CRANFIELD UNIVERSITY

Léo Unbekandt

Investigation and implementation of resource allocation algorithms for containerized web applications in a cloud environment

School of Engineering

Computational and Software Techniques in Engineering

MSc

Academic Year: 2013 - 2014

Supervisor: Mark L. Stillwell

August 2014

CRANFIELD UNIVERSITY

School of Engineering

Computational and Software Techniques in Engineering

MSc

Academic Year: 2013 - 2014

Léo Unbekandt

(leo@unbekandt.eu)

Investigation and implementation of resource allocation algorithms for containerized web applications in a cloud environment

Supervisor: Mark L. Stillwell

August 2014

This thesis is submitted in partial fulfilment of the requirements for the degree of Master of Science

Declaration of Authorship

I, Léo Unbekandt, declare that this thesis titled, 'Investigation and implementation of resource allocation algorithms for containerized web applications in a cloud environment' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Léo Unbekandt

Date: 18th. August 2014

Abstract

Operating system-level virtualization, commonly featured as containers, has taken an important place in the current IT ecosystem. They allow people to deploy applications with an additional abstraction level. Whatever is the underlying hardware, or the version of the operating system, a container will always run the same way.

This technology has increased the flexibility of an infrastructure, and even more of a cloud based infrastructure. This thesis defines a platform to test resource allocation algorithms in a concrete environment, not a simulation. The scope of this work has been reduced to bin packing algorithms which are the most common for resource allocation and load balancing. Furthermore, the studied tasks are only web applications, for their ability to be stateless, and to migrate easily.

The defined platform is based on *Docker* and on the decentralized cluster management and service discovery tool *Consul*. Different utilities have been developed. They create the ability of spawning containers on all the servers of the cluster. Moreover, those containers are gathered by service. If two containers are part of the service "abstract", all the requests on `http://abstract.thesis.dev` are shared between them.

Online bin packing and offline bin packing algorithms have been used to respectively study the resource allocation and the load balancing problems Experiments have been designed to measure their impact on the performance of the running processes part of the infrastructure.

Keywords: operating system-level virtualization, containers, load balancing, resource allocation, bin packing, cloud environment, virtual machines, web applications

Acknowledgements

First and foremost, I must sincerely thank my supervisor Dr Mark Stillwell for his advices, useful feedbacks and abovall his sympathy. Motivation and concentration wasn't always easy to find, as I've been working professionaly in parallel of my MSc to fund a company. Mark was understanding and helped me to revise my goals for this thesis and to find the energy I needed.

Thank you to Pauline, our course administrator for her good work all over this year, she has been helpful and friendly and I personnaly think that it is something very important to tie the students of a course together and create a good atmosphere.

I am also grateful to Cranfield University for offering me the opportunity to study here by offering a scolarship to me. I'd like to thank Stéphane Genaud, director of the French Engineering School **ENSIIE**, who invested time to create this partnership betweew the school and Cranfield University.

Of course I'm thankful to all the people I've met in Cranfield, with whom I've shared my difficulties and my doubts and who reconforted me in the idea that I wasn't as late as I thought in my work, or at least we were late altogether.

Source code license

All the source code developped in the scope of the experiments done in this thesis are developed under the MIT License. The integrality of the examples are publicly available on GitHub https://github.com/Soulou

Contents

List of Figures

List of Tables

Abbreviations

VM	**V**irtual **M**achine
SaaS	**S**oftware **a**s **a** **S**ervice
PaaS	**P**latform **a**s **a** **S**ervice
IaaS	**I**nfrastructure **a**s **a** **S**ervice
SLA	**S**ervice **L**evel **A**grement
DBMS	**D**ata**B**ase **M**anagement **S**ystem
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
HTML	**H**yper **T**ext **M**arkup **L**anguage
CSS	**C**ascading **S**tyle**S**heet
JS	**J**ava**S**cript
XML	e**X**tensible **M**arktup **L**anguage
JSON	**J**ava**S**cript **O**bject **N**otation
PID	**P**rocess **ID**entifier
CM	**C**onfiguration **M**anager

Introduction

Optimization of allocation is a common problem to every developer or system administrator who has to scale software programs in a distributed setting such as a cloud environment. What is the best distribution for the set of applications we need to deploy? Everybody is looking for the best price-performance ratio. This thesis deals with the problem of resource allocation and gives the design of an experimental infrastructure which can be considered as an "enterprise" infrastructure.

The final goal is to have a convenient platform to perform experiments in order to evaluate the efficiency of resource allocations algorithms using different heuristics. The aim of this work is to be able to have an overview of what is possible and to give recommendations concerning good practices and interesting algorithms to use in a real world situation.

In the last decade, the hardware has kept improving year after year, especially processors, which have been improved at doing operations simultaneously. Infrastructure owners have been struggling to create ways to split those resources securely and efficiently, so they may be used by several isolated customers. To answer this need, technologies related to the concept of virtualization [3] have been created, enabling to split servers into isolated sub-servers, sharing resources from a common host.

Amazon Web Service started commercialising their service of on-demand virtual servers named **Elastic Compute Cloud** in 2006. It was the first actor of a large market, which has since grown hugely. Based on the concept of "pay as you

go", their customers are able to adapt, in real time, their infrastructure consumption, impacting directly the money they are spending. This attractive feature gave birth to the first cloud services like **Dropbox** which was first based on Amazon infrastructure.

An interesting definition of the Cloud Computing has been written by the National Institute of Standards and Technology [4]:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

This thesis doesn't focus on the allocation of virtual machines in a set of physical servers, but at a higher level, the allocation of application containers. Actually, VMs have been used extensively and studied for more than a decade now, but this is not the only way to achieve virtualization. Containers are considered as lightweight virtual machines. Instead of being global (virtualization of hardware and operating system), they focus on the isolation at the application level. This method is getting more and more attraction: companies have started requesting infrastructure based on it [5].

When using containers, it is a common practice to split its product into a set of loosely coupled softwares, which are communicating together using a common communication protocol. The most often, the web (HTTP) is used, and those services are sending and receiving requests through HTTP APIs. One of the main advantages of those applications is that they are stateless. As a result, it is much more easy to move them from one server to another, this is why this thesis will focus on these particular softwares.

The main question which will be answered is: how those services can be managed when they are containerized and how is it possible to apply resource allocation

methods in a concrete infrastructure to manage all the containers and their resource consumption?

⇒ In the first chapter, a literature review provides a large background concerning the motivation behind resource allocation and load balancing, showing that a lot of different perspectives have been used by researchers these last years. A particular focus is given to bin packing algorithms which will be used later in this work.

⇒ Then, the second part provides details about the scope of the thesis. Containerized web applications features and limits are covered.

⇒ The third part is a preliminary experiment to measure the ability to share the CPU cores of a server between containers.

⇒ The design of the experimental platform in a cloud environment is defined in the chapter 4. The different roles of the servers are explained, as well as the deployment of the complete software suite in another infrastructure.

⇒ Finally, the last chapter shows different experiments done in the previously created infrastructure and some discussions concerning the use of bin packing algorithms.

Chapter 1

Literature Review

1.1 Motivation

Cloud computing is currently a major trend in the IT culture. Companies are more and more using cloud infrastructure, even if their reasons are still blurred and have been strongly altered by a skilled salesperson. However, behind this marketing layer, there are some concrete technological evolutions and advantages in cloud related products.

The legitimate question is "Why should people use cloud services? ". Whether it concerns virtual machines, whether it is linked to containers, the answers are multiple. Valentina Salapura explains how a virtualized environment improves the resiliency of an infrastructure [6]. As cloud infrastructure should be highly available and fault tolerant, in order to provide a high-quality SLA (Service Level Agreement) and attract customers, the overall quality of the hosting is in most of the cases better that what can be done by the customers themselves. It is not their job, they don't have as much knowleldge as the provider which is specialized in this field.

The **IaaS** provider owns the infrastructure, commonly composed by several physical machines (PMs), sometimes located in different data centers. Each of these

PMs contains a variable number of virtual machines (VMs), rented by the customers. In the scope of a **PaaS**, the provider has the control on the way the users access those instances. Containers are often used to isolate the users with each other. The interesting problematic concerns the assignment of these VMs on the PMs or the containers on the VMs, what is the optimal distributions of the containers among the different servers?

At the scope of the physical server, Thomas Setzer and Alexander Stage based their study on the statement that energy represents up to 50% of operating costs of an infrastructure [7]. That's why there is a need to optimize it. For consumers of commercial *IaaS* offers, the main goal is to use the minimum number of virtual machines while having enough resources for all the applications running on their current infrastructure. They do not directly pay the electricity, it is included in the price paid to the provider, the focus is on the level of performance directly.

1.2 Algorithms

Cloud computing is a hot topic in the IT industry. New problematics have appeared, the resource allocation problem is one of them. The different algorithms listed in this document gather publications about resource allocation or reassignment.

In the first part, a general solution method is studied. Using Linear Programming, an optimized result is able to be found. However as this problem is NP-hard, it is not possible to use Linear Programming anymore when the amount of objects or the number of resources are getting big. This is why we have to use heuristics. Heuristics decrease the quality of the results, but allow the computation to be executed much more fastly.

Among those heuristic algorithms, we are going to have a special focus on the bin packing heuristics which are the methods used later in this thesis. An overview

of other methods like the Ant Colony heuristics, network flows based, or genetic algorithms will be proposed.

1.2.1 Linear Programming

Also known as Linear optimization. It is specialisation of mathematical programming, which is focused on linear functions. The main goal of linear programming is to find a maximum or a minimum to a linear function given a set of constraints. In other words: maximizing profits while minimizing costs. In the scope of resource allocation, it is required to define the different variables, the function to optimize and the constraints linked to the variables.

In the work *Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads* [8], the authors are working with linear programming. The aim of their study is to define a way to optimize the number of allocated virtual machines splitted in different cloud infrastructures. Different constraints are defined to setup the scope of the function to minimize.

Equation 1 Example of linear optimization problem

$$Minimize \sum_{k=1}^{A} \sum_{l=1}^{T_k} \sum_{i=1}^{I} \sum_{j=1}^{C} (y_{klij} \cdot (ni_{kl} \cdot pi_j + no_{kl} \cdot po_j) + \sum_{s=1}^{S} (p_{ij} \cdot x_{klijs}))$$

Equation 1 is the problem they want to solve, in this case a cost minimization problem. How can we minimize for each task t of each application k in each virtual machine i of each cloud infrastructure j the price of the input and output bandwidth ($ni \cdot pi_j$ and $no_{kl} \cdot po_j$) and the price the requested virtual machines ($x_{klijs} \cdot p_{ij}$) at each unit of time (S)

Equation 2 Example of constraints in a linear program

$$\forall j \in [1, C], s \in [1, S] : \sum_{k=1}^{A} \sum_{l=1}^{T_k} \sum_{i=1}^{I} cpu_i \cdot x_{klijs} \leq maxcpu_j$$

The *Equation 2* defines a constraint from the linear problem, which explains that in each cloud, at each unit of time, the sum of all the tasks running on all the

virtual machines instantiated should be less than the number of CPUs available. (There is note that in the case of public clouds, the amount of CPU is considered unlimited so this constraint becomes void).

The work of M. Stillwell, F. Vivien, and H. Casanova [9], which focuses virtual machine resources allocation in heterogeneous environment, also starts by defining a formal model based on linear programming. However, as explained in this publication, resolving such a problem requires an exponential time, linked to the amount of allocations to achieve. As a result using directly this solution on an important workload is not feasible.

Linear optimization relaxation can be used to simplify the original problem and transform it from an exponential complexity to a polynomial complexity **(author?)** [10]. The "random rounding" (RRND) is a probabilistic approach which modifies some of the constraints by a weaker one.

Equation 3 Application of random rounding

$$\text{constraint before: } 0 \leq x \leq 1$$
$$\text{constraint after: } x_r \in 0, 1$$
$$x_r = 1 \text{ with a probability of } x, \text{ otherwise: } 0$$

However, the RRND approach is quickly discarded as the results are not good enough in the case of resource allocations in heterogeneous environment.

1.2.2 Bin packing

Bin packing is one of the most common method to calculate resource allocation or re-allocation. It consists in representing "bins" associated to a storage capacity and "items" which have to be packed into those bins.

1.2.2.1 Different variants

Two main types of bin packing algorithm coexist. On the one hand, those considered as "offline". They consider that we have access to all the items to find the optimal packing on the different bins. This problem is a NP-hard problem, there is no, to this day, a polynomial way to solve this problem. That is why to answer this problem in a reasonable duration, different heuristics have to be defined in [11]:

Algorithm Name	Description
First Fit (FF)	Pack the item in the first bin with a large enough capacity
Best Fit	Pack the item in the bin which will have the less capacity after packing
Worst Fit	Opposite of Best Fit: Pack the item in the bin with the biggest capacity
Next Fit	Same as FF except that instead of reconsidering the first bin after packing, the current one then the next one is considered
*-Fit Decreasing	First, sort the items in a decreasing order, then apply any of the *-Fit algorithm

Those different algorithms reduce the complexity of the packing operation to $O(n \log n)$. But as [11] title explains: they are "Near-Optimal". The issue is finally to find the best ratio optimality/complexity.

On the other hand, the "online" algorithms, which, on the contrary, are packing items at the time they are arriving. In this case bins are already partially filled with other items, and it is not always possible to move those. Thus, the main goal is to find the best assignment for the newly coming item. Previous *-Fit could be

directly used. However, it is really limited to pack one item in a set of bin, this is why different algorithms have been developed

To answer more precisely to the cloud resource allocation problem, some people have defined some variants of those two main categories of bin packing algorithms. G. Gambosi and A. Postiglione and M. Talamo have developed a "relaxed online bin packing" algorithm [12]. It may be represented as a mix between online and offline bin packing. When a new item has to be packed, it allows an additional limited number of moves among the currently packed items.

Another interesting variant is the dynamic online bin packing defined by Joseph Wun-Tan. It differentiates itself from standard online bin packing by allowing items to be removed from bins. Static online bin packing does not allow these items changes, once an item has been placed it does not move anymore.

1.2.2.2 Their application in resource allocation

In the scope of containers assignment on a set of hosts, the bins are the different servers and the items are the services we want to host. Some additional aspects have to be considered: applications need different resources like memory, CPU, persistent storage (disk), network input/output. Often, the items are multidimensional, and it is named multidimensional vector bin packing. Moving a VM or container from one host to another has a cost which may be important for virtual machines, so the number of parallel migrations may be limited.

In *Adaptive Resource Provisioning for the Cloud Using Online Bin Packing* [13], the authors consider first, that a virtual machine only has one dimension, its CPU consumption. They reject "strict" online bin packing, because we often don't know exactly the future consumption of a virtual machine, so it is necessary to move it afterward, when we can measure it. Moreover, as VMs can be migrated easily, there is no reason not considering it if the resulting performance is better. "Relaxed online bin packing" allows movements when a new item is packed, but an item cannot be resized. "Dynamic online bin packing" is thought inadequate in

this context too, but often, when an virtual machine has to move the best solution is not always to remove it then repack it, but to move others instances which are easier to move.

This is why in [13], they decided to build an online bin packing algorithm which suits virtualised environments: "Variable Item Size Bin Packing", its characteristics are the following.

- As relaxed online bin packing, it allows movements when a new item is packed
- Stronger limit of movements, to avoid executing too many migrations
- A `change` operation is defined to modify the size of an item in a bin

They extend their algorithm to multidimensional vectors by considering the biggest value among the different dimensions of a vector, so the problem returns to one-dimension. Using this way to simplify the problem is working in some cases. Commonly when a resource consumption increases the others are following. For example an application having a high network bandwidth requirement, would also have a high CPU consumption. Finally, they admit that this solution would work quite poorly in the case of instances with non-proportional requirements.

In [9], we have seen that the first approach of the author was around linear programming, but the main part of their work is defining a way to apply multidimensional vector bin packing to heterogeneous environments. On a first side, they deal with the multidimensional aspect of this problem. It is necessary to specify how to sort the items because there is not natural way to sort these vector.

- Value of the maximal dimension
- Sum of all dimensions
- Ratio of the max/min
- Difference max-min

- Lexicographic order
- None

Most of the previous algorithms are not considering the way the bins are used. In this publication, as it is targeting heterogeneous infrastructure, the order the bins are sorted when executing any algorithm matters. All the previous way to sort the items can be applied to the set of bins.

All these previous possibilities of ordering among the virtual machines and physical hosts are combined and result in a "meta" algorithm (METAHVP) which takes the best result out of the different combinations of one items ordering and one bins ordering. After individual analysis, some sort types are removed from the meta algorithm to improve its runtime. (METAHVPLIGHT)

The simulation achieved to test these heuristics are comparing the results to those which have been found using the linear programming method and those obtained using greedy algorithms (*-Fit). The conclusion is that METAHVP has the best results over all the other, and METAHVPLIGHT achieves this result in on tenth of METAHVP's runtime.

Finally, according to what we want to study there are several possible solutions using bin packing. Semantically, it is really comfortable to compare bins with physical servers and items with virtual machines, it allows a very natural vision of this problem.

1.2.3 Others

To deal with mathematical optimization and approximative solution of NP-complete problems, Ants colony algorithms, genetic algorithms and some other famous methods, based on statistical analysis.

1.2.3.1 Ant colony algorithms

In [1, 14, 15], the ant colony algorithms are studied. As we can see in the following graph:

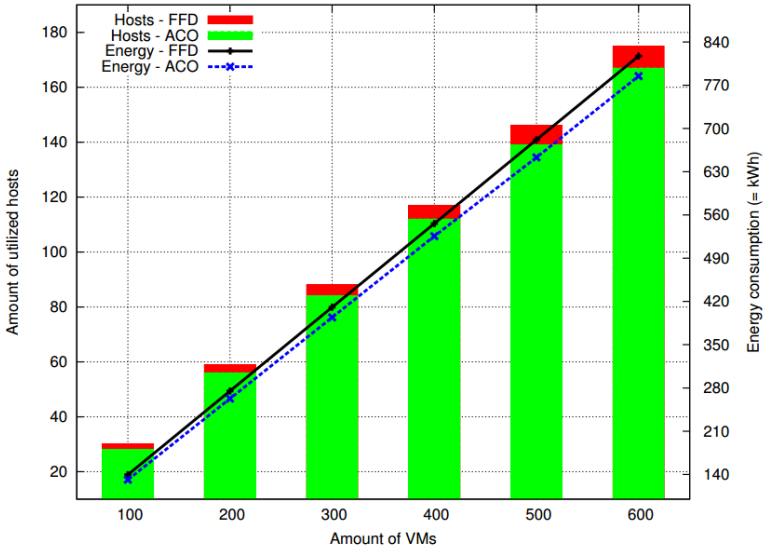

FIGURE 1.1: Comparison between First Fit Decreased and Ant Colony algorithms in [1]

The simulation shows that the ant colony gets better performance than a simple greedy First Fit Decreasing, however this improvement is not free:

VMs	**Policy**	**Hosts**	**Execution time**	**Energy (= kWh)**	**Energy gain (= %)**
100	FFD	30	0.39 sec	139.62	
	ACO	28	37.46 sec	131.41	5.88
200	FFD	59	0.58 sec	275.13	
	ACO	56	4.51 min	262.83	4.47
300	FFD	88	0.77 sec	410.65	
	ACO	84	15.04 min	394.28	3.98
400	FFD	117	1.03 sec	546.16	
	ACO	112	34.23 min	525.75	3.73
500	FFD	146	1.39 sec	681.67	
	ACO	139	1.17 h	653.17	4.18
600	FFD	175	1.75 sec	817.19	
	ACO	167	2.01 h	784.75	3.96

FIGURE 1.2: Runtime of First Fit Decreased and Ant Colony algorithms in [1]

When the number of nodes becomes bigger, the time spent to find the optimal allocation grows hugely, it is thousands times longer than a simple First Fit Decreasing for 3 to 5 percents of improvement. For analysis purpose it is something interesting to get better results, but in a realistic point of view, this operation can not take several hours as it should be repeated often.

1.2.3.2 Genetic algorithms

Genetic algorithms (GA) are heuristics based on natural selection. Generations of solutions are mutating, inheriting with and from each other to result in close to optimal results. [2] and [16] focused on them to solve the virtual machines assignment problem. In the work of David Wilcox et al.[2], simulations are comparing GA with *-Fit algorithms.

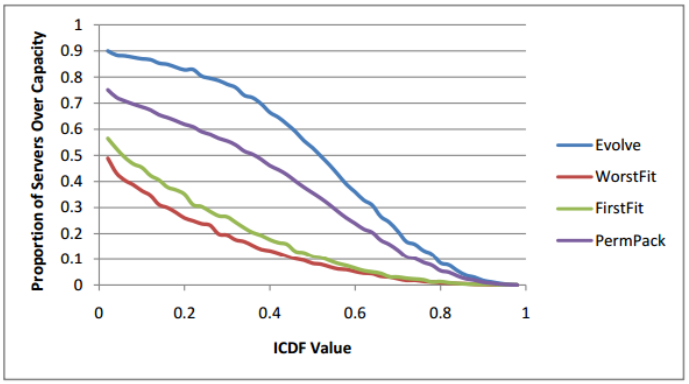

Fig. 6. A comparison of the proportion of servers over capacity.

FIGURE 1.3: Results of simulations using a genetic algorithm[2]

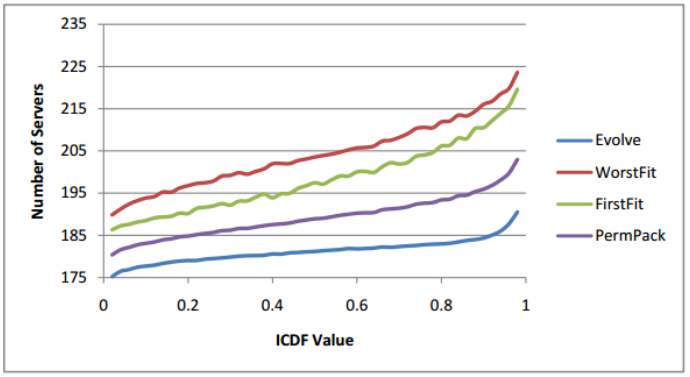

Fig. 7. A comparison of the the number of servers found.

FIGURE 1.4: Results of simulations using a genetic algorithm[2]

On the following graphs, ICDF stands for "Inverse Cumulative Distribution Function" also known as "quantile function", the authors use it to represent the load: "Using the icdf, we can specify a percentile value and obtain a corresponding load which can be passed to the assignment algorithm".

The conclusion which is that GA tends to consume less physical hosts, at any load, the number of PMs is largely under the amount of servers used by the other bin packing algorithms. As a direct consequence, the PMs which are over-capacitated (where the amount of VMs exceed the resource capacity of the physical sever), is much more high. For this reason, this approach can hardly be used in environment where a SLA (Service Level Agreement) has to be respected, because if there are overloaded servers, some applications or tasks running of them will be slowed by this situation.

1.2.3.3 Network flows

Network flows are basically directed graphs where each edge has a capacity and a flow. The main property is that each node of this graph must have an equal sum of flows from the edges directed to it and leaving from it, except for two particular type of nodes: "the source node" and "the sink node".

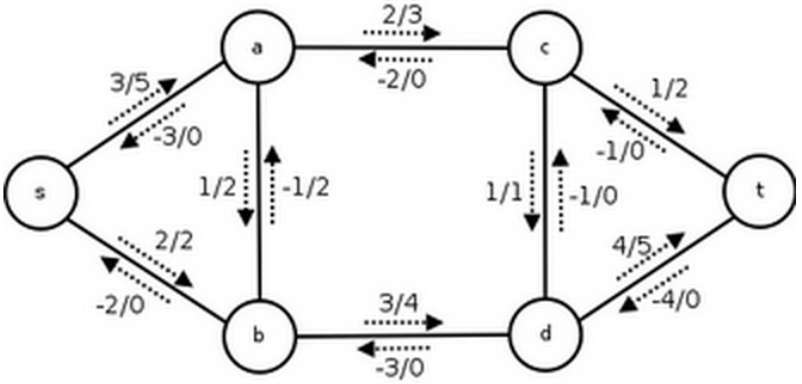

FIGURE 1.5: Example of network flow directed graph

Some people have used this concept to build a model to solve the resource allocation problem, to fine a close to optimal solution. Kimish Patel, Murali Annavaram and Massoud Pedram worked on resource assignment in datacenter[17], considering an heterogeneous environment as in [9]. Each set of similar servers, considered as a pool of servers is represented by a node, with a capacity different from each other according to the differences between two pools of servers.

Unfortunately, this technique does not seem to be used for virtual machines allocation, and the link between this method and the problem we are dealing with is not obvious at all.

1.3 Real data analysis

Most of the cited works in the literature review are basing their work on simulations. In the experiments, simulation tools like SimGrid[18] or CloudSim[19] are used to simulate the behavior of one or multiple cloud infrastructures.

The data may be generated randomly or following some statistical rules, but often, workloads are based of extract of real workload. Typically, Google is releasing workloads of its own production infrastructure.

In 2012, Google sponsored the ROADEF contest (Operational Research and Decision Support French Society)[20]. The contest was focusing the machine reassignment problem based on Google workload. Each attendee had to find the best solution. Some of them resulted in an official publication like "Heuristics and matheuristics for a real-life machine reassignment problem" [21]. They based their work on linear programming. However in [22, 23], the authors have used around the bin packing algorithms. Unfortunately, the work of the winner has not been published so we are not able to see which algorithm has been used to achieve the best reassignment.

Chapter 2

Container load balancing in cloud environment

2.1 Definition

Operating system-level virtualization has been existing for a long time, the premises of this technology can be found at the early times of computing in the 1960s. It wasn't named the same way, but companies like **IBM** had to allow users to split the computing time of a mainframe for example. Different operating system weren't started, but processes were strongly isolated inside one unique OS.

Operating system-level virtualisation was not common in most infrastructures until the nineties where common open-source kernels started to implement features related to it. **Jails** appeared first in the BSD kernel (1998). Then, Sun developed Solaris (Sun UNIX operating system) **zones** in 2005, the same year as the **OpenVZ** implementation over the Linux kernel. But tools which really popularized this concept to a broader audience is named **LXC** for LinuX Containers, introduced in 2009.

Containers are running over the same operating system as the host server, they are sharing the same drivers, but all the processes contained in them are running in

an isolated environment. The memory consumption, the CPU usage, the network and disk IO are monitored and managed by these container engines sending the corresponding instructions to their respective kernel.

This technology has been more and more used in the industry these last 3-5 years, more and more companies are adopting it. It may be to offer services for companies like OpenShift (Red Hat), Cloud Foundry, Heroku, MongoLab, etc. or to manage the hosting of their own projects: Google, Ebay, Spotify. What kind of applications are containerized? Any software is able to run in a container, the work done by **?**] shows in the field of HPC, containers are mature enough to replace virtual machines and get better performance.

This is a completely different approach to process isolation compare to classical virtual machines where hypervisors and VMs have been following the paradigm where everything is virtualized, creating overhead and slower performance, then optimization is added afterward.

The main idea behind containers is, based on the host operating system, only the required devices/features will be virtualized, and finally the level of performance is close to native efficiency. Already in 2007, some articles have shown the possibility to use container-based virtualization instead of hypervisors and virtual machines as a high-performance alternative [24].

2.2 Docker container engine

2.2.1 A bit of history

Docker is a recent project. Introduced in May 2013 by the Platform as a Service provider **Dotcloud** with the ambition to create a standard way to manage multi-platform containers, **Docker** has rapidly been promoted as a mainstream project supported by all the main tech companies. Their catchword is "Build once, run everywhere", but this is only a half truth.

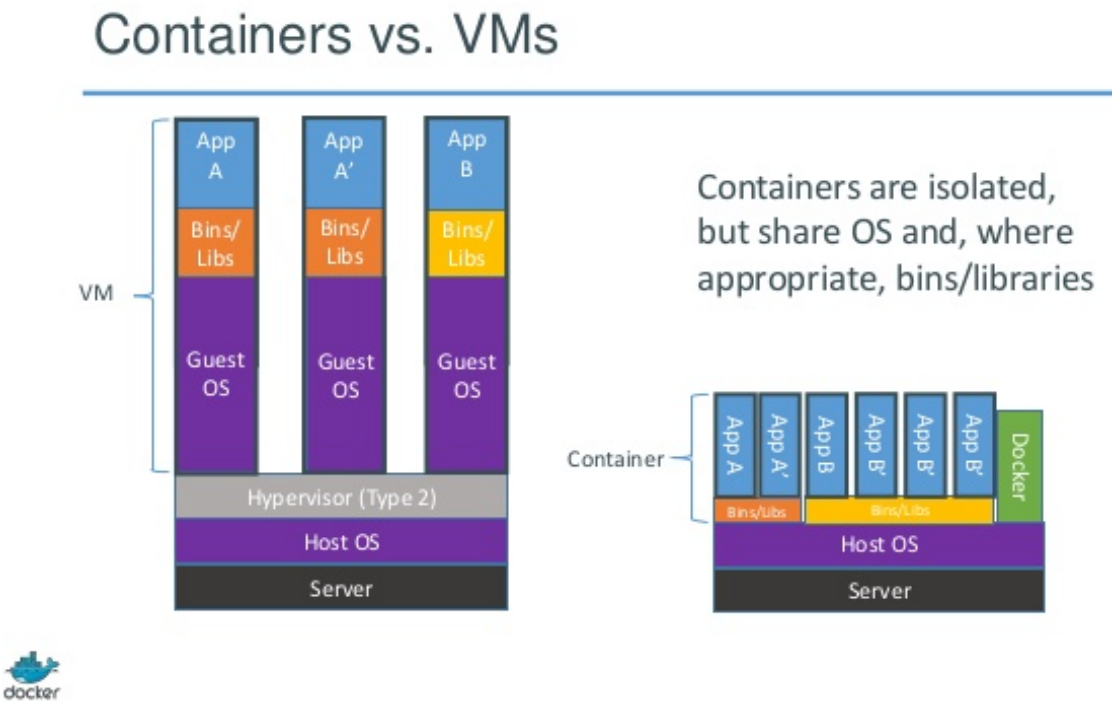

FIGURE 2.1: Structural difference between containers and VMs

The project has been created as a REST-ish API server, using LXC tools to manage the containers themselves, however LXC is, as its name shows (LinuX Container) specific to Linux-based operating systems, at that time it was even more restricted, **Docker** was only working on Ubuntu Linux amd64. This is why the "run anywhere" was a bit biased at that time. One year later, the project has abandoned LXC to create their own library named *libcontainer*, which is able to run on mostly any Linux. In the future, the project leaders want docker to be able to run on any kind of containerization system : BSD Jails, Solaris Zones, and Linux Containers, to become the real interoperable standard for containerization.

2.2.2 Its contribution to operating system level virtualization

As stated previously, containers were existing for a long time before **docker**, but this project has succeeded to create a wave of motivation and keen interest around this technology. People were so enthusiastic that **Dotcloud** pivoted their activity, to focus on **docker** and changed its name to **Docker inc.**

What has been brought by this container engine is a simple way to manage and deploy containers one a large amount of server through a simple API. This simplification took over the LXC tools which were known to be difficult to handle.

Docker also uses copy on write filesystems, it means that if different applications are using the same base file system, it only needs to be present once. This precise characteristic let users create containers in milliseconds as there is no file to copy.

Currently, thanks to **Docker inc.** containers are really fashionable, every tech companies is looking at them and their evolution and more and more people try to get rid of heavy virtual machines, because even if containers do not have all the features of virtual machines, they are good enough in a lot of cases.

2.3 Advantages

Cloud providers are looking for the best way to setup a multi-tenant architecture which is secure enough and fast enough. "Containers" is an answer to this issue. For example, the company **MongoLab** is hosting thousands of MongoDB databases. Data is something critical for any company, so **MongoLab** needs to isolate each instance of MongoDB from each other. We can assume that most of the databases they are hosting don't have a really high traffic. Having a virtual machine for each of those instances is clearly something oversized and would result on high provisioning overhead (duration of virtual machine boot), storage overhead (1 full operating system per instance), etc. This company is using containers because, it allows them to isolate the databases, to provision them instantly, and the files required to run MongoDB are only present once on their servers (physical or virtual).

The interface proposed by **Docker** is easy to manipulate and is using standard technologies. There is no need to use a custom protocol to communicate with it, everything is transiting through HTTP. There is now an easy way to isolate

applications. Compared to virtual machines, the usage difficulty level has been highly decreased.

2.4 Limits

Containers are not able to live-migrate from one host to another with a standard linux kernel yet. This feature is possible with an **OpenVZ** patched kernel because those patches implement the checkpoint/restore operations for the containers, but for a vanilla Linux kernel, it does not exist yet. Some developers/hackers are trying to clean the code of OpenVZ and push the features to the mainstream kernel with the [25] project, but so far the results are mostly drafty and unstable.

This main limit results in the difficulty to host stateful applications like a database. It can be isolated in a container but we don't have the possibility to move it without any downtime, the container has to be stop first then restarted on another host. This is particularly blocking in the case of production environment where every downtime leads to money loses for instance.

2.5 Web Application

As containerized stateful applications can not be cleanly load balanced among a set of servers (a downtime is required), stateless web applications will be targeted, as stated in the introduction of this work.

A Web application is an applicative server which uses the web standards to communicate with clients. There are two main types of web services. The websites, which are rendering HTML/JS/CSS web pages to users, and web services defining an API and answering with standard data formats like XML or JSON. Both of them are using HTTP as transfer protocol.

By the nature of HTTP, web applications are mostly stateless. Each resource request is done using a new connection (except the case of reusing opened connections). When a web application is stateful it is linked to the application itself which is linking information to a local session or connection.

These last 5 years, more and more of the web services have been written based on some or all the principles of the REST method which declares as "best practice" to create complete stateless applications. Additionally, another manifesto, the “12 Factors [26]”, has become a standard set of good practices for web development (website and web services)

The main advantage of stateless services is that they are able to scale horizontally easily: the first step is to spawn new instances of the service, and then modify the routing table of a frontal reverse-proxy. As a result the requests will be distributed among all the instances.

2.6 Application balancing in the infrastructure

When a web application has to be moved from one host to another, there should be no unavailability and the current requests have to stopped gracefully. To solve the first issue, the following walkthrough has to be followed:

1. Create a new instance of the application - Instantiate a new container of a web application
2. Wait until the instance is available - TCP ping the application until a connection is established
3. Change reverse proxy routing to route requests to the new container and not the old one
4. Stop the old container to free its resources

To solve the second issue, it should be handle by the application itself. When the system is querying the old container to stop. It actually sends a signal to it. In most systems (Systemd, Upstart at the system level, or Heroku and Dotcloud at the PaaS level), SIGTERM is sent, then the application has some time to shutdown. In the case where the application is still running a while after receiving the signal, SIGKILL is sent to get rid of the process.

2.7 Operations on containers

2.7.1 Load balancing and consolidation

The load balancing process consists in moving applications in order avoid having overloaded and underloaded hosts.

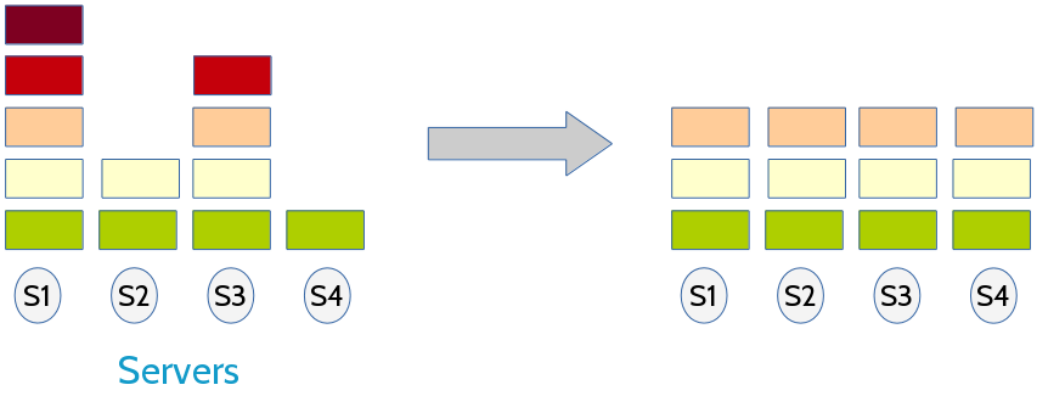

FIGURE 2.2: Schema of a load balancing process

The load balancing operation can be considered as a routine in order to avoid overloaded and underloaded servers. It has to be run periodically in order to move containers from overloaded host to more available hosts. But also to avoid servers to be nearly empty, this is the process of consolidation. In this case the server costs money and it would be better if the applications on the underloaded servers could be moved.

2.7.2 Resource Allocation

When a new application has to start on the cluster, a container has to be created. At that step, it is required to find the best server to host this newly spawned application

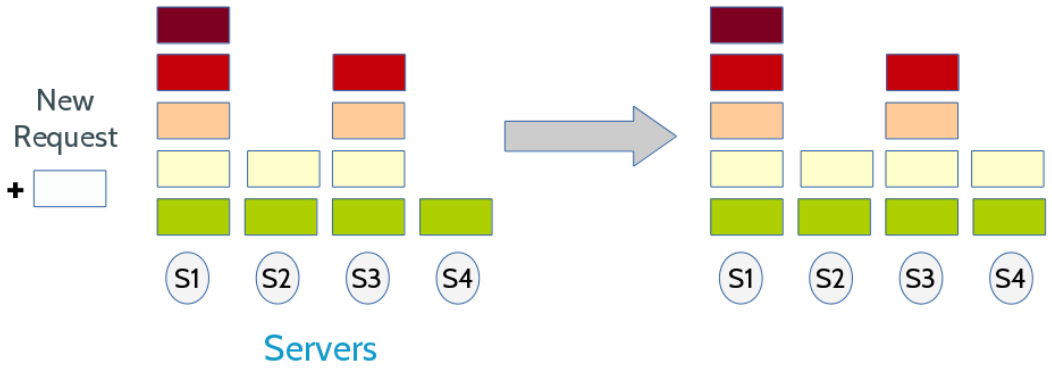

FIGURE 2.3: Schema of a resource allocation process

For that step, it is required to find the most available server, because deploying an application on a node which is already under an important load can have repercussions on all the different containers hosted on it.

Chapter 3

CPU allocation and scheduling for containerized processes

3.1 Goal of the experiment

This experiment has been defined to study the ability to isolate containers CPU usage using Linux control groups (cgroups). They are integrated to the linux kernel to apply limits on the resource access right of each container.

This experiment aims at studying how these cgroups are working and how do they actually share and isolate the CPU resources among the different containers.

3.2 Metrics

3.2.1 Input

The number of CPUs that an application consumes has to be clearly defined. In each container, an application developed to consume a given number of CPU cores will be launched. The source code of the application can be found at https://github.com/Soulou/msc-thesis-cpu-burn.

```
# Parameter n: Number of core to consume
./msc-thesis-cpu-burn -nb-cpus=<n>
```

The second input corresponds to the number of shares a container can access on the CPUs of the running computer. This number is arbitrary as the shares are relative to each other.

> If a container does not have any cpu share number specified, the default value is: 1024

It is expected that if there are two containers, one with 1024 cpu shares and the other with 2048 CPU shares, the second container will have access to 2048/1024 = 200% of the resources, for a single CPU: 33% and 66%.

3.2.2 Output

As the input concerns the maximal CPU usage, the output the experiment is going to get, each second, the actual consumption of each running container.

3.3 Setup

3.3.1 Hosts

To test the capacity of the isolation by cpu shares, two different environments will be used. As the result are expected to be relative to the hardware their should not be any major differences between both, but as a sanity test, it is important execute it on two différents contexts

The first one my personal laptop, here are its characteristics:

- CPU: Intel® Core™ i7-3537U CPU @ 2.00GHz (2 cores with hyperthreading)
- Memory: 8 GB RAM DDR3
- Disk: 256GB Solid State Drive

Then we'll study the results of the same experiment on a 4 cores virtual machine based on an OpenStack cluster:

- CPU: 4 KVM vCPUs
- Memory: 8 GB RAM
- Disk: Virtual HDD 80GB

3.3.2 Deployment

In order to simplify the reproduction of these experiments, the different applications have been packaged into container images. They can be found on the docker public repository:

- `soulou/msc-thesis-cpu-burn`
- `soulou/msc-thesis-docker-cpu-monitor`

In order to deploy them, simply install Docker on your host (http://docs.docker.com/installation/), then use the `docker pull` to get the container images locally.

```
docker run -d soulou/msc-thesis-cpu-burn -nb-cpus=<n>
...
# Run more instances according to what your want to test
...
docker run -i -t \
```

```
-v /var/run/docker.sock:/var/run/docker.sock \
-v /sys/fs/cgroup/cpuacct/docker:/cgroup \
soulou/msc-thesis-docker-cpu-monitor -cgroup-path=/cgroup
```

The cpu monitoring service will display in columns the cpu consumption of each container running on the host (including itself), the data are displayed to be quickly usable by a third-party data analysis tool like **R** or to draw graph with **Gnuplot**

3.4 Expected results

Four different experiments have been done:

- 4 Processes with equal CPU shares :
  The tested hosts have a total of 4 cores, normally 4 processes using 1 core each should be able to share it equally, and each of the process should be able to get 100%

- 4 Processes with different CPU shares 128-256-512-1024 :
  For the same reason as the previous experiment, the shares should not change the results. Even if some processes have less priority over the CPU, as there is enough cores for all the processes, they should all be able to run their process at its maximum potential.

- 6 processes with equal CPU shares :
  This case is different, as there is a higher number of processes compared to the amount of available computation units. With an equal amount of CPU shares for each process, it is expected that each process will get 66% in average of CPU time. The results can't be stable as the mount of cores is not a divisor of the number of applications. In other words, there is no way the operating system can allocate an equal share of core per process, as the context of a process is linked to one core. An application can't be 33% on one core, and 33% on the other one at the same time.

- 6 processes with different CPU shares 32-64-128-256-512-1024 :
  According to the rule defined previously, a process with 64 shares should have twice more CPU time than a process with 32 shares but twice less than a 128-shares process.

3.5 Results

All the following graphs represent the percentage of CPU time per process in function of the time in seconds.

3.6 On the laptop

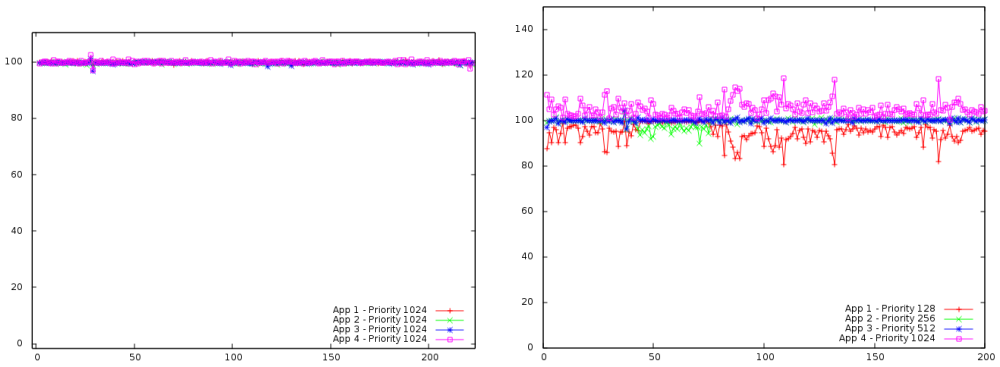

FIGURE 3.1: 4 Processes with equal[1] and different[2] CPU shares

Using 4 processes, the expectations are reached, even if there are some small differences between the execution with equal shares and the one without, it is clear that each service can use one complete core whatever are its CPU shares.

When 6 processes are executing on the host the observed behavior is different. When shares are equal, the cpu consumption of each process is completely unstable. As explain in the expectation for this experiment, theoretically each process should have 66%, but as it's not possible because a process is only attached to one core at a precise time, the operating system is moving the processes during all the calculations. This is why the curves are so changing. But overall, if we measure

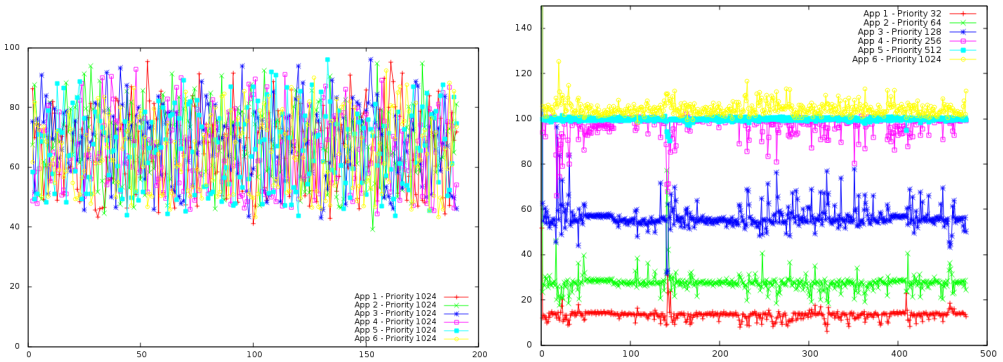

FIGURE 3.2: 6 Processes with equal[1] and different[2] CPU shares

the average and median of the CPU consumption of each application, the result is 66%, so the expectation is reached.

In the case where 6 processes are running with different CPU shares, the results are linked to what has been planned, but not only. The process with the minimum amount of shares (32) is using $\approx$ 15% of CPU, then the one with 64 shares has $\approx$ 30% of CPU consumption, and then, the third one has $\approx$ 60% of processor usage. These values are effectively each time twice higher as the previous one. However this rule is not respected afterwards. Three of the process are able to use one full core event if their shares are respectively really different (256, 512, 1024)

3.7 On the virtual machine

As previously said, the results of this experiment in another environment should not be fundamentally different. As the results are relative percentages, the same figures should be found.

When 4 processes are running, the results are not fundamentally different with the previous ones. There are some instability which may be linked to the virtual machine environment, but concretely, each process got 100% of a core.

Once more, the results are what we could expect. It is interesting that the consumption of 6 processes looks much more constant in this environment. It may

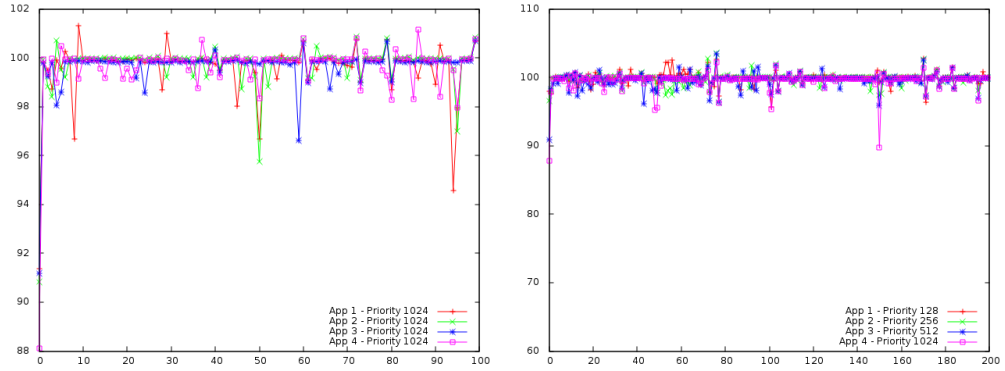

FIGURE 3.3: 4 Processes with equal[1] and different[2] CPU shares

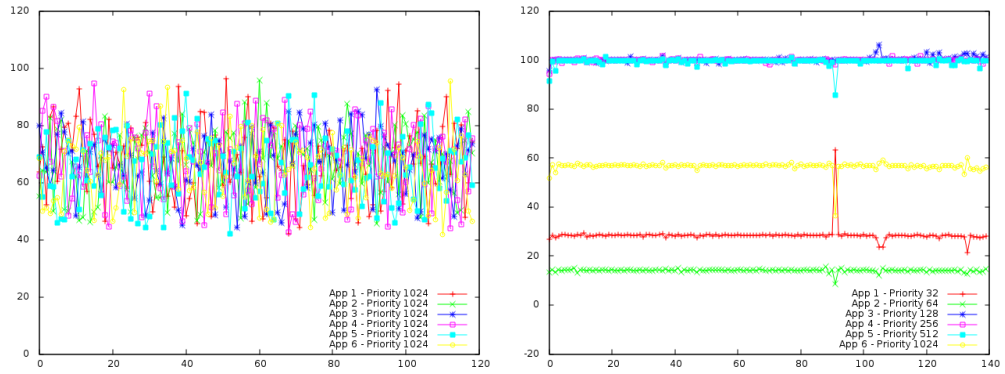

FIGURE 3.4: 6 Processes with equal[1] and different[2] CPU shares

be interesting to investigate that in order to know if it's because there are less applications running on the server than the laptop or if the vCPUs used by the virtual machine are rounding of the real CPU consumption on the physical host provided by the hypervisor. It doesn't change fundamentally the results, but there is a difference.

3.8 Additional analysis

We have seen that with 6 processes on 4 cores, 3 of them are able to get a full CPU, in this case the processes are limiting the experiment. That is why, the operation has been repeated with the following parameters.

```
docker run -d -c <shares> soulou/msc-thesis-cpu-burn -nb-cpus 2
```

In this case, each container will try to get 2 cores, so 200% of CPU.

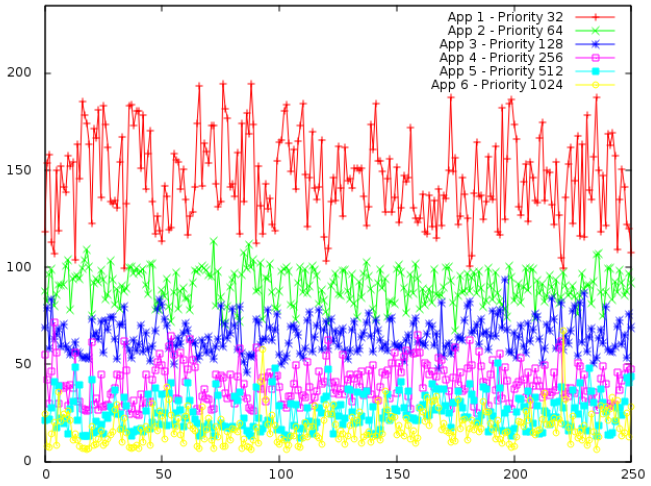

FIGURE 3.5: 6 Processes using 2 cores with different CPU shares

	App6	App5	App4	App3	App2	App1
Median	144.0	90.11	63.29	38.58	23.59	16.04
Mean	145.6	90.22	64.48	40.20	25.07	17.52

TABLE 3.1: Median and mean of the different application CPU usage in %

Compared to the previous experiment, we can first observe that the limit is not the process anymore, none of the applications have reached their maximal potential 200%. However, the observed relation between two tasks has been lost. The more shares it owns, the more CPU usage a process has, accordingly to the shares. The operating system is respecting the shares, in a best effort. The priority of one process over another can not be precisely defined.

3.9 Conclusion of the experiment

The main goal of this experiment was to show that it is possible to give different priority levels to different applications running on a given host. This is something really important because in the scope of multiuser infrastructure, resources

should be manageable. The **cgroups** already allow to set hard limit on memory consumption, but the CPU limits are different. The **cgroup** cpuset has not been mentioned in this experiment, it is another solution for CPU resource management but it is not integrated in **docker**. The **docker** developers are currently working on the addition of more resource control features but as the project is still quite recent, they are still not implemented.

The obtained results are in line with the expectations, as long as there is less processes than cores, the shares have no effect, but when the relation changes and the number of processes is increasing, the shares can have an important impact on how an application will be able to use the available cores. From a Quality of Service perspective, it allows to regulate the processes on a server base, to decrease the "Bad neighbour" effect, when one or different processes are slowing down all the other processes of a host.

Chapter 4

Definition of the experimental platform

Before speaking of the experiments themselves, it is important to define how those experiments are setup. This chapter is devoted to the design of the infrastructure used for the experimental study. The target is the ability to test algorithms using a realistic infrastructure and not to create a simulation of it.

4.1 Hardware Infrastructure

Different kinds of cloud coexist, if **Amazon Web Services** provide services which are part of a "public" cloud, this is not the only way to use a cloud infrastructure: private cloud or hybrid clouds mixing private and public cloud infrastructures are being developed more and more. Thanks to open-source projects like [27], cloud environments can be installed on private servers having commodity hardware.

In this thesis, the experiments are done on a private cloud infrastructure, powered by Openstack (version: Grizzly). But they could equaly be run on a public cloud, or even bare metal servers. The amount and the capacities of the virtual machines are different according to the experiment, but all the instances are always in the same private network.

This document won't cover how to install Openstack but there are plenty of tutorials on the web. Moreover this setup can be done in a public cloud infrastructure like Amazon Web Services EC2, there is no difference. We are going to use two distinct kinds of node. The agents, which execute the different web applications, and the controller, which is the interface to control the complete infrastructure.

4.2 Software Infrastructure

4.2.1 Operating System

All the virtual machines are running **Ubuntu Server 14.04 LTS**, this choice has been lead by the fact that this Linux distribution is probably the most standard worldwide and because *python 3.4* is required to run some libraries of the project.

The cloud version of the distribution has been chosen[1] if order to be compatible with Openstack and correctly boot. To add the image to an openstack cluster, only one command is required:

```
glance add name="ubuntu-trusty" is_public=true \
  container_format=ovf disk_format=qcow2 < /path/to/file.img
```

To deploy, run, stop and migrate our applications, **Docker** will be used. More precisely its REST API. Actually all the requests to Docker are done through HTTP requests on a unix socket. (**Docker** is using a unix socket owned by root for security reasons, to avoid remote access to the host)

4.2.2 Service discovery

One of the common difficulties in cloud infrastructure gathering numerous virtual machines, is the service discovery. It is possible, of course, to use a configuration

[1]Download page of Ubuntu 14.04 Cloud :http://cloud-images.ubuntu.com/releases/14.04/

manager to generate static settings on each node. However this system doesn't scale well and is not fault tolerant. That is what it is a bad idea to write anything statically when deploying such infrastructure.

The project **Consul** is used to achieve the feature. Consul is decentralized solution for service discovery based on two protocols. On the one hand, it is using a gossip protocol to manage the communication between nodes. This feature let **Consul** creates a decentralized cluster of servers. When a new node is available, it just needs to communicate with one node, whichever it is, to join the complete cluster and get access to the shared resources. On the second hand, the process is using a consensus algorithm to elect a leader node, which has the responsibility to keep the data consistent. Write operations have to be validated by the leader node, then spread to the rest of the servers. If the leader node crashes, another node is automatically elected by the others nodes.

Consul main usage is service discovery, so each node is registering its running services to consul which will spread the information among the whole cluster. That is how services get to know each other.

The application is developed using the **Go** programming language, so the installation is trivial. To achieve it, whatever is the operating system, downloading the binary from the website `http://www.consul.io/downloads.html` and executing it enough. The configuration of each node service is done through a set of JSON files which have to be defined in **Consul** configuration directory.

4.2.3 Balancer agent

On each server which has to execute web applications, the installation of the balancer agent is required. It is a HTTP server written using python3. The source code can be found on GitHub [2]. The installation is straightforward.

```
git clone \
```

[2] `https://github.com/Soulou/msc-thesis-container-balancer-agent`

```
  https://github.com/Soulou/msc-thesis-container-balancer-agent
cd msc-thesis-container-balancer-agent
virtualenv -p /usr/bin/python3 .
source bin/activate
pip install -r requirements.txt
```

As the agent has to communicate with **Docker** is should be run as root:

```
sudo -E python agent.py
```

The server has two distinct roles. The first one is to execute instructions coming from a controller, the interface is an HTTP API. You can find its documentation in the Appendix A: HTTP API of the agent server

The other role of the agent is to achieve real time monitoring of the server itself and of each container running on it. Different threads starts in parallel with the HTTP Server. The reason why it is necessary to use separate threads is that at a given time it's not possible to get some relative data. This is how the different metrics are gathered by the agent for the host:

- CPU: The interface from the Linux Kernel to read the CPU usage is `/proc/stat`. When this virtual file is read, the kernel fills it with the current information about the CPU usage, the interruptions and the processes. However those data are cumulative. So each second the data are fetched, and compared to the CPU usage of the previous second. The data are un *User Hz*, this unit represent a tick in the user space, 100 of them are generated per second, so the value shown in this file are close to hundredths of second.

- Memory: This value is easier to access, the kernel provides `/proc/meminfo` which contains the real time data usage. There is no extra work to do in order to the clean pieces of information.

- Network I/O: `/proc/net/dev` contains all the information related to all the network interfaces of the server, in this file is displayed the amount of bytes

and packets sent and received by each of them. The values are also cumulative, that's why the agent has to keep track of them. As a result when a request is done to get the system usage, the right data can be sent directly and it's not required to wait 1 second.

For the containers:

- CPU: The CPU usage of each container is accounted separately thanks to the *cpuacct* cgroup feature of the Linux Kernel. The communication from the userspace is done through a virtual file system located at `/sys/fs/cgroup`. As a result, for docker we can find the correct data at that path: `/sys/fs/cgroup/cpuacct/docker/:container_id/cpu.usage`. As precedently, the value is in *User Hz*, so the process to calculate the actual CPU usage is similar as the `/proc/stat` analysis.
- Memory: The cgroup *memory* manages the memory usage and limits per container, it is enough to read */sys/fs/cgroup/memory/docker/:container_id/memory.usag* to get the interesting piece of information.
- Network I/O: It is a bit more difficult to monitor the network usage of a container, as the resource management mechanisms is not part of the cgroup, it is another feature of the Linux Kernel called "network namespace". In order to get access to it, different steps are required.
  1. Find the PID of a process in the monitored container by looking in `/sys/fs/cgroup/:cgroup/docker/:container_id/tasks`
  2. Access the network namespace file located in `/proc/:pid/net/ns`
  3. Create a link of this namespace to `/var/run/netns/:container_id`
  4. Use IP command to get stats from the desired namespace `ip netns exec :container_id netstat -i`

To sum up, three distinct threads are running in the Balancer agent, the HTTP server, the host system monitoring, and the containers monitoring. Those threads

are used to be able to get accurate data instantly, otherwise 1 second should be waited to get interesting data.

3

4.2.4 Balancer controller

The second main brick of this infrastructure is the controller. The role of this software is to control all the different agents, to give them the instructions about which container to start and which container to stop. There is no need to configure this service particularly, the knowledge about the composition of the infrastructure is acquired through requests to the local **Consul** agent.

The running applications on the cluster are gathered by *service*. Each of them can contain a variable amount of containers hosted on the different agent.

Moreover, it updates dynamically the routing tables of the different services running on the infrastructure. If a service called "service-test" owns two containers, the incoming requests will be routed to both of those containers by following a round-robin algorithm. (C1 - C2 - C1 - C2 - ...).

Hipache[4] is used as a front HTTP working as reverse proxy, it is in charge of routing incoming requests to the different containers. Why **Hipache** has been used instead of a more classical server like **Apache**. The main reason is that **Hipache** is dynamically configurable thanks to different backends. The most common is the **redis** backend. **Redis** is a key-value store with really high performance as all the dataset stay in memory, and is asynchronously written to the disk. So when the controller sends request to start or stop a container, it also connects to a **redis** instance to update hipache configuration.

The controller also exposes a HTTP API which is described in the Appendix B HTTP API of the controller server. Part of this API is only a proxy to the

[3] https://github.com/Soulou/msc-thesis-container-balancer-agent
[4] https://github.com/dotcloud/hipache

agents endpoints, in order to aggregate all the cluster data. On the other side, there are the active actions, linked to container scheduling in the infrastructure.

This component of the infrastructure is critical for the experiment detailed later in this work. The controller is charged to choose the nodes where the containers have to start. To select a strategy which will be used when a new controller has to start, it has to be specified as options:

```
python3 controller.py --strategy random
```

In the previous example, the new containers will be placed randomly on the different available nodes. Others strategy are available:

- random: Choose randomly a node
- round-robin: Choose one node after the other, following a loop

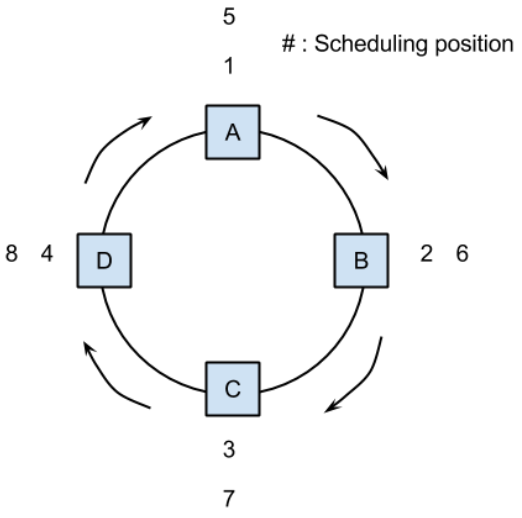

FIGURE 4.1: Scheme of a round-robin scheduling

- first-fit: Use the online bin packing algorithm First-Fit
- best-fit: Use the online bin packing algorithm Best-Fit
- worst-fit: Use the online bin packing algorithm Worst-Fit
- best-worst-fit: Use a mix of best and worst fit
- first-worst-fit: Use a mix of first and worst fit

4.2.5 Balancer client

The balancer client, is a command line tool to command the controller, it shows in a human-understandable way, the JSON results the controller and allows a user to execute requests easily. Its documentation can be found in the Appendix C: " Client command line tool documentation".

[5]

4.3 Deployment

All those services and tools have to be deployed on the instances of the infrastructure. More and more configuration managers (CM) are released those days to answer the increasing needs to deploy applications in volatile infrastructures, composed of temporary virtual machines. To setup all the different pieces of this infrastructure, **Ansible** has been used. There are two main models of CMs, it may be centralized around a server, like **Chef Server** or **Puppet**, where the nodes synchronize themselves with this server. The devops do not connect directly to the nodes but give instructions to the central server. The second category are those who are run directly from a devops workstation, like **Chef Solo**, **Ansible**, **SaltStack**. The choice of **Ansible** has been done for it's ability to deploy nodes in parallel and by it's easiness of prototyping, its syntax is pretty simple.

All the mecanisms of deployment can be found on following *GitHub* page: https://github.com/Soulou/msc-thesis-cloud-builder. The script requires that **Openstack** credentials are stored in environment variables. (`OS_USERNAME`, `OS_TENANT_NAME`, `OS_PASSWORD`, `OS_AUTH_URL`, `OS_REGION_NAME`). Then it uses the API to instantiates VMs, and then deploy all the services on them using ansible [6].

[5] https://github.com/Soulou/msc-thesis-container-balancer-client
[6] Ansible configuration: https://github.com/Soulou/msc-thesis-ansible

Chapter 5

Study of algorithms for containers allocation and load balancing

5.1 Experimental applications

The experimental process aims at measuring the performance of a set of containers before and/or after using any resource allocation algorithm. First, we have to define which applications will be containerized for those operations. Those applications have to be deployed on all the agent servers and have to behave as standard web services. Whatever is the language or the framework used (except PHP), all the dependencies of the applications are loaded on application startup, so afterwards, there is no more file to read from the disk. It results in low input/output for most of the applications and this metric will not be measured.

The first service which has been used during the experiments is an application which is calculating N elements of the Fibonacci suite[1]. This web application has only one endpoint: `GET /:n`, which returns the N^{th} items, of the Fibonacci sequence. This application only consumes CPU, the memory footprint is really negligible.

[1] **Docker** image: `soulou/msc-thesis-fibo-http-service`

The second application has been designed to consume a precise amount of resource during a specified duration[2]. Its endpoint is `GET /:memory/:time` with the memory in megabytes and the time in milliseconds. For instance, `GET /10/100` will consume precisely 10MB for a duration of 0.1s. Thanks to this application, we can measure precisely how much should consume the application with a load of N requests per second of a certain amount of memory during a precise timelapse. The memory is allocated and initialized. As a result, even if we are requesting amount of memory, it also generates an important CPU load in the loops for memory application.

The source code of both services can be found on **GitHub**:

- https://github.com/Soulou/msc-thesis-fibo-http-service
- https://github.com/Soulou/msc-thesis-memory-http-service

5.2 Load generation

An important question when measuring performance is the way to generate the requests. Do they have to reflect a realistic user load, is it possible to do it, is it pertinent? In this case, we are going to measure raw performance over the cluster, to gather data about the efficiency of a specific algorithm. When user load simulation algorithm is used, the benchmark is irregular, and finally getting information about the real impact of a given algorithm may be more difficult to do.

5.2.1 Tools

To generate web traffic, HTTP requests generators are required. Historically, the most used tool is part of the **Apache** web server tools, and is called Apache Bench *Command line name: ab*, released under the open-source *Apache Licence*. This

[2] **Docker** image: `soulou/msc-thesis/memory-http-service`

utility is not really used anymore because it is single-threaded which. This is a very limiting caracteristic, only one CPU core is used, it may not be enough to saturate a target and get a correct measure of the performance.

The tool which has been mostly used in the scope of this work is named **wrk**[3]. This is a benchmark tool able to send requests using a given number of connections, used in different parallel threads. (The opposite of `ab`, which sends all the request concurrently using one thread)

Usage example:

```
wrk -c 10 -t 2 -d 1m http://service1.thesis.dev
```

This example will send 10 requests concurrently during 1 minute using two threads. So we are sure that the targeted URL will receive a maximum of 10 request at a given time.

One reproach which can be made to this tool is that it doesn't adjust the number of threads automatically. By default, two threads are used, but if we need more parallel connections, the user has to define it by hand. But is most of the case, giving a thread per core on the underlying computer is the best practice.

5.2.2 Different kinds of load

Combining **wrk** and the two previously defined web application, different kinds of load can be generated. Thanks to the "memory" HTTP service, it is even possible to emulate different kinds of application.

5.2.3 Lightweight services

In some cases, web applications are micro-services and their job is to do a small particular task. Commonly, requests are really quick and the treatment of each

[3] https://github.com/wg/wrk

of them has a low memory footprint. However as those HTTP requests may be numerous, the overall processor and memory usage can be important.

In out infrastructure, one container of such a service may be represented by instances of the memory services with requests using a few megabytes of memory, are done really quickly: `http://micro-service.thesis.dev/5/100`. All the requests to this endpoint are compliant with the previous constraints. The following data show how such application behave in an environment where they don't have to share resources[4]:

- One instance:
  `wrk -c 20 'http://memory-service.thesis.dev/5/100'` — 82.05req/s — CPU 106%, 24MB
  `wrk -c 40 'http://memory-service.thesis.dev/5/100'` — 93.74req/s — CPU 112%, 22MB
  There is no big difference when running 20 or 40 connections in parallel, so we can assume that we have reached the maximum capacity of the instance. As announced previously, the memory usage is really low, but the application is using one complete core.

- Two instances on two different hosts:
  `wrk -c 20 'http://memory-service.thesis.dev/5/100'` — 123.23req/s — CPU 109% | 80%, 15MB | 14MB
  `wrk -c 40 'http://memory-service.thesis.dev/5/100'` — 165.77req/s — CPU 117% | 97%, 17MB | 31MB
  The exepectations were that two instances would be able to execute twice the number of requests compared to the previous case. With 20 connections, it is not the case, but with 40, the service has been able to execute 165 requests per second, which is more or less the double as previously. 20 connections was not enough. This lack can be seen by the fact that the CPU usage of one of the instances is not one complete core but 80%.

[4]All the measure have been done 5 times, and the given result is the average

- Two instances on the same host:
  `wrk -c 20 'http://memory-service.thesis.dev/5/100'` — 101.11req/s — CPU 97% | 98%, 17MB | 18MB
  `wrk -c 40 'http://memory-service.thesis.dev/5/100'` — 127.77req/s — CPU 115% | 117%, 18MB | 17MB
  The instances have 2 vCPUs, so theoretically two instances of a same service on one node should be able to run at maximumal performance, but in practice, the results are not as good, a ratio of 1.5 has been reached using 40 parallel connections. This is a perfect illustration of a "bad neighbour" situation, the performance of each container is jeopardized by the other instance. That is something which can be (at least partially) solved with scheduling algorithms, to avoid as much as possible this case.

Note: The values of CPU usage are often above 100% when an application is using on core. The amount of *User HZ* is read every second, so it may be an inacurracy of this timer, or it may be that the time used by the kernel to schedule processes is taken into account. As a result, the process uses 100% and the kernel uses some CPU time on another core.

5.2.4 Bigger services

The opposite of previous microservices are more important web services, consuming more memory per request, with a lower number of request per second:

- One instance:
  `wrk -c 20 'http://memory-service.thesis.dev/50/500'` — 10.79req/s — CPU 122%, 326MB
  `wrk -c 40 'http://memory-service.thesis.dev/50/500'` — 9.30req/s — CPU 116%, 293MB

- Two instances:

  `wrk -c 20 'http://memory-service.thesis.dev/50/500'` — 19.48req/s — CPU 122% | 80%, 259MB | 182MB

  `wrk -c 40 'http://memory-service.thesis.dev/50/500'` — 23.26req/s — CPU 102% | 98%, 302MB | 283MB

- Two instances on one node:

  `wrk -c 20 'http://memory-service.thesis.dev/50/500'` — 19.48req/s — CPU 109% | 106%, 153MB | 170MB

  `wrk -c 40 'http://memory-service.thesis.dev/50/500'` — 19.43req/s — CPU 105% | 101%, 202MB | 195MB

Except the fact he the servers suppor less requests as each of them is heavier, the behaviour between a light application and this one is prett similar. It scales out correctly, doubling the mount of requests the service can manage. With this application even if the memory consumption is getting higher, the limiting resource is still the processor. The fact that the CPU is in most of the case the resource causing a bottleneck is something common. [7, 13]

In the previous examples we have done performance benchmark, the maximum performance of the services have been measured. But with **wrk**, it is possible to send a lower amount of requests to simulate a normal load. When 1 to 10 connections are used (instead of 20 or 40) like in the previous experiment, the resulting CPU consumption would vary respectively between 10 and 100%

5.3 Bin packing algorithms

Bin packing algorithms have been defined in the Chapter 1, but here is a short reminder about the main ideas behind them. A bin packing algorithms consist in

packing items into bins. Each bin has a capacity and each item weighs certain size. There are two main categories: online bin packing and offline bin packing. The first type is defined by the following setup. The bins are already filled with items, but we can't have details about those items. There is a new item to pack, the online bin packing algorithms have to choose the best bin for the new item. The offline bin packing algorithms consider that all the bins are empty, and it has to pack all the items into the bins. All the information about each item is available.

We speak of bin packing algorithms with the plural, because there isn't only one way to pack items, offline bin packing is an NP-Hard problem, finding the optimal solution can be very long. That's why a large number of heuristics exist to get a good enough result in a short enough period.

5.3.1 Relation with resource allocation

In our situations, the items are the appications containers and the bins are the different hosts, here OpenStack virtual machines. Different operations can be done when packing application containers:

- Consolidation: Use the minimum of hosts to run correctly all the containers
- Load balancing: Balance the load over the available hosts
- Container Allocation: Choice of the best node to run a new container

First, it is important to define the fact that the resource consumption of a container is completely ephemeral. As the resources are not completely dedicated to a container, but shared with each other, the container may have to be moved soon after its allocation, when its load has been defined. That's why using only resource allocation is not enough to balance correctly the infrastructure.

As the consumption of CPU is variable, it is important to keep a security reserve. If the amount of requests to a container increases, it has to be able to get more

CPU time, otherwise it will the stucked and all the requests would be impacted. That's why, there is a default margin of 20% in the platform we are using. It means that if a server is loaded up to 50% and that a new container estimated at 40% has to be packed. The server wouldn't be considered as available as the total is over 80% (for a one core server)

5.3.2 Container allocation — online bin packing algorithms

5.3.2.1 Type of new items

Online bin packing means that we have to pack new items into bins without having the knowledge of what is already present in the bins. So bins have a remaining capacity, and each new items is known or partially known. In some cases, the new items are completely unknown.

In this infrastructure there are two different cases which can be distinguished:

- The new container is part of a new service: there is no information about the number of incoming requests, and what is the memory footprint of the new process
- The new container is from an existing service: in this case, the CPU consumption and memory usage can be estimated by looking the other containers of a similar service. For instance, if two containers of `service1` are running using both 20% of CPU, it is probably true that the consumption of the new container will be capped at 20%

5.3.2.2 Algorithms and Heuristics

In the following part, different algorithms are going to be used. Some dummy methods will be used as comparative methods. They are 'random' and 'round-robin'. With those strategies, no bin packing is executed, the choice of the server is determined randomly or through a round-robin loop.

Three basic algorithms of bin packing are used:

- First Fit: The item is packed on the first bin which is able to contain it
- Best Fit: The bin which is able to contain the item with the smallest capacity
- Worst Fit: The bin which is able to container the item with the biggest capacity

Our nodes and containers are monitored over three metrics: “CPU usage”, “Memory usage”, “Network I/O”. As it has been shown previously, the only resource wich is a bottleneck from far the the CPU. As a result the algorithms have been implemented the following manner: when the item is inspected to see if it can fit on one node, all the metrics are compared, if one of them is insufficient, the allocation on this node is rejected. However in best fit and worst fit, when we need to select the best, or the worst, only the CPU is considered.

5.3.2.3 Experiment - Comparaison of algorithms

This experiment consists in starting a set of containers using different algorithms to choose the destination node. There are 8 virtual machines with 2 cores, 7 of them are agents running the container and one is controller and load balancer to route the HTTP requests to the instances. The containers are splitted into 3 services of various sizes:

	Number of containers	Memory Usage	Request Duration	Parallel connections
Service 1	4	10MB	10ms	30
Service 2	4	25MB	200ms	20
Service 3	2	50MB	300ms	10

Table 5.1: Description of the experimental HTTP services

A container has been launched every 5 seconds to let the time to the monitoring system to take it into account and to be able to measure its usage. The HTTP

requests are sent even before the first container has been created, be cause the front-end server **Hipache** will drop the request answering the HTTP error *502 bad gateway*. As soon as a container from the given service is available the requests are correctly routed to it.

The metric which is measured is the one important from an user perspective. The number of requests answered per second. We are expecting worst-fit, random and round-robin to get better results that first-fit and best-fit because they don't try to optimize the number of used servers, but they will spread the services on all of them.

For each algorithm, 5 executions have been done and the displayed value is the average of the the result of each of the tries. One execution lasts 2 minutes, and the obtained value is the amount of requests answered per second.

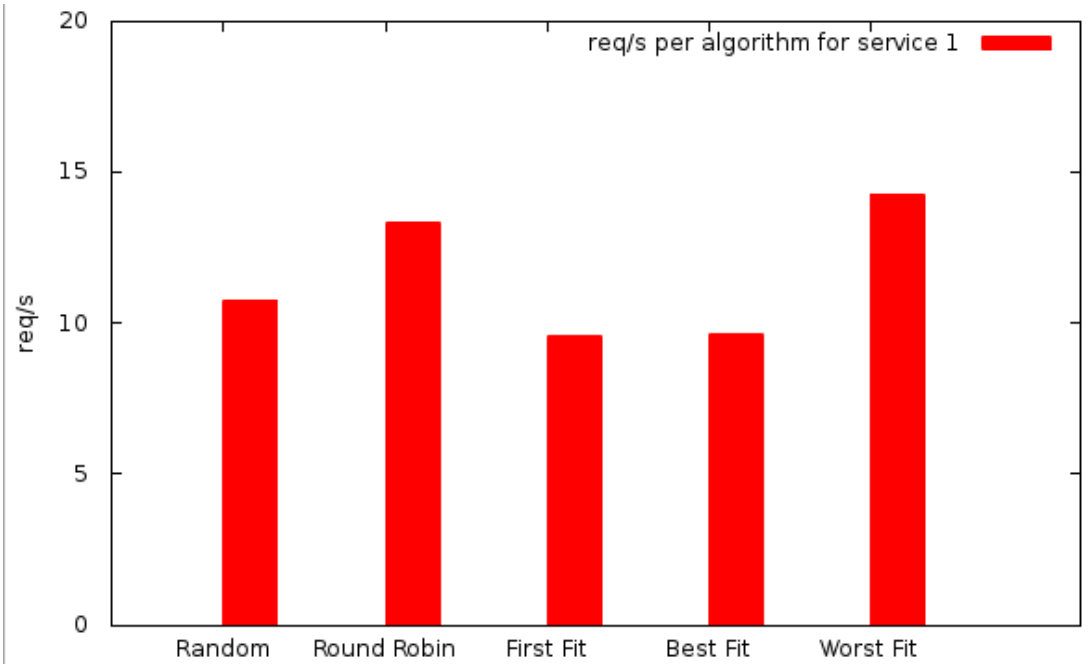

FIGURE 5.1: Service 1: Number of requests per second for various algorithms

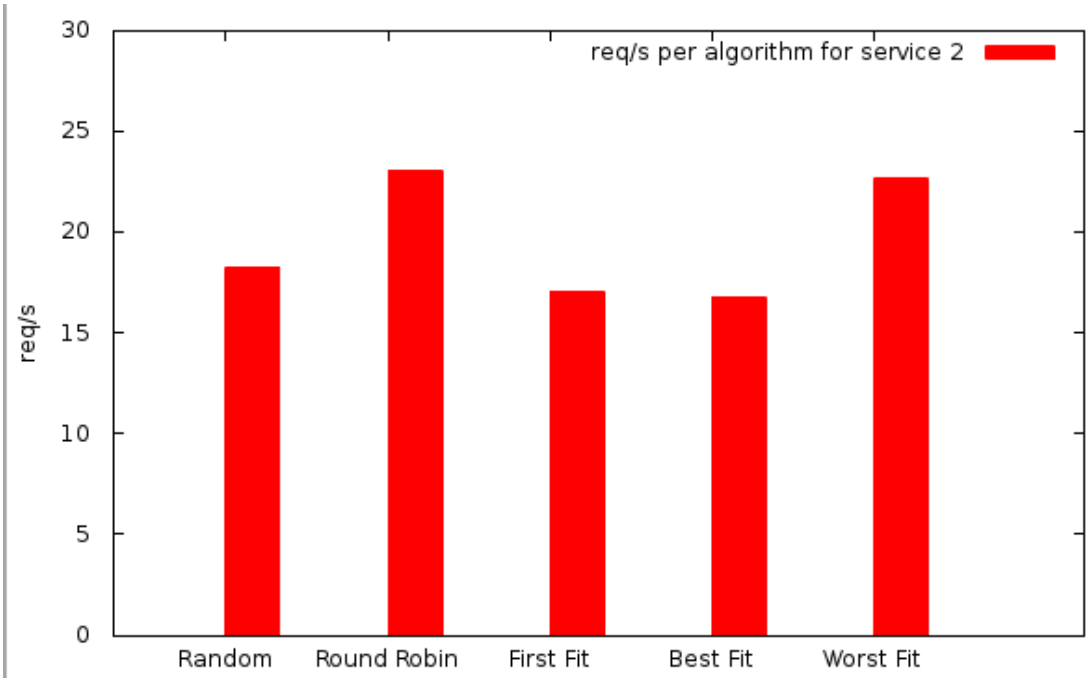

FIGURE 5.2: Service 2: Number of requests per second for various algorithms

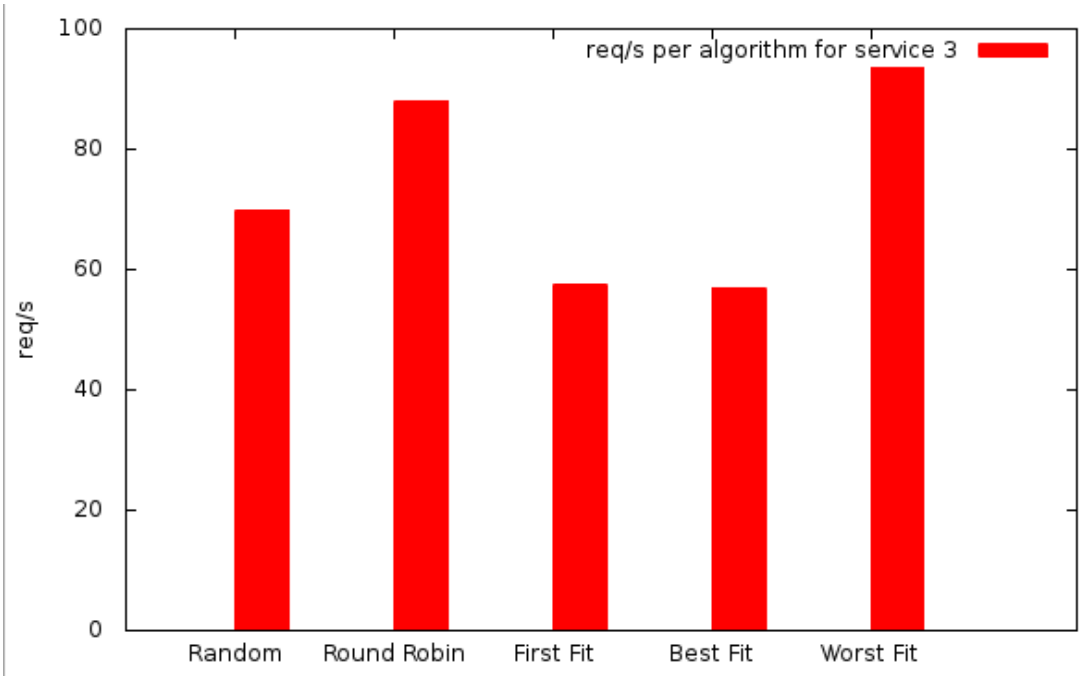

FIGURE 5.3: Service 3: Number of requests per second for various algorithms

Algorithm	Number of used nodes	Performance rank
Random	7	3
Round-Robin	7	2
First-Fit	4 or 5	4
Best-Fit	3 or 4	5
Worst-Fit	7	1

TABLE 5.2: VMs allocation summary

5.3.2.4 Results and discussions

A first remark about those results is that, whatever is the service, and the amount of requests it has to deal with, they behave the same way for each algorithm. The graphs have the same shape with another ratio because some container deal with much more queries.

Then, we can observe that as expected the best-fit and first-fit algorithms are resulting to poorer performance. But what is shown in the allocation summary is that those algorithms managed to use less virtual machines to pack all the processes. They allocate resources on less than 5 of the 7 availables nodes.

Here is one of the biggest challenge, to get close to the highest performance while using the minimum number of bins. There is direct solution for this problem it really depends what is the main goal of the operation. If it is money saving, reducing the number of virtual machines has a meaning, but the quality of service, as the performance, should be high enough to respect the SLA.

Something which is more surprising is that an algorithm considered as "intelligent", which is worst fit, is only doing slightly better, and sometimes even worse, than a simple algorithm like round robin.

The different *-fit algorithms need an estimation of the size of the incoming item to pack. As stated previously, if the containers is part of an existing service, the average of the usage of the other containers of this service will be used. Otherwise, no estimation is done. This second condition seems to damage those algorithm results.

To get over it, Some hybrid *-fit algorithms have been tried:

- Best-Worst Fit: This algorithm is using best-fit all the time except when the new item is from a new service, in this case, worst fit is used to pack it
- First-Worst Fit: Similar to best-worst fit but using first fit instead of best-fit

These variants should allocate more VMs than the base algorithms they are based on, but they should be much more efficient for the same reason. As we have seen

that the results are mostly identical for the three differents services, we will focus on the third one, it is not worthy to study the three of them.

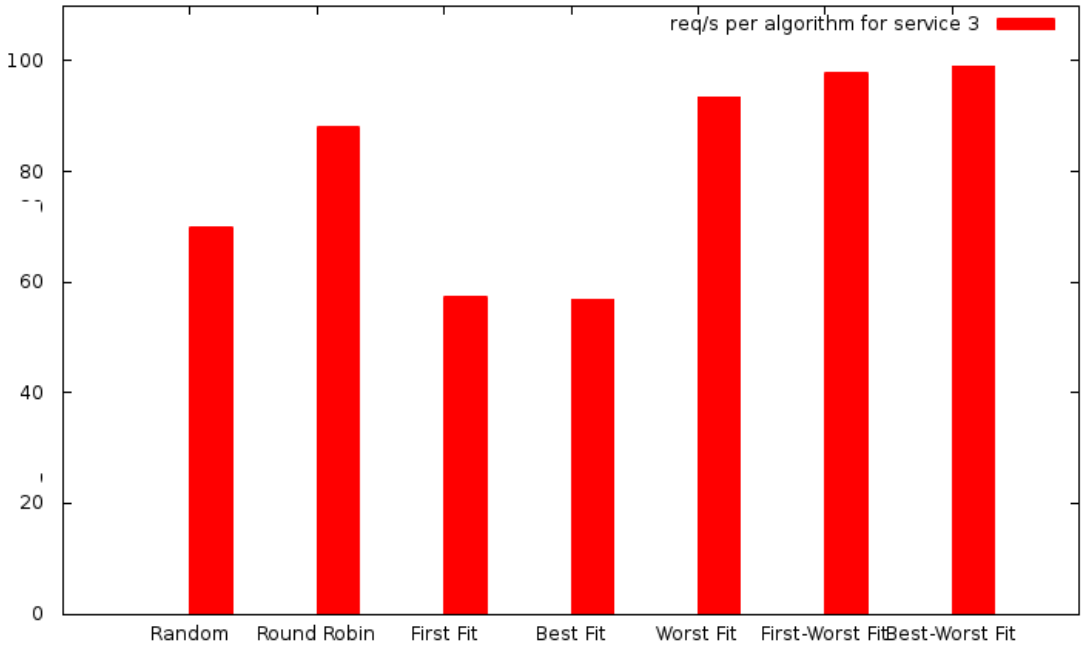

FIGURE 5.4: Comparaison of *-fit algorithms performance

Algorithm	Number of used nodes	req/s for service 3	Performance rank
First-Worst Fit	6 or 7	97.80	2
Best-Worst Fit	5 or 6	99.05	1
Worst-Fit	7	93.41	3

TABLE 5.3: Allocation summary with variants of *-fit algorithms

With these new results, we have successded to get slightly better results than the standard worst-fit, and highly superior than the base algorithm each of the variant is based on. First-Worst Fit and Best-Worst Fit are getting more or less the same results but they are not using the same amount of nodes. From that point, best-worst is clearly the best algorithm among them. We could alter those algorithms with some additional steps like sorting the bins before starting the pack operation like it has been done in [9]. However in this precise case it would be without any effect. First-fit with ascending sorted bins result in Best-fit and when the sort is descending it leads to Worst-fit, and we already know the results of these methods.

We have seen that different methods exist, but to get a good ratio performance/node usage, it may be required to mix the algorithms according to the situation and the execution environment. In our case, a mix of Best-fit and Worst-fit has definitely got the most interessant results.

5.3.3 Consolidation and load balancing — offline bin packing algorithms

5.3.3.1 Impact on the infrastructure

When an offline bin packing algorithm is applied on computing infrastructure, to apply the new configuration, it results in a lot of migrations. When we are dealing with full virtual machines, a migration has a cost and it is difficult to imagine migrating several instances from one same host in parallel. With containers, this issue doesn't exist and even less with stateless application

When a container has to be migrated, a new one is being started on the destination server, then the routing table to the service is updated and finally the source container is stopped. This operation can be done in less than a second. Compared to VM migration which can last minutes and use a lot of disk, network and CPU resources, the process which is used here costs almost nothing.

If migrations were costly, caution should be taken when choosing the algorithm, to avoid to do too many of them.

5.3.3.2 Algorithms

In this case, testing simple algorithms like round-robin or random would not be interesting. They would result in the same data as what we have seen in the part about online bin packing.

Offline bin packing algorithms are close to the online bin packing algorithm, the differences are that we have all the pieces of information about the running containers so there is no need to estimate any value. Then, there is not only one item to pack anymore, but a list of items. An additional question is in which order do the items have to be considered by the algorithms. In [7], [28] and [29] they all

refer to first-fit decreasing or best-fit decreasing as reference algorithms. It means that the data are used from the biggest to the smallest.
As a result 4 algorithms will be tested: Best-Fit Decreasing (BFD), First-Fit Decreasing (FFD), the algorithm proposed by Michaël Gabay in [29] and the variant detailed in [9] where items are sorted decreasingly.
The used parameters are the following:

```
Gabay: heuristic=bin_balancing
Gabay: heuristic=bfd_item_centric
Gabay: heuristic=bfd_bin_centric
Stillwell: item_sort=d:maximal, pack=pack_by_items
```

5.3.3.3 Experiment

The following experiment is based on the same services as the previous experiment. This choice has been made because they have already been benchmarked, and their range of performance is already known. Refer to the Table 5.3.2.3.
To obtain the data of the following graph, the underneath experiment has been done 5 times for each algorithm:

1. Start the load generation against the three services
2. Deploy containers using a resource allocation algorithm which is not optimal. Best-fit has ben chosen because of the low result it has given in the previous experiment. This situation leads to overloaded nodes
3. Run a first benchmark to get the performance before the offline bin packing algorithm execution
4. Balance the infrastructure with the chosen algorithm
5. Run a second benchmark to get the performance after the operation
6. Calculate the average of the performance variation for the three services

Then once the previous protocol has been done 5 times, the average of the result is the value displayed on the plot.[5]

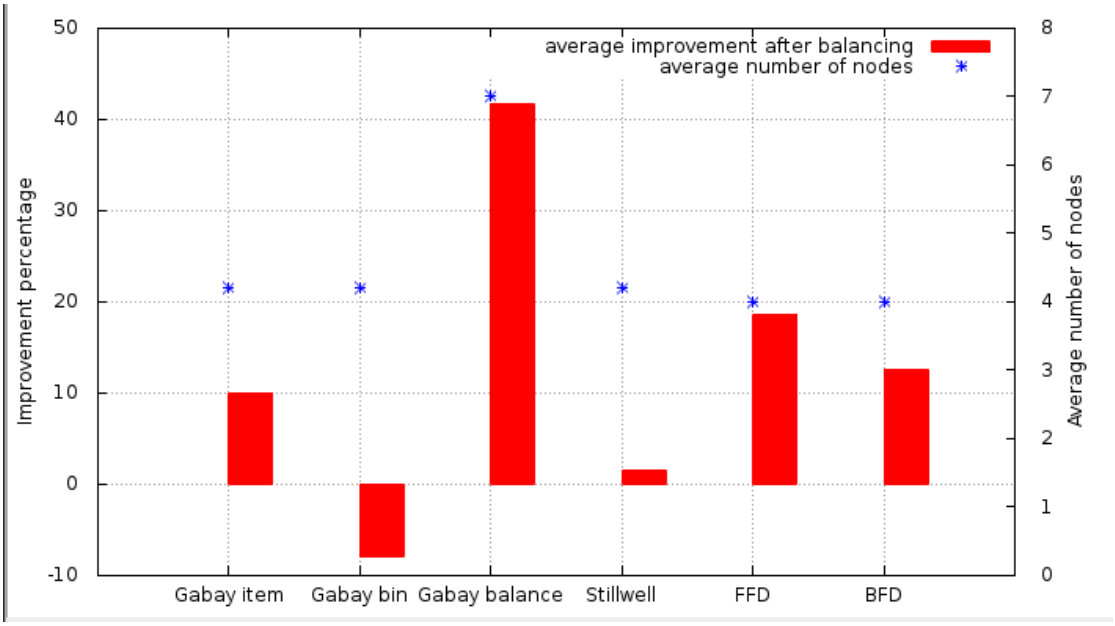

FIGURE 5.5: Performance improvement in function of the algorithm

5.3.3.4 Results and Discussions

The results are quite surprising. If *Gabay balance* is shelved, the different results are using the same amount of nodes, but their result are unexpected. [9] and [29] based their work on BFD and FFD, and build variations of them to fit specific need, which is to work in an heterogeneous environment. Obviously it was not the case here, but we could expect that those algorithms would have done at least as good as BFD or FFD.

It is also possible that this error is due to a bad integration of the algorithm in the container platform. Extensive testing would be required to find if the behavior is buggy. It may be linked to how the data are normalized. Everything is based on the bins. The server which as the highest capacity results in the unit bin, with 1 as capacity for each of his metrics, then all the containers consumptions (items) and other bins are normalized using this reference. After this step, a reserve is subtracted from the capacity, to avoid the algorithms to pack servers

[5]The detailed results for the experiment can be found here: https://github.com/Soulou/msc-thesis-container-balancer-client/tree/master/experiment4

up to 100% again. By default this reserve is 20%, and it seemd it suits well the implementations of BFD and FFD. But Gabay's and Stillwell's algorithm are not working as they should. According to their estimation in their work, they should both be more performant than BFD and FFD.
The heuristic *Gabay Balance* got great results which are really closed to a simple worst fit allocation, all the nodes have been used and no effort has been done to limit that. But, as a results, great performance is present. If the goal of the operation is to balanced the load of the server, this approach is probably the most efficient.
Comparing the two basics algorithms, FFD got slightly better results than BFD, the difference is higher than the difference of those algorithms for online bin packing, but the trend is similar.

Finally, the platform has proven that it is possible to get interesting metric from it. In the previous experiments, containers have been deployed and migrated with success resulting in performance variations. The three main operation linked to resource allocation/re-assignment can be tested using this infrastructure. If the online bin packing results are interesting, the data collected for offline be packing are not resulting to a spcific recommandation.

Conclusion

The purpose of this work has been to explain why containers are a new efficient way to isolate and deploy web applications. Then, we have seen that it is possible to apply quotas to restrict how a container behaves. The next step has been to design a software platform, aiming at deploying containers easily into a cloud environment, but also, to be able to move them between all the available nodes. Furthermore, this set of server appplications are gathering real time metrics about the running containers and servers. With these data, bin packing algorithms have been made possible and the last chapter of this thesis has shown different experiments concerning different ways to use them to allocate resource, consolidate the infrastructure or balance the load of the servers.

The developed platform has shown that it is able to manage efficiently containerized web applications. This feature has lead to interesting results concerning resource allocation. When a new container has to be instantiated, a mix of simple bin packing heuristics has given really interesting results. Combined together, best fit and worst got the best ratio “number of used nodes by number of answered requests per second”

The field of resource allocation is getting more and more important now, because infrastructures are getting more and more heavily distributed and I have the feeling I have only scratched the surface of what can be done. However, I think the developed platform is good enough and could be extended in order to be used for further experiments.

This writing has principaly focused on the obtained performance for containers able to run without any limit. This is something which is really common. A lot of

service providers are based on a “fair use” policy. In this case they do their best to run everything, but they can warn or give explicit quotas to users abusing this statement. If we get outside this scope, other experiment should be done to get new results.

Working in a real environment instead of a simulation changes a lot of settings, I think that with the developed tools, a lot more testing and experiments could be done, especially concerning load balancing. Another possible extension could be statistical predictions about the future consumption of a service, based on statistical and probabilistical models.

A particular effort has been made to make this work reproducible. All the tools are open-source softwares and protocols standard. Thanks to **Ansible** the environment should be easy to deploy on any infrastructure, physical, virtual or hybrid. Even if the obtained results in another deployment of the platform would be slightly different, it should follow the same trends as those obtained in this thesis.

Appendix A

HTTP API of the agent server

```
/*
 * Get current system info of the node itself (CPU, Memory, Net)
 */

// GET /status

// Code: 200 - OK
// Content-Type: application/json
{
  "cpus": {
    "cpu0": 44,
    "cpu1": 22,
  },
  "memory": 27440000,
  "network": {
    "eth0": {
      "tx": 98765,
      "rx": 12345
    }
  }
}

================================================

/*
```

```
 * Get list of all running containers on the node
 */

// GET /containers

// Code: 200 - OK
// Content-Type: application/json
[
  {
    "Id": "0123456789abcdef",
    "Image": "soulou/msc-thesis-memory-http-service",
    "Ports": [
      {
        "PublicPort": 49127,
        "PrivatePort": 3000
      }
    ],
    "Names": [ "service1-1-837" ],
    "Created": 1723454345,
    "Status": "Up"
  },
  {
    ...
  }
]

================================================

/*
 * Get information about a specific container
 */

// GET /container/:container_id

// Code: 404 - Container not found

// Code: 200 - OK
// Content-Type: application/json
```

```
// Docker JSON representation of a container:
// See: http://goo.gl/JrR6f6

================================================

/*
 * Get information about the resource status of a container
 * (CPU, Memory, Net)
 */

// GET /container/:container_id/status

// Code: 404
//   - Container not found
//   - Data not ready (container launched for less than a second)

// Code 200 - OK
// Content-Type: application/json
{
  "cpu": 44,
  "free_memory": 123456,
  "memory": 2340354,
  "net": {
    "rx": 123456,
    "tx": 123564
  }
}

================================================

/*
 * Create a new container
 * Params:
 *   image: Docker image to use
 *   service: Name of the service
 *   port (optional): Choose the given port instead
 *     of allocating one randomly
```

```
 */

// POST /containers

// Code 201 - Container created and started
// Content-Type: application/json

// Docker JSON representation of a container:
// See: http://goo.gl/JrR6f6

================================================

/*
 * Stop and delete the container with the given ID.
 */

// DELETE /container/:container_id

// Code 204 - Container deleted, no content in response
```

Appendix B

HTTP API of the controller server

```
/*
 * Get the status of a given node, like the different
 * resources consumption and total
 */
// GET /node/:host/status

// Code 404 - Agent not found

// Code 200 - OK
// Content-Type: application/json
{
  "cpus": {
    "cpu0": 44,
    "cpu1": 22,
  },
  "free_memory": 12300000,
  "memory": 27440000,
  "network": {
    "eth0": {
      "tx": 98765,
      "rx": 12345
```

```
    }
  }
}

================================================

/*
 * Return a list of containers running on a given host
 */
// GET /node/:host/containers

// Code 404 - Agent not found

// Code 200 - OK
// Content-Type: application/json
[
  {
    "Id": "0123456789abcdef",
    "Image": "soulou/msc-thesis-memory-http-service",
    "Ports": [
      {
        "PublicPort": 49127,
        "PrivatePort": 3000
      }
    ],
    "Names": [ "service1-1-837" ],
    "Created": 1723454345,
    "Status": "Up"
  },
  {
    ...
  }
]

================================================

/*
 * Return the status of all the nodes of the cluster.
```

```
 */
// GET /nodes/status

// Code 200 - OK
// Content-Type: application/json
{
  "192.169.0.1" : {
    "cpus": {
      "cpu0": 44,
      "cpu1": 22,
    },
    "memory": 27440000,
    "network": {
      "eth0": {
        "tx": 98765,
        "rx": 12345
      }
    }
  },
  "192.168.0.2" : ...
}

==================================================

/*
 * This endpoint aims at executing an algorithm
 * of load balancing. The name of the algorithm
 * has to be given in the strategy parameter.
 * The containers and the agents nodes resource
 * usage is gathered then normalized, then the
 * algorithm is executed. As a result of the
 * problem solving, a set of migration is applied
 * to move the containers according to the algorithm
 * solution.
 */
// POST /balance
// Params:
//   strategy: "first-fit-decreasing"
```

```
//    opts: "heuristic=xxx;item_sort=xxx"

// Code 200 - OK
{
  "items": [[0.2, 0.3, 0], [...]],
  "bins": [[1.0, 2.0, 1.0], [...]],
  "problem": {
    "algorithm": "first-fit-decreasing",
    "mapping": [0,0,0,1,2,3,4],
  },
  "migrations": [
    {
      "Service": "service-1",
      "Started": {
        "host": "182.168.0.2",
        "id": "0123456789abcdef"
      },
      "Stopped": {
        "host": "182.168.0.2",
        "id": "0123456789abcdef"
      }
    },
    {
    ...
    }
  ]
}

==================================================

/*
 * Return a list of all the containers running on the cluster
 * hosted by each alive node.
 */
// GET /containers

// Code: 200 - OK
// Content-Type: application/json
```

```
[
  {
    "Host": "192.168.1.2",
    "Id": "0123456789abcdef",
    "Image": "soulou/msc-thesis-memory-http-service",
    "Ports": [
      {
        "PublicPort": 49127,
        "PrivatePort": 3000
      }
    ],
    "Names": [ "service1-1-837" ],
    "Created": 1723454345,
    "Status": "Up"
  },
  {
    ...
  }
]

==================================================

/*
 * Create a new container
 */

// POST /containers

// Code: 201 - Container started

// Docker JSON representation of a container:
// With an additional field: "Host"
// See: http://goo.gl/JrR6f6

==================================================

/*
 * Stop and delete a container on a given host
```

```
 */

// DELETE /container/:host/:container_id

// Code: 404 - Host or Container not found
// Code: 204 - Container stopped

================================================

/*
 * Migrate a container, to the most available node
 */

// POST /container/:host/:container_id/migrate

// Code 200 - Container migrated
{
  "Service": "service-1",
  "Started": {
    "host": "182.168.0.2",
    "id": "0123456789abcdef"
  },
  "Stopped": {
    "host": "182.168.0.2",
    "id": "0123456789abcdef"
  }
}
```

Appendix C

Client command line tool documentation

- Start a container

```
./client.py start [--image <image name>] <service name>

##
# The Image parameter correspond to the docker image which should be
# run on the agent, as at that time, the identify of this agent is not
# known, take care that all the agents have the requested docker image.
# By default a fibonacci web service is executed.
#
# The service name is the name of the group of containers that the
# newly created process will be appended, if the group doesn't exist,
# it is created.
#
# Example
#

./client.py start --image soulou/memory-http-service service1

-> Host: 192.168.114.13
-> ID: e6bbb8853678b67882ea6919c559d80a20
-> Service: service1
-> Image: soulou/msc-thesis-memory-http-service
```

```
-> Port: 49342
```

- Stop a container

```
./client.py stop --service <service_name> | --all <host> | <host> <ID>

##
# Three different options are available to stop a container.
# * All the container of a service ban be stopped
# * All the container of a precise server may be stopped
# * A precise container can be shutdown by giving its host and its ID
#
# Example
#

./client.py stop --service <service_name>

Container f6fe748fbf from 192.168.114.14 has been stopped
Container e7dc0f68f3 from 192.168.114.17 has been stopped
Container 6a28e4b2ab from 192.168.114.15 has been stopped
```

- Get status of the infrastructure

```
./client.py status

##
# Give the information about all the running containers
#
# Output example
#

+----------------+-------+-----------+
| Node           |  Port |  Service  |
+----------------+-------+-----------+
| 192.168.114.14 | 49196 |  service3 |
| 192.168.114.14 | 49195 |  service3 |
| 192.168.114.13 | 49342 |  service1 |
+----------------+-------+-----------+
```

```
+---------------------------------------+
|                 Image                 |
+---------------------------------------+
| soulou/msc-thesis-memory-http-service |
| soulou/msc-thesis-memory-http-service |
| soulou/msc-thesis-memory-http-service |
+---------------------------------------+

+-----------------+---------------------+
|        ID       |      Created At     |
+-----------------+---------------------+
| 3c9345c19e00d21 | 2014-07-28 10:51:34 |
| 9a2e2b22448f766 | 2014-07-28 10:51:33 |
| e6bbb8853678b67 | 2014-07-29 15:52:15 |
+-----------------+---------------------+
```

- Migrate an application to another node

```
./client.py migrate <host> <id>

##
# This operation asks the controller to migrate a container
# to another node
#
# Example
#

./client.py migrate 192.168.1.3

Container 192.168.1.3 - 9a2e2b2244 -> 192.168.1.18 - 406d5ce46d
```

- Load balance infrastructure

```
./client.py balance [--opt opt1=val1 [--opt opt2=va2 ...]] <algorithm>

##
# Run a load balancing or consolidation
# operation on the complete infrastructure
# The algorithm can be specified the following
```

```
# are available:
#
# Implemented in the controller
# [first-fit-decreasing, best-fit-decreasing]
#
# Implemented in third-party library
# [stillwell, gabay2013vsvu, brandao2013mvp]
# https://github.com/marklee77/vpack
#
# Example
#

./client.py balance \
  --opt heuristic=bin_centric \
  --opt bin_measure=do_nothing \
  --opt item_measure=do_nothing \
  gabay2014vsvu

{'bins': [{'capacity': [0.8, 0.8, 0.8],
           'remaining_capacity': [1.0, 1.0, 1.0]}, ...],
 'items': [[0.0, 0.0004, 0.0], ...]
 'migrations': [{'Service': 'service2',
                 'Started': {'Id': '7724621e2dcccb472',
                             'Node': '192.168.114.14'},
                 'Stopped': {'Id': 'd3f8a1e45481e3945',
                 'Node': '192.168.114.13'}}, ...],
 'result': {'algo-runtime': 0.00013580999999973642,
            'bincount': 7,
            'datetime': '2014-08-10 18:04:28.343690',
            'failcount': 0,
            'family': 'gabay2013vsv',
            'kwargs': {'bin_measure': 'do_nothing',
                       'family': 'gabay2013vsv',
                       'heuristic': 'bin_balancing',
                       'item_measure': 'do_nothing'},
            'mapping': [0, 1, 2, 3, 4, 5, 6, 0, 1, 2],
            'problem-argshash': None,
            'solver-githash': '__GITHASH__',
```

```
            'total-runtime': 0.00021365099999925974,
            'verified': True}
}
```

- Get node or nodes status

```
./client.py node_status <host>
./client.py nodes_status

##
# Get the system metrics for on particular node or all the nodes
#

./client.py nodes_status
[ '192.168.114.19': {'cpus': {'cpu0': 0, 'cpu1': 0},
                     'free_memory': 3736440,
                     'memory': 4048292,
                     'nb_containers': 0,
                     'net': {'docker0': {'rx': 0, 'tx': 0},
                             'eth0': {'rx': 3728, 'tx': 1858},
                             'eth1': {'rx': 0, 'tx': 0},
                             'lo': {'rx': 0, 'tx': 0},
                             'veth2074': {'rx': 0, 'tx': 0},
                             'vetha4f2': {'rx': 913, 'tx': 1796},
                             'vethc390': {'rx': 0, 'tx': 0},
                             'vethd466': {'rx': 633, 'tx': 964},
                             'vethec34': {'rx': 0, 'tx': 0}}}}
]
```

Appendix D

Projects related to the thesis

- https://github.com/Soulou/msc-thesis-container-balancer-agent

  The repository contains the code of the agent, which is running on every hosting server, monitoring the host and its container in real time.

- https://github.com/Soulou/msc-thesis-container-balancer-controller

  The controller of the infrastructure which finds the agents through **Consul**. It manages the allocation and the load balancing operations in the infrastructure.

- https://github.com/Soulou/msc-thesis-container-balancer-client

  The client is a simple CLI tool to communicate with the controller. Otherwise it would be necessary to write the HTTP requests by hand, which is not really confortable.

- https://github.com/Soulou/msc-thesis-fibo-http-service

  Webservice generating numbers of the fibonacci sequence. Used as a docker image to generate load.

- https://github.com/Soulou/msc-thesis-memory-http-service

  Webservice allocating a precise amount of memory during a precise duration, used to simulate different kinds of webservices

- https://github.com/Soulou/msc-thesis-cpu-burn

Small software design to use completely a given amount of CPU cores.

- https://github.com/Soulou/msc-thesis-container-cpu-monitor

  Tool to generate data about the consumption of each container run by **Docker**.

- https://github.com/Soulou/msc-thesis-cloud-builder

  Set of tools designed to deploy VMs automatically in an **OpenStack** environment.

- https://github.com/Soulou/msc-thesis-ansible

  Ansible playbook to deploy everything required to run the exeperiment on the VMs dpeloyed by the Cloud Builder.

Bibliography

[1] E. Feller, L. Rilling, and C. Morin, "Energy-aware ant colony based workload placement in clouds," in *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*, 2011.

[2] D. Wilcox, A. McNabb, K. Seppi, and K. Flanagan, "Probabilistic virtual machine assignment," in *The First International Conference on Cloud Computing, GRIDs, and Virtualization*, 2010.

[3] B. Paul, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualisation," in *19th ACM Symposium on Operating Systems Principles*, 2003.

[4] P. Mell and T. Grance, "The NIST definition of cloud computing," *Recommandation of National Institute of Standards and Technology*, Spetember 2011.

[5] r. f. F. Ben Kepes, "Docker raising monster funding round because... vmware disruption," http://www.forbes.com/sites/benkepes/2014/08/06/docker-raising-monster-funding-round-because-vmware-disruption/.

[6] V. Salapura, "Cloud computing: Virtualization and resiliency for data center computing," in *30th International Conference of Computer Design*, 2012.

[7] T. Setzer and A. Stage, "Decision support for virtual machine reassignments in enterprise data centers," in *Network Operations and Management Symposium*, 2010.

[8] R. V. den Bossche, K. Vanmechelen, and J. Broeckhov, "Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads," in *3rd International Conference on Cloud Computing*, 2010.

[9] M. Stillwell, F. Vivien, and H. Casanova, "Virtual machine resource allocation for service hosting on heterogeneous distributed platforms," in *Parallel & Distributed Processing Symposium*, 2012.

[10] N. E. Young, "Randomized rounding without solving the linear program," in *6th ACM-SIAM Symposium on Discrete Algorithms*, 1995.

[11] D. S. Johnson, "Near-optimal bin packing algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, 1967.

[12] G. Gambosi, A. Postiglione, and M. Talamo, "Algorithms for the relaxed online bin-packing model," *SIAM Journal on Computing*, vol. 30, no. 5, 2000.

[13] W. Song, Z. Xiao, Q. Chen, and H. Luo, "Adaptive resource provisioning for the cloud using online bin packing," in *Transactions on Computers*, 2013.

[14] L. Chimakurthi and M. K. SD, "Power efficient resource allocation for clouds using ant colony framework," *CoRR*, 2011.

[15] L. Zhu, Q. Li, and L. He, "Study on cloud computing resource scheduling strategy based on the ant colony optimization algorithm," *IJCSI International Journal of Computer Science*, vol. 9, no. 5, 2012.

[16] D. Wilcox, A. McNabb, and K. Seppi, "Solving virtual machine packing with a reordering grouping genetic algorithm," in *IEEE Congress on Evolutionary Computation (CEC)*, 2010.

[17] K. Patel, M. Annavaram, and M. Pedram, "Nfra: Generalized network flow-based resource allocation for hosting centers," *IEEE Transactions on Computers*, vol. 62, no. 9, 2013.

[18] INRIA, "Simgrid official website," http://simgrid.gforge.inria.fr/.

[19] M. University, "Project page of cloudsim," https://code.google.com/p/cloudsim/.

[20] R. members, "Roadef contest homepage," http://challenge.roadef.org/.

[21] R. Lopes, V. W. Morais, T. F. Noronha, and V. A. Souza, "Heuristics and matheuristics for a real-life machine reassignment problem," *International Transactions in Operational Research*.

[22] R. Masson, T. Vidal, J. Michallet, P. H. V. P. andVinicius Petrucci, A. Subramanian, and H. Dubedout, "An iterated local search heuristic for multi-capacity bin packing andmachine reassignment problem," *Expert Syst. Appl.*, vol. 40, no. 30, 2013.

[23] B. Kell and W.-J. van Hoeve, "An mdd approach to multidimensional bin packing," *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems Lecture Notes in Computer Science*, vol. 7874, 2013.

[24] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *EuroSys European Conference on Computer Systems*, 2007.

[25] CRIU, "Checkpoint/restore in userspace."

[26] . Factors, "The twelve factors."

[27] openstack, "official website for the openstack project," http://www.openstack.org/.

[28] M. Carlo, M. Michela, and P. Giuseppe, "Self-economy in cloud data centers: Statistical assignment and migration of virtual machines," 2011.

[29] M. Gabay and S. Zaourar, "Variable size vector bin packing heuristics – application to the machine reassignment problem," 2014.

[30] C. Christopher, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, 2005.

[31] L. Colombus, "Roundup of cloud computing forecasts update, 2013," http://www.forbes.com/sites/louiscolumbus/2013/11/16/roundup-of-cloud-computing-forecasts-update-2013/, November 2013.

[32] P. P. Arkency, "Microservices - 72 resources," http://blog.arkency.com/2014/07/microservices-72-resources/.

[33] J. Wun-Tat, C. T.-W. Lam, and P. W. Wong, "Dynamic bin packing of unit fractions items," *Theoretical Computer Science*, vol. 409, no. 8, 2008.