a host that is currently experiencing 120% overbooking, this request would have its SLA violated since it would be affected by more than 16% overbooking. To avoid this situation, we can only allow requests that have been cut to be assigned to hosts with a class lower than the task class. However, if we cut a class 3 request by 33%, it would only be allowed to run on a host class 1, same for a class 4 request. To avoid this limitation, the cuts applied for class 3 and 4 requests, depend on the host they are at. For example, if a task class 3 is running on class 2 host, the cut would be of 17% (33% (class 3 overbooking toleration) - 16% (class 2 overbooking toleration)), since the remaining 16% would be inflicted indirectly by the overbooking tolerated by that host. All the restrictions are as follow:

- Class 1 requests do not receive cuts;

- Class 2 requests can only receive a cut if they are assigned to a class 1 host;
- Class 3 requests can receive a full cut if they are assigned to a class 1 host. If they assigned to class 2 host, they can only receive a cut equal to: 33% (class 3 value) - 16% (class 2 value), i.e. 17%. They cannot receive cuts for class 3 and 4 hosts;
- Class 4 requests can receive a full cut if they are assigned to a class 1 host. If they are assigned to class 2 host, they can only receive a cut equal to: 50% (class 4 value) - 16% (class 2 value), i.e. 34%. If the task is at a class 3 host then they can only receive a cut equal to: 50% (class 4 value) - 33% (class 3 value), i.e. 17%. They cannot receive cuts for class 4 hosts.

Having understood what a cut is, its restrictions and its benefits, we can finally look into Algorithm 3.2. At line 6 it starts by checking if the request fits considering the same conditions as in Algorithm 3.1 at line 4. Although this check is done at the previous algorithm, this is done again because the selected hosts for the simple algorithm are different from the hosts selected ones for the cut algorithm, for the reasons stated on the previous section. Therefore, it might be possible to allocate on these hosts without resorting to cuts, thus avoiding cutting unnecessary tasks.

If this first check fails, the next step is to try to fit the request by cutting it and checking if it fits (line 9). When making this check, it is always taken into consideration the current class of the host to try and leverage as much overbooking as we can, in order to achieve a better resource utilization. If it does fit, then the request is cut (line 10) and allocated to that host.

Otherwise, if the request class is higher than the host class (line 13), it continues to the next host because it is not worth to cut at this host. This is the case because, if the request class is higher than the host class, then it is likely that this host contains a majority of tasks that are below the request class (we only allow tasks to be cut if they are higher or equal than the request class) therefore not being worth the time searching this host for tasks to cut.

As was explained previously, the request can only be cut if the host class is lower than the request class. For the same reason, tasks can only be cut if their classes are lower than the host class. Therefore, if the host class is greater or equal than the request, only tasks whose class is higher than the request (because if the request is assigned successfully, the class of the host would change and the classes would have their overbooking levels violated) (lines 16 and 17) can be cut. To illustrate this, an incoming class 3 request can only be cut if it is being assigned to either a class 1 or 2 host. As another example, if an incoming class 2 request is being assigned to a class 1 host, it cannot be cut and only class 3 and 4 tasks that are active on that host can be cut.

A check is also performed for class 4 requests because there are no tasks higher than class 4. Oth-

erwise, we are safe to cut classes equal or higher than the incoming request, if it is not a class 1 request, because we cannot cut class 1 requests (lines 18 and 19).

Algorithm 3.2 Cut algorithm

```
1:  function CUT(listHostsLEE_DEE, request)
2:      for listHostsLEE_DEE as selectedHost do
3:          cutLIST = null
4:          listTasks = null
5:
6:          if requestFits(selectedHost, request) then
7:              returntrue
8:          if request.CLASS != 1 and afterCutRequestFits(selectedHost, request) then
9:              newRequest = cutRequest(request)
10:             return true
11:         else if request.CLASS > selectedHost.CLASS then
12:             continue
13:
14:         if selectedHost.CLASS >= request.CLASS and request.CLASS != 4 then
15:             listTasks = getListTasksHigherThanRequestClass()
16:         else if request.CLASS != 1 then
17:             listTasks = getListTasksEqualHigherThanRequestClass()
18:
19:         memoryReduction = 0
20:         cpuReduction = 0
21:
22:         for listTasks as task do
23:             cpuReduction+ = task.CPU * task.CutToReceive
24:             memoryReduction+ = memory.CPU * task.CutToReceive
25:             cutLIST.Append(task)
26:             if fitsAfterCUT(request, cpuReduction, memoryReduction) then
27:                 cutRequests(request, cutLIST)
28:                 return true
29:         end for
30:     end for
31:     return false
```

When the list of tasks from the Task Registry (lines 17 or 19) is retrieved, the lower classes are appended to the higher ones, so that it attempts to cut from the higher classes first. Since the list is ordered by total resources utilization (done by the Task Registry) and by class, the first of the list is the best candidate for a cut.

Line 28 checks if the request fits taking into consideration the overbooking restrictions described earlier. The tasks are checked iteratively until the request fits or does not fit at all, trying the next host. This is done iteratively instead of checking all of them at once, to avoid cutting unnecessary tasks.

To reduce the amount of tasks the Scheduler needs to check for a request to fit, the Task Registry only sends the tasks that respect the cut restrictions explained earlier on this section.

If one reaches line 34, it means that we cannot cut enough tasks at any host to allocate this request, therefore we must try to kill tasks to fit this request (algorithm 3.3).

3.4.3 Kill algorithm

Regarding Algorithm 3.3, only tasks with their class higher than the request can be killed (line 4 and 5, the check on 4 is because there are no classes higher than 4). This provides the opportunity to co-locate similar task classes, leaving other hosts to be able to have more overbooking, increasing the overall energy efficiency. However, class 4 tasks that are services, since they are most likely not to be utilizing their resources fully, we decided to kill them if the request is a job, that is more likely to use the resources more efficiently than a service (line 7).

The reasoning beyond the remainder of the algorithm is similar to Algorithm 3.2, checking if the requests fits by killing the tasks obtained from the Task Registry, if not, move to the next host. Killed tasks are rescheduled to other hosts (line 18). If after checking all hosts the request does not fit in any, then it cannot be allocated and we warn the client.

Algorithm 3.3 Kill algorithm

```
 1: function KILL(listHostsDEE, request)
 2:     for listHostsEED_DEE as selectedHost do
 3:         possibleKillLIST = null
 4:         if request.CLASS != 4 then
 5:             possibleKillLIST = selectedHost.getListTasksHigherThanRequestCLASS()
 6:         else
 7:             possibleKillLIST = selectedHost.getListTasksClass4NonJob()
 8:         killLIST = null
 9:         memoryReduction = null
10:         cpuReduction = null
11:
12:         for possibleKillLIST as task do
13:             cpuReduction+ = task.CPU
14:             memoryReduction+ = memory.CPU
15:             killLIST += task
16:             if requestFits(request, killLIST) then
17:                 kill(killLIST)
18:                 reschedule(killLIST)
19:                 return true
20:         end for
21:     end for
22:     return false
```

Summary

As could be seen by the previous chapter, there are different strategies to optimize energy efficiency, each with its advantages and disadvantages. This chapter presented our proposal to extend Docker Swarm with an overbooking strategy, maximizing its advantages and minimizing its disadvantages. We maximize its advantages by going further than the overbooking strategies found in the literature, adding the concept of cuts. In order to minimize the disadvantages, we propose a kill algorithm that mitigates the issue of extremely high resources utilization, a risk of using an overbooking strategy. The solution aims at increasing resource utilization based on resource utilization regions that provide different energy efficiency levels. The next chapters continues with more details about our solution, this time at lower level of abstraction, with the implementation details.

4 Implementation

In the previous chapter we saw how our solution was designed and why, at a higher level of abstraction. In this chapter we will go to a lower level of abstraction, by first describing the overall system setup, presenting how all components are integrated with Docker Swarm for example. Then we will go into important aspects of the components implementations, vital for the overall performance of our solution. We finish with the software architecture and what extensions were added to Docker Swarm.

4.1 System setup and operation

At this section is described the important configurations for hosts discovery and for communications between components.

4.1.1 Discovery service

In order to start containers on remote hosts, Docker Swarm uses a discovery service. As seen on Section 2.1.3, Docker Swarm provides a default discovery service but also supports different discovery services, such as key-value stores or DNS. We started using the default discovery service but soon came to realize that it was too slow and a different approach was required. The default discovery service required constant communications with the Docker Hub (Section 2.1.2), which is a slow process when compared to using a local discovery service without requiring external connections outside of the cloud environment.

We decided to use a key-value store discovery service for being better supported by Docker Swarm than a DNS discovery service. We opted to use Consul[1], for being simple to learn and having a good integration with Docker Swarm.

4.1.2 Inter-components communications

Now that Docker Swarm can communicate with the hosts, the next step is to make Docker Swarm be able to communicate with the components we introduced on the previous chapter. As described on Section 2.1.3, Docker Swarm does not have any of the components presented on the previous chapter.

[1] https://www.consul.io/

The components communicate with each other using Representational State Transfer (REST) endpoints. These endpoints represent the interactions exchanged between components explained Section 3.2.2. Throughout the remainder of the section, when we refer to messages, we refer to the ones of Section 3.2.2. Lets start with the Host Registry endpoints presented on Fig. 4.1.

Host Registry
1 - /host/list/{requestclass}&{listtype} (GET)
2 - /host/listkill/{requestclass} (GET)
3 - /host/updateclass/{requestclass}&{hostip} (GET)
4 - /host/updateresources/{hostip}&{cpu}&{memory} (GET)
5 - /host/createhost/{hostip}&{totalmemory}&{totalcpu} (GET)
6 - /host/updatetask/{taskid}&{newcpu}&{newmemory}&{hostip}&{cpucut}&{memorycut} (GET)
7 - /host/reschedule (POST)
8 - /host/killtask (POST)
9 - /host/updateboth/{hostip}&{cpu}&{memory} (GET)
10 - /host/updatecpu/{hostip}&{cpu} (GET)
11 - /host/updatememory/{hostip}&{memory} (GET)

Figure 4.1: Host Registry endpoints

Endpoints 1 and 2 refer to message 1. They are used by the Scheduler when it requests hosts to attempt to schedule the request. The first endpoints gets hosts for the simple or cut algorithm (Section 3.4) based on the *listtype* value provided. Endpoint 2 gets hosts for the kill algorithm (Section 3.4.3).

When a request is scheduled (message 3), the Scheduler sends information to the Host Registry via endpoints 3 and 4. The former checks if the host class should be updated based on the request class sent by the Scheduler, while the latter updates the allocated resources of the host identified by the IP.

Endpoint 5 is used by the Monitor to communicate to the Host Registry that a new host joined the cloud environment and can now be considered for scheduling decisions.

When cuts are performed at tasks, the Scheduler uses endpoint 6 to send the new CPU shares and memory value for the task after the cut, so the Host Registry can update it (Section 4.2.2). It also sends the amount of CPU shares and memory value that was cut so the allocated resources of that host are updated. This endpoint corresponds to message 5.

For the reasons stated on Section 3.2.1, the Host Registry is responsible for rescheduling tasks that were killed. Endpoint 6 serves this purpose, while endpoint 7 is used to update the allocated resources of the host. The whole task information (Section 3.1) is sent via POST. Both refer to message 8.

The remaining endpoints refers to message 12 and are used by the Monitor when a resource utilization update is required. Why there are three different endpoints for this purpose will be seen later on Section 4.2.3.

The Task Registry defines the remaining endpoints shown at Fig. 4.2

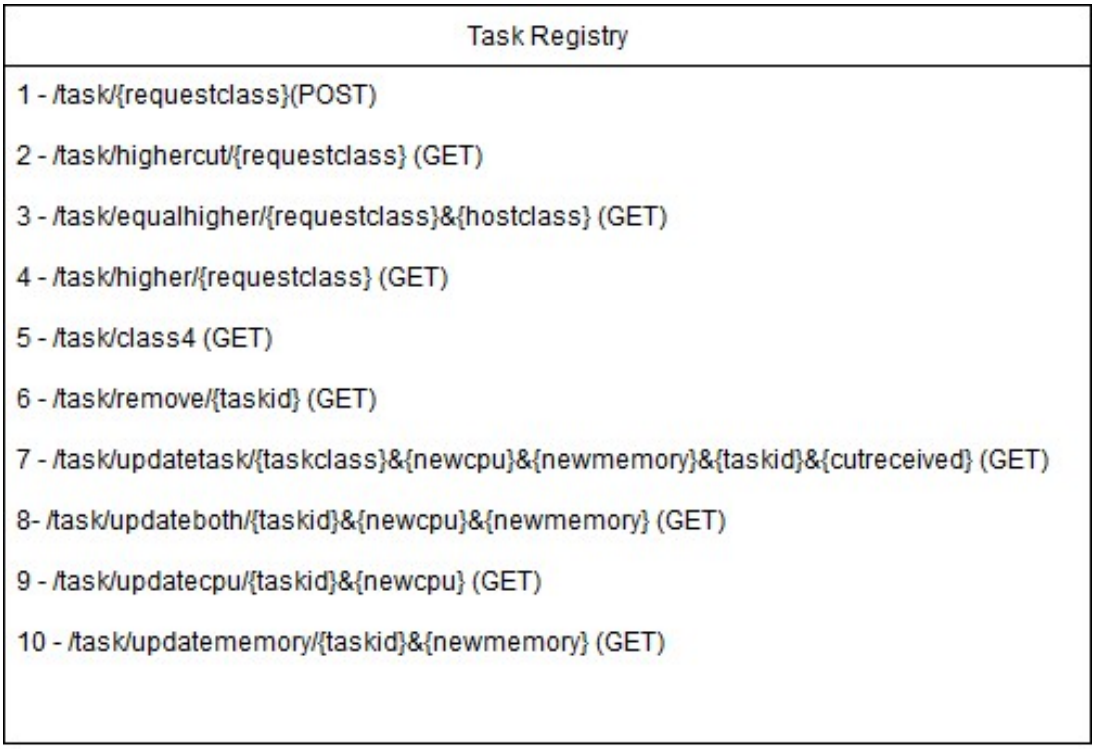

Figure 4.2: Task Registry endpoints

The first endpoint refers to message 4. The Scheduler uses it to send all the request information in order for the Task Registry to create an entry for it, on its data structures. Endpoints 2 - 5 refer to message 5 when the Scheduler requests a list of tasks for the cut or the kill algorithm. The first two are used by the cut algorithm and the last two by the kill algorithm. When a task has terminated (message 9), the Task Registry is warned by the Scheduler via endpoint 6. When a cut decision is made, the Scheduler uses endpoint 7 to tell the Task Registry which task must have its allocated resources updated. This refers to message 6. The last three endpoints are the same reasoning as the last three endpoints of the Host Registry and will be explained later at Section 4.2.2.

4.2 Components

On this section we provide more details about the implementation of the components that comprise our solution, starting with the Scheduler, Host Registry and Task Registry, and finishing with the Monitor.

4.2.1 Scheduler

As seen on Section 2.1.3, Docker Swarm has filters that can be used to restrict the number of hosts to which a request can be scheduled. This is not useful for our case since we use more complex restrictions that cannot be performed through the use of filters, which are only intended for simple restrictions. However, we leverage filters to pass information to our algorithm which otherwise would be required to be passed with the image, which is not acceptable because, if for example, a client wants to start an Ubuntu desktop, it doesn't make sense for him to edit the image to include the information required for the request to be scheduled.

The information required to make a scheduling decision was first introduced at Section 3.1. The number of CPU shares, memory and the request image the client provides is accessible inside Docker Swarm. However, the request class and type are custom and these are the two types of information that we use filters to make them accessible inside Docker Swarm.

4.2.2 Host Registry and Task Registry

The Host Registry is responsible for many different concurrent tasks (Section 3.3.1), making it susceptible to bottlenecks and having inconsistencies within its data structures. The Task Registry is more lightweight (Section 3.3.2), although it also deals with constant changes within its data structures. Both solutions that we found for these problems are applied at both registries in a similar way, therefore we present them both together at this section. However there are some differences that are highlighted when relevant.

Sorting: The constant insertions could result in bottlenecks and scalability problems since the data structures will grow very large in real cloud deployments. Therefore a quick, but simple insertion algorithm is required.

Both registries contain data structures ordered by ascending order of total resources utilization (EED region for the Host Registry, Section 3.3.1), but Host Registry also contains data structures ordered by descending order of total utilization resources (LEE and DEE region), so two versions are required, one for ascending and another for descending ordering.

Binary search [64] is a common and simple algorithm used to find elements in a list with O(log N) complexity. We decided to adapt this algorithm to, instead of searching for an element, to search for an index position indicating the place we want to insert.

Our adaption for the ascending order works as follows. As the search value we use total resources

utilization. When the lower bound is greater than the upper bound it means we have reached the desired position. However, since the order matters, an additional check is required, because at the current index, there could be a host/task that contains a total resources utilization higher than the host/task total resources utilization that we want to insert. In case the search value is lower then that of the point to be inserted, then it can return the current index position. If the search value is higher, then it is inserted at index + 1 position because at the index position, the host/task has a total resources utilization higher than the host/task we want to insert. If the current host/task at the index position has a total resources utilization equal to the search value, then the search has also finished. It returns the index position but it could also be index position + 1 since they have the same total utilization resources, it would not make a difference.

The algorithm for sorting in descending order has the same rationale but the checks are reverted, that is, when the lower bound is lower than the upper bound, then we've reached the desired position, also performing the two additional checks as in the ascending algorithm.

Data structures implementation: Now we are going to look in more detail at how the data structures are actually implemented, with the goal of achieving the fastest insertion, deletion and updating times. First, we will look at the Host Registry then at Task Registry because they are implemented differently.

Host Registry: As seen on Section 3.3.1, each region will have 4 lists, one for each overbooking class. For a quick access, each region will be accessed through a map (e.g. named regions) where the key is a string with the region (LEE, DEE or EED) and the value is a struct (similar to C++ structs, there are no classes in Go) as follows:

```
struct {
classHosts map[string][ ]*Host
}
```

ClassHosts is another map whose key is a string with the host class (1,2,3 or 4) and the value is a pointer to a slice of a Host struct. This struct contains all the information regarding a host (e.g. its IP, total resources utilization, resources allocated, etc).

Docker Swarm is implemented in Go[2]. Instead of arrays, they have *slices* which is similar to arrays but has a dynamic size, i.e. we can add or remove elements. For example, a slice of ints would be declared as : *[ ]int*. So in this case, a slice of a pointer to a struct named Host is: *[ ]*Host*.

[2]https://golang.org/

These maps grant a very quick access to the hosts we want to access, useful for example, when the Scheduler asks for lists of hosts with restrictions about region and class as seen on Section 3.4. As an example, if we want to access all the class 3 hosts at region EED, we simply use the following: *regions["EED"].classHosts["3"]*.

However, this approach is very inefficient if we want to access a single host (to update the utilization of resources for example). To change the resources of a certain host at region DEE, class 1, we would have to iterate through the slice regions["DEE"].classHosts["1"][i], where i is initially 0 and is incremented until we reach the host we wanted. Besides having a complexity of O(N), it also has the additional overhead of accessing two maps, despite being quick, it is not negligible.

To solve this problem we decided to create another map (e.g. named hosts) with the host IP as key (since it is unique) and as a value, we use a pointer to a Host struct, the same Host struct as above. To access a host cpu utilization and update it we can now simply use: hosts["193.146.164.10"].CpuUtilization = 0.23.

The changes performed at the hosts map will also be reflected at the regions map (and vice-versa) since they share the same pointer to the Host struct. Using this approach also increases concurrency, consequently increasing overall performance. If we did not use a separate map (hosts) to access hosts in a single fashion, in order to make an update to a host, besides iterating through all hosts within the regions map, we would also have to lock that map in order to ensure it remains consistent, allowing only one host to be updated at a time which has severe performance implications.

In the regions map, we have to use a slice for the hosts, because the hosts at that map need to be ordered in order to maximize the performance of the scheduling algorithm (Section 3.4) and we can not order values in a map.

Task Registry: The Task Registry data structures are simpler than Host Registry's (Section 3.3), but the rationale is the same. Again, for a quick access, we use a map (e.g. named classTasks) whose key is a string (1, 2, 3 or 4, representing the task class) and the value is a pointer to a slice of a struct called Task. This struct Task, contains all the information regarding a task (e.g. its ID, allocated resources, etc). If we to want to access all class 2 tasks we can simply use *classTasks["2"]*. Again, we have to use a slice because the tasks need to ordered (Section 3.4).

The same problems mentioned for the Host Registry also exist here. To deal with this, we also have another map (e.g. named tasks) with the task ID as key (since it is unique) and as a value, we use a pointer to a Task struct, the same as above. To access a task and update its allocated memory, we can change it directly: *tasks["af4848743"].Memory = 5004328974* , where af4848743 is the task ID.

As for the Host registry, the changes performed at the tasks map will also be reflected at the classTasks map since they share the same pointer.

Synchronization: Throughout the chapters it was mentioned several times that the data structures need to be properly synchronized in order to avoid inconsistencies within the data structures. An inevitable tradeoff of applying synchronization is performance penalties. However, the data structures need to remain consistent so there is a tradeoff we need to couple, but we can try to diminish it as much as possible.

To accomplish this, we defined fine-grained locks so the data structures locked are the minimum to have everything consistent. The synchronization performed at the Host and Task Registry are different but the concept is the same.

To achieve these fine-grained locks mentioned in the previous paragraph, we resort to maps and structs again. There is a map (e.g. named mapLocks) with a string key representing the host class (LEE, DEE or EED) and as a value the following Lock struct is used:

```
struct {
classHosts map[string]*sync.Mutex
    lockRegion *sync.Mutex
}
```

Using the above struct, we can have more coarse-grained locks (using lockRegion) by locking at region level if required, or more fine-grained locks by locking at class level through the classHosts map, whose key is a string representing the host class (1,2,3 or 4) and the value is Go's internal lock. As an example of a coarse-grained lock, if we want to lock all hosts on region LEE we can use: *mapLocks["LEE"].lockRegion.Lock()*. If we want to use a more fine-grained lock to lock all hosts on region DEE and class 3 we can do the following: *mapLocks["DEE"].classHosts["3"].Lock()*.

The Task Registry uses the exactly same procedure for synchronization but since it has simpler data structures, it uses a single map (e.g. named lockTasks) with a string key representing the task class (1,2,3 or 4) and as a value Go's internal lock. So in order to lock access to class 4 tasks for example, we use: *classTasks["4"].Lock()*.

4.2.3 Monitor

To finish the components implementation description, we are going to describe how the monitor measures resource utilization and how it decides when to send an update or not, to the Host Registry or the Task Registry.

The Monitor has two threads, one to monitor the host resources utilization and the other to monitor tasks resource utilization. First, lets look into how the host's resources utilization are measured.

Host resources monitoring: Every 30 seconds, it is checked if an update should be sent to the Host Registry. However, measurements are not collected every 30 seconds, which could lead to errors. For example, imagine that between measurements (in a 30 seconds interval), the CPU utilization is 50% but at the very last second it spikes to 90%, and the next second (at 31 seconds), would drop again to 50%. The measurement collected would be 90% and it shouldn't because it was just a CPU spike during 1 second. This sort of issues would have significant consequences to the algorithm, because it would state that the host was under a heavy load when it actually wasn't, and we must wait another 30 seconds for the correction to be fixed (if there isn't another spike), loosing many requests in the meantime.

To deal with this issue, between the 30 seconds interval, samples are collected every 3 seconds. At the end of the 30 seconds, we average all the samples collected during that interval and use those values (CPU and memory) to check if an update should be sent.

In order for the update to be sent to the Host Registry, a condition must be verified. The difference (either CPU or memory) between the last update sent and the current measurement must be higher than a threshold, otherwise it is not worth to send the update, which would result in wasted communications between the components. The threshold is defined at 10 *p.p.* That is, an update is only sent if there is a different from at least 10 *p.p* between the last measurement sent and the current measurement. To illustrate this, consider that 30 seconds ago, a CPU measurement of 31% was sent. If the current measurement is 42%, an update is sent. However, if the current measurement is 35%, that measurement is not sent.

To avoid sending two messages, if both the CPU and memory threshold are exceeded, then both updates are sent together in the same message. Otherwise it sends an individual message for each update.

The threshold, the time between samples and the time between measurements are configurable and can be adapted to the environment or the type of workloads that the cloud environment deals with. For a highly volatile environment with constant variations, these configuration values can be lowered. In an environment where the workloads are constant, the configuration values can be increased significantly to avoid wasting resources by taking constant measures that do not change. We decided our configuration values based on a middle ground between a high volatile environment and a constant environment.

To collect resource usage information we use System Information Gatherer And Reporter (Sigar)[3]. It provides a simple and efficient way to access OS/hardware information. It has another significant advantage that it is cross platform (i.e, works in Windows and Unix based systems). This is important because, although Docker is still in beta version for Windows and Mac, it is just a matter of time for it to be widely

[3]https://github.com/hyperic/sigar

available at those systems. Therefore we had the concern to make this component cross-platform. The other components do not have this concern because they do not use any software platform dependent.

Tasks resource monitoring: The rationale behind tasks resource monitoring is the same as of the host monitoring. The configuration values are also the same except that the time between measurements is 45 seconds instead of 30. We increased the value because tasks resource utilization is not as volatile as the hosts resources utilization.

A single thread is used to measure every tasks since it is not practical to have a thread for each task and also because Docker has a command called *stats*[4], that returns a live stream resource usage statistics of all the containers running on a certain host. We leverage this command to get CPU and memory utilization of each task.

At the next Section we will see how the structs mentioned on this section are implemented as well as many of the information presented on Chapter, such as the communication components.

4.3 Software architecture

Represent by Fig. 4.3 is our software architecture composed by the Scheduler, the Host Registry and the Task Registry. For space reasons, the Monitor software architecture is represented separately at Fig. 4.4. At Fig. 4.3 is detailed everything that was explained on the current and the previous chapter. On this Section we will approach details that were not covered yet, for full clarity about our solution. We also present our extensions to Docker Swarm in order for our solution to be implemented.

At the bottom of the figure are the structs that contain information about the tasks and the hosts, shared between different components. At the Task struct the *Original CPU and Original Memory* are used when kills are performed. Imagine the case that a task receives a cut, but then is killed consequently being rescheduled. It needs to be rescheduled with its original values (pre-cut), and these are stored at Original CPU and Original Memory.

At the top of the figure are the structs mentioned on the previous Section. At the left top is a struct called TaskResources. As seen, the Host Registry is a complex component, all the computation we can take away from it, the better for the overall system performance. This struct is used with this in mind.

As referred on Section 3.3.1, when a task has terminated, the host class might need to be updated. The Task Registry has all the information required to perform this check since it has information about

[4]https://docs.docker.com/engine/reference/commandline/stats/

all the current active tasks on the host. Therefore, once a task has terminated and is the most restrictive task running on the host (the lower class task), it is checked if there are more tasks with the same class as the one that was terminated still running on the this host. If there aren't, Update is set to true, PreviousClass to the task class that terminated, and the NewClass to the next task class running on this host. If there are still tasks with the same class as the one that was terminated, Update is set to false and do not set any value for PreviousClass and NewClass since they will not be used. As for the CPU and Memory values, they are used in the struct because, the Host Registry needs to update the resources allocated, independently if Update is true or false. IP is used to identify the host as seen on Section 4.1.2.

At the top right of the figure are all the extended packages and the respective *class* files.(identified by the .go extension). Our algorithm presented on Section 3.4 is extended in the Strategy package where the other Docker Swarm scheduling algorithms also reside. We are now going to look at the necessary modifications performed at Docker Swarm original source code. These extensions are presented and the rationale behind the extensions can be found on the Appendix of this document.

To finish this section, the Monitor software architecture is illustrated at Fig. 4.4. Everything presented was already covered on previous sections, such as currentTasks for example, seen on Section 3.2, which maintains the active tasks on the host the Monitor belongs to.

Summary

This chapter presented the details about the implementation of our proposed solution, including configurations to Docker Swarm and the components. It started by detailing how the internals of the components were implemented, with performance, scalability and consistency concerns in mind. Next, was presented a detailed description of our software architecture, which included all our components, shared data structures between components and extensions performed to Docker Swarm. In the next chapter we present the evaluation results of the solution presented in this and the previous chapter, comparing with the different Docker Swarm scheduling algorithms.

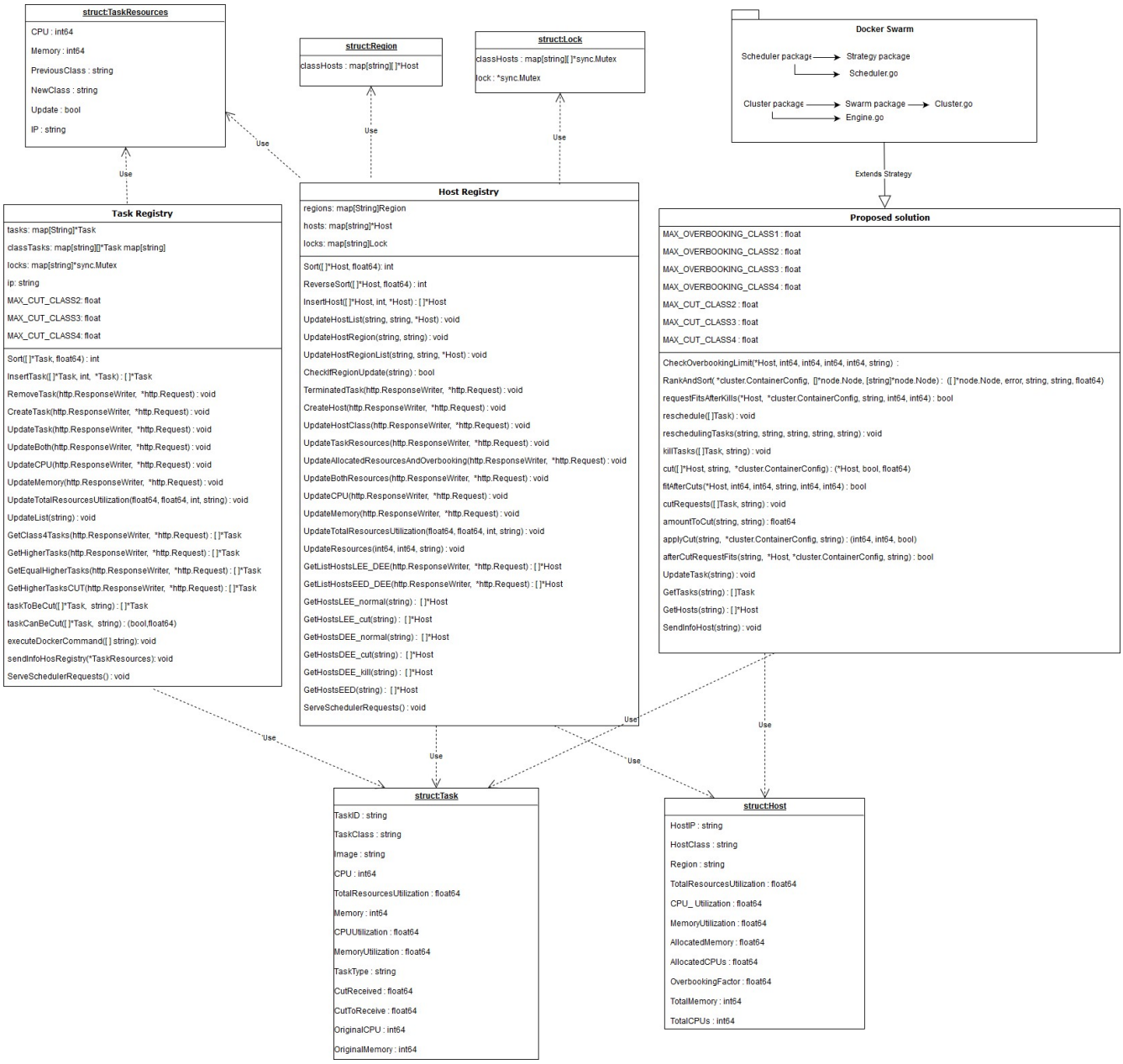

Figure 4.3: Software architecture containing the Scheduler, Host Registry and Task Registry

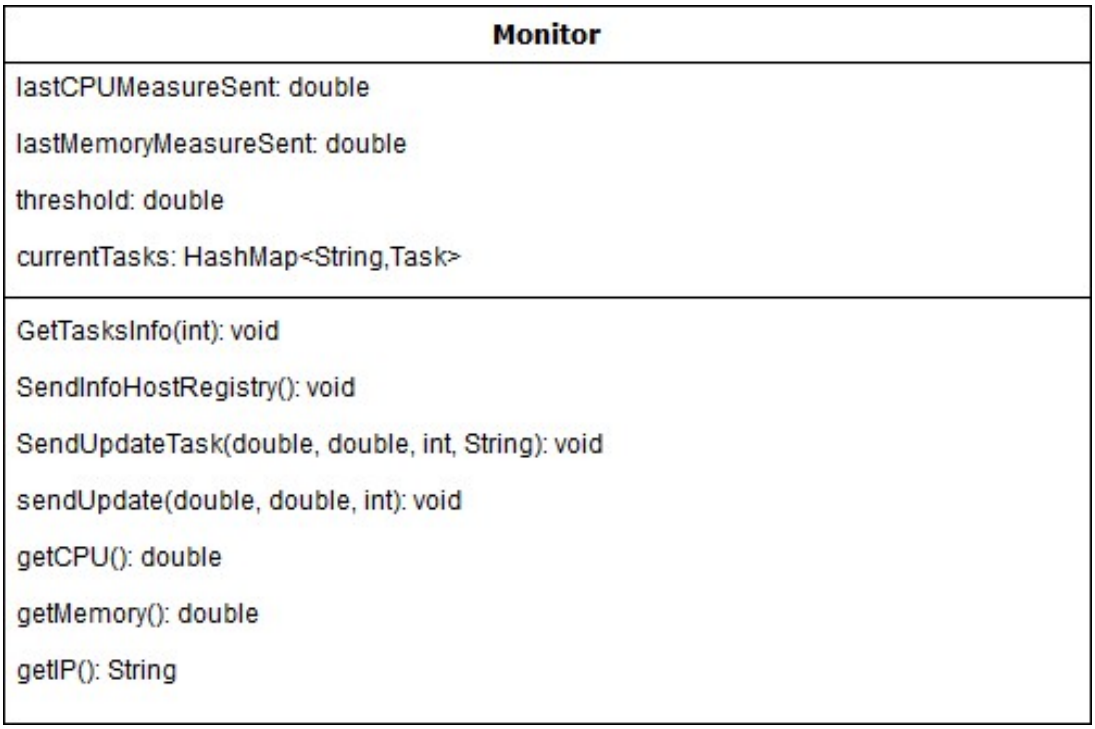

Figure 4.4: Monitor software architecture

5 Evaluation

This chapter describes the experiments carried out to evaluate the proposed solution agaisn't the three Docker Swarm scheduling algorithms, *random*, *spread* and *binpack*, as seen on Section 2.1.3. We start by describing how the evaluation was carried out, followed by its results.

5.1 Experimental setup

Evaluating cloud solutions, in order to be realistic, requires thousands of machines. Unfortunately, for academic purposes, these amounts are normally not available to students, which have to resort to simulation. However, simulations have many drawbacks, such as being in closed, safe environments not susceptible to "noise" as in real deployments, therefore not producing realistic results. Since the proposed solution is focused on cloud environments, susceptible to different kinds of disturbances, we decided not to use simulation, instead making a real deployment.

However, our solution was deployed on 6 hosts only, provided by INESC-ID. This is not ideal but we preferred it over simulation to see how it would behave in a real environment, not susceptible to the drawbacks provided by simulations. We thought at some point to complement our real deployment with a simulation deployment but it would be another work to incorporate realistic resource usage variance in the simulation. Although, it is open for future work has all the other things that could have been improved with more time as will be seen on Section 6.1.

As already mentioned, our solution was deployed on 6 hosts. These hosts are powered by an Intel Core i7-2600K CPU @3.40GZ, 11926 MB RAM and HDD 7200RPM SATA 6GB/s 32MB cache. One host (host 2) will serve as the Manager (Section 3.1) and the remaining hosts will serve as workers, executing client's requests.

Our goal, was to evaluate our solution in the most realistic environment possible. To this end, we also intended to use real workload traces. However, there are none available for Docker Swarm, which also hinders realistic simulation with reduced overhead. There has not been many research on benchmarking Docker Swarm either. We only managed to find benchmarking tools that evaluate scheduling speed and not scheduling quality (e.g. scheduling decisions). Due to this, we had to create our own custom

workload and extensions to collect metrics. These are presented next.

5.1.1 Workload generation

Using a custom workload is never as realistic as real workload traces, however we tried to mitigate this difference as much as possible by attempting to design representative workloads for testing. Making it unbiased was also our concern, that is, so it does not favor our solution against the ones we are comparing with. Our evaluations lasted one hour in order to have as much variability as possible. The workloads generated were saved and used on all attempts on the different scheduling algorithms so that they are tested with the same conditions. The following requirements for each workload had to be generated:

- CPU requirement;
- Memory requirement;
- Request makespan;
- Workload type;
- Request rate;
- Request class;

The CPU and Memory requirements are the amount of resources the clients requests for that workload. These are generated using an exponential distribution. We decided to use an exponential distribution since it provides a good variability. An exponential distribution provides this opportunity because in a real environment, we can expect more small requests than large ones.

The number generated by the exponential distribution was mapped to a CPU and memory value. For CPU, the minimum value depends if it was a service or a job. If it is job, the minimum CPU assigned was 204 CPU shares (equal to approximately 20% of a single core utilization). This value was chosen because if a client choses a job, then a CPU lower than 20% would be unrealistic. If it was a service, the minimum CPU shares was 2, because services do not require as much CPU as jobs. As for the maximum, it was 1024 CPU shares (equal to 100% of a core utilization). For Memory requirements, the limits are the same for jobs and services, the minimum was 256 MB and maximum was 2GB.

The request makespan was also generated by the exponential distribution. This makespan was used to control workload's lifetime. Since the evaluations lasted one hour, we needed to limit the duration of the workloads so that new requests could be scheduled. After this makespan elapsed, the task was terminated. The minimum value was 30 seconds and the maximum was 30 minutes. Again, through the

exponential distribution, we ensure there are more requests with small makespans than requests with makespans close to 30 minutes. If we had more long requests than small ones, we could not test the quality of our solution since the machines would have their resources fully occupied for a long period of time.

The workload type was chosen randomly between the four types of workloads that we have selected. The types and respective application used for that type are the following:

- CPU-intensive (job): FFMPEG[1]
- Memory-intensive (service): Redis[2]
- CPU and Memory intensive (job): Deep-learning[3]
- Non-intensive (service): Timeserver[4]

For each of these application's types, we have selected real and popular Docker applications (with the exception of the non-intensive), in order to be representative of each type. For CPU-intensive we have chosen **FFMPEG**, a video encoding application. Video encoding is usually also memory intensive but from our tests, FFMPEG's memory usage is minimal and it uses considerable CPU, therefore we use it as being CPU-intensive. The requests that have this workload type, enconded a video big enough so that the job only terminated after the makespan had elapsed.

We used **Redis** as the memory intensive application. Redis is an in-memory key/value store. When a request with this workload type was scheduled, a Redis server was launched at a certain port and was filled with random data. A short-while after, requests were issued to that Redis Server (such as, checking if a key exists, retrieving values, etc).

For the CPU and memory intensive, we have chosen a **Deep-learning** application, where a neural network is trained to zoom in images. As for the CPU intensive application, this application must not be concluded before the makespan elapses. For this reason, we have chosen a high-resolution image that the application will zoom in.

Finally for the non-intensive application, we created a Docker application called **Timeserver** which simply returns the time when requested.

[1] https://hub.docker.com/r/jrottenberg/ffmpeg/
[2] https://hub.docker.com/r/redis/
[3] https://hub.docker.com/r/alexjc/neural-enhance/
[4] https://hub.docker.com/r/sergiomendes/timeserver/

Besides a better resource utilization, we have to ensure that our solution does not break any SLAs. To ensure that this is not the case, as will be seen on Section 5.1.3, we evaluated how long the jobs took to execute and how long the services took to respond to requests. For this purpose, we defined requests rates to continuously issue requests to the services to the test the quality of our solution. Until the services makespan expired, requests (based on the request rate) were issued every 10 seconds. The request rate was defined based on the memory (since one of the two types of services was memory intensive and the other is non-intensive) assigned to the service we are going to evaluate. The higher the memory the higher the request rate:

- < 500MB: 20 requests;
- < 1GB: 40 requests;
- < 2GB: 80 requests;

The last thing to was generated was the request class. We have given more probability for classes 2 and 3 because we believe that these would be the most used in a real situation. Class 4 since it has a big depreciation, it would be less used than classes 2 and 3, however in our view, it would still be more used than class 1 requests due to the lack of benefits (in terms of compensation) this class provides. The probabilities for a request to be a certain class is then the following:

- Class 1: 10%;
- Class 2: 30%
- Class 3: 45%
- Class 4: 15%

Next we will see which and how metrics are collected in order to produce the results presented on Section 5.2.

5.1.2 Metrics collection

To evaluate our work, we focused mostly on comparing our solution with the current existing ones by comparing: **scheduling speed**; **failed/successful allocations**; **resource utilization** (CPU and Memory) throughout the experiment; **job makespans**; **services response times**. We also did an individual evaluation to our solution, to see how much it resorts to cuts and kills.

For measuring the scheduling speed, we used an already existing Docker Swarm benchmark tool, swarm-bench[5]. This tool launches requests and measures the time it takes for them to be scheduled.

[5]https://github.com/aluzzardi/swarm-bench

We extended this tool to also measure successful and failed allocations.

For the remaining metrics, there were no existing tools. To gather information, we had to extend our components to gather this information. The resource utilization information was gathered by the Host Registry whenever an update comes from the Monitor.

Measuring job makespan consists in measuring the amount of time the job takes to execute. As already mentioned on this Chapter, we terminate the job before it finishes based on the makespan value generated. In order to measure job makespan, we launched specific jobs with a known makespan (measured previously) without that makespan value so they are not finished beforehand. Every 5 minutes we launched these jobs, alternating between the CPU-intensive job and CPU/Memory intensive job.

To measure services response times, we made requests to those services. However, we did not know beforehand where those services would run and which port they would use since it depended on the scheduling decision performed by the Scheduler. We created a web server at the Manager, that received from the Task Registries, information on where to issue requests. This web server measured the amount of time it took to receive all responses. Once a request was allocated, the Task Registry read the request rate value and sent a GET request to the web server with, the port number that service was using, if it was redis or timeserver (because the type of request for each was different, therefore they need to be differentiated) and the IP of the Task Registry. Now the web server has all the information it needs to issue requests to services.

When a cut or kill occurred, the Host Registry counted those occurrences. It also gathered how much CPU and memory was cut for posterior analysis.

5.1.3 Evaluation execution

As was previously mentioned, the evaluation lasted 1 hour. For each scheduling algorithm (including our solution), each evaluation was ran three times and averaged at the end, in order to avoid extreme results. All evaluations will used the same traces.

Sending all requests at once is not realistic so we decided to send two requests per second to the Manager. We kept sending requests until a memory or CPU limit is reached. The full memory capacity of the 5 hosts combined is roughly 60GB and the full CPU capacity is 40970 CPU shares. We defined the limit as being 50% (i.e. 200% overbooking) more than the full capacity. So the limit was 90GB for Memory and 61440 CPU shares for CPU. We impose these limits because if we kept sending requests

regardless of these limits, they would just be discarded because there would be no resources available to cope with them.

While sending requests, we saved the makespan values for each request. When either of the limits are reached, we averaged all the makespan values and waited for that value before resending requests again. By waiting this period of time, we ensured that there were resources available when we restarted sending requests.

Now that we have seen how the traces are generated, how the metrics are collected and how the evaluation is executed, the next section presents the results of the evaluations carried out.

5.2 Evaluation results

The previous sections presented the different types of evaluations carried out. At this Section, is presented the results of those evaluations, starting with the results related to the main objective of our work, achieving a better resource utilization to maximize energy efficiency. We will see that our solution allows significantly more requests to be allocated, achieving an overall better resources utilization. A natural and unavoidable tradeoff of our solution is a comparatively slower scheduling speed to the other solutions, these differences will be exposed. As seen on Section 2.2.2, another tradeoff of overbooking could be that jobs or services, can take longer times to finish or to respond, respectively. We will see if this is the case in our solution. To finish, the cuts/kill ratio is presented and we will see how they are useful, especially the cuts, in order to increase the amount of requests that can be allocated.

We intended to evaluate our solution against the three Docker Swarm scheduling algorithms. However, it was not possible to evaluate the random strategy. This is due to the fact that this strategy does not check for resources constraints when making scheduling decisions. It keeps allocating requests regardless if the CPU or memory limit of the machine has been reached. So, in these circumstances, it makes no point in testing if it as it would severely overallocate and degrade performance.

5.2.1 Successful and failed allocations

The results obtained are presented at Table 5.1. We can quickly see that our solution (named as Energy) has a significantly higher success rate than the two solutions provided by Docker Swarm. By having such high fail rates, the other solutions would require more machines than our solution does, consequently using more energy. A natural (but outside of the scope) extension to our work, producing even better energy gains would be to have machines be turned on/off dynamically, according to the

	Successful allocations	Failed allocations	Success rate	Failure rate
Spread	1229	904	57.7%	42.3%
Binpack	1256	967	56.5%	43.5%
Energy	1404	274	83.7%	16.3%

Table 5.1: Successful and failed allocations

overall resources usage. By having such a high success rate we could have much less machines turned on compared with the other solutions.

From the results presented at Table 5.1 we can also see that our solution deals with less requests than the other two approaches, in an one hour evaluation. This derives from the fact that our algorithm is comparatively slower than the other solutions (Section 5.2.3), due to our solution keeping the resources almost fully utilized for a longer period of time as will be seen on the next Section.

This tradeoff is compensated by the high success rate and higher absolute value of successful allocations, since it managed to successfully allocate more requests than both solutions, despite dealing with less requests than those solutions. As will be seen on Section 5.2.3, these values would be lower if more machines were added as can be extrapolated by the data presented on that Section.

5.2.2 Resources utilization

On this Section is presented the results regarding resource utilization. This is an important evaluation because in order to achieve the best energy efficiencies, the hosts must be in the DEE (between 50% and 85% utilization rates, where the desired rates are closest to 85%) region the most time possible. This evaluation shows how the different solutions behave. We start by analyzing CPU usage, followed by memory usage.

CPU usage: Since we only use 5 hosts (1 of the 6 is the Manager), we present the results for CPU utilization for the 5 hosts individually, for each of the solutions: Spread is represented by Fig. 5.1, Binpack by Fig. 5.2 and Energy by Fig. 5.3.

By looking at the graphs, we can see that our solution (Energy) besides achieving a better overall CPU utilization, it is also more consistent than the other approaches. Spread and mainly Binpack, have many fluctuations between high and low CPU utilizations. By looking at Fig. 5.2, we can see several periods of time where there are under-utilized hosts (below 50% CPU utilization), for example, between 0-10 minutes and more severe between 20-40 minutes. Spread, Fig. 5.1, is not as bad as Binpack but we can also detect several points with a moderate CPU utilization, such as between 0-10 minutes and from 20-30 minutes.

These two are examples of how overbooking can be very useful in maximizing resource utilizations.

Despite having the resources fully allocated, since they are not being used 100% of the time (services only receive requests from time to time as was explained), this resource inefficiency happens. This is even more salient in real life scenarios, where clients after ask for much more resources than they actually need. This clearly indicates that more resources could be allocated to some of the hosts to make them more efficient. This is illustrated by the results of our solution, Fig. 5.3, where most of the time, the hosts have more than 70% resource utilization. There are some exceptions however, for example for host 1 at roughly 5 minutes and some hosts between 20-30 minutes. However these lower peaks are still significantly higher than the peaks of Spread and Binpack. The former lower peaks can drop lower than 40% and the latter bellow 30% while ours does not go bellow 50%. Besides the lower peak being significantly higher, they are also shorter. They quickly go up to good CPU utilization levels.

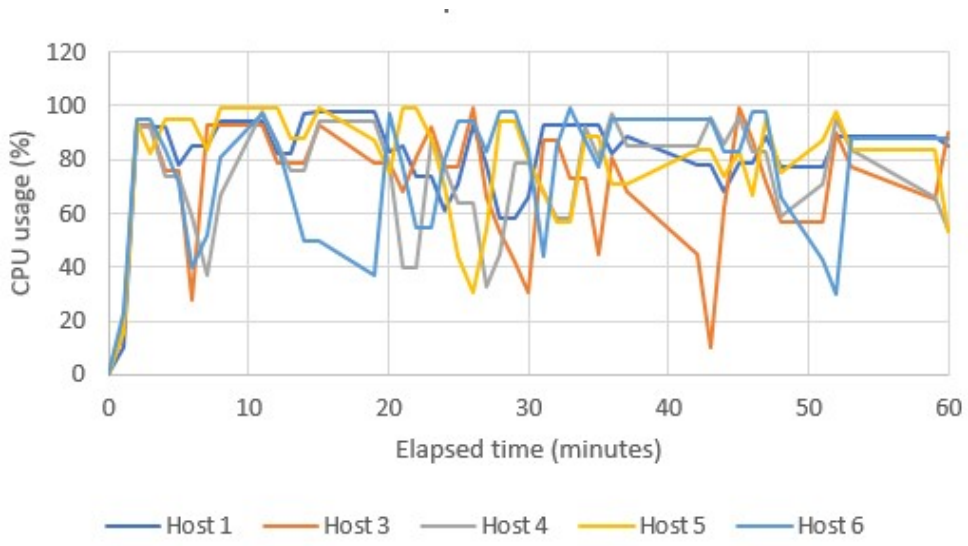

Figure 5.1: Spread - hosts CPU utilization

To provide an easier comparison, a graph is presented with the average CPU utilization of all the hosts of each solution, Fig. 5.4. As expected from the previous results, we can see that our solution (green line) is more consistent than the other two, fluctuating most of the time between 75% and 88%. The Binpack solution (blue line) is most of the time bellow 80%. Spread (orange line) is better than Binpack, as expected as seen on the previous graphs, but worse than our solution, most of the time it is bellow the green line, with some exceptions.

However, the results presented by this graph can be elusive. If we looked at this graph without looking at the previous graphs, we could deduce that Spread has a good performance and Binpack as well (on this graph Binpack is always above 50%). But, as we saw on the previous graphs, Spread and Binpack, while they have hosts with a good CPU utilization, they have other hosts with a bad CPU utilization and the average balances both, hiding the fact that there hosts with significant resource inefficiency.

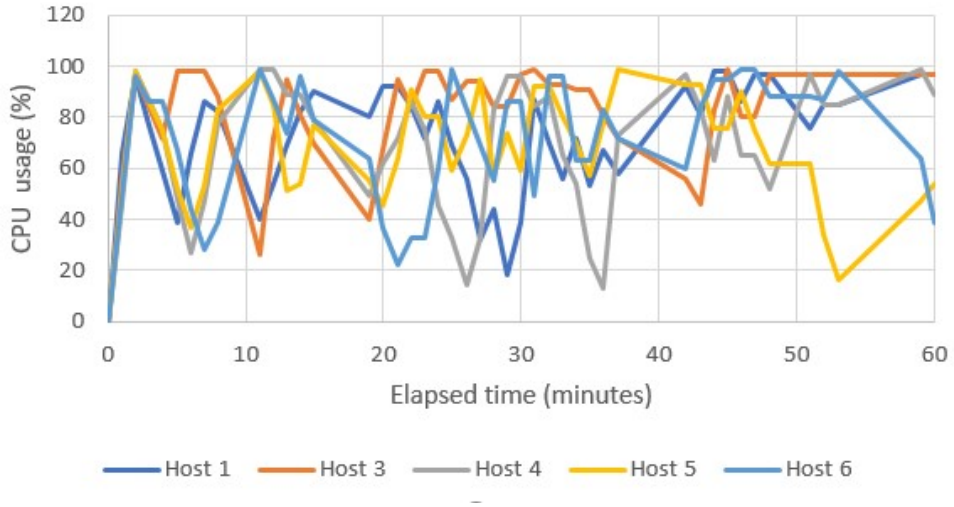

Figure 5.2: Binpack - hosts CPU utilization

Memory usage: For the same reasons as for the CPU evaluation, we start by presenting the memory utilization of each host individually for each solution and the end, present a graph for the average memory utilization of all the hosts. Spread hosts memory utilization is represented by Fig. 5.5, Binpack's by Fig. 5.6 and Energy's by Fig. 5.7.

Again, our solution presents better results than the existing solutions provided by Docker Swarm. As for CPU, Binpack has the worst performance, where only one host (Host 3 at the 3rd minute) manages to surpass 60% memory utilization and only for a short period of time. Spread is more consistent in the 40-60% zone than Binpack, especially after the 30 minutes mark, but it only goes up more than 60% a few times along the whole evaluation.

Our solution in terms of memory results, does not provide consistent results at the same level as the CPU utilization results. But it performs significantly better than our competitors. The other solutions surpassed the 60% mark briefly. Our solution achieves it throughout the whole evaluation as can be seen, and it even goes over 80% memory utilization at some points.

Fig. 5.8 presents the results regarding the average hosts memory utilization of each solution. Despite being more inconsistent as we saw on Fig. 5.7, our solution provides bigger improvements regarding memory utilization over CPU utilization as can be seen by at Table 5.2. We can see at this table that our solutions provides a 5.6 *p.p* improvement over Spread and 8.2 *p.p* over Binpack, regarding CPU utilization. The memory utilization improvement is much more significant, achieving an improvement of 15,8 *p.p* over Spread and 18,9 *p.p* over Binpack.

Despite this big improvement, the desirable result would be a more consistent one, as with CPU utilization, illustrated at Fig. 5.3. Also, 55.7%, despite being in the DEE region, is still very close to

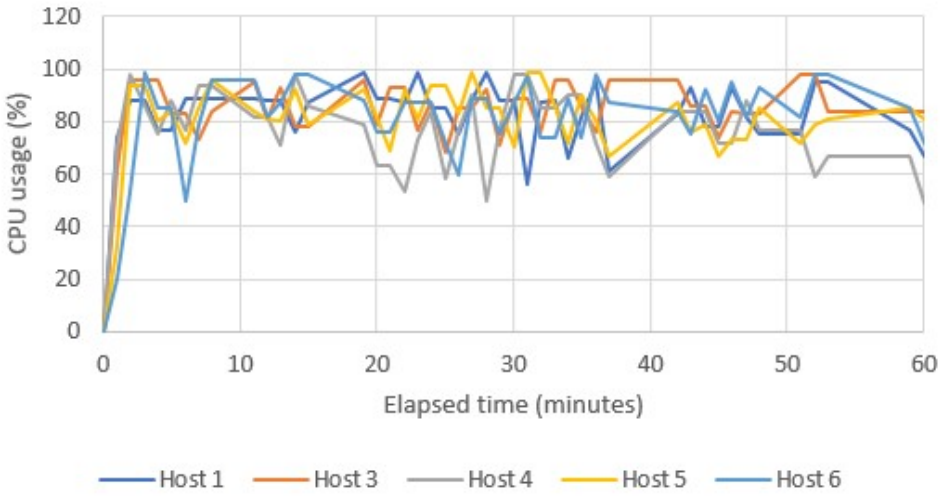

Figure 5.3: Energy - hosts CPU utilization

	Average CPU utilization	Average Memory utilization
Spread	74.9%	39.9%
Binpack	72.3%	36.8%
Energy	80.5%	55.7%

Table 5.2: Average CPU and Memory utilizations

the LEE region, therefore improvements are still required to push this value higher, maximizing memory utilization.

5.2.3 Scheduling delays

The significant improvements analyzed in the previous two sections, unfortunately, do not come without a price. This section presents the results regarding the scheduling delays, i.e. the time to schedule requests. One of the tradeoffs of our solution is the extra time it takes to schedule a request, due to having more resources allocated and being more complex than the other two solutions. Table 5.3 has detailed information about the time it takes to schedule throughout the evaluation of each solution. The type of information is divided into: average scheduling time, the 50th, 90th and 99th percentiles. The values are calculated between the elapsed times. For example, the average time for 0-5 minutes is the average time it takes to make scheduling decisions between that period.

By looking at the average values, as expected, our solution performs worse than Spread and Binpack. Despite being more complex, our solution is only slightly slower when the system has more resources free, has can be seen by the 50th percentile at 0-5, 25-30 and 50-55. This last note shows that this scheduling delay can be decreased if more machines are added.

Looking at the extreme results, the 90th and 99th percentiles, we can see that these values are

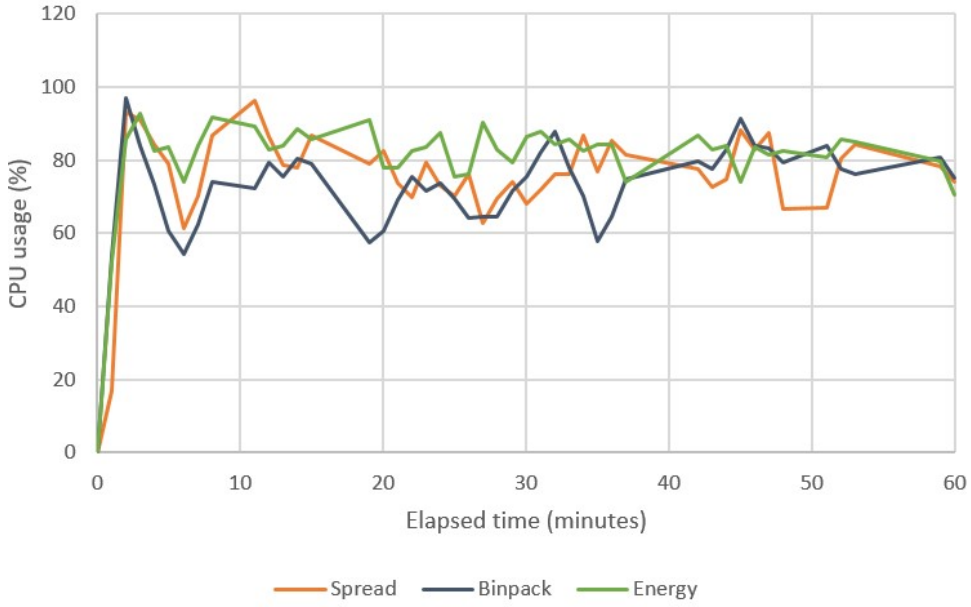

Figure 5.4: Average hosts CPU utilization of each solution

significantly higher than the 50th percentile for all solutions, but more emphasized on our solution. Although only 10% of the requests suffer from this long scheduling time, in a real environment, 10% could represent millions of requests, potentially bringing financial repercussions to the CSP. To deal with this issue, priorities could be given to classes, in order that class 4 requests were the ones that fall into the 90th and 99th percentiles and the lower class requests would always fall under the 50th percentile.

5.2.4 Response times

By using an overbooking strategy, allocating more resources than the limits of the machines, there are risks regarding the response times provided by both jobs and services. Our solution, as explained on Section 3.4.1, was designed to avoid these issues by giving priority to schedule requests on hosts that are on the LEE region and the DEE region, where a best-fit approach is used to try and avoid those hosts to enter the EED region where besides, starting to experience less energy efficiency, the response times could also be affected due to the high utilization of resources. For this purpose, the kill algorithm is responsible to try and bring the host back to the DEE region, avoiding both these problems just mentioned.

On this Section we will see that despite allocating more requests as was seen on Section 5.2.1, our solution is close to our competitors response times, therefore, we can safely claim that we increased resource utilization, improving energy efficiency, without significant penalties regarding Quality of Service, avoiding SLA violations.

Solution (ms) /Elapsed time (minutes)	Spread	Binpack	Energy
0-5	Average: 2904,87 50th: 9.90 90th: 10002,95 99th: 22974,34	Average: 5278.21 50th: 10.19 90th: 18696.95 99th: 27176.24	Average: 18469.96 50th: 11.94 90th: 72720.68 99th: 128048.71
5-10	Average: 3354,43 50th: 10,215 90th: 11422,62 99th: 44155,3	Average: 3846.83 50th: 9.23 90th: 14873.89 99th: 29386.84	Average: 55869.52 50th: 2190.31 90th: 193995.17 99th: 222260.83
10-15	Average: 5720,64 50th: 9,64 90th: 14440,4 99th: 68623,37	Average: 4821.98 50th: 10.1 90th: 20066.55 99th: 37442.99	Average: 32865.09 50th: 1467.7 90th: 113716.57 99th: 150872.68
15-20	Average: 5720.68 50th: 9.64 90th: 16972,52 99th: 68623.37	Average: 4233.91 50th: 10.37 90th: 16680.14 99th: 35839.41	Average: 33644.27 50th: 838.11 90th: 121751.3 99th: 169663.67
20-25	Average: 4287.1 50th: 10.1 90th: 12138.77 99th: 56623.41	Average: 6569.64 50th: 10.16 90th: 22410.08 99th: 61825.73	Average: 45443,75 50th: 1107.14 90th: 185867.72 99th: 273002.71
25-30	Average: 4281.98 50th: 10.49 90th: 14833.28 99th: 33677.65	Average: 4368.09 50th: 10.09 90th: 18108.94 99th: 28256.79	Average: 33264.43 50th: 15.31 90th: 13997.85 99th: 202526.65
30-35	Average: 4561.09 50th: 10.77 90th: 14162.09 99th: 51546.53	Average: 4630.1 50th: 9.56 90th: 13269.16 99th: 40586.64	Average: 37849.88 50th: 1464.58 90th: 145200.94 99th: 204593.55
35-40	Average: 4979.68 50th: 9.98 90th: 20431.98 99th: 64590.9	Average: 4518.57 50th: 10.45 90th: 17867.58 99th: 36027.47	Average: 55226.02 50th: 1139.49 90th: 199052.23 99th: 225747.18
40-45	Average: 6165.39 50th: 10.09 90th: 12955.69 99th: 97868.28	Average: 5677.06 50th: 9.13 90th: 20915.75 99th: 50454.16	Average: 42314.51 50th: 735.23 90th: 315134.33 99th: 173454.34
45-50	Average: 3762.55 50th: 10.74 90th: 13457.83 99th: 24530.57	Average: 7112.71 50th: 9.52 90th: 20472.89 99th: 85855.37	Average: 32578,95 50th: 1323.67 90th: 142789.02 99th: 234586.78
50-55	Average: 4685.35 50th: 10.35 90th: 14950.33 99th: 51035.95	Average: 6079.21 50th: 9.73 90th: 22136.46 99th: 51136.6	Average: 23251.23 50th: 13.79 90th: 19237.15 99th: 223237.94
55-60	Average: 5376.25 50th: 10.95 90th: 19209.29 99th: 51372.29	Average: 6011.39 50th: 9.86 90th: 24105.28 99th: 57326.5	Average: 31247.08 50th: 1024.541 90th: 153109.42 99th: 202359.59

Table 5.3: Scheduling delays

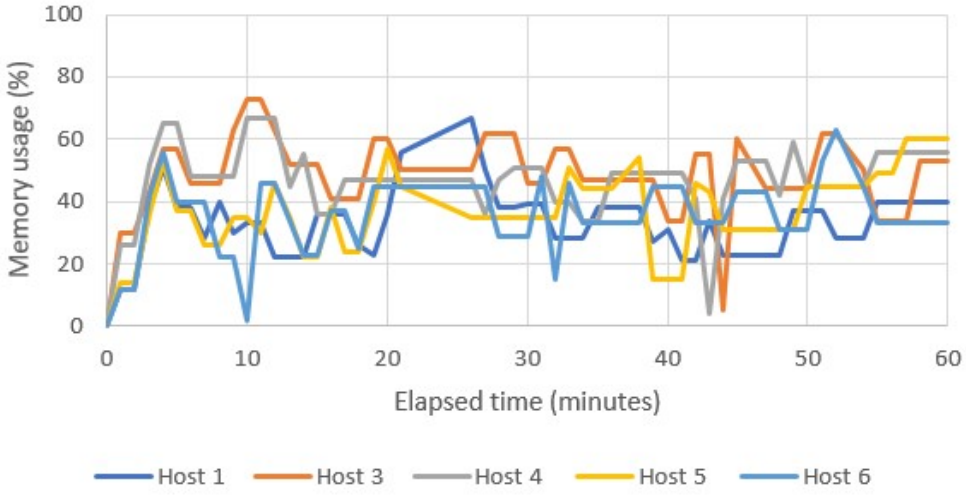

Figure 5.5: Spread - hosts Memory utilization

Table 5.4 presents the response times obtained for each type of workload used. Redis - 20 indicates that a request rate of 20 to access Redis was used, the same applies for the following columns. For the CPU-intensive workloads, FFMPEG, we can see that our solution has a better average time than the other two, although it has a higher 50th percentile compared with Binpack. We achieve better response times at this type of workload because of what was explained on the last paragraph. By leveraging the kill algorithm, we avoid extremely (not always though as can be seen by Fig. 5.3, there are periods where the utilization goes above 90%) high CPU utilization rates, which would lower response times. This can be seen by the results obtained by Spread, which are significantly higher than Binpack since it has much higher CPU utilization rates as was seen on Section 5.2.2.

For the CPU/Mem intensive workloads, Deep-learning, we can see that our solution no longer has the best results, but is still better than Spread (better average and 75th percentile results). This decrease in performance compared with Binpack and Spread for this type of workload is unavoidable, because we have significantly more memory utilization rates than the other solutions.

Next we have the Redis results, the memory-intensive workload. Redis produced some unstable results as can be seen by the fact that Redis-80, for Binpack, has better results than Redis-40 and Redis-20, which should not be the case, since Redis-80 is twice the request rate of Redis-40, and four times Redis-20. We thought this could be due to coincidences, such as Redis-80 requests are performed at certain periods where the hosts are experiencing low resources utilization, although this would be unlikely since three tests were performed. To see if this was the case, we did an extra one hour test for each solution, just gathering Redis metrics, but again produced these unstable results. If the results were consistent, we could expect increasing times from Redis-20 to Redis-40, and from Redis-40 to Redis-80. We assume that our solution here would achieve worse times than the other solutions

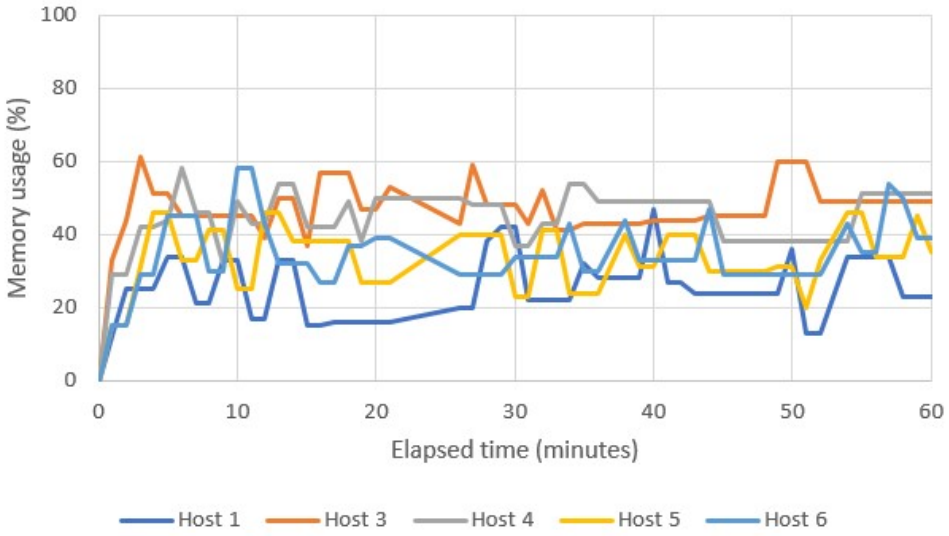

Figure 5.6: Binpack - hosts Memory utilization

Workload (ms) / Solution	FFMPEG	Deep-learning	Redis - 20	Redis - 40	Redis - 80	Timeserver - 20	Timeserver - 40	Timeserver - 80
Spread	Average: 333.43 50th: 273 75th: 485	Average: 151.41 50th: 140 75th: 177	Average: 480.53 50th: 115 75th: 587	Average: 560.48 50th: 168 75th: 619.5	Average: 455.08 50th: 322 75th: 880	Average: 1126.04 50th: 800 75th: 944	Average: 2193.75 50th: 1645 75th: 2513.25	Average: 3460.5 50th: 3208 75th: 3477.25
Binpack	Average: 266.51 50th: 189.5 75th: 402.5	Average: 146.76 50th: 137 75th: 163.5	Average: 365.28 50th: 166 75th: 413	Average: 335.91 50th: 197 75th: 220	Average: 239.4 50th: 244 75th: 284	Average: 1475.67 50th: 818 75th: 1126.75	Average: 2380.2 50th: 1669 75th: 2047	Average: 3544.22 50th: 3196 75th: 3477.25
Energy	Average: 250.87 50th: 199 75th: 367	Average: 149.56 50th: 140 75th: 171	Average: 313.2 50th: 247 75th: 393	Average: 393.67 50th: 149 75th: 528	Average: 436.14 50th: 276 75th: 242	Average: 1727 50th: 804 75th: 1315	Average: 2547.48 50th: 1768 75th: 2817	Average: 3570.33 50th: 3332 75th: 3782

Table 5.4: Response times

because of what was seen with CPU/Mem-intensive workloads due to the memory impact, potentially worsening as the request rates increased.

Finally we have the non-intensive workloads, the Timeserver. Here our solution performs slightly worse than the other solutions at all requests rates. The positive note is that the degradation does not increase as the request rates increase. It actually got closer to the other two solution at the highest request rate (Timeserver-80), as can be seen by difference the between the average times, which is lower than at Timeserver-20 and Timeserver-40.

5.2.5 Cuts and kills

To finish this chapter, we will look into the two parts of our solution that are crucial to the operation of whole system, cuts and kills. Cuts play an important role in allowing more requests to fit on the hosts. On this Section, we will see how much we resort to cuts and how much resources do we gain by leveraging this approach. Then we will look at how many kills, important to keep the system balanced, were executed throughout the evaluation.

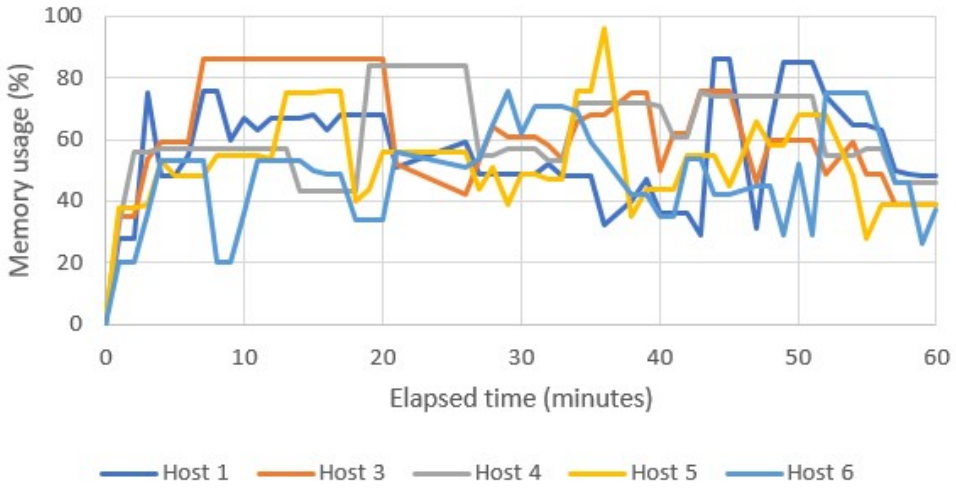

Figure 5.7: Energy - hosts Memory utilization

A total of **636 cuts** were performed throughout the evaluation. This resulted in **112736 CPU shares** and **189.3919 GB memory** being cut. These values are the reason why we achieve such a high allocation successful rate (Section 5.2.1). If we resorted only to overbooking such as other approaches in the literature, the successful allocation would be lower because 112736 CPU shares and 189.3919 GB memory could not have been allocated.

This approach is our biggest contribution because, as can be seen by these results, if it has such a big impact with just 5 machines in an hour experiment, with more machines and more time, we can expect this approach to be even better. However, it depends on the willingness of the client to have its resources cut. This could be arranged by compensating the clients based on the class they choose. The higher the class, the higher the compensation. Another possibility is to have a more sophisticated Manager, which dynamically cuts and undoes the cuts, according to how much resources the tasks are currently using, always leaving a safety margin in case there is a peak in utilization, which the Manager is not quick enough to react.

Kills are important to avoid the hosts from entering extremely high utilization values. However, as for the cuts, the clients must be willing to accept their tasks to be killed, again being properly compensated. Only **202** kills (14,4% of the successfully allocated requests) were executed throughout the experiment. Even if those 202 tasks that were killed could not be successfully rescheduled and if we considered them as not being allocated, we would still have a higher successful allocation rate than Docker Swarm's solutions. This approach has two limitations. The first one was already mentioned, and it is that tasks that were killed may not have been successfully rescheduled, this happened with 36 (18,2%) of the killed tasks and 2.56% of all the tasks. To fix this issue, if the task that was killed could not be allocated

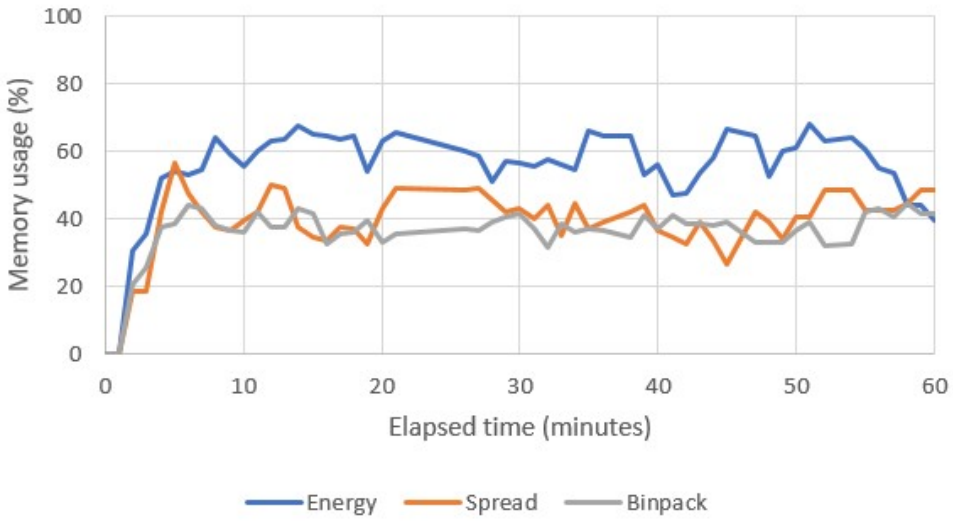

Figure 5.8: Average hosts memory utilization with each solution

twice, we lowered the request class in order to compensate for the delay in rescheduling. However this brought a problem which we did not had time to fix so we did no go forward with this solution. This produced a cascading effect, that is, this was making all tasks eventually go to class 1, because the ones being rescheduled, since their class decreased, they were killing tasks and those killed would have their classes reduced and kill more tasks and so on, until they all reached class 1, where we could not apply overbooking or cuts. Another possible solution is to keep it on hold until it can be scheduled, providing the client with proper compensation for the time his requests are not being serviced. However this solution is still not ideal and perhaps could be combined with priorities, in order to be more fair with the clients that had their requests successfully scheduled. However this requires further consideration and is left as future work, as the purpose of this proposal is to show the potential of the kill algorithm in order to keep the system resources balanced.

A second limitation is regarding jobs. When a job is killed, it must start what it was doing from the beginning. As future work, this could be solved by using live migrations, therefore not requiring the job to start from the beginning when it is allocated to a new host, it is a simple matter of the orchestrator, issuing a migration or a checkpoint instead of a kill.

Summary

Despite being a commercialized solution for some years, there are no benchmarks that evaluate Docker Swarm's scheduling decision quality, only its scheduling speed. For this reason we had to create our own benchmarks to evaluate our solution and extend it to Docker Swarm's scheduling strategies

in order to perform a proper comparison between each solution. Therefore the chapter starts with a detailed description of how the evaluation was setup, executed and evaluated. The chapter finishes with the evaluations results together with an explanation of the reasoning beyond those results. The first results are a comparison between the solutions, where our solution managed to achieve a higher success allocation rate, which contributed to the higher CPU utilization and even higher memory utilization rates comparing to our competitors, still offering similar QoS results. A drawback of our solution is the time it takes to schedule a request, although, this is unavoidable as could be seen throughout this chapter. Finally results obtained regarding the cut and kill algorithm are presented, crucial for both the good results and maintaining QoS levels, important to avoid SLA violations. The next chapter concludes by summarizing the document and pointing out some limitations and how they can be dealt with in future work.

6 Conclusion

Despite all the effort done by academia, the problem of energy consumption in data centers persists and needs to be addressed. In this work we started by identifying the current solutions that exist and their challenges, in order to identify opportunities so that we can contribute to the literature. Due to the lack of work regarding containers, we defined our objective, develop an energy-efficient scheduling algorithm using Docker. To understand containers, we started by studying their predecessors, modules and components. Afterwards, we synthesized the different types of containers and their orchestrators, analyzing them after their study. Throughout the years, many mechanisms and strategies for energy-awareness were proposed. We wrapped them up, describing and analyzing them accordingly.

The analysis of the related work enabled us to make our design choices, choosing Docker as container platform, Docker Swarm as the orchestration platform and overbooking as the strategy to achieve the proposed goal of this thesis. Due to the simplicity of Docker Swarm scheduling algorithms, simply applying an overbooking strategy would be enough to achieve better results. However, we decided to go further than this, proposing the cut concept. The cut combines perfectly with the overbooking strategy, although some concerns have to be taken into consideration as was seen, to avoid prejudicing the clients. The kill algorithm demonstrated its potential in keeping the system resources balanced, avoiding global SLA violations. However, it still needs more improvements in order to be viable in a real deployment.

The results obtained in the evaluation revealed that there are many allocated resources wasted due to not being fully utilized. These results highlight the opportunity for applying an overbooking strategy and this thesis shows that it is possible to push further the allocated resources, achieving a better energy efficiency, using less machines, which itself allows for more energy savings.

6.1 Future Work

Some limitations were already pointed out throughout the chapters but here, they are wrapped up. The main limitation is the increased time to schedule requests, especially for extreme cases. Despite being an unavoidable consequence due to our solution keeping the resources almost fully utilized, different solutions to mitigate this issue have been proposed on Chapter 5 and can be applied depending on the environment the solution is deployed.

Another limitation that was already pointed out, was regarding the rescheduling of tasks for two reasons. The first being that jobs had to be stopped and then started from the beginning in a new host and the second being that they may not be successfully rescheduled. We tried to fix the second problem but it resulted in the problem mentioned on Section 5.2.2. In that Section we also provide suggestions that can be used as future work to fix these issues. With these issues fixed, we could increase the amount of times the algorithm resort to kills, decreasing even further the amount of time hosts spend in the EED region, achieving a better energy efficiency and better QoS results, regarding jobs and services response times.

The Manager is a centralized solution as others orchestrators. For cloud environments with thousands of hosts, a centralized approach would not be viable. As future work, this solution can easily be extended to a distributed solution, by having several managers assigned to a certain amount of hosts. For example, Manager 1 deals with hosts0-host100, Manager2 with hosts 101-200 and so on, with some coordination between them, ensuring that the requests are evenly distributed between them, avoiding under-utilized hosts.

The results presented at Section 5.2 highlighted how an overbooking strategy combined with cuts and kills can improve significantly resource utilization and allocation rates. Thus, it would be interesting to perform studies on how much CSPs would benefit from this approach by having real container traces provided by CSPs. Another study that could be performed is that if clients would be interested in such an approach, where their requests resources are reduced or placed on hosts experiencing overbooking, and how much they would gain from it.

Unfortunately when the evaluations were performed, there were not enough power meters available so we could not measure the energy consumption of the hosts, leaving it for future work, evaluating with the suggestions proposed for future work already implemented. However, based on the work of [63] and since we achieved utilization rates close to the ones mentioned at that work that provide the better energy efficiency, we can assume that the goal was achieved, although it is still an assumption. Assumptions a side, we did achieve better utilization rates with both CPU and memory, a relevant result in itself, having significantly more requests allocated thanks to a high allocation successful rate, thus requiring less machines than the others solutions, which would in turn consume more energy.

Bibliography

[1] C. L. Philip Chen and C. Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data," Information Sciences, vol. 275, pp. 314–347, 2014.

[2] A. Botta, W. De Donato, V. Persico, and A. Pescapé, "Integration of Cloud computing and Internet of Things: A survey," Future Generation Computer Systems, vol. 56, pp. 684–700, 2016.

[3] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, and A. Rabkin, "A view of cloud computing," Communications of the ACM, vol. 53, no. 4, p. 50, 2010.

[4] W. Van Heddeghem, S. Lambert, B. Lannoo, D. Colle, M. Pickavet, and P. Demeester, "Trends in worldwide ICT electricity consumption from 2007 to 2012," Computer Communications, vol. 50, no. 0, pp. 64–76, 2014.

[5] T. Bawden, "Global warming: Data centres to consume three times as much energy in next decade, experts warn," Independent, 2016. [Online]. Available: http://www.independent.co.uk/environment/global-warming-data-centres-to-consume-three-times-as-much-energy-in-next-decade-experts-warn-a6830086.html

[6] B. Whitehead, D. Andrews, A. Shah, and G. Maidment, "Assessing the environmental impact of data centres part 1: Background, energy use and metrics," Building and Environment, vol. 82, no. December 2014, pp. 151–159, 2014.

[7] T. Kaur and I. Chana, "Energy Efficiency Techniques in Cloud Computing: A Survey and Taxonomy," ACM Computing Surveys, vol. 48, no. 2, pp. 1–46, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2830539.2742488

[8] M. Dayarathna, Y. Wen, and R. Fan, "Data Center Energy Consumption Modeling : A Survey," IEEE Communications Surveys & Tutorials, vol. 18, no. September, pp. 1–1, 2015.

[9] J. Simão and L. Veiga, A Taxonomy of Adaptive Resource Management Mechanisms in Virtual Machines: Recent Progress and Challenges. Cham: Springer International Publishing, 2017, pp. 59–98. [Online]. Available: https://doi.org/10.1007/978-3-319-54645-2_3

[10] J. E. Smith and R. Nair, "The architecture of virtual machines," Computer, vol. 38, no. 5, pp. 32–38, 2005.

[11] J. Simão and L. Veiga, "Partial utility-driven scheduling for flexible sla and pricing arbitration in clouds," IEEE Transactions on Cloud Computing, vol. 4, no. 4, pp. 467–480, Oct 2016.

[12] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," ACM SIGOPS Operating Systems Review, vol. 41, no. 3, p. 275, 2007.

[13] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "Efficient Virtual Machine Sizing for Hosting Containers as a Service," Proceedings - 2015 IEEE World Congress on Services, SERVICES 2015, pp. 31–38, 2015.

[14] L. Tomas, C. Klein, J. Tordsson, and F. Hernandez-Rodriguez, "The straw that broke the camel's back: Safe cloud overbooking with application brownout," Proceedings - 2014 International Conference on Cloud and Autonomic Computing, ICCAC 2014, pp. 151–160, 2015.

[15] A. Havet, V. Schiavoni, P. Felber, M. Colmant, R. Rouvoy, and C. Fetzer, "GENPACK: A generational scheduler for cloud data centers," Proceedings - 2017 IEEE International Conference on Cloud Engineering, IC2E 2017, pp. 95–104, 2017.

[16] T. Mitral, "Early Experience With Mesa," no. April, p. 138, 1977.

[17] N. Wirth, "Hochschule Eidgenössische Technische Hochschule Zürich," 1976.

[18] D. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," Information Processing, vol. 71, no. 5, pp. 339–344, 1972.

[19] D. Box, Essential COM. Addison-Wesley, 1997.

[20] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," Computer, vol. 33, no. 3, pp. 78–85, 2000.

[21] F. Plášil, D. Bálek, and R. Janec, "SOFA / DCUP : Architecture for Component Trading and Dynamic Updating Faculty of Mathematics and Physics Department of Software Engineering ˇ ží," Proc. Fourth Int'l Conf. Configurable Distributed Systems (ICCDS '98), pp. 43–52, 1998.

[22] B. I. Page and B. B. Economist, “The Rise and Fall of CORBA,” 21st Century, vol. 12, no. July, pp. 319–350, 2007.

[23] I. Crnkovic, S. Sentilles, a. Vulgarakis, and M. R. V. Chaudron, “A Classification Framework for Software Component Models,” IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 593–615, 2011.

[24] A. L. Tavares and M. T. Valente, “A gentle introduction to OSGi,” ACM SIGSOFT Software Engineering Notes, vol. 33, no. 5, p. 1, 2008.

[25] The FreeBSD Documentation Project, The FreeBSD Handbook, 2016, no. 48818. [Online]. Available: http://ftp.freebsd.org/pub/FreeBSD/doc/en/books/handbook/book.pdf

[26] S. J. Vaughan-Nichols, “New approach to virtualization is a lightweight,” Computer, vol. 39, no. 11, pp. 12–14, 2006.

[27] Introduction to Oracle Solaris Zones, 2015, no. September. [Online]. Available: http://docs.oracle.com/cd/E53394_01/pdf/E54762.pdf

[28] R. Morabito, J. Kjällman, and M. Komu, “Hypervisors vs. lightweight virtualization: A performance comparison,” Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015, no. October, pp. 386–393, 2015.

[29] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center,” Proceedings of the 8th USENIX conference on Networked systems design and implementation, pp. 295–308, 2011.

[30] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant Resource Fairness : Fair Allocation of Multiple Resource Types Maps Reduces,” Ratio, vol. 167, no. 1, p. 24, 2011.

[31] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: Wait-free Coordination for Internet-scale Systems,” USENIX Annual Technical Conference, vol. 8, p. 11–11, 2010.

[32] D. Ongaro and J. Ousterhout, “In Search of an Understandable Consensus Algorithm,” Atc ’14, vol. 22, no. 2, pp. 305–320, 2014.

[33] J. Koomey, “Growth in Data Center Electricity use 2005 to 2010,” Analytics Press., pp. 1–24, 2011.

[34] A. Greenberg, J. Hamilton, D. a. Maltz, and P. Patel, “The Cost of a Cloud : Research Problems in Data Center Networks,” ACM SIGCOMM Computer Communication Review, vol. 39, no. 1, pp. 68–73, 2009.

[35] C. Gu, H. Huang, and X. Jia, "Power metering for virtual machine in cloud computing-challenges and opportunities," IEEE Access, vol. 2, pp. 1106–1116, 2014.

[36] Z. Jiang, C. Lu, Y. Cai, Z. Jiang, and C. Ma, "VPower: Metering power consumption of VM," Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS, no. October 2016, pp. 483–486, 2013.

[37] L. Liu, H. Wang, X. Liu, X. Jin, W. B. He, Q. B. Wang, and Y. Chen, "GreenCloud," Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session - ICAC-INDST '09, no. June, p. 29, 2009.

[38] M. Kurpicz, A. C. Orgerie, and A. Sobe, "How Much Does a VM Cost? Energy-Proportional Accounting in VM-Based Environments," Proceedings - 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, pp. 651–658, 2016.

[39] R. Koller, A. Verma, and A. Neogi, "WattApp : An Application Aware Power Meter for Shared Data Centers," International Conference on Autonomic Computing, p. 10, 2010.

[40] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya, "Virtual Machine Power Metering and Provisioning," 1st ACM Symposium on Cloud Computing (SoCC '10), pp. 39–50, 2010.

[41] B. Krishnan, H. Amur, A. Gavrilovska, and K. Schwan, "VM power metering," ACM SIGMETRICS Performance Evaluation Review, vol. 38, no. 3, p. 56, 2011.

[42] A. K. Sahoo, "Energy Efficient Scheduling Using DVFS Technique in Cloud Datacenters," vol. 4, no. 1, pp. 59–66, 2016.

[43] Z. Tang, L. Qi, Z. Cheng, K. Li, S. U. Khan, and K. Li, "An Energy-Efficient Task Scheduling Algorithm in DVFS-enabled Cloud Environment," Journal of Grid Computing, vol. 14, no. 1, pp. 55–74, 2016.

[44] G. Wang, S. Wang, B. Luo, W. Shi, Y. Zhu, W. Yang, D. Hu, L. Huang, X. Jin, and W. Xu, "Increasing Large-scale Data Center Capacity by Statistical Power Control," Proceedings of the Eleventh European Conference on Computer Systems, pp. 8:1–8:15, 2016.

[45] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory Power Management via Dynamic Voltage/Frequency Scaling," Proceedings of the 8th ACM International Conference on Autonomic Computing, pp. 31–40, 2011.

[46] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "A Survey and Taxonomy of Energy Efficient Resource Management Techniques in Platform as a Service Cloud," IGI Global, pp. 410–454, 2016.

[47] M. H. Kabir, G. C. Shoja, and S. Ganti, "VM Placement Algorithms for Hierarchical Cloud Infrastructure," 2014 IEEE 6th International Conference on Cloud Computing Technology and Science, pp. 656–659, 2014.

[48] A. Beloglazov and R. Buyya, "Energy efficient allocation of virtual machines in cloud data centers," CCGrid 2010 - 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing, pp. 577–578, 2010.

[49] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Resource pool management: Reactive versus proactive or let's be friends," Computer Networks, vol. 53, no. 17, pp. 2905–2922, 2009.

[50] A. Beloglazov and R. Buyya, "Adaptive Threshold-Based Approach for Energy-Efficient Consolidation of Virtual Machines in Cloud Data Centers," Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science, no. December 2010, p. 6, 2011.

[51] I. S. Moreno, R. Yang, J. Xu, and T. Wo, "Improved energy-efficiency in cloud datacenters with interference-aware virtual machine placement," Autonomous Decentralized Systems (ISADS), 2013 IEEE Eleventh International Symposium on, pp. 1–8, 2013.

[52] B. Li, J. Li, J. Huai, T. Wo, Q. Li, and L. Zhong, "EnaCloud: An energy-saving application live placement approach for cloud computing environments," CLOUD 2009 - 2009 IEEE International Conference on Cloud Computing, pp. 17–24, 2009.

[53] V. M. Raj and R. Shriram, "Power aware provisioning in cloud computing environment," 2011 International Conference on Computer, Communication and Electrical Technology (ICCCET), pp. 6–11, 2011.

[54] M. Xu, A. V. Dastjerdi, and R. Buyya, "Energy Efficient Scheduling of Cloud Application Components with Brownout," CoRR, no. August, 2016.

[55] W. Huang, Z. Wang, M. Dong, and Z. Qian, "A Two-Tier Energy-Aware Resource Management for Virtualized Cloud Computing System," Scientific Programming, vol. 2016, 2016.

[56] A. Verma, P. Ahuja, and A. Neogi, "Power-aware dynamic placement of HPC applications," Proceedings of the 22nd annual international conference on Supercomputing ICS 08, no. November, pp. 175–184, 2008.

[57] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "CoScale: Coordinating CPU and memory system DVFS in server systems," Proceedings - 2012 IEEE/ACM 45th International Symposium on Microarchitecture, MICRO 2012, pp. 143–154, 2012.

[58] A. Beloglazov and R. Buyya, “Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in Cloud data centers,” Concurrency Computation Practice and Experience, vol. 24, no. 13, pp. 1397–1420, 2012.

[59] J. Tordsson, L. Tom, L. Tomas, and J. Tordsson, “An Autonomic Approach to Risk-Aware Data Center Overbooking,” IEEE Transactions on Cloud Computing, vol. 2, no. 3, pp. 292–305, 2014.

[60] P. Vojtáš, “Fuzzy logic programming,” Fuzzy Sets and Systems, vol. 124, no. 3, pp. 361–370, 2001.

[61] A. Beloglazov, J. Abawajy, and R. Buyya, “Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing,” Future Generation Computer Systems, vol. 28, no. 5, pp. 755–768, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.future.2011.04.017

[62] A. Shehabi, S. J. Smith, D. A. Sartor, R. E. Brown, M. Herrlin, J. G. Koomey, E. R. Masanet, N. Horner, I. L. Azevedo, and W. Lintner, “United States Data Center Energy Usage Report,” Lawrence . . . , no. June, 2016. [Online]. Available: https://eta.lbl.gov/publications/united-states-data-center-energy

[63] L. Sharifi, N. Rameshan, F. Freitag, and L. Veiga, “Energy efficiency dilemma: P2P-cloud vs. Data-center,” Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom, vol. 2015-Febru, no. February, pp. 611–619, 2015.

[64] D. Knuth, The Art of Computer Programming. Addison-Wesley, 1971.

A Docker Swarm extensions

A.1 Scheduler.go extension

This class has functions that initiate the process of selecting a host to schedule the request to. The **selectNodesForContainer** function initiates this process by applying filters (Section 2.1.3) to narrow down the list of possible of hosts to simplify the algorithm. Since our solution does not use filters, we had to edit this function to skip this process. This illustrated by Fig. A.1. From lines 48-50 is visible our extension to this code. The condition at line 48 exists in order to make sure the other algorithms are not affected by our extension. Only when the strategy selected for Docker Swarm is energy (our solution), then the filters check are skipped (performed at line 52).

At the parameters list for this function we also included nodesMap, an important optimization for our algorithm which does not affect the other scheduling algorithms since they do not use it, nodesMap will be explained next.

```
46  func (s *Scheduler) selectNodesForContainer(nodes []*node.Node, nodesMap map[string]*node.Node, config *cluster.ContainerConfig, soft bool) ([]*
47
48          if s.Strategy() == "energy" {
49                  return s.strategy.RankAndSort(config, nodes, nodesMap)
50          }
51
52          accepted, err := filter.ApplyFilters(s.filters, config, nodes, soft)
53
54          if err != nil {
55                  return nil, err,"0", "", 0.0
56          }
57
58          if len(accepted) == 0 {
59                  return nil, errNoNodeAvailable, "0", "", 0.0
60          }
61
```

Figure A.1: Scheduler.go extension

A.2 Cluster.go extension

After a host has been selected for the request, the code flow returns back to cluster.go, who is responsible for creating the container at the selected host. This had to be extended this code for two different reasons. The first change is because the request to be scheduled can have suffered a cut,

therefore we must edit its CPU shares and memory before creating the container, illustrated by Fig. A.2 between lines 256-263. Since a request has been successfully scheduled to a host, as seen on Section 3.2.2, the Task Registry must be updated. This is done at line 279 which calls a function we created who is responsible for making all the preparations to send the whole task information to the the Task Registry. This code snippet belongs to function **createContainer**.

We had to make an additional extension at this class, this time at the **listNodes** function. This is due to a limitation in our approach, which limits our scheduling performance slightly. When making a scheduling decision, the default Docker Swarm algorithms have access to the hosts directly through internal mechanics and make decisions based upon it. For portability reasons, we access the hosts not through internal mechanics, but through the Host Registry. However, the Host Registry does not contain all the host information. When host is selected for allocating the request, we need to map it with the one Docker Swarm has in its internal mechanics. For this reason, we created nodesMap which contains as key, the host IP and as value, the host. When a scheduling decision is done, based on the IP of the host selected, we use nodesMap to retrieve the host and return it, in order for Docker Swarm to continue the scheduling procedure.

We don't use the Docker Swarm internal mechanics for host information as the other scheduling algorithms because the hosts need to be sorted based on their total resources utilization. Changing the internal mechanics to allow this sorting was possible, but having this computation and all the others Host Registry performs at the Scheduler would stress it too much.

Our extension to listNodes can be seen at Fig. A.3, where at line 904 nodesMap is declared having a string as key (the host IP) (line 914) and as a value the host, which at Docker Swarm is represented by *node.Node.

A.3 Engine.go extension

This class has functions responsible for managing the whole system, such as creating and updating containers. We leverage this class function **refreshContainer**, which is responsible for refreshing the state of a container, to detect when a task has finished and send a message to Task Registry, informing it (Section 3.2.2).

Our extension is pictured at Fig. A.3. Line 789 checks if the container has terminated (represented by the state being equal to exited). The first condition, *exists*, is there to ensure that we do not send the information to Task Registry twice. For some reason we could not understand, sometimes the exited status would occur more than one time for the same container, therefore creating inconsistencies on Task Registry if this check is not performed.

```
256         if strategy == "energy" {
257                 //if this condition is true then we must apply a cut to the request in order to fit it
258                 if cut != 0.0 {
259                                 config.HostConfig.CPUShares = int64(float64(config.HostConfig.CPUShares) * cut)
260                                 config.HostConfig.Memory = int64(float64(config.HostConfig.Memory) *  cut)
261                                 cutReceived = 1 - cut
262                 }
263         }
264         c.scheduler.Unlock()
265
266         container, err := engine.CreateContainer(config, name, true, authConfig)
267
268         if err != nil {
269                 log.WithFields(log.Fields{"NodeName": n.Name, "NodeID": n.ID}).WithError(err).Error("Failed to create container")
270         } else {
271                 containerFlag := name
272                 if containerFlag == "" {
273                         containerFlag = stringid.TruncateID(container.ID)
274                 }
275                 log.WithFields(log.Fields{"NodeName": n.Name, "NodeID": n.ID}).Debugf("Scheduling container %s to ", containerFlag)
276         }
277
278         if strategy == "energy" && err == nil{
279                 go SendInfoTask(container.ID, requestClass, config.HostConfig.CPUShares, config.Image, config.HostConfig.Memory, requestType, cu
```

Figure A.2: Cluster.go extension 1

At lines 792 and 793 the IP of the host this container was running is retrieved. The remaining of the code is to create the code to access, through a GET request, the endpoint at Task Registry, responsible for dealing with terminated tasks (Section 3.2.2).

```
898  // listNodes returns all validated engines in the cluster, excluding pendingEngines.
899  func (c *Cluster) listNodes() ([]*node.Node, map[string]*node.Node) {
900          c.RLock()
901          defer c.RUnlock()
902
903          //for faster lookup when making a scheduling decision
904          var nodesMap map[string]*node.Node = make(map[string]*node.Node)
905
906          out := make([]*node.Node, 0, len(c.engines))
907          for _, e := range c.engines {
908                  node := node.NewNode(e)
909                  for _, pc := range c.pendingContainers {
910                          if pc.Engine.ID == e.ID && node.Container(pc.Config.SwarmID()) == nil {
911                                  node.AddContainer(pc.ToContainer())
912                          }
913                  }
914                  nodesMap[node.IP] = node
915                  out = append(out, node)
916          }
917
918          return out, nodesMap
919  }
```

Figure A.3: Cluster.go extension 2

```
787          //if these conditions verify then the container has finished and we must alert the task registry
788          //because it cannot be performed here.
789          if(!exists && containers[0].State == "exited") {
790                  lastIDSent[ID] = ID
791
792                  hostInfo := strings.Split(e.IP,":")
793                  hostIP := hostInfo[0]
794
795                  //to task registry to be removed
796                  req, err := http.NewRequest("GET", "http://"+hostIP+":1234/task/remove/"+ID, nil)
797                  req.Header.Set("X-Custom-Header", "myvalue")
798                  req.Header.Set("Content-Type", "application/json")
799
800                  client := &http.Client{}
801                  resp, err := client.Do(req)
802                  if err != nil {
803                  panic(err)
804                  }
805                  defer resp.Body.Close()
806          }
807
```

Figure A.4: Engine.go extension