Energy4Cloud

Sérgio da Silva Mendes
sergiosmendes@tecnico.ulisboa.pt

Instituto Superior Técnico, INESC-ID,
Av. Prof. Doutor Anbal Cavaco Silva - 2744-016 Porto Salvo, Portugal

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 work we make a thorough analysis 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. We finish by proposing the architecture, the algorithms and how we are going to evaluate our energy-efficient scheduling of Docker containers in Cloud environments, with SLA concerns.

Keywords: Energy efficiency, scheduling, Cloud, containers, Docker,

Table of Contents

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 consume more and more hardware resources (e.g. Big Data applications [1]) and with the emergence of IoT (Internet of Things) [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 of servers. This solution is appealing for businesses and people, who, instead of buying the hardware infrastructure, can rent it and continuously adapt to their needs. 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 CSPs (Cloud Service Providers) [4]. Besides the costs for CSPs, this amount of energy required has significant environmental consequences [5]. These issues bring the urgent need for energy-aware policies for cloud environments [6]. 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 [7]. Cooling costs are highly correlated with server energy costs. If we can reduce the energy consumption or maximize the energy efficiency of a server, 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 VMs (Virtual Machines) [8], has been extensively used to enhance resource utilization. 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.

1.2 Challenges

However, an alternative to VMs to virtualize resources has been proposed, containers [9]. 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 OS (Operating System) 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.

The state-of-the-art regarding energy-aware strategies for cloud environments mostly focus on using VMs and not containers. On our research, we only managed to find one work that takes energy and containers into consideration [10]. Their work has some limitations due to the of usage computationally intensive computations which can be an overkill on real cloud environments. 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 consumption, provides a good opportunity to contribute to the literature with a solution that provides energy-aware scheduling for containers on cloud environments. Thus, our objective is to propose a scheduling algorithm that promotes energy efficiency in the context of cloud environments, managed by Docker containers, based on resource utilization monitorization and levels of energy efficiency, without violating SLAs (Service Level Agreement). We are going to develop a prototype of the solution in order to evaluate it in a realistic environment. This evaluation will be performed according to a set of relevant metrics drawn from related work, comparing with relevant related systems.

1.4 Document roadmap

This work is organized as follows: Our research on the related work about our proposal is described thoroughly on section 2. Section 3 presents our proposed solution to accomplish the objectives proposed. Section 4 presents how we intend to evaluate our proposed solution and Section 5 concludes. Our work scheduling is presented on the Appendix Section.

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[11] and Modula [12]. 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 [13]. 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 components 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 API, therefore allowing different components to interact with each other.

CORBA (Common Object Request Broker Architecture)[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 **DCOM** (Distributed Component Object Model), a distributed version of COM (Component Object Model)[14]. 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 [15] and SOFA [16], had significant limitations mainly due to their implementation complexity and scalability problems [17]. Despite these partially failed approaches, they inspired other component models to be implemented [18]. The one presented next, OSGi, is the most successful to date.

OSGi[2] (Open Services Gateway Initiative) 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 [18].

[1] http://www.corba.org/
[2] https://www.osgi.org/

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 [19], 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 (Virtual Machine) and can share code among each other.

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 BSD (Berkeley Software Distributed) systems and later to the other Unix-based operating systems. 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 [20]. The virtualization achieved by FreeBSD was very promising and new technologies quickly started to appear, e.g. Solaris zones [21].

The concept of *zones* was introduced by Solaris in 2004 [22]. 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 **algorithms**. 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 **orchestration** (also known as **scheduling**) **algorithms** that allows application containers to be efficiently deployed.

OS containers: OS containers (also known as system containers) are very similar, in principal, to VMs (Virtual Machines). 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 [23]. 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 LXC (Linux Containers, OpenVZ[4] and Linux VServer[5].

[3] https://www.osgi.org/developer/benefits-of-using-osgi/
[4] https://github.com/OpenVZ
[5] http://www.linux-vserver.org/

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: **ACI** (App Container Image), **ACR** (App Container Runtime) and **ACD** (App Container Discovery). 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. 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 IPC (Inter Process Communication), they can use the same 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.

[6] https://linuxcontainers.org/
[7] https://coreos.com/rkt/docs/latest/app-container.html
[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

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.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.

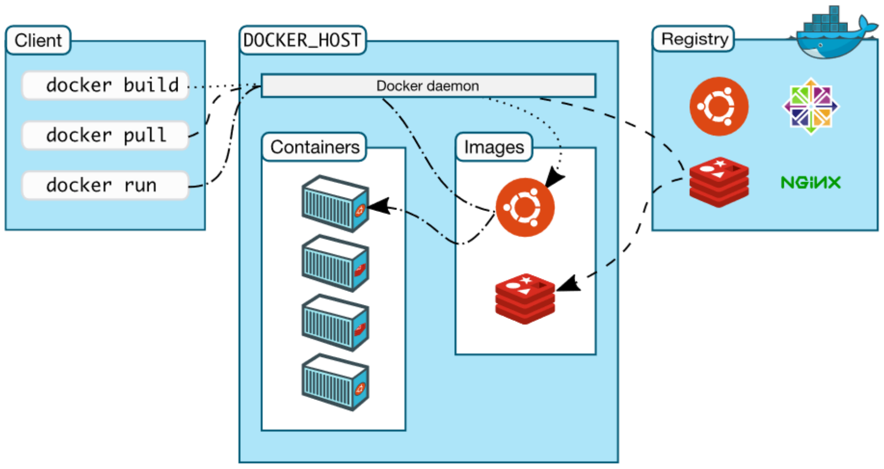

Fig. 1. Docker Architecture[11]

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.

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

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

Docker as could be seen above by the number of the different features Docker has in comparison with 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 Algorithms 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 algorithms 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 [24] 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. presents Mesos architecture.

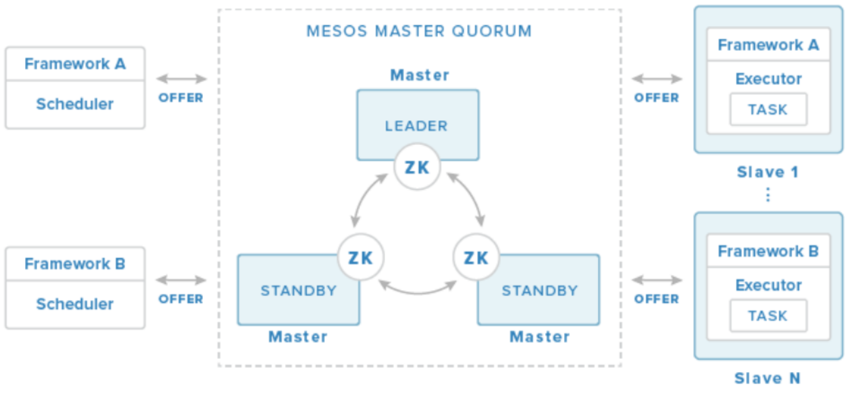

Fig. 2. Mesos Architecture[16]

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 [25]. 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 [26] 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).

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

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 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;
- 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. 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 [27] for fault-tolerance.

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

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., Etcd. They also support DNS for service discovery.

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: In table 1, we present 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 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 [28]. Servers have been identified has the main source of power consumption in data centers [29] 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.

[23] https://github.com/docker/libkv
[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 1. Orchestrators summary

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 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 [30]. 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 [31] 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 [30]:

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 PDUs (Power Distribution Units), but there are others approaches [32]. 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. PMCs

(Performance Monitor Counters)[26] are 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 [33], memory and CPU [34], and more [7].

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 [35], 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 [36], other techniques only use PMCs and many more different techniques as can be seen here [30].

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: Dynamic Voltage and Frequency Scaling 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 [37–39]. 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 [40]. Being

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

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 [41]. 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 [41] 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 PM (Physical Machine) 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 [42]. 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 [41], 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 [43, 44] or dynamic thresholds [45]. 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 underloaded, 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 [41], 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* [46]. 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 very risky strategy [47]. 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 [48] 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 [49] 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**. 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 [10]. 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 [50].

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 [35] 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 LLC (Last Level Cache) 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 [34] 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 **OQI** (Oracle Query Interface) 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, 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 [51]. 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* [52] 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 MC (Memory Controller) and DIMM (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 [53] 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 MIPS (Millions Instructions Per Second), 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 NAS (Network Attached Storage), 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 thre 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 **MMT** (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 **MC** (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 [54] 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 **AC** (Admission Control) 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 **KOB** (Knowledge DB) and is passed to the **RA** (Risk Assessment) module that, based on fuzzy logic programming [55], 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 **SOS** (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 [56], 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 [57].

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 IaaS (Infrastructure as a Service) and PaaS (Platform as a Service). 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 [10] propose finding efficient VM sizes for hosting containers in such way that the workload is executed with minimum wastage of energy. 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 [58]. 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.

Analysis: To better understand the differences and similarities between the different works we studied, we present table 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 [40]. 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 [52]	DVFS	CPU and memory	Performance	- Does not work on overload environments - Continuous estimations
Dynamic consolidation algorithm [53]	Consolidation	CPU, memory and network bandwidth	Possible overloads and Performance	- Does not work on overload environments
Autonomic risk-aware overbooking [54]	Overbooking	CPU, memory and I/O	Performance	- Simulation and emulations required
Energy efficient brownout enabled algorithm [56]	VM Placement and Brownout	CPU	Performance	- Limited functionality
VM Sizing for hosting CaaS [10]	VM Sizing	CPU, memory and I/O	Performance	- Heavy calculations

Table 2. Relevant related works summary

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, 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 request's fixed sizes imposed by CSPs are a significant source of energy inefficiency, therefore creating an opportunity for increasing the energy efficiency.

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 a high level our system consists of two components, like Docker Swarm, a manager and hosts. The process starts by a client submitting a request, indicating the **number of CPUs**, the **amount of memory required** for that request and the **request type**, if it is 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. data analytics application). Afterwards the client must negotiate a SLA with the CSP. Besides the usual SLA clauses, we introduce a new one, which will be important for our algorithm in order to avoid SLA violations. The user must select the maximum overbooking he is willing to tolerate. We provide four classes for the client to choose:

- Class 1: 100% overbooking (i.e. the client does not tolerate overbooking);
- Class 2: 120% overbooking;
- Class 3: 150% overbooking;
- Class 4: 200% overbooking.

After the client chooses the request resource requirements and its class, the Manager receives this information and according to it, among all hosts, it selects the one which provides the better energy efficiency for this request, allocating the request to it.

3.2 Architecture

Now we are going to describe in more detail, the components inside the Manager and the hosts, and how they interact with each other. The architecture is depicted on Fig.3. When a request arrives, it goes directly to the **Scheduler** and, if possible, it will schedule that request into one of the N hosts. To make decisions, besides the information regarding the request, the Scheduler requires

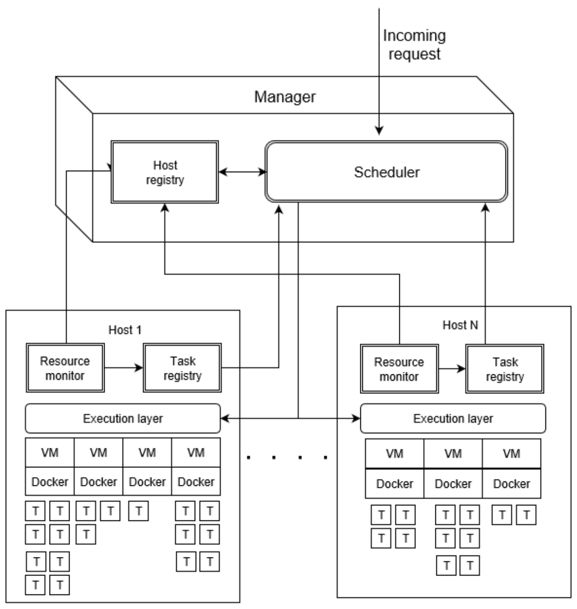

Fig. 3. System architecture

additional information about the hosts. This is provided by the **Host Registry** and the **Task Registry**. The Host Registry contains general information about the hosts (e.g. total resources utilization of each host) while the Task Registry contains more specific information about each host (e.g. current tasks being served by the host). 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, sometimes 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.

In order to make the best scheduling decisions, the Host Registry and the Task Registry must be constantly updated about resources utilization. For this purpose, the **Resource Monitor** is responsible for measuring resource utilization on each host and sending updated information to the Host Registry and Task Registry. A potential risk of using an approach like overbooking is the vulnerability to overloads as was explained on Section 2.2.2. The Resource Monitor is also responsible to detect overloads, killing tasks to leave the overload situation and sending the killed tasks to the Scheduler to be rescheduled.

Once a host is selected to schedule the request, the **Execution Layer** will allocate the request in the respective Docker host. If the request is not compatible with any of the current Docker hosts (e.g. different Linux version), a new Docker host must be launched for that request.

3.3 Data structures

We are going to have three regions which map resource utilization (CPU and Memory) with energy efficiency based on the study performed by [59]. **LEE** (Low Energy Efficiency) refers to the region that has the lowest energy efficiency due to under-utilized resources (0-50% total resources utilization). We want to transit hosts on that region to the **DEE** (Desired Energy Efficiency) region as quickly as possible, where we have optimal energy efficiency (50-85% total resources utilization). Our goal is to keep them at region DEE because heavily used resources have negative impact on the energy efficiency, region **EED** (Energy Efficiency Degradation) (>85% total resources utilization).

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.4. 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. These lists introduce an increased space utilization but they simplify the algorithm significantly, compensating for this increased space utilization.

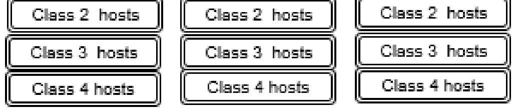

Fig. 4. Host Registry data structures

Each host will have the following information associated with it: **host ID**, **% of total resources utilization**, **% of CPU utilization, % of memory utilization, host class, amount of allocated resources, total host resources, overbooking factor**. The total resources utilization is represented as max{% of CPU utilization, % of memory utilization} and the overbooking factor is the max{% of CPU utilization/100, % of memory utilization/100}. We use the max because it is what is restraining more the tasks on the host. 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.

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 we want to bring them up to the the DEE region as quickly as possible. Therefore the scheduling algorithm will try and schedule the requests on the first elements of the list since they are closest to the DEE region. Since the DEE region is the one that we want hosts to be, we order it 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 cuts and kills (explained later), since the hosts on that region are already experiencing high resource utilization, we don't want them to be receiving more requests which would only aggravate their energy inefficiency. What we want is to bring them down to the DEE region as quick as we can, therefore we order it by ascending order so that the first on the list is the closest to the DEE region.

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. Each task will have the following information associated with it: **Task ID, allocated resources, task type, cut received.** These lists will be ordered by ascending order of their allocated resources, which is useful for cut and kill decisions as will be seen next.

3.4 Algorithms

We will have three core algorithms. The first, tries to schedule the request, taking some restrictions into consideration as we shall see next. However, if we cannot fit the request with the first algorithm, we have 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 with Algorithm 1.

Algorithm 1 Scheduler 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:          return
10:     listHostsEED_DEE = getHostsListsEED_DEE()
11:     if kill(listHostsEED_DEE, request) then
12:         return
13:     warnClient()
```

The goal of Algorithm 1 is, first, to try and schedule the request either in the LEE or in the DEE region without cutting or killing tasks. We start by getting the hosts that are in the LEE region, then we append to that list the hosts on the DEE region (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. We only get the lists of hosts that respect this condition: $request.CLASS \leq host\ Class$ 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. We then try to schedule. **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. $overbooking\ factor < host\ class$).

If we can't schedule the request 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, because by cutting a task and putting a request there, it would increase the overbooking on that host, worsening the energy inefficiency that is already felt by hosts on that region. On line 7 (still on Algorithm 1) we fetch the lists of hosts again, because this time, we will aggregate them differently than what we did on line 2. Here the lists are fetched respecting this condition: $request.CLASS \geq host\ Class$. 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), even though that at those hosts there could be tasks greater or equal than the request, we believe it is not worst the cost of searching all these hosts. Like line 2, we will also aggregate by classes, following an ascending order. Again, we do this 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 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 we get the list of hosts on regions EED and we append the hosts on region DEE. We give priority 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, we will consider all the hosts on that region no matter their class. We will also aggregate them differently than we did 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. We do it this to avoid the problem that was mentioned at the end of the last paragraph.

Before explaining Algorithm 2, we are going to explain what we mean by cutting a task. The cut means that we are decreasing the resources attributed 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 we still have space for a class 4 request, 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, 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 20%. However cuts for class 3 and 4 can be performed differently as will be explained next.

For cutting, some restrictions apply. First, we have two choices, either we cut the request being scheduled or we 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. 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 20%. 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 50%, 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 requests class 3 and 4 will be depending on the host they are at. For example, if a task class 3 is running on class 2 host, the cut would be of 30% since the remaining 20% would be inflicted indirectly by the overbooking tolerated by that host.

As was just explained on the previous paragraph, we can only cut the request if the host class is lower than the request class. For the same reason, we can only cut tasks on a host if their classes are lower than the host class. Therefore, if the host class is greater or equal than the request, we can only cut tasks whose class is higher than the request (lines 5 and 6), otherwise, we are safe to cut classes equal or higher than the incoming request, if its not a class 1 request, because we cannot cut class 1 requests (lines 11-13). As was also mentioned earlier, we give priority to cutting the request (if it is not class 1). Line 7 checks if the request fits by cutting it. If it does not fit, we add it to the **cutLIST** and search for more tasks to cut on this host so that this request can fit.

When we get the list of tasks from the Task Registry (lines 6 and 12), we append the lower classes to the higher so that we cut from the higher classes first. Since the list is ordered by resource utilization (done by the Task Registry) and by class, the first of the list is the best candidate for

Algorithm 2 Cut algorithm

```
1: function CUT($listHostsLEE\_DEE, request$)
2:     for $listHostsLEE\_DEE$ as $selectedHost$ do
3:         $cutLIST = null$
4:         $listTasks = null$
5:         if $selectedHost.hostClass >= request.CLASS$ then
6:             $listTasks = selectedHost.getListTasksHigherThanRequestClass()$
7:         else if $request.CLASS$ != 1 and $afterCutRequestFits(selectedHost, request)$ then
8:             $newRequest = cutRequest(request)$
9:             $allocateRequest(selectedHost, newRequest)$
10:            return $true$
11:        else if $request.CLASS$ != 1 then
12:            $listTasks = getListTasksEqualHigherThanRequestClass()$
13:            $cutLIST$ += $request$
14:        for $listTasks$ as $task$ do
15:            if $task.CLASS == 1$ then
16:                break
17:            $cutLIST$ += $task$
18:            if $fitsAfterCUT(request, cutLIST)$ then
19:                $cutRequests(request, cutLIST)$
20:                $allocateRequest(selectedHost, request)$
21:                return $true$
22:        end for
23:    end for
24:    return $false$
```

a cut. Line 18 checks if the request fits taking into consideration the overbooking restrictions described earlier. If one reaches line 24, 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).

Finally, regarding Algorithm 3, we only allow tasks to be killed if their class is higher than the request, because it 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 to not 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 4 and 5). However if it is a class 4 and a service, it is not worth to kill a service for another (lines 6 and 7). If it is not a class 4 (lines 8 and 9) we get the tasks that are higher than the request and aggregate the same way we did on Algorithm 2. The reasoning beyond the remainder of the algorithm is similar to Algorithm 2.

4 Evaluation methodology

In this section we present the metrics and the benchmarks that we are going to use to evaluate our work. The metrics are the following:

- **Energy efficiency:** Our most important metric and what we aim to optimize;
- **Scheduling makespan:** Introducing energy-awareness to the scheduling algorithm can degrade the time it takes to schedule requests. We want this penalty to be as lower as possible;
- **Task makespan:** Due to overbooking and cuts, tasks makespan is going to be higher, however it must conform to its class constraints;
- **Hosts in each region:** The core to achieve energy efficiency are the three regions presented on Section 3. To understand how the algorithm is behaving, we need to to understand how are the hosts transitioning between these three regions;
- **Allocated resources and resources utilization:** These two are very important and correlated in order to understand the potential of overbooking;
- **Overbooking factor:** This is an important metric because it is what guarantees that SLAs are not violated;
- **Overloads duration:** An inherent risk of using overbooking is that overloads might happen. The important is to react quickly and deal with that overload;

Algorithm 3 Kill algorithm

```
function KILL($listHostsDEE, request$)
    for $listHostsEED\_DEE$ as $selectedHost$ do
        $possibleKillLIST = null$
        if $request.CLASS == 4$ and $request.TYPE == JOB$ then
            $possibleKillLIST = selectedHost.getListTasksClass4()$
        else if $request.CLASS == 4$ then
            return $false$
        else
            $possibleKillLIST = selectedHost.getListTasksHigherThanRequestCLASS()$
        $killLIST = null$
        for $possibleKillLIST$ as $task$ do
            if $request.CLASS == 4$ and $request.TYPE == SERVICE$ then
                $killLIST$ += $task$
            else if $request.CLASS$ != $4$ then
                $killLIST$ += $task$
            if $requestFits(request, killLIST)$ then
                $kill(killLIST)$
                $reschedule(killLIST)$
                $allocateRequest(selectedHost, request)$
                return $true$
        end for
    end for
    return $false$
```

- **Number of cuts and kills:** Due to the restrictions applied by using task classes, we need to understand how much the algorithm is resorting to cuts and kills in order to schedule requests;
- **Space overhead:** Space overhead introduced by the data structures;
- **Ordering overhead:** Background cost of maintaining the data structures ordered;
- **Successful/Failed allocations:** We want to compare the the amount of successful and failed allocations to understand how restrictive is our algorithm.

Although not a metric *per se*, it is also important to understand if the algorithm is **scalable** since the target of our algorithm are cloud environments, which can have tens of thousands of machines.

We are going to compare our solution with the three Docker Swarm scheduling strategies presented on Section 2.1.2, using the metrics we just presented, besides the overheads related with the data structures we introduce, the overbooking factor and number of cuts and kills, because Docker Swarm does not use overbooking and it also does not cut and kill tasks to schedule a request. For this purpose, we are going to use real-world workload traces from workloads executed during 10 days by thousands of PlanetLab VMs provisioned for multiple users [53, 60], in order to benchmark our work in a realistic way.

5 Conclusion

Despite all the effort done by academia, the problem of energy consumption in data centers still 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 and we finish this work by proposing our solution and how we intend to evaluate it.

References

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. 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.
6. 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
7. 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.
8. J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005.
9. 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.
10. 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.
11. T. Mitral, "Early Experience With Mesa," no. April, p. 138, 1977.
12. N. Wirth, "Hochschule Eidgenössische Technische Hochschule Zürich," 1976.
13. D. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Information Processing*, vol. 71, no. 5, pp. 339–344, 1972.
14. D. Box, *Essential COM.* Addison-Wesley, 1997.
15. 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.
16. 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 ží," *Proc. Fourth Intl Conf. Configurable Distributed Systems (ICCDS 98)*, pp. 43–52, 1998.
17. B. I. Page and B. B. Economist, "The Rise and Fall of CORBA," *21st Century*, vol. 12, no. July, pp. 319–350, 2007.
18. 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.
19. A. L. Tavares and M. T. Valente, "A gentle introduction to OSGi," *ACM SIGSOFT Software Engineering Notes*, vol. 33, no. 5, p. 1, 2008.
20. The FreeBSD Documentation Project, *The FreeBSD Handbook*, 2016, no. 48818. [Online]. Available: http://ftp.freebsd.org/pub/FreeBSD/doc/en/books/handbook/book.pdf
21. S. J. Vaughan-Nichols, "New approach to virtualization is a lightweight," *Computer*, vol. 39, no. 11, pp. 12–14, 2006.
22. *Introduction to Oracle Solaris Zones*, 2015, no. September. [Online]. Available: http://docs.oracle.com/cd/E53394_01/pdf/E54762.pdf
23. 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.
24. 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.
25. 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.
26. 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. 1111, 2010.

27. D. Ongaro and J. Ousterhout, “In Search of an Understandable Consensus Algorithm,” *Atc ’14*, vol. 22, no. 2, pp. 305–320, 2014.
28. J. Koomey, “Growth in Data Center Electricity use 2005 to 2010,” *Analytics Press.*, pp. 1–24, 2011.
29. 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.
30. 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.
31. 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.
32. 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.
33. 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.
34. 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.
35. 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.
36. B. Krishnan, H. Amur, A. Gavrilovska, and K. Schwan, “VM power metering,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 3, p. 56, 2011.
37. A. K. Sahoo, “Energy Efficient Scheduling Using DVFS Technique in Cloud Datacenters,” vol. 4, no. 1, pp. 59–66, 2016.
38. 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.
39. 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.
40. 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.
41. 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.
42. 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.
43. 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.
44. 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.
45. 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.
46. 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.
47. 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.
48. 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.
49. 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.
50. 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.

51. 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.
52. 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.
53. 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.
54. 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.
55. P. Vojtáš, “Fuzzy logic programming,” *Fuzzy Sets and Systems*, vol. 124, no. 3, pp. 361–370, 2001.
56. M. Xu, A. V. Dastjerdi, and R. Buyya, “Energy Efficient Scheduling of Cloud Application Components with Brownout,” *CoRR*, no. August, 2016.
57. 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
58. D. W. M. Pelleg, “X-means: Extending k-means with efcient estimation of the number of cluster,” *Seventeenth International Conference on Machine Learning*, 2000.
59. L. Sharifi, N. Rameshan, F. Freitag, and L. Veiga, “Energy efficiency dilemma: P2P-cloud vs. Datacenter,” *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, vol. 2015-Febru, no. February, pp. 611–619, 2015.
60. K. Park and V. S. Pai, “CoMon: a mostly-scalable monitoring system for PlanetLab,” *ACM SIGOPS Operating Systems Review*, vol. 40, no. 1, p. 6574, 2006.

A Appendix - Work scheduling

Tasks	**Details**	**Duration**
Preparation	- Introduction to GO programming language - Setup development environment - Docker and Docker Swarm code study and tutorials	January (1 week) February (3 weeks)
Implementation	- Setup and implement components of the architecture - Data structures implementation - Implement algorithms	February (1 week) March (4 weeks) April (4 weeks)
Implementation conclusion	- Conclude any unimplemented functionality - Bugs detection and fix	May (4 weeks)
Performance measurements	- Setting up test environments - Collection and preparation of the traces to benchmark the solution - Evaluate implemented solution	June (4 weeks) July (2 weeks)
Thesis final report writting	- Write thesis report	July (2 weeks) August (4 weeks)
Review and submission	- Report review and submission	September (2 weeks)
Documentation	- Document design choices, code and tests	January - September
Bi-weekly meetings	- Analyze work progression	January - September

Table 3. Work scheduling