Energy4Cloud

Sergio Mendes
INESC-ID
Instituto Superior Tcnico

***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, we propose and implement an extension to Dockers orchestrator, Docker Swarm, with an energy-efficient scheduling algorithm, based on maximizing resources utilization to levels where the energy efficiency is maximized. our solution improved CPU utilization by 5.6 *p.p* and 8.2 *p.p* over Spread and Binpack (Docker Swarm scheduling strategies) respectively, and improved memory utilization by 15.8 *p.p* and 18.9 *p.p* over the same strategies, during an one hour evaluation. Despite the comparatively longer scheduling times w.r.t other approaches, this is largely compensated due to the fact that our solution manages to allocate more requests, having obtained a successful allocation rate of 83.7% against 57.7% and 56.5% of Spread and Binpack respectively.**

I. Introduction

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

A solution to the insufficient hardware resources was the adoption of Cloud [3], which led to the creation of massive data centers with tens of thousands or even more, servers. However, as mentioned in the previous paragraph, due to emerging trends, more hardware resources are going to be required, consequentially increasing the size of data centers.

Besides operations costs, this increase will also reflect on the energy consumed by these massive infrastructures, which already consume a significant amount of energy, incurring high costs for Cloud Service Providers (CSPs) [4]. This recent report [5] shows that data centers consumed almost 416.2 terawatt hours of energy in 2015. These issues bring the urgent need for energy-aware policies for cloud environments [6].

On traditional cloud environments (e.g. data centers), virtualization using Virtual Machines (VMs) [7], 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 [8].

However, containers [9] have been proposed as an alternative to VMs to virtualize resources. Containers are more lightweight than VMs, containing only the required application binaries to run a specific process and nothing more, not requiring a full guest Operating System (OS) instance. Since they are significantly more lightweight than VMs, a better resource utilization can be achieved using containers. Achieving an even better resource utilization than VMs and considering that VM energy-aware strategies already provide a significant reduction on energy costs, containers are an excellent opportunity to further increase this reduction.

Having energy concerns also has drawbacks that need to be considered. There is always, at least, one tradeoff that is inevitable. It can either be, longer time to schedule requests or reduced Quality of Service (QoS), regarding response times of the workloads that were scheduled. These tradeoffs have to be mitigated.

Despite already existing many solutions for all these different challenges, the state-of-the-art regarding energy-aware strategies for cloud environments mostly focus on using VMs and not containers. In our research, we only managed to find two works that takes both energy and containers into consideration [10], [11]. The first has some limitations due to the of usage computationally intensive computations (through the use of X-means) which can be an overkill on real cloud environments. The second work approach can lead to hosts not serving any requests due to using a static amount of hosts for profiling and others for long duration requests. Therefore, if there are no requests to be profiled or there are only short duration requests, those hosts will not be used, wasting energy. Energy is also not considered on the current platforms for managing containers (e.g. Docker Swarm, Kubernetes). Their scheduling decisions, are not energy-aware.

The lack of state-of-the-art approaches to schedule containers, taking into consideration such an important issue as is energy, provides a good opportunity to contribute to the literature with a solution that provides energy-aware scheduling for containers on cloud environments. Thus, we propose a scheduling algorithm that promotes energy efficiency in the context of cloud environments, managed by Docker containers, based on maximizing resource utilization according to levels of energy efficiency, without violating Service Level Agreements (SLAs). We have developed a prototype of the solution in order to evaluate it in a realistic environment. This evaluation was performed according to a set of relevant metrics drawn from related work such as CPU and Memory utilization over time, comparing with relevant

related systems.

This paper is organized as follows: Our research on the related work about our proposal is described on Section 2. Section 3 presents our proposed solution to accomplish the objectives pro- posed on this Section. Section 4 details the algorithms that support the proposed solutions. Section 5 presents how we evaluated our proposed solution and Section 6 concludes.

II. Related Work

As was mentioned on the previous section, our goal is to optimize energy efficiency where containers are concerned. In order to do so, some decisions have to be made, such as which scheduling strategy to use. These decisions were made based on our analysis of the related work.

The first important decision was deciding which container technology to use. The two most mature open-source solutions are **Docker**[1] and **Rocket**[2] (or rkt). Due to being daemon-less and not executing as root (as opposed to Docker which the daemon runs as root), Rocket provides more security guarantees than Docker. It is also simpler than Docker, since Docker provides significantly more different features such as Docker Compose[3], in comparison with Rocket. However this simplicity is also one of Rocket disadvantages, since these extra features Docker provides can be useful in different scenarios. Also, Rocket is still in the process of maturation while Docker is already a stable solution, already being deployed on production environments. For being more mature, we chose Docker as the container technology.

To schedule containers on cloud environments, there are three major orchestrator platforms: **Mesos** [12], **Kubernetes**[4] and **Docker Swarm**[5]. 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 provides a good reliability. Finally Docker Swarm uses the standard Docker API which simplifies the learning curve. None of the three solutions is significantly better than the others, in fact, they only differ on small aspects as could be seen by this brief analysis. We chose Docker Swarm because it has the closest architecture to the one we propose on the next section.

The last step is choosing a strategy for scheduling in an energy efficient manner. As was already mentioned on the previous section, we only managed to find two works that schedule containers in an energy efficient manner. However, VM strategies for scheduling VMs in an energy-efficient way can be leveraged for containers since both VMs and containers serve similar purposes. A panoply of strategies exist [8] but the most significant strategies are **VM Placement** [13], **Consolidation** [14], **Overbooking** [15], **Brownout** [16] and **VM Sizing** [10]. 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 [17]. Of these strategies we opted for an overbooking strategy, which consists on allocating more resources beyond the hosts nominal capacity. The amount of resources that are wasted due to fixed size requests imposed by CSPs are a significant source of energy inefficiency, therefore creating an opportunity for increasing the energy efficiency by maximizing resource utilization. A report made on the USA [18] shows that approximately 30% of the servers on a data center are either idle or under-utilized, highlighting even further how an overbooking approach can be important to solve this important problem by being able to allocate beyond the machine nominal capacity.

The other two works that perform energy-efficient scheduling with containers use different approaches. The authors in [10] use a **VM Sizing approach**. They propose finding efficient VM sizes for hosting containers in such way that the workload is executed with minimum wastage of resources. The challenge is therefore finding an optimal size such that applications have enough resources to be executed.

GenPack [11] uses a VM Consolidation and VM Placement strategy. The authors propose a framework to schedule containers extending Docker Swarm. The general idea is to have the hosts divided into three distinct groups, which they refer to as generations. The containers will run on each generation depending on their resource profile. Containers that have not been profiled before, i.e. whose workload is unknown, are placed into the nursery generation for profiling resources requirements and energy consumption. The hosts in this generation are static, that is, regardless if they are servicing containers or not, the hosts are always running. Once those containers are fully profiled, the containers are migrated to a host on the young generation based on resource matching (if the host as enough resources host the container and it also considers the properties and requirements of the containers,e.g. if it is CPU intensive container). Hosts in this generation are in charge of containers whose lifetime is short and if the container lifetime surpasses a defined threshold, it advances to the next generation, the old generation. On the old generation, the containers are consolidated in such a way that the minimum number of hosts are used. The amount of hosts in this generation is also static as in the nursery generation.

Next we will describe the architecture of our solution, providing a high level view of it.

III. Architecture

At high level our system consists of two components, a manager and hosts, similarly to other Cloud scheduling platforms like Docker Swarm. The process is depicted at Fig.1. It starts by a client submitting a request, indicating

[1]https://www.docker.com/
[2]https://coreos.com/rkt/
[3]https://docs.docker.com/compose/overview/
[4]https://kubernetes.io/
[5]https://docs.docker.com/swarm/overview/

the request requirements. The request type refers to a **service** (does not have a finite execution time, e.g. a web server) or a **job** (if it has a finite execution time, e.g. calculating a factorial). The image refers to what the container is going to execute (e.g. an Apache web server). As for the classes, we provide four classes for the client to choose:

- Class 1: No overbooking;
- Class 2: 120% overbooking;
- Class 3: 150% overbooking;
- Class 4: 200% overbooking;

Class 1 requests do not tolerate overbooking. These requests must run on hosts that are not experiencing overbooking. As for the other classes, they tolerate 1 - (100/requestClassValue) overbooking. As an example, for a class 3 request: 1(100/150) = 0.33, therefore these request classes can run on hosts that have up to 33% more resources allocated than its nominal capacity. After this process, the Manager receives this information and according to it, among all hosts, it selects the one which maximizes overall resource utilization, allocating the request to it.

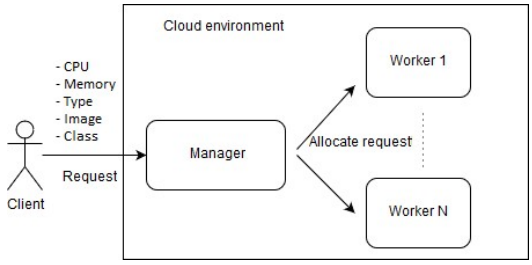

Fig. 1. Use case

1) System Architecture: Fig.2 describes the architecture in more detail, the components inside the Manager and the hosts, and how they interact with each other. Next is provided a brief overview of each component.

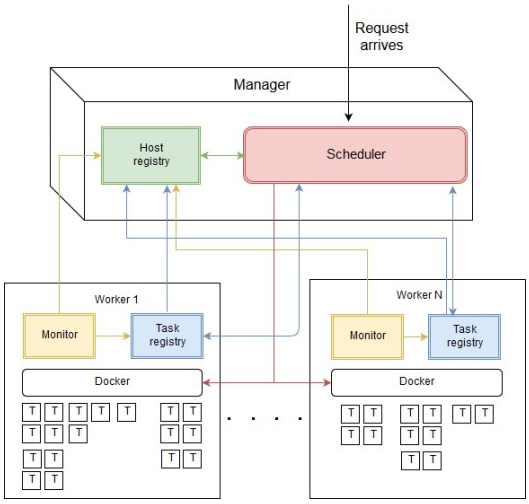

Fig. 2. System Architecture

Scheduler: The Scheduler is the first component the request interacts with. This component is Docker Swarm, which was extended to include our scheduling algorithm which is presented on the next Section.

Host Registry: This component maintains general information about the hosts (e.g. total resource utilization of each host) on these data structures. This component is also responsible for rescheduling tasks (sends them back to the Scheduler) that are killed by the Scheduler and updating task information when a cut is performed.

The **cut** means that we are decreasing the resources assigned to that task. This is different from overbooking, because overbooking affects all the tasks on a host, while a cut affects a single task. This is useful for example, imagining that in a class 1 host there is 2GB RAM available and comes a class 4. request which requires 3 GB RAM, if we put the request there, it would increase the overbooking factor over 1 which is unacceptable on a class 1 host. But if we cut it (e.g. to 2GB RAM), then we can fit it there without bringing the overbooking factor over 1. The cut is equal to the overbooking that a class tolerates, so, for example, a class 2 task, would have its resources decreased by 16%. **Kills** refer to tasks that are killed in order to allow lower level classes to be allocated to the host. The main purpose of resorting to kills is to avoid the hosts to reach extremely high resource utilization levels which would reflect in a degradation of energy efficiency.

Task Registry: This component contains more specific information about each host (e.g. current tasks being served by the host). It is also responsible for killing the tasks chosen by our algorithm.

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

To make decisions, besides the information regarding the request, the Scheduler requires additional information about the hosts. This is provided by the Host Registry and the Task Registry. Information on the Host Registry is the first to be considered, therefore being directly available at the Manager to avoid communication overheads. However, more specific information might be needed about what is running on each host. When that is the case, the Scheduler will request that information from the Task Registry of the host it requires that additional information.

Besides requesting information, the Scheduler also sends information to both registries. When a request is scheduled, the Scheduler informs the Host Registry to which host the request was scheduled and the corresponding request information (resources requirements, request type and request class). It also informs the Task Registry that a task was just created sending the same information. Upon receiving this information, each Registry will update its data structures accordingly. Next we will see how these data structures are built.

A. Key Data Structures

Our strategy for achieving better resource utilization and consequently, better energy efficiency, is based on the study performed by [19], which states that the energy consumed is proportional to the resource utilization and that energy efficiency starts degrading at high levels of resources utilization. Based on this, we decided to have three regions which map resource utilization (CPU and Memory) with energy efficiency:

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

The LEE region refers to the region that has the lowest energy efficiency, due to under-utilized resources. We want to transit hosts on that region to the DEE region as quickly as possible, where an optimal energy efficiency is achieved. Our goal is to keep the hosts at region DEE, because heavily used resources (hosts at region EED) have a negative impact on the energy efficiency, increasing the energy consumption.

Host Registry: This component 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. What defines a host class is the lowest level class task currently running at that host.

The region a host belongs depends on the current total resources utilization of the host. The total resources utilization is represented as max{% of CPU utilization, % of memory utilization}, since the highest of these two values is what is restraining more the utilization of the overall host resources. The overbooking factor is the max {CPU shares allocated/Total CPU shares, Memory allocated/Total memory}. Again, we use the max because it is what is the most restraining. As an example, if the overbooking factor is 1.3, it means we have 30% more resources allocated on that host than the total amount of resources of that host.

The lists on the regions LEE and DEE are ordered by descending order of total resources utilization and EED by ascending order. The hosts on the LEE region are ordered by descending order, because the goal is to make the hosts leave this region of energy inefficiency, bringing them up to the the DEE region as quickly as possible. Therefore the scheduling algorithm will try to schedule the requests on the first elements of the list since they are closest to the DEE region. Since the DEE region is the desired region for hosts to be, we order its lists by descending order, to use a best fit approach, i.e. put as much requests on a host to maximize it but at same time avoid entering the EED region. The EED list will only be used for kills as will be seen on the next section. The hosts on that region are experiencing high resource utilization, therefore we dont want them to be receiving more requests which would only aggravate their energy efficiency. What we want is to bring them down to the DEE region, therefore we order the lists by ascending order so that the first on the list is the closest to the DEE region.

Task Registry: As mentioned earlier, the Task Registry contains specific information about the tasks running on the host. Per host, there will exist four lists, one per overbooking class. The information of the Task Registry will only used for the cut or kill algorithm. Since the objective is to maximize resource utilization, priority is given to cutting or killing tasks that are using less resources. To achieve this, Task Registry data structures are ordered by ascending order of their total utilization resources.

The next section presents the algorithms that leverages these data structures.

IV. Algorithms

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

A. Scheduling algorithm

The goal of the scheduling Algorithm 1 is, first, to try and schedule the request either in the LEE or in the DEE region, without resorting to cuts or kills. It starts by getting the hosts that are in the LEE region, then the hosts on the DEE region are appended to that list (line 2). We prioritize scheduling in the LEE region so that those hosts can leave that region of energy inefficiency. Since the lists are ordered by descending order of total resources utilization, as we saw the previous section, the first elements of the lists are always the best candidates in order to achieve the goals of the hosts on each region.

Algorithm 1 Scheduling algorithm

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

The hosts retrieved (line 2) must respect this condition: ***request.CLASS*** $\geq$ ***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. The next step (line 4) is to try and schedule the request. **requestFits** function checks if the host has enough resources to couple with the resources the request demands. It also checks, if after the allocation, the overbooking allowed by the host is not violated (i.e. overbooking factor < host class).

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

On line 7 a lists of hosts is fetched again. This time, the hosts are aggregated differently than before. Here, the lists are fetched respecting this condition: ***request.CLASS ≤ hostClass***. 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). Like line 2, they are also aggregated by classes, following an ascending order.

If we cannot cut anything to fit the request, our last chance is to try and kill tasks in order to fit the request (line 11). On line 10 the list of hosts on regions EED is obtained and we append the hosts on region DEE. Priority is given to killing tasks on region EED, because by killing tasks and assigning a new request to it, we could bring that host back to the DEE region. Since kill is our last resort to fit a request, all the hosts on that region are considered regardless of their class.

B. Cut and kill algorithm

Before explaining Algorithm 2, it is important to mention some restrictions to cutting. We give priority to cutting the incoming request rather than the already running tasks, because cutting a task involves more overhead than cutting a request, due to the updates that have to be performed at the data structures. The following restrictions are due to the fact that when combining overbooking and cutting, class SLAs could be violated if these restrictions are not followed:

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

Having understood what a cut is, its restrictions and its benefits, we can finally look into the Cut Algorithm. At line 6 it starts by checking if the request fits considering the same conditions as in Algorithm 1 at line 4. Although this check is done at the previous algorithm, this is done again because the selected hosts for the simple algorithm are different from the hosts selected ones for the cut algorithm, for the reasons stated on the previous section. Therefore, it might be possible to allocate on these hosts without resorting to cuts, thus avoiding cutting unnecessary tasks.

Algorithm 3.2 Cut algorithm

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

If this first check fails, the next step is to try to fit the request by cutting it and checking if it fits (line 9). If it does fit, then the request is cut (line 10) and allocated to that host. Otherwise, if the request class is higher than the host class (line 13), it continues to the next host because it is not worth to cut at this host. This is the case because, if the request class is higher than the host class, then it is likely that this host contains a majority of tasks that are below the request class therefore not being worth the time searching this host for tasks to cut.

As was explained previously, the request can only be cut if the host class is lower than the request class. For the same reason, tasks can only be cut if their classes are lower than the host class. Therefore, if the host class is greater or equal than the request, only tasks whose class is higher than the request can be cut (lines 16 and 17).

A check is also performed for class 4 requests because there are no tasks higher than class 4. Otherwise, we are safe to cut classes equal or higher than the incoming request, if it is not a class 1 request, because we cannot cut class 1 requests (lines 18 and 19)

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

Line 28 checks if the request fits taking into consideration the overbooking restrictions described earlier. The tasks are checked iteratively until the request fits or does not fit at all, trying the next host. This is done iteratively instead of checking all of them at once, to avoid cutting unnecessary tasks. To reduce the amount of tasks the Scheduler needs to check for a request to fit, the Task Registry only sends the tasks that respect the cut restrictions. If one reaches line 34, it means that we cannot cut enough tasks at any host to allocate this request, therefore we must try to kill tasks to fit this request.

Regarding the kill algorithm, its rationale is very similar to the cut algorithm except for some cases. Only tasks with a higher class than the request can be killed. This provides the opportunity to co-locate similar task classes, leaving other hosts to be able to have more overbooking, increasing the overall energy efficiency. However, class 4 tasks that are services, since they are most likely not to be utilizing their resources fully, we decided to kill them if the request is a job, that is more likely to use the resources more efficiently than a service. Killed tasks are rescheduled to other hosts. If after checking all hosts the request does not fit in any, then it cannot be allocated and we warn the client.

V. Implementation

In Section 3 we saw how our solution was designed and why, at a higher level of abstraction. In this section we will go to a lower level of abstraction,looking at how the system is setup and the components implemented.

A. System setup

In order to start containers on remote hosts, Docker Swarm uses a discovery service. Docker Swarm provides a default discovery service but also supports different discovery services, such as key-value stores or DNS. The default discovery service requires constant communications with the Docker Hub[6], which is a slow process when compared to using a local discovery service without requiring external connections. We decided to use a key-value store discovery service for this purpose, Consul[7], for being simple to learn and having a good integration with Docker Swarm.

Next we will go a little deeper into the components to understand how they are implemented.

B. Components

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

[6]https://hub.docker.com/

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

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

Binary search [20] is a common and simple algorithm used to find elements in a list with O(log N) complexity. We decided to adapt this algorithm to, instead of searching for an element, to search for an index position indicating the place we want to insert.

Data structures implementation: Now we are going to look in more detail at how the data structures are actually implemented, with the goal of achieving the fastest insertion, deletion and updating times. For both registries the rationale is the same with only slight differences.

As seen on Section 3, at the Host Registry, each region will have 4 lists, one for each overbooking class. For a quick access, each region will be accessed through a map (e.g. names regions) where the key is a string with the region (LEE, DEE or EED) and the value is a struct (similar to C++ structs, there are no classes in Go, which is the language Docker Swarm is implemented) as follows:

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

ClassHosts is another map whose key is a string with the host class (1, 2, 3 or 4) and the value is a pointer to a slice[8] of a Host struct. This struct contains all the information regarding a host (e.g. IP).

These maps grant a very quick access to the hosts we want to access, useful for example, when the Scheduler asks for lists of hosts with restrictions about region and class. As an example, if we want to access all the class 3 hosts at region EED, we simply use the following: regions[EED].classHosts[3]. However, this approach is very inefficient if we want to access a single host. To solve this problem we decided to create another map (e.g. named hosts) with the host IP as key (since it is unique) and as a value, we use a pointer to a Host struct, the same Host struct as above. To access a host cpu utilization and update it we can now simply use: hosts[193.146.164.10].CpuUtilization=0.23.

Using this approach also increases concurrency, consequently increasing overall performance. If we did not use a separate map (hosts) to access hosts in a single fashion, in order to make an update to a host, besides iterating through all hosts within the regions map, we would also have to lock that map in order to ensure it remains consistent, allowing only one host to be updated at a time which has severe performance implications. The same rationale is used to implement the data structures on the Task Registry.

Synchronization: The data structures need to remain consistent despite the concurrent changes. To accomplish this, we defined fine-grained locks so the data structures locked are the minimum to have everything consistent.

To achieve these fine-grained locks mentioned in the previous paragraph, we resort to maps and structs again.

[8]https://blog.golang.org/go-slices-usage-and-internals

There is a map (e.g. named mapLocks) with a string key representing the host class (LEE, DEE or EED) and as a value the following Lock struct is used:

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

Using the above struct, we can have more coarse-grained locks (using lockRegion) by locking at region level if required, or more fine-grained locks by locking at class level through the classHosts map, whose key is a string representing the host class (1,2,3 or 4) and the value is Go internal lock.

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

To finish the components implementation description, we are going to describe how the Monitor measures resource utilization and how it decides when to send an update to the Host Registry or the Task Registry.

Host resources monitoring: Every 3 seconds samples are collected. After 30 seconds, we average all the samples collected during that interval and use those values (CPU and memory) to check if an update should be sent.

In order for the update to be sent to the Host Registry, a condition must be verified. The difference (either CPU or memory) between the last update sent and the current measurement must be higher than a threshold. The threshold is defined at 10 p.p.

To collect resource usage information we use System Information Gatherer And Reporter (Sigar)[9]. It provides a simple and efficient way to access OS/hardware information.

Tasks resource monitoring: The rationale behind tasks resource monitoring is the same as of the host monitoring, except that the time between measurements is 45 seconds instead of 30. We increased the value because tasks resource utilization is not as volatile as the hosts resources utilization. We leverage Docker built-in command, stats[10], to get CPU and memory utilization of each task.

VI. Evaluation

This chapter describes the experiments carried out to evaluate the proposed solution against the two Docker Swarm scheduling algorithms, spread and binpack. We start by describing how the evaluation was carried out, followed by its results.

A. Experimental setup

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

Our prototype was deployed 6 hosts only, provided by INESC-ID[11]. These hosts are powered by an Intel Core i7-2600K CPU @3.40GZ, 11926 MB RAM and HDD 7200RPM SATA 6GB/s 32MB cache. One host served as the Manager and the remaining hosts served as workers, executing clients requests.

Due to the lack of tools to benchmark Docker Swarm scheduling decision quality, we only managed to find benchmark tools to evaluate scheduling speed, we had to create our own custom workload and extensions to collect metrics.

Our evaluations lasted one hour in order to have as much variability as possible and three evaluations were executed for each solution. The workloads generated were saved and used on all attempts on the different scheduling algorithms so that they were tested with the same conditions.The following requirements for each workload was generated: **CPU requirement; Memory requirement; Request makespan; Workload type; Request rate; Request class.**

CPU and memory requirements are generated using an exponential distribution. We decided to use an exponential distribution since it provides a good variability. The number generated by the exponential distribution was mapped to a CPU and memory value. For CPU, the minimum value depends if it was a service or a job. If it was job, the minimum CPU assigned is 204 CPU shares (equal to approximately 20% of a single core utilization). If it was a service, the minimum CPU shares was 2, because services do not require as much CPU as jobs. As for the maximum, it was 1024 CPU shares (equal to 100% of a core utilization). For Memory requirements, the limits are the same for jobs and services, the minimum was 256 MB and maximum was 2GB.

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

The workload type was chosen randomly between four types of workloads that we have selected. For each of these applications types, we have selected real and popular Docker applications (with the exception of the non-intensive), in order to be representative of each type. The types and respective application used for that type were the following: **FFMPEG**[12] is the CPU-intensive workload, a video encoding application. We used **Redis**[13] as the memory intensive application which is an in-memory key/value store.

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

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

[11]https://www.inesc-id.pt/

[12]https://hub.docker.com/r/jrottenberg/ffmpeg/

[13]https://hub.docker.com/r/redis/

For the CPU and memory intensive, we have chosen a **Deep-learning**[14] application, where a neural network is trained to zoom in images. Finally for the non-intensive application, we created a Docker application called **Timeserver**[15] which simply returns the time when requested.

The last thing to be generated was the request class. We give more probability for classes 2 and 3 (30% and 45% chance respectively) because we believe that these would be the most used in a real situation. Class 4 (15% chance) since it has a big depreciation, it would be less used than classes 2 and 3, however in our view, it would still be more used than class 1 requests (10% chance) due to the lack of benefits (in terms of compensation) this class provides.

We compared our solution with our competitors using the following metrics: **scheduling speed**; **failed/successful allocations**; **resource utilization (CPU and Memory)** throughout the experiment;**job makespans**;**services response times**. We also did an individual evaluation to our solution, to see how much it resorts to **cuts** and **kills**, as well as how much CPU and memory was cut.

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

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

B. Evaluation results

We will see that our solution allows significantly more requests to be allocated, achieving an overall better resources utilization. A natural and unavoidable tradeoff of our solution is a comparatively slower scheduling speed to the other solutions, these differences will be exposed. A possible consequence of overbooking could be that jobs or services, can take longer times to finish or to respond, respectively. We will see if this is the case in our solution. To finish, the cuts/kill ratio is presented and we will see how they are useful, especially the cuts, in order to increase the amount of requests that can be allocated.

Successful and failed allocations: The results obtained for the successful and failed allocations are presented at Table 1. We can quickly see that our solution (named Energy) has a significantly higher success rate than the two solutions provided by Docker Swarm. By having such high fail rates, the other solutions would require more machines than our solution does, consequently using more energy. We can also see that our solution deals with less requests than the other two approaches, in an one hour evaluation. This derives from the fact that our algorithm is comparatively slower than the other solutions, due to our solution keeping the resources almost fully utilized for a longer period of time as will be seen afterwards.

[14]https://hub.docker.com/r/alexjc/neural-enhance/

[15]https://hub.docker.com/r/sergiomendes/timeserver

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

TABLE I

SUCCESSFUL AND FAILED ALLOCATIONS

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

Resources utilization: By looking at the graph at Fig.4, which represents the average **CPU utilization** of the worker hosts throughout the evaluation, we can see that our solution (Energy) achieves an overall better CPU utilization. We can see that our solution (green line) is more consistent than the other two, fluctuating most of the time between 75% and 88%. The Binpack solution (blue line) is most of the time bellow 80%. Spread (orange line) is better than Binpack, but worse than our solution, most of the time it is bellow the green line, with some exceptions.

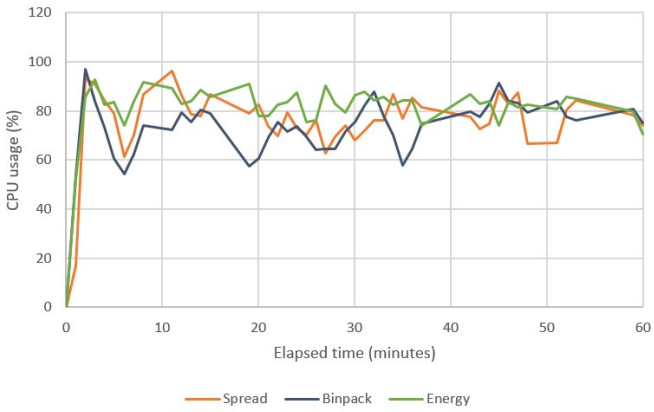

Fig. 3. Average hosts CPU utilization of each solution

Despite Spread and Binpack having the resources fully allocated, since they are not being used 100% of the time, this resource inefficiency happens. This is even more salient in real life scenarios, where clients after ask for much more resources than they actually need. This clearly indicates that more resources could be allocated to some of the hosts to make them more efficient. This is illustrated by the results of our solution, where most of the time, the hosts have more than 70% resource utilization.

Memory: Again, our solution presents better results than the existing solutions provided by Docker Swarm. The other solutions never surpass the 60% mark. Our solution achieves

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

TABLE II

AVERAGE CPU AND MEMORY UTILIZATIONS

it constantly throughout the whole evaluation as can be seen on Fig.4.

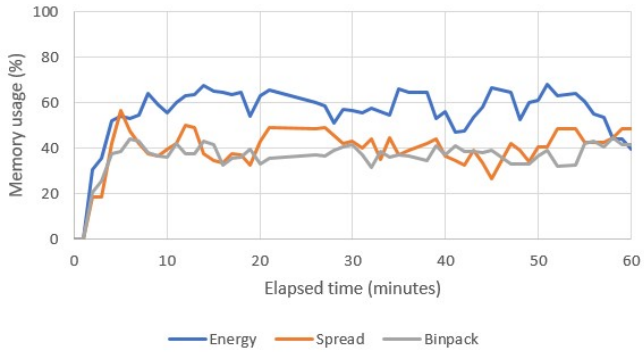

Fig. 4. Average hosts memory utilization of each solution

Despite being more inconsistent than CPU, our solution provides bigger improvements regarding memory utilization over CPU utilization as can be seen by at Table II. We can see at this table that our solutions provides a 5.6 p.p improvement over Spread and 8.2 p.p over Binpack, regarding CPU utilization. The memory utilization improvement is much more significant, achieving an improvement of 15.8 p.p over Spread and 18.9 p.p over Binpack.

Scheduling delays: The significant improvements analyzed previously, unfortunately, do not come without a price. This section presents the results regarding the scheduling delays, i.e. the time to schedule requests.

Table III presents a summary about the time it takes to schedule a request on each solution. By looking at the average values, as expected, our solution performs worse than Spread and Binpack. Despite being more complex, our solution is only slightly slower when the system has more resources free, has can be seen by the 50th percentile at 0-5, 25-30 and 50-55. This last note shows that this scheduling delay can be decreased if more machines are added. For the remaining elapsed time, the 50th percentile oscillated between 735.23 and 2190.91 seconds.

Response times: Now we will see that despite allocating more requests as was seen previously, our solution is close to our competitors response times.

Table 3 presents the response times obtained for each type of workload used. Redis - 20 indicates that a request rate of 20 to access Redis was used, the same applies for the following columns.

For the CPU-intensive workloads, FFMPEG, we can see that our solution has a better average time than the other

Solution (ms) / Elapsed time (minutes)	**Spread**	**Binpack**	**Energy**
0-5	Average: 2904,87 50th: 9.90 90th: 10002,95 99th: 22974,34	Average: 5278.21 50th: 10.19 90th: 18696.95 99th: 27176.24	Average: 18469.96 50th: 11.94 90th: 72720.68 99th: 128048.71
15-20	Average: 5720.68 50th: 9.64 90th: 16972,52 99th: 68623.37	Average: 5720.68 50th: 9.64 90th: 16972,52 99th: 68623.37	Average: 33644.27 50th: 838.11 90th: 121751.3 99th: 169663.67
25-30	Average: 4281.98 50th: 10.49 90th: 14833.28 99th: 33677.65	Average: 4368.09 50th: 10.09 90th: 18108.94 99th: 28256.79	Average: 33264.4 350th: 15.31 90th: 13997.85 99th: 202526.65
50-55	Average: 4685.35 50th: 10.35 90th: 14950.33 99th: 51035.95	Average: 6079.21 50th: 9.73 90th: 22136.46 99th: 51136.6	Average: 23251.23 50th: 13.7 990th: 19237.15 99th: 223237.94

TABLE III

TIME TO SCHEDULE REQUESTS

two, although it has a higher 50th percentile compared with Binpack. For the CPU/Mem intensive workloads, Deep-learning, we can see that our solution no longer has the best results, but is still better than Spread (better average and 75th percentile results). This decrease in performance compared with Binpack and Spread for this type of workload is unavoidable, because we have significantly more memory utilization rates than the other solutions.

Next we have the Redis results, the memory-intensive workload. Redis produced some unstable results as can be seen by the fact that Redis-80, for Binpack, has better results than Redis-40 and Redis-20, which should not be the case, since Redis-80 is twice the request rate of Redis-40, and four times Redis-20. We assume that our solution here would achieve worse times than the other solutions because of what was seen with CPU/Mem-intensive workloads due to the memory impact, potentially worsening as the request rates increased.

Finally we have the non-intensive workloads, the Time-server. Here our solution performs slightly worse than the other solutions at all requests rates.

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

Kills also play and important role, avoiding the hosts from entering extremely high utilization values. Only **202 kills** (14,4% of the successfully allocated requests) were executed throughout the experiment. Even if those 202 tasks that were killed could not be successfully rescheduled and if we considered them as not being allocated, we would still have a higher successful allocation rate than Docker Swarms solutions.

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

TABLE IV
RESPONSE TIMES

VII. CONCLUSION

Despite all the effort done by academia, the problem of energy consumption in data centers persists and needs to be addressed. In this work we started by identifying the current solutions that exist and their challenges, in order to identify opportunities so that we can contribute to the literature. Due to the lack of work regarding containers, we defined our objective, develop an energy-efficient scheduling algorithm using Docker.

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

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

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

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

[8] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] 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.

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