

UiO : **Department of Informatics**  
University of Oslo

# Will You Carry Me?

Distributed Load Balancing in Cloud Data Centre with  
Stable Marriage

Disha Sangar

Master's Thesis Spring 2017





# Will You Carry Me?

Disha Sangar

23rd May 2017



# Abstract

With an increasing number of computers and virtually driven technology, the desire to find performance- and cost efficient solutions has risen in the world of research. In search of better solutions, we have started to look outside the field of computer science for inspiration.

Cloud computing, virtualization and load balancing are emerging terms that come with the power to run a modern data centre with better conditions than ever before. Running a sheer virtualized data centre with the help of *Virtual Machines* (VM) enables the possibility to save resources. The unique method of reallocating VMs between physical hosts with Live Migration has started a new movement for System administrators.

*Stable Marriage* is a new concept presented in this thesis. Inspired from Nobel Peace Prize winners Alvin Roth and Lloyd Shapley's work on Stable Matching [17], a similar approach to present load balancing in a cloud based data centre is introduced.



# Contents

<b>1</b>	<b>1</b>
1.1	1
1.1.1	3
1.1.2	3
1.1.3	4
<b>2 Background</b>	<b>7</b>
2.1	7
2.1.1	9
2.2	11
2.2.1	11
2.3	12
2.3.1	13
2.4	14
2.4.1	14
2.4.2	15
2.4.3	15
2.5	16
2.5.1	17
2.6	18
2.6.1	18
2.7	19
2.7.1	20
2.8	20
2.9	22
2.10	25
2.10.1	25
2.10.2	26
2.10.3	27
<b>I The project</b>	<b>31</b>
<b>3 Approach</b>	<b>33</b>
3.1	33

3.2	Mapping the load balancing problem to Stable marriage . .	34
3.3	Design . . . . .	35
3.3.1	Modelling . . . . .	35
3.3.2	Load balancing Schemes . . . . .	36
3.4	Implementation . . . . .	37
3.4.1	Environment . . . . .	37
3.4.2	Framework . . . . .	37
3.4.3	Data Collection & Comparison . . . . .	37
3.5	Result & Analysis . . . . .	38
3.5.1	Configuration Experiments . . . . .	38
3.5.2	Scheme experiments . . . . .	39
3.6	Expected Results . . . . .	39
3.7	Appraising Properties . . . . .	39
3.8	Discussion & Conclusion . . . . .	40
3.9	Challenge Prediction . . . . .	40
<b>4</b>	<b>Result I - Design</b>	<b>41</b>
4.1	Model . . . . .	41
4.2	Overview of a functioning framework . . . . .	41
4.3	Formal Notations . . . . .	43
4.4	Bin Packing with Stable Marriage . . . . .	44
4.4.1	Stable Marriage Animation . . . . .	44
4.4.2	Libvirt Live Migration . . . . .	47
4.4.3	VMs . . . . .	47
4.4.4	Node communication . . . . .	48
4.5	Schemes . . . . .	48
4.5.1	Scaling the percentile . . . . .	49
4.5.2	Stable Marriage Algorithm - Migrate Large & Migrate Small . . . . .	49
4.5.3	Distance Vector . . . . .	50
<b>5</b>	<b>Result II – Implementation</b>	<b>51</b>
5.1	Model Overview . . . . .	52
5.2	Environment Configuration . . . . .	52
5.3	Virtual Configuration . . . . .	53
5.3.1	Network of VMs . . . . .	54
5.3.2	VM Communication & Message Exchange . . . . .	56
5.3.3	Deployment of a VM . . . . .	57
5.4	Stable Marriage . . . . .	59
5.5	Schemes . . . . .	63
5.5.1	Migrate Large First . . . . .	63
5.5.2	Migrate Small First . . . . .	63
5.6	Summary . . . . .	64
<b>6</b>	<b>Result III – Experiment &amp; Analysis</b>	<b>65</b>
6.1	Testing . . . . .	65
6.2	Simulation Experiments . . . . .	65
6.3	Simulation . . . . .	67

6.3.1	Summary . . . . .	67
6.3.2	Small scale migration - I . . . . .	68
6.3.3	Analysis . . . . .	69
6.3.4	Small scale migration - II . . . . .	70
6.3.5	Analysis . . . . .	71
6.3.6	Workaround . . . . .	71
6.3.7	Medium Scale migration - I . . . . .	72
6.3.8	Analysis . . . . .	73
6.3.9	Medium Scale Migration - II . . . . .	74
6.3.10	Analysis . . . . .	75
6.3.11	Large Scale Migration - I . . . . .	76
6.3.12	Analysis . . . . .	77
6.3.13	Large Scale Migration - II . . . . .	78
6.3.14	Analysis . . . . .	79
6.4	Real Experiment . . . . .	80
6.4.1	Migrating Small Scale . . . . .	80
6.4.2	Analysis . . . . .	81
6.4.3	Migrating Large Scale . . . . .	81
6.4.4	Analysis . . . . .	82
6.5	En route Proposal Acceptance . . . . .	82
6.5.1	Five Rejected Proposals - Large . . . . .	83
6.5.2	Three Rejected Proposals . . . . .	84
6.5.3	Thirty three Rejected Proposals . . . . .	84
6.5.4	Twelve Rejected Proposal . . . . .	85
6.5.5	Analysis - Scheme Large . . . . .	85
6.5.6	Three Rejected Proposals - Small . . . . .	85
6.5.7	Ten Rejected Proposals . . . . .	86
6.5.8	Eleven Rejected Proposals . . . . .	86
6.5.9	Eleven Rejected Proposals . . . . .	87
6.5.10	Analysis - Scheme Small . . . . .	88
<b>7</b>	<b>Discussion</b>	<b>89</b>
7.1	Background . . . . .	89
7.2	Design & Implementation . . . . .	90
7.2.1	Proposals . . . . .	90
7.2.2	Schemes & Live migration . . . . .	91
7.2.3	Migrate Large Vs. Migrate Small . . . . .	92
7.3	Results & Analysis . . . . .	92
7.3.1	Dedicated vs. Non-dedicated links . . . . .	93
7.3.2	Managing Scaling in cloud computing . . . . .	93
7.4	Future Work . . . . .	94
<b>8</b>	<b>Conclusion</b>	<b>95</b>
	<b>Appendices</b>	<b>101</b>

<b>A</b>	<b>Working Environment</b>	<b>103</b>
A.1	create isc hosts dhcp.py . . . . .	103
A.2	vm-net . . . . .	104
A.3	deploy_large.py . . . . .	105
A.4	deploy_small.py . . . . .	110
<b>B</b>	<b>Artificial Simulation</b>	<b>115</b>
B.1	Many to Many Move Smallest.py . . . . .	115
B.2	Many to Many Move Largest.py . . . . .	120

# List of Figures

2.1	Non-virtualized vs. Virtualized system . . . . .	8
2.2	Full -vs. Paravirtualization . . . . .	9
2.3	Full virtualization . . . . .	10
2.4	Paravirtualization . . . . .	11
2.5	Cloud services . . . . .	13
2.6	Simulation of Live migration . . . . .	14
2.7	Difference between Centralized, Decentralized and Dis- tributed [9] . . . . .	18
2.8	Stable Matching . . . . .	22
2.9	Python Logo . . . . .	22
2.10	Eclipse Logo . . . . .	23
4.1	Proposal accepted . . . . .	42
4.2	Proposal Rejected . . . . .	42
4.3	Set of over/under utilized servers . . . . .	45
4.4	PM1 proposes to PM3 . . . . .	45
4.5	PM3 rejects PM1 seeing no benefit to this marriage. . . . .	46
4.6	PM4 accepts PM1's proposal . . . . .	46
4.7	Migration successful . . . . .	46
4.8	PM with various flavours . . . . .	47
4.9	VM Chart . . . . .	49
4.10	VMs with their allocated values . . . . .	50
5.1	Chapter overview . . . . .	51
5.2	Design . . . . .	52
5.3	Overview of the Physical lab structure . . . . .	53
5.4	Physical attributes . . . . .	53
5.5	Physical Lab details . . . . .	54
5.6	Flow Diagram of the Stable Marriage Implementation . . . . .	59
6.1	Mean of CI for Small Migration scheme . . . . .	66
6.2	Mean of CI for Large Migration scheme . . . . .	66
6.3	Table of Experiment details . . . . .	67
6.4	Migration of 60 Large VMs . . . . .	68
6.5	Migration of 60 Small VMs . . . . .	68
6.6	Imbalance example . . . . .	69
6.7	Migration of 125 Large VMs . . . . .	70
6.8	Migration of 125 Small VMs . . . . .	70

6.9	Migration of 2500 Large VMs . . . . .	72
6.10	Migration of 2500 Small VMs . . . . .	72
6.11	Flavours of VMs . . . . .	73
6.12	Migration of 5000 Large VMs . . . . .	74
6.13	Migration of 5000 Small VMs . . . . .	74
6.14	Flavours of VMs . . . . .	75
6.15	Migration of 10.000 Large VMs . . . . .	76
6.16	Migration of 10.000 Small VMs . . . . .	76
6.17	Flavours of VMs . . . . .	77
6.18	Migration of 20.000 Large VMs . . . . .	78
6.19	Migration of 20.000 Small VMs . . . . .	78
6.20	Flavours of VMs . . . . .	79
6.21	Small Imbalance Before vs. After . . . . .	80
6.22	Large Imbalance Before vs. After . . . . .	82
6.23	Descending Imbalance . . . . .	83
6.24	Descending Imbalance . . . . .	84
6.25	Descending Imbalance . . . . .	84
6.26	Descending Imbalance . . . . .	85
6.27	Rounds of Proposal . . . . .	86
6.28	Rounds of Proposal . . . . .	86
6.29	Rounds of Proposal . . . . .	87
6.30	Rounds of proposal . . . . .	87

# Acknowledgments

My deepest gratitude goes first to my Professor and supervisor Anis Yazidi, who has been my mentor and guide throughout my education. Thank you for dedicating your time and knowledge with great excitement to work on this project. The door to your office was always open whenever I ran into trouble or had any questions about my research, you constantly steered me in the right direction whenever I needed it.

I would also like to extend my gratitude to Kyrre Begnum and Hårek Haugerud for their essential input and support. Thanks to Oslo and Akershus University College and the University of Oslo for giving me a chance to take part in this master program.

This thesis has been a journey on which I have learned so much more than different aspects of computer science. I have met people whom have shared their immense knowledge to help pave my path towards the finish line. I could never have done it without their love, support and motivation.

Finally, I must express my profound gratitude to my parents, my siblings and family. With unfailing support and continuous encouragement and love throughout my years of study. To my lovely bouquet of friends who never stopped believing in me and always kept encouraging me with their love and support, you all know who you are. Thank you.

- Disha



# Chapter 1

## 1.1 Introduction

IT services have grown rapidly over the past 10 years and we have become dependent on them. Major systems and Internet based services have grown to such a scale that we now use the term “hyper scale” to describe them. Furthermore, hyper scale architectures are often deployed in cloud based environments, which offer a flexible pay-as-you-go model.

From a system administrator’s perspective, optimizing a hyper scale solution implies introducing system behaviour that can yield automated reactions to changes in configurations and fault occurrences. For instance, auto scaling is a proposed behaviour model for websites to optimize cost and performance relative to usage patterns.

There are two different perspectives on how an automated behaviour can be implemented. One of the perspectives is to implement the behaviour in the infrastructure, which is the paradigm embraced by the industry. The other alternative is to introduce behaviour as a part of the virtual machine, which opens a possibility for cloud independent models. This means that its behaviour would work in any cloud based environment.

This space is still largely unexplored from system administrator’s perspective. We need more knowledge on algorithms and more understanding of their potential to be able to introduce them in a real working environment. Such understanding of algorithms system administrator’s perspective would pave the way towards larger scale co-ordination and swarm intelligence based solutions.

Cloud computing refers to services or solutions that are delivered in real time over the internet. A cloud provider is a company or an institution, which provides these services [32]. More companies are now abiding to cloud computing, instead of having to spend fortunes on building and creating a physical computer infrastructure [22]. This is an effective way of allowing companies to focus on having flexible and accessible machines at all times, which does not necessarily consume massive amount of resources. High power consumption by hardware facilities and cooling system is today one of the biggest challenges companies face, as well as storage of massive data in physical form.

With today's data centres being mostly overutilized, trying to sustain the service during periods of peak has become the foremost goal. It was estimated that in 2006, the energy consumed by IT infrastructures in USA was about 61 billion kWh, which corresponded to 1.5 % of the total amount of electricity produced in total [21].

Several approaches exist to counter the issues of energy consumption where virtualization can be seen as enabler. The term virtualization refers to the abstraction of computer resources. Virtualization helps minimize the number of active *Physical machines* (PM) in data centres by creating *Virtual Machines* (VM) and storing these on a cloud platform. There are now several open-source projects (Openstack, Apache, OpenNebula, etc.), which offer good virtualization solutions, just as good as any other reputed companies' solutions (Amazon, Microsoft, etc). These services often offer a web based front-end interface for managing VMs, such as creating, initiating, stopping and terminating these.

Virtualization technologies allow less physical infrastructure, however the underlying physical infrastructure is needed, as the virtualized environment runs on top of this. The foremost goal is to find a solution that minimizes the amount of active nodes in an infrastructure to gain the benefit of virtualization. The two most common methods of achieving this goal is Consolidation and Load Balancing, with the help of Live Migration.

Live Migration offers the possibility to re-allocate VMs in their current state to another server based on need. When the amount of VMs utilizing a server is low, the VM can either be migrated to another server to balance both servers load, or shut down the old server completely after the migration saving energy and resources. Migration can also be very useful for maintenance of a server. The whole purpose of a migration is to pack VMs on the least number of PMs in a well-known dimension, NP hard problem, bin-packing, and minimizing the amount of physical nodes [8].

Virtual Machine Monitors (VMM) or hypervisors offer the possibility to monitor and manipulate VMs in any desired state, through several different software. Nowadays, Live migration is a common practice among System administrators who can run several updates at one time and avoid down time, which is a large benefit for companies which can not afford down time.

To be able to respond to the large and emerging demand of cloud and virtually driven technology, research is lead to seek solutions that can handle the large amount of data. Various studies on self-organizing approaches have been emerging in the recent years. These studies mainly focus on finding algorithms that can optimize the system in an efficient order and at the same time reduce the energy consumption made due to the high performance of the compute node.

Today there is an extensive research on consolidation and very little on load balancing. Ideally, companies want to be cost-effective and

address performance issues at the same time. The industry has chosen consolidation, because this reduces the amount of active servers. The problem is however, that servers are now more prone exhaustion. Load balancing is a distributed solution it is computationally faster, but not always optimal.

Having a disseminated system makes the system much faster and very *scalable* as well as less expensive. The aim of this paper is to provide an answer through the definitions and implementation of the Stable Marriage algorithm along with two schemes to experiment migration and see how different scales can affect the results of a data centre. To be able to test this some constraints will be necessary and a way to monitor these results in action.

The potential gain with this research is to find an efficient and less complex way of operating a data centre. Stable Marriage is a new contribution and holds many promises due to its distributed nature, which can be a positive contribution to research as it can solve many practical problems. The algorithm is an inspiration from the Nobel Peace Prize holders for their contribution in the field of economics for *Stable Matching*, by Lloyd Shapley and Alvin Roth.

### 1.1.1 Problem Statement

*Can we achieve load balancing in a data centre using simple message exchange between physical machines in a distributed manner?*

Along with:

*How can we borrow the principals of Stable Marriage to gain load balance in a distributed data centre?*

### 1.1.2 Definitions

**Distributed** is defined as a system or cluster which does not have one single point of management. The opposite of Distributed is Centralized. The purpose of a distributed system is to disassemble one central manager, as mentioned above, in a centralized solution if one point fails the entire system fails. To avoid this there are several distributed approaches, for instance the very renowned peer-to-peer structure.

To **optimize** something means to make the best effective use of a system or a resource. Things have been ever evolving in the field of computer science and usually newer technologies are a better and optimized version of an old technology. The term is often used to describe to take advantage of improving systems, to optimize performance or cut energy consumption for instance.

### **1.1.3 Chapter Summaries**

A short introduction to each chapter is presented as a guide. This will help provide an insight to what the

#### **1.1.3.1 Background**

The background chapter gives an insight to all the theories and underlying research of concepts which are later introduced in the various chapters of the project.

#### **1.1.3.2 Approach**

This section describes the manner of the approaches which the project requires in order to get a complete framework. Here the design, models and basic introduction to the different objectives of the forth coming chapters are presented.

#### **1.1.3.3 Result I - Design**

This is the chapter where the overview of the framework is presented. The ideas behind the structure in form of drawings and models, which should describe the wanted end-result for the framework. Some of the studies and terminologies from the background chapter may provide a deeper insight with the design chapter.

#### **1.1.3.4 Result II - Implementation**

This is probably the heaviest chapter in terms of technicality. The approach and design chapter introduced the wishful goal, however, this is the chapter where all the implementation takes place and is discussed. The details will be provided in form of pseudocode and models to accompany some analysis for the sake of readability.

#### **1.1.3.5 Result III - Results Analysis**

This chapter highlights the results gathered from the experiments conducted throughout the project. Since the project is very large, the data gathered will be equally large in volume. Graphs, tables and analysis for each experiment will accompany to provide a guide for the readers sake.

#### **1.1.3.6 Discussion & Conclusion**

There are always some unforeseen circumstances, what did not work out? What did work out? How was the results compared to the expectations? Has any prior research prepared this outcome? These are some of the questions which are discussed and concluded in this particular chapter.

#### **1.1.3.7 Appendices**

The code of the framework is divided into two categories; A and B. In the first section the code for the working environment is provided. Whereas in the second sections the code for the simulations is provided.



## Chapter 2

# Background

This chapter presents all the underlying concepts on which this project is built. Research, terminologies and introductions to tools and methods used will be presented. Instead of inventing the wheel all over again, earlier research can contribute to avoid such problems even though the concepts introduced in this chapter are new.

### 2.1 Virtualization

The concept of virtualization was introduced in early 1960's by International Business Machines (IBM), which made it possible to have multiple hosts sharing the same hardware at the same time. In the 80's however, the decrease in hardware cost made it impossible for virtualization to gain its stardom. In modern times the networked environments presented difficulties such as security, reliability, cost inefficiency and complexity, which welcomed virtualization, back to address these difficulties.

According to Amza et al. *"A virtual machine abstracts the computing resources of a physical machine into a virtual resource"* [3]. The processing capacity of servers have consistently increased the later years allowing virtualization to enhance data centre abilities by abstracting the OS and applications from the hardware and assign them to VMs. This has presented an endless of possibilities a hardware could not handle alone and made servers more tolerant than ever [34].

Figure 2.1 demonstrates how a non-virtualized system is built compared to a virtualized system. In a non-virtualized environment the OS controls the access to the hardware resources, in a virtualized system however, the VMM controls the access to the hardware resources [6, 34].

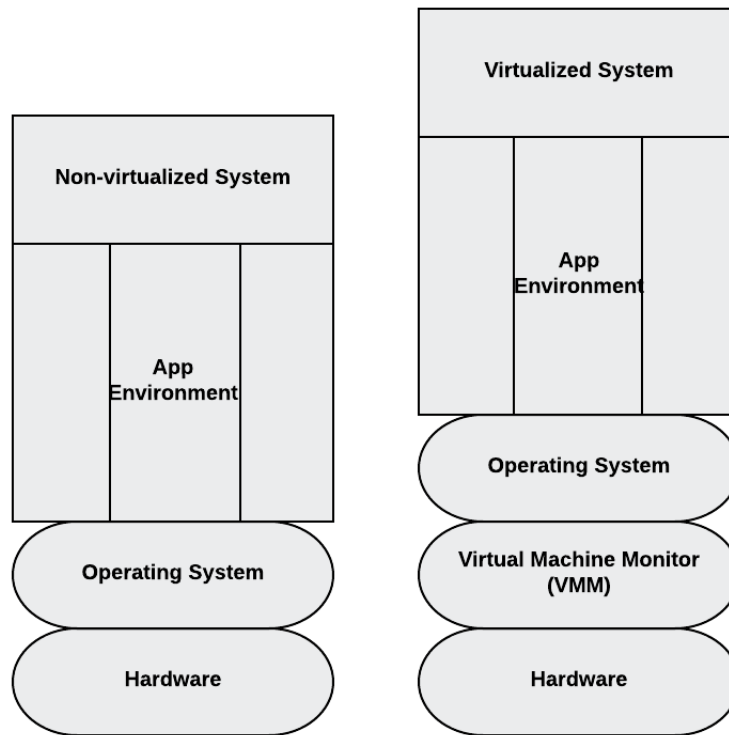


Figure 2.1: Non-virtualized vs. Virtualized system

The instruction set is separated in two categories; privileged and non-privileged, under two modes called *user* and *supervisor* [34]. The instructions given by a non-privileged user can only be executed in user mode, on the other hand if an OS or application environment issues any non-privileged instruction; the machine directly executes it. This is known as full virtualization. In the x86 architecture there are four levels of privileges. Rings 0-3, in a non-virtualized environment the OS executes in ring 0 and the applications in ring 3. In a *paravirtualized* environment the VMM runs at ring 0, the guest OS at ring 1 and the applications in ring 3 [6, 34].

### 2.1.1 Virtualization types

There are several types of virtualization; full virtualization, paravirtualization, Hardware assisted virtualization, I/O virtualization, memory virtualization etc. There is a range of different strategies to pursue virtualization, however to set the focus on the two more common technique the next section cover full virtualization and paravirtualization.

Figure 2.2 explains the difference of the architecture of a fully virtualized environment vs. paravirtualized environment.

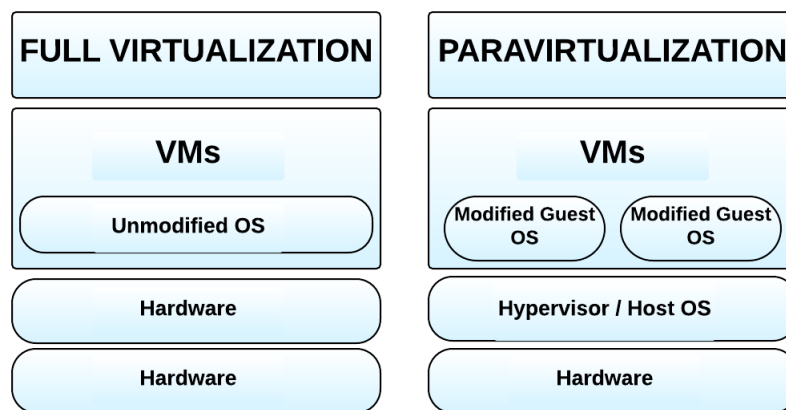


Figure 2.2: Full -vs. Paravirtualization

#### 2.1.1.1 Full virtualization

Full virtualization offers security, simplifies migration and portability. In terms of complexity, this is probably the least painful approach. In full virtualization single or multiple OSs and the application it holds can run on top of a virtual hardware. Each instance on the OS requires that every feature of the hardware is reflected into one of several VMs, known as *guest operating system (Guest OS)*. These guest OSs are managed by a VMM, which then has control of every set of instructions between the guest OSs and the physical hardware [24]. In this approach the a binary translation of OS requests translation of kernel code to replace non-virtualizable instructions [34].

The system is fully virtualized when it is completely decoupled from the underlying hardware. The guest OS is not aware that it is virtualized and this is the only form of virtualization that requires no form of hardware assistance in order to virtualize sensitive or privileged instructions. The VMM does all of this [34]. VMware was one of the pioneers who introduced this technique with the x86 architecture allowing a full virtualization [25, 34]. Figure 2.3 shows what a fully virtualized system looks like.

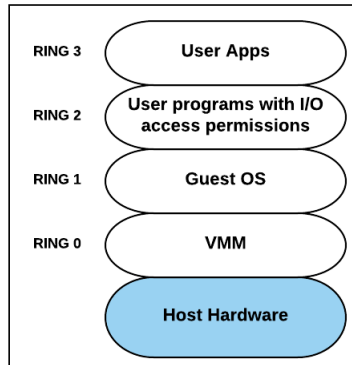


Figure 2.3: Full virtualization

#### 2.1.1.2 Paravirtualization

In Greek the word "Para" stands for "beside" or "alongside" translating the word "paravirtualization" to "alongside virtualization" [34]. Unlike full virtualization this approach requires the guest OSs to be modified in order to be operated in a virtual environment, with lightweight hardware [28].

Compared to full virtualization 2.1.1.1, instead of binary translation of the instruction set the VMM and guest OS communicate with each other, to improve the quality and performance of the system. This involves modifying the OS kernel to replace non-virtualizable instructions with *hypercalls* which communicate directly with the virtualization layer, as seen in figure 2.4.

Paravirtualization works on a bit more advanced level compared to full virtualization. Due to the fact that this approach requires deep OS kernel modifications, it can offer great support and maintainability with production issues [34].

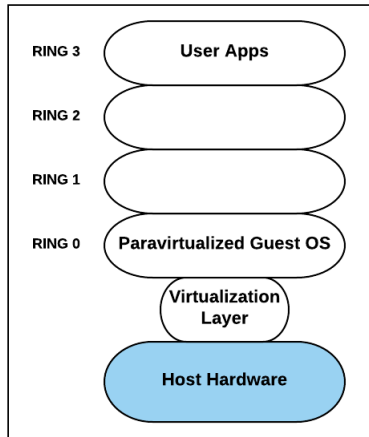


Figure 2.4: Paravirtualization

## 2.2 QEMU & KVM

QEMU is a software which allows virtualization. There are several modes in which the software operates, where “full system emulation” is amongst one of them. This also allows QEMU to host a variety of machines as guests in this specific mode. This is especially helpful when simulating a real data centre, where the need to create and host a large number of VMs is required. This is possible because a compiler in QEMU translates binary code which is destined for one CPU type to another [32].

Amongst the multiple hosting modes QEMU allow, KVM-hosting is one of them. KVM which stands for *Kernel Virtual Machine* is a full virtualization solution for the Linux x86 architecture. KVM is a hypervisor, another variant of QEMU and if run accordingly it can also transform the Linux Kernel into a hypervisor, meaning all the guests running on this kernel would work as if they were processes on the host [41].

### 2.2.1 Libvirt

Libvirt is a collection of software that offers a convenient way to manage VMs and other virtualization functionalities. Libvirt is developed by Red Hat and offers an interface (API), a daemon (Libvirt) and a command line utility (Virsh). The foremost goal of Libvirt is to provide a way to manage multiple virtual hosts. Some of the major features libvirt offers are host management and Remote machine support. With the VM management functionality the possibility to conduct operations like start, stop, pause, save, migrate and restore are offered [16].

Along with VM management, libvirt also offers the possibility to create virtual switches (or use existing one). This is a very resourceful option as it makes transfer of network traffic easier. In some cases, one needs to

be able to edit the network definition per wishes and apply these changes instantly. Libvirt makes it possible to add new static MAC and IP mappings for the DHCP server on the network. The Virsh command will let the user edit and make changes, and these will not take effect until the network is destroyed and re-started. One drawback of this approach is that all the guest interfaces lose their network connectivity with the host until re-attached again [23].

It is common to use a combination of QEMU, KVM and Libvirt when setting up a virtualized environment on a Linux platform.

## 2.3 Cloud Computing

A quick search for *Cloud computing* on any search engine will generate a large sum of results. There are also several paid advertisement placements from companies offering “fast, reliable and secure” cloud services. Cloud computing allows storing data or applications over a network instead of having them locally stored on a physical hard drive [18]. The network that is distributed is referred to as a cloud. Though there is no standard definition of Cloud Computing, researchers and authors seem to agree that it mostly consist of many clusters of distributed computers, which form the term *cloud*.

Cloud computing offers three important business models, which has made cloud computing even more popular for businesses to invest in. Cloud computing is an effective way of allowing companies to focus on having flexible and accessible machines at all times, instead of focusing on spending money on creating their own infrastructure. It also enables companies to consume compute resources as a utility, just like electricity and is therefore a *Pay-as-you-go* model.

The three major business models are known as *Software as a Service* (SaaS), *Platform as a Service* (PaaS) and *Infrastructure as a Service* (IaaS) [19].

SaaS is a method of software delivery which makes it possible to allow data to be accessible from any device with an internet connection and web browser. SaaS first emerged in the early 1960’s as a shared resource environment such as cloud computing. This method is beneficial as companies do not need to invest in extensive hardware to host software.

PaaS is a model which offers hardware and software tools to its users as a service. PaaS provider hosts the hardware and software on its own infrastructure, which is an advantage as it does not replace an entire infrastructure, instead a business relies on PaaS for key services, for example Java development or application hosting. This is also an advantage as it frees its users from having to install in-house hardware to develop or run new applications.

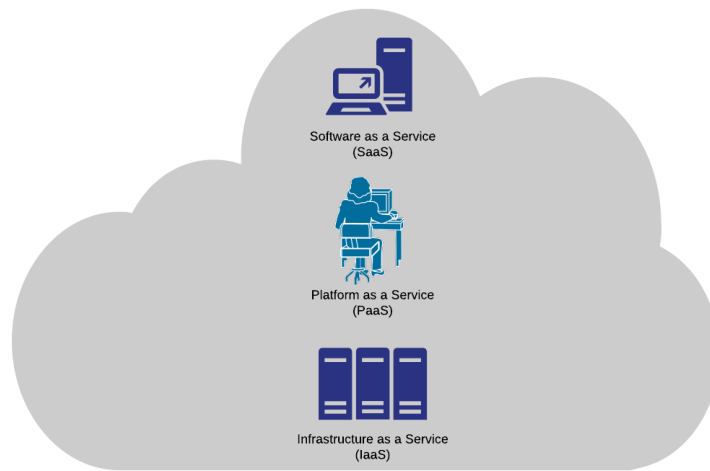


Figure 2.5: Cloud services

The current commercial offering of cloud computing became apparent in late 2007 [26]. The recent uptake of cloud computing and the high demand of cloud solutions show that until recently most research base the work on load balancing with assumed homogeneous nodes, which is unrealistic. A data centre is dynamic and the need for dynamic and heterogeneous systems are necessary to provide on-demand resources of services [26].

### 2.3.1 Distributed Load Balancing for Cloud Computing

Load balancing is essential for operating a dynamic data centre. In a survey by Al Nuaimi et al. [1] the challenges with load balancing in cloud computing is narrowed down in four different categories. All are linked to the fact that the research in cloud computing with load balancing in the focus is often done in a constrained environment. This means that algorithms are often designed to be efficient only for intranet or nodes that are closely located, where no communication delay is negligible. The issue with designing algorithms in these cases has been the many factors that needs to be taken into account, such as speed, network links and distance [1, 26]. Load-balance is a critical function among storage nodes, by load balancing the resources can be well utilized and maximize performance [12].

There is a need to develop a more efficient way that can control the load balancing dynamic in a better way than today. The benefit of algorithms for load balancing is that they are preferred to be less complex. In terms of implementation and operations, the more complex the algorithm is the more complex the process is. Hence, less complex algorithms can be more preferable, rather than more complex algorithms to suit the dynamic situation of a cloud data centre [1].

## 2.4 Live Migration

Live Migration is the process of moving the state of a running VM or application to any other host or different PM, without interrupting or effecting the state of the VM being migrated. RAM, CPU, storage and network connectivity of the VM is transferred from the original machine to the destination of the migration.

In a data centre there can be up to hundreds of physical server racks with PMs running. Imagine the PMs only consisting of one VM on each PM, this would be a total waste of energy and resources. Live migration exists for the purpose of avoiding such a scenario, by migrating and distributing one or multiple VMs on to one or several underutilized PMs. This helps freeing up space and saving consumption [10].

Migration can be used for several purposes, either for the purpose described above -to free space - or it can be used to balance load in a overutilized data centre. Load balancing is the term used to avoid a situation where a PM can become overutilized. A desired solution is to have the load balanced equally across one or more PMs.

In simple terms, migration is the process of moving the state of a running VM or application to any other host or different PM, without interrupting or effecting the state of the VM being migrated. RAM, CPU, storage and network connectivity of the VM is transferred from the original machine to the destination of the migration [22].

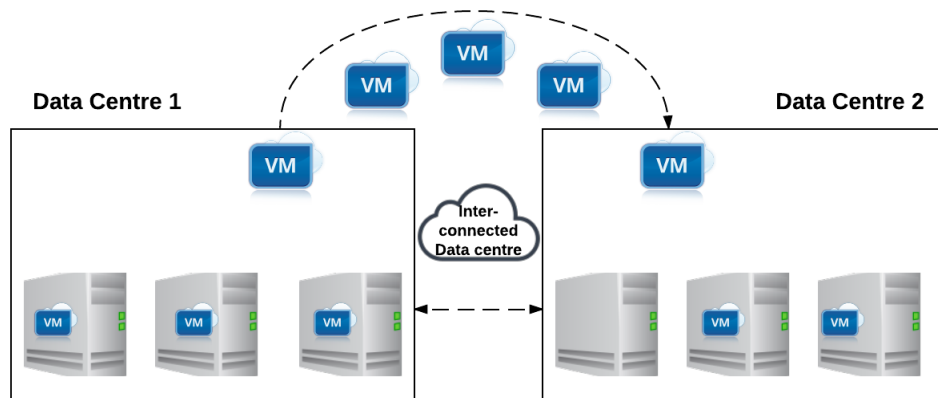


Figure 2.6: Simulation of Live migration

### 2.4.1 Process of Live Migration

Though the migration in itself isn't challenging, moving the contents of a VM's memory from one PM to another it is important that the transfer of the VM happens in a manner that can help reduce the downtime and *total migration time* [5].

Total migration time is the time it takes from one VM to move its memory and details from one host to another, till it arrives the destination host and is running. A migration can happen in the following phases:

- Push Phase
- Stop-and-copy Phase
- Pull phase

In the *Push Phase* the source VM continues to run, while some certain pages of the VM is sent over the network to the new host destination. In the *Stop-and-copy phase*, the source VM is stopped and the remaining pages are transferred to the end destination VM and the VM is again started at the new destination. The last phase is the *Pull Phase* in this phase the VM is initiated again and up on the new destination [5].

There are also the many phases that a migration process goes through. However, to cover the most common and basic ways, *Pre-Copy Migration* and *Post-Copy Migration* is presented to understand what happens before and after a migration.

### 2.4.2 Pre-Copy Migration

This is the process where the VM, which is supposed to transfer from Host A to Host B, is prepared. The hypervisor copies all the memory pages from Host A to Host B, while the VM is still running on Host A. If some of these memory pages become *dirty* (change) during the process, they will be re-copied as long as the *dirty rate* allows it. Dirty rate is the rate at which the disk or memory changes during the migration [5, 32].

### 2.4.3 Post-Copy

Between the pre-copy and post-copy migration process there is the *Reservation* process where a request on Host B for the VM from Host A is made. This is to ensure there is enough resources to be able to have a successful migration. The next process is the *Iterative pre-copy* process where all the pages are copied over from Host A to B, including the dirty pages. Next the *Stop-and-copy* phase stops the instance running at Host A and redirects the network traffic to Host B, the remaining memory pages(if any) and CPU state is transferred. In the *Commitment* phase, Host B acknowledges to Host A that it has received the OS image. Host A will respond and discard the original VM. Host B is now the primary host. The last *Activation* process, is when the VM hat was migrated to Host B is activated. The post-migration code runs to re-attach the drivers to new host and advertise traffic correspondingly [5].

## 2.5 Load Balancing vs. Consolidation

Load balancing is a distinct term which is self-explanatory. For the common man it means the same as a load distributed over two or more sets, made to carry a load of any kind. This can be compared to a regular trip to the grocery store, where all the groceries bought are distributed into multiple grocery bags. To make the load feel even, one would want to have the same amount of weight in each bag. This analogy can be transferred to servers. Load balancing is a technique of distributing the amount of connections to a machine over multiple machines, based on how much traffic is directed at the node.

The technique of load balancing exists to handle the problem of overutilization of a node. By overfilling the grocery bag, a certain risk of the plastic tearing up always exists. In the same way each server can only handle a maximum amount of load before it violates the maximum capacity and exhausts the server. High amount of CPU and memory usage are the main problem which can cause exhaustion and a potential system break down.

The grocery bag analogy can also introduce another approach which is a highly popular solution in the industry today, *Consolidation*. To be effective and save money, one would skip on distributing the load and fill the groceries in one bag as efficiently as possible. In non-human language, this means that the load of the servers should be packed onto least possible servers to save resources. This reduces the total number of active servers but this does not necessarily always reduce the risk of server exhaustion.

The ideal way to deal with traffic and VMs is to create a data centre that is scalable at all times. Load on servers can be dynamic because the percentage can change over time as the load can increase and decrease, active and inactive servers can start/stop depending on the traffic. The problem is not the dynamic, it is how the load varies and how a void is created due to the fact that a VM is started up and shut down at different times. This causes the system to be thrown out of balance.

There are several algorithms which try to address the void-problem, where the deployment of an VM is decided beforehand. For instance in this project a solution which is adjustable and can adopt to any number of incoming requests is presented. This approach should not only work for smaller environments but in larger data centers as well. The deployment of the VM is decided by the available number of servers and not pre-decided deployments. In this way the void-problem is avoided and the exhaustion problem justified.

### 2.5.1 Load Balancing strategies

Load balancing has many benefits, it improves the performance of each individual node, hence the overall system performance. There is a short response time, higher throughput, higher reliability, low cost but high gain. Most of all it has an incremental growth [31].

Load balancing offers a wide range of strategies or algorithms to address problems with imbalance in data centers. There are two categories which load balancing can be defined under and these are *Static Load Balancing* and *Dynamic Load Balancing*.

In Static load balancing algorithms each process is assigned to the processors according to its performance, there are no reassignment or changes of the tasks. The number of jobs in each node is fixed, and the nodes do not collect any information [2, 31]

In Dynamic load balancing algorithms each job can be reassigned at the run time depending on load being transferred from heavy to light node. No decision can be taken until the process is done and executed. Dynamic load balancing is mostly used for systems that are heterogeneous, due to the difference of speed, link speed, varying memory and load values.

In this project the Stable Marriage algorithm is implemented exactly to gain load balance in a dynamic infrastructure, to strengthen and optimize performance of a system.

## 2.6 Centralized, Decentralized, Distributed & Swarm Intelligence

The term *Distributed* was briefly introduced along with *Centralized* in the introduction. Each approach is different and is not to be confused with each other. As the terms implies, a *Centralized* system is where there is one central manager who controls the entire system. A *Decentralized* system is a system which operates on local information to accomplish one mission. *Distributed* system is where several components or computer nodes interact with each other through message exchange in order to achieve a common goal. *Swarm intelligence* is the behaviour of a decentralized system where the concept is usually employed on artificial intelligence, some examples include ant colony, bird flocking, honeybee intelligence and so on [9, 37–40].

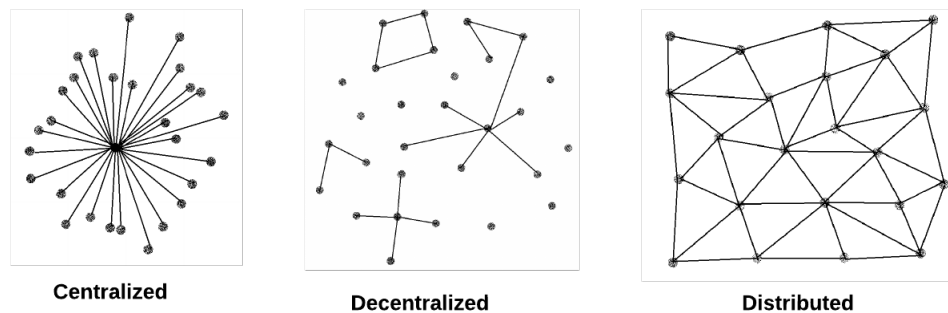


Figure 2.7: Difference between Centralized, Decentralized and Distributed [9]

Figure 2.7 shows the distinct difference between the three main approaches. In terms of maintenance, a centralized system is easier to maintain. There is one central manager, if something needs repair it doesn't take long. Centralized systems are also good if a fast growth is needed and scaling is not a concern. A decentralized system might be a tad difficult to maintain but works well in terms of management, these are especially good for problem solving projects. Distributed systems are often hard to maintain, but in return very stable, a small failure will not effect the rest of the system. In terms of scalability, distributed systems are one of the best options [9].

### 2.6.1 Self-organizing Algorithms

Automation and autonomic compute systems are well known terminologies. It's a common misconception that systems today are completely automated. There is still a long way to go before systems will be completely free of user intervention. The term *Autonomic* was first applied in the IT-sector

by IBM. This implied that systems are able to *self-manage, self-configure, self-protect and self-repair*. The term *Self-organizing* can be defined as a set of dynamical mechanisms which has the capability to interact among low-level components of a system [4].

A system is built up by many components and layers, research in recent years on self-organizing systems have started to surface. The studies mostly focus on finding algorithms which can make systems even more efficient, both in terms of efficiency in virtualized environments which helps create a system which can manage itself and as well as less user intervention. A very notable approach to obtain self-organizing computer systems in recent years is the focus on nature-inspired algorithms. The term nature can be defined in many ways, in this case with inspiration from other fields than computer science.

The reason for this is simple, many of the algorithms seem to show good results. By using inspired behaviour from humans, animal kingdom etc, many rules can be applied as algorithms in science. Section 2.10 introduces a few studies on such inspirations.

## 2.7 Bin Packing

There have been numerous sections which have described an ideal situation for a data centre. The terms *efficiency* and *resource minimization* are repeatedly used. Up until now it is evident that in any given data centre the goal is to pack the physical resources to minimum, while avoiding situations that can create problems with the efficiency and performance. *Bin Packing* does exactly this in mathematical terms [10, 13, 35].

Bin packing is the concept which allows a way to pack any given number of objects of different sizes. Bin packing is known as an *NP-hard* problem. NP-hard is a term which is shortened from *Non-deterministic polynomial-time hard* [11, 36]. In the field of computer science a problem is NP-hard when an algorithm is needed for solving it, and that algorithm can be applied to any NP-problem, it can also be translated to "*At least as hard as any NP-problem, although it might be harder*" [43].

The NP-hardness problem applies for virtual environments because of the constant change in the environment. The dynamic situation of a data centre makes it hard to use bin packing as an ideal algorithm, because before the algorithm manages to calculate an optimal bin packing solution for the system, the state is changed [10, 43].

In a data centre over the recent years the emerging problems with physical resources, power consumption and overutilization has been a hot topic. The extreme cost and inefficient way of solving these issues have led to research focusing on Bin Packing solutions. There are many approaches to bin packing, such as 2D, 3D, linear packing or weight packing. Even though Bin packing seems to be an ideal approach, it can be a slow process due to its long calculation time as there are several permutations [10].

## **2.7.1 Bin Packing algorithms**

To name some algorithms which are commonly used a brief introduction of the First fit algorithm and Best fit algorithm is presented below.

### **2.7.1.1 First Fit Algorithm**

The first fit algorithm is based on placement of an item in the first available bin. This does not consider the optimal situation. An item is supposed to be put into the oldest (earliest opened) bin into which it fits first. If the item does not fit in the first available bin, only then is it possible to open a new bin [11, 33].

### **2.7.1.2 Best Fit Algorithm**

The best fit algorithm bases its placement of items on the weight of the items. Each item has a given weight, and the algorithm is supposed to place each item one at the time. The best fit bin for given item is the bin that has the least capacity remaining, but still enough to fit the weight of the given item. In this way the capacity is not violated, but not wasted either. The best fit algorithm makes it possible to fit as many items into as few bins as possible [11, 33].

## **2.8 Stable Matching**

In 2012 the Noble Peace Prize in economics was awarded to Lloyd Shapley and Alvin Roth. Their contribution was an extended development of a real world problem theory from the 1960s. The problem revolved around practical real-world-problems, such as assigning doctors to hospitals, students to schools and human organs for transplantation. Shapley made the early contributions while Roth unexpectedly adopted the contributions two decades later. Roth was investigating the market for U.S doctors. With his further research, findings generated analytical development as well as practical design of market institutions.

The fundamentals of economics is based on demand and supply, where analysis often studies markets where prices adjust so that the supply equals the demand. This is done because in practice the markets functions well on these premises. The problem with this model is however, that prices cannot be used to allocate resources. For instance in the case of human organs for transplantation, monetary payments are ruled out on ethical grounds, in cases like these an allocation still has to be made, so on which basis should this allocation take place as effectively as possible?

Shapley et al. started working on different approaches and analysis of allocation mechanisms. It is a rather abstract idea, if rational people who know what their best interest is and behave accordingly, engage in unrestricted mutual trade, the outcome should be efficient. If otherwise,

individuals would devise new trades that were ideal for their situation. An allocation where no individual perceives any gain from any further trade is called *Stable*. Stability is a central notion in the concept of cooperative game theory, this is an abstract area of mathematical economics which seeks to know how any constellation of rational individuals might cooperate to choose an allocation.

Shapley applied the stability idea to a special case in 1962, in a short paper he examined the case of *pairwise matching*. Pairwise matching was the idea how individuals can be paired up when they all have different views regarding who would be the best match. The matching was analysed at an abstract level where the idea of marriage was used as an illustrative example.

For this experiment they tested how ten women and ten men should be matched, while respecting their individual preferences. The main challenge was to find a simple method that would lead to stable matching, where no couples would break up and form new matches which would make them better off. The solution was *deferred acceptance*, a simple set of rules that always led straight to the stable matching.

Deferred acceptance can be set up in two different ways, either men propose to women or women propose to men. If women propose to men the process begins with each woman proposing to the man she likes the best. Each man then looks at the different proposal he has received, if any, and regards the best proposal and rejects the others. The women who were rejected in the first round, then move along to propose to their second best choice. This will continue in a loop until no women want to make any further proposals. Shapley et al proved this algorithm mathematically and showed that this algorithm always leads to stable matching.

The specific way the algorithm was set up turned out to have an important distributional consequence. It mattered a great deal whether the right to propose was given to the women or to the men. If men proposed this led to the worst outcome from the women's perspective. This is because if women proposed some women would end up with men they liked even better. No woman would be worse off than if the men had been given the right to propose [17].

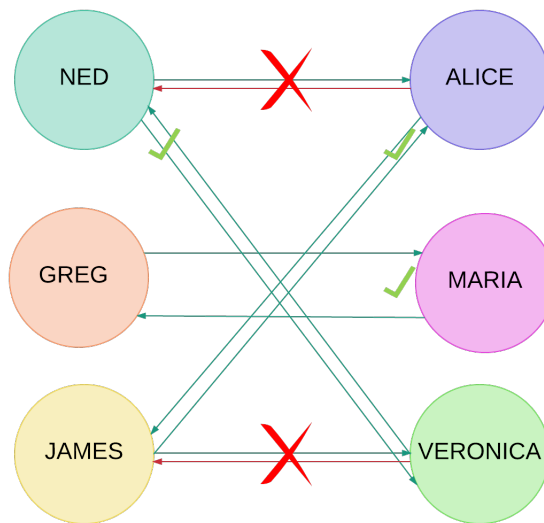


Figure 2.8: Stable Matching

The model 2.8 presents the selection process for Stable Matching. On the right side are the women with their preferences and to the left are the men with their.

## 2.9 Tools

### 2.9.0.0.1 Python

Python is a programming language known to be simple and easy to learn. The syntax accentuates readability, which the industry has a benefit of considering this reduces program maintenance, in addition the code is pleasant and orderly to read. The Python dictionary is based on the English language, which makes it globally easier accessible. **Version: 2.7**



Figure 2.9: Python Logo

### 2.9.0.0.2 Eclipse

Eclipse is a Java-based program-platform that allows the programmer to create a customized development environment (IDE) from plug-in

components, built by other Eclipse members all around the world. In short, Eclipse is used as a editor to develop programs with many different languages like C, C++, PHP and Ruby etc. Customarily Eclipse does not have any editors where Python can be downloaded as a default, PyDev is however, a plug-in that can run with Eclipse and allow Python scripts to run and compile like any other editor. **Version: Neon Release (4.6.0)**



Figure 2.10: Eclipse Logo

#### 2.9.0.0.3 Git and Bit Bucket

Git is a well known platform for developers who want to ensure backup of their code. The hours spent working are not worth a loss if faced with technical issues. The source code of this project was securely stored through Git and Bit Bucket for version control.

#### 2.9.0.0.4 Dropbox

Dropbox is a cloud solution where users can store their files and access them with their user-account at any device where Dropbox is found. To make sure that the project was safely stored, Dropbox was also used to ensure safety of the files and easy access from any device with the updated version at all times.

#### 2.9.0.0.5 QEMU & KVM

KVM stands for Kernel Virtual Machine and it is built into the Linux OS. KVM is a hypervisor program that allows different and multiple operating systems to share a single hard-core host. Many guest OS's can work with KVM. QEMU is also a hypervisor just like KVM. The difference is that QEMU can make use of KVM when running a target architecture that is the same as the host architecture.

#### 2.9.0.0.6 LIBVIRT

Libvirt is a toolkit used to manage virtualization hosts. Libvirt supports KVM, QEMU and many other hypervisors. With the help of Virsh, an interactive shell it is possible to script and manage tasks on all Libvirt domains, networks and storage.

Section 2.2 gives a deeper insight to the technologies QEMU, KVM and Libvirt.

#### **2.9.0.0.7 TinyCore**

To create a realistic simulation of a data centre every detail needs to be recreated as it would be in a real data centre. TinyCore offers the possibility to create customised Linux based VMs, which works just the way a Linux node would. This allows several guest OS's to run on top of one host OS, which can simulate a data centre.

#### **2.9.0.0.8 ISC DHCP**

To provide each VM with an IP address and MAC address, the open source software ISC DHCP is chosen. It offers solution for implementing DHCP servers, relay agents and clients for small local networks or larger enterprises.

## **2.10 Relevant Research**

This section provides the background for this project. The subsections will be divided into two different categories; research on Load balancing and research on consolidation, as well as some important research on nature inspired algorithms. Usually the path to achieve load balancing or consolidation is based on somewhat same approaches such as inspired algorithms from different fields aside from the computer science field.

In recent years there has been very little research on load balancing and there is comparatively more research on consolidation. Considering that both approaches use inspiration from different fields and try to solve the same problems in computer science, it is important to study their approaches and results. Hence, in this section relevant studies for both the approaches are presented.

### **2.10.1 Load Balancing in Cloud computing through Nature inspired Algorithms**

#### **2.10.1.1 Load balancing in structured Peer-to-Peer systems**

Rao et al. proposes a solution which is quite considerate to scalability by using the peer-to-peer protocol as mentioned in section [thesis Peer2-Peer resource allocation]. With the help of each peer knowing its neighbour we get the view to the entire system. Rao et al. also uses data hashing table (DHT) in their article, which stores the unique ID of each peer in a table making it easier and faster to find a peer based on a unique ID.

Rao et al. present three different but simple techniques to achieve load balancing in a structured system. The three different approaches are the way they balance the load by migrating nodes from one place to another. The first technique one-to-one picks two random virtual machines, where one is a heavy node and the other one is a light node. Each light node can periodically pick a random ID and then perform a look up operation to find the node that is responsible for that ID. If that node is a heavy node, then a transfer may take place between the two nodes. The second scheme is the "One-to-many" scheme, in this scheme a heavy node is allowed to consider more than one light node at the time and migrate to the lightest one after choice. The third and last technique is the many-to-many scheme, which is an extension of the first two schemes. In this however, there is a concept of a global pool where each heavy node drops off their weight. This happens over three phases; unload, insert and dislodge. Without going into much detail this is where heavy nodes come to drop their "weight" and unload it until they become a light node. The idea is to transfer all virtual servers from the pool to light nodes without creating any new heavy nodes [27].

The system is simulated with 4096 nodes and all three schemes do very well in terms of scalability. This research considers the multi-dimensionality problem and as the entire point of a load balancer is to utilize resources in the best possible way, considering scalability.

### 2.10.2 Extended scheduler for efficient frequency scaling in virtualized

Dynamic Voltage and Frequency Scaling (DVFS) is introduced as a hardware technology to dynamically modify the processor frequency, depending on the CPU needs, in order to reduce energy consumption. Kamga et al. [14] presents a solution consisting of VM scheduler and DVFS to conduct some experiments. With seven powerful nodes and help of Xen to virtualize the environment for experiments, DVFS seemed to be a powerful tool, but generated low VM performance.

#### 2.10.2.1 Ant Colony optimization

The first time an *Ant-Colony* algorithm was introduced was by Marco Dorigo in 1991 while he was writing his PhD thesis. There are several ant-colony based algorithms. Figure 4 shows the list of the different algorithms presented over time.

The Ant Colony Optimization algorithm has been used to solve optimization problems. Generally the idea is based upon the way the real ants solve problems using pheromones. They are capable of finding the shortest path from a food source to their nest by leaving pheromones on the ground which can be picked up by other ants. All ant algorithms use pheromones as a chemical messenger [44].

In one of the studies, Yaseen G. in [7], sorts the problem of the algorithms into two different rules the ACO system applies, local pheromone rule update and global pheromone updating rule. The study further suggests that an ACO algorithm include two more mechanisms, tail evaporation and optionally daemon actions.

By studying the behaviours we can create rules based on the information, by understanding how the ants think by dropping pheromone on the ground, which then other ants detect, following this path and determining the shortest path from one source to another. By understanding these, researchers divide the behaviour into short “list”/rules, which can be applied to a compute node to make the node behave in a similar way, to find the shortest path during a migration from one compute node to another compute node, closest to its own source [7, 44].

#### 2.10.2.2 Honeybee behaviour

Swarm intelligence has been a popular choice for research in recent years. Karaboga [15] introduces new research on swarm optimization algorithm through research on honeybees. To understand the concept of swarm optimization, Karaboga [15] divides the fundamental concepts into two different sections self-organization and division of labour. These are necessary properties to obtain swarm intelligent behaviour, such as solving problems with distributed systems that can self-organize and adapt in their given environments. The first concept of self-organization can be

divided into four sub-fundamentals: Positive feedback, negative feedback, fluctuations and multiple interaction.

The positive feedback is a behaviour, which enhances recruitment and reinforces trail laying and following group of other bee in “dances” the bees do. Whilst negative feedback counterbalances the positive feedback to help stabilize the collective pattern. This might occur in terms of available foragers, food source exhaustion, crowding or competition at the food sources.

Fluctuations is defined as random walks, errors, random task switching among swarm individuals, these are vital for the creativity and innovation of the swarms. The randomness intact within swarms is often crucial for structures and finding possible new discoveries and solutions.

Imagine being an employee at a company, you are hired to take care of the food stock coming in and out of the place. This is the work the forager bees are hired to do. They are employed as specialists at a particular food source. They carry the information about the profit, its distance from nest to food source and share this information with other bees.

The unemployed bees/scouts can be imagined to be as unemployed humans searching for a job, a better job, a job nearby or a job with better income, and then share this information with other bees.

Karaboga described a simple and flexible technique, which was simulated through two different functions. *Sphere and Rosenbrock Valley*, the latter being a known optimization problem. The simulation was tested on very limited set of problems. The numbers however seem promising and the algorithm is suitable for unimodal and multi-modal numerical optimization problems.

### **2.10.3 Consolidation through peer-to-peer algorithms**

#### **2.10.3.1 Gossiping Algorithm V-MAN**

Marzolla et al., proposes a decentralized gossip-based algorithm, called V-MAN to address to the issues regarding consolidation in their thesis. V-MAN algorithm is an algorithm based on a gossiping protocol\*. The entire focus of the algorithm is on the VM itself, which means it is independent from any sort of application or instrument to be a central controller. The algorithm – V-MAN - is performed periodically to create new arrangements for existing VMs onto fewer servers, maximizing empty servers.

Marzolla et al., assumes that the cloud system has a communication layer, so that any pair of server can exchange messages in between. In section of related work Marzolla et al., talks about another thesis Bio-inspired algorithm in which the algorithm used in the thesis, they use scouts on the migration work where the scouts are allowed to move from one PM (physical machine) – to be able to recognize which compute node might be a suitable migration destination for a VM. This is completely opposite of what V-man does. Where V-MAN does not rely on any subset

like scouts, instead each server can individually cooperate to identify a new VM location, which makes V-MAN scalable. It is also to be noted that any server can leave or join the cloud at any time.

The thesis is based on simulation conducted with four different experiments. Each experiment has V-MAN implemented but the first two experiments use a limited amount of servers and VMs in the experiment and no server can leave and join the cloud. The two latter experiments, experiment 3 and 4, simulate a more realistic scenario, where servers can leave and join the cloud and the amount of nodes are a realistic numeric.

In the end the thesis has a very promising result. The results show that using V-MAN converges faster – less than 5 rounds of message exchanging between the servers – and the algorithm was also resilient to server failure.

A gossip protocol is a style of computer-to-computer communication protocol inspired by the form of gossip seen in social networks. Modern distributed systems often use gossip protocols to solve problems that might be difficult to solve in other ways, either because the underlying network has an inconvenient structure, is extremely large, or because gossip solutions are the most efficient ones available[20].

### **2.10.3.2 Cooperative VM Consolidation**

In an extensive research performed by Sedaghat et al. A distributed algorithm and gossiping protocol is considered to gain an effective approach to consolidation. Sedaghat et al, use Peer-to-Peer protocol to achieve energy efficiency and increase the resource utilization. Peer-to-Peer protocol provides mechanisms for nodes to join, leave, publish or search for a resource-object in the overlay or network. This also considers multi-dimensionality – because the algorithm needs to be specified to be dimension aware, (each PMs Proportionality should be considered).

Each node is a peer where a peer sampling service, known as newscast, provides each peer with a list of peers whom are to be considered neighbours. Each peer only know  $k$  random neighbours, mapping its local view. In the research by Sedghat et al, using the gossip-based protocol, there are continuously decisions made in order to improve a common value. Which is defined as the total imbalance of each pair at the time of decision-making and the goal is to reduce this imbalance by redistributing the VMs among them.

The thesis also modifies and uses a dimension aware algorithm to consider – as mentioned above – the multi-dimensionality of the problem. The algorithm is an iterative algorithm which starts from an arbitrary VM placement. When the algorithm is converged a reconfiguration plan is set so the migration of the VMs can start. The reconfiguration will re-consolidate only if the three steps in the thesis are followed[30].

### 2.10.3.3 Peer-to-Peer resource Allocation

The next thesis also uses a distributed algorithm and gossiping protocol to gain an effective approach to consolidation. Sedaghat et al., use Peer-2-Peer protocol to achieve energy efficiency and increase the resource utilization. Peer-2-Peer protocol provides mechanisms for nodes to join, leave, publish or search for a resource-object in the overlay or network. This also considers multi-dimensionality – because the algorithm needs to be specified to be dimension aware, (each PMs Proportionality should be considered).

Each node is a peer, where a peer sampling service, known as newscast provides each peer with a list of peers whom are to be considered neighbours. Each peer only know  $k$  random neighbours, mapping its local view. In the thesis, using the gossip-based protocol, there are continuously decisions made in order to improve a common value. Which is defined as the total imbalance of each pair at the time of decision-making and the goal is to reduce this imbalance, by redistributing the VMs among them.

The thesis also modifies and uses a dimension aware algorithm, to consider – as mentioned above – the multi-dimensionality of the problem. The algorithm is an iterative algorithm, which starts from an arbitrary VM placement. When the algorithm in converged a reconfiguration plan is set so the migration of the VMs can start. The reconfiguration, which will re-consolidate only if the three steps in the thesis are followed.

Simulation of 100.000 physical machines and 200,000 VM requests. Results: 75% resource utilization, whereas the aim is 90% - done in less than 7 cycles [29, 30].



**Part I**

**The project**



## Chapter 3

# Approach

This chapter outlines the problem statement: *Can we achieve load balancing in a data centre using simple message exchange between physical machines in a distributed manner?.*

As well as:

*How can we borrow the principal of Stable Marriage to gain load balance in a distributed data centre?*

The problem statement introduces a set of important underlying questions such as scaling problems, optimization of performance and energy efficiency issues. In this section the approach to provide a ground for research through implementation of Stable Marriage is presented.

The approach will outline some of the following sections:

- Design
- Environment
- Schemes
- Algorithm design
- Implementation Design
- Expected Results

### 3.1 Objectives

To outline the structure of the project that has been introduced through various inspirational studies and concepts in the background chapter, there will be three main objectives in this project. In the first phase the design of the project is presented, the implementation phase next and finally the results and analysis. Each objective will be presented through a detailed breakdown of the structure with designated sections.

Below is a list of what the idea behind each objective is:

#### 1. Design

- (a) Create a model of the simulated and real life environment.
- (b) Create a model of the Stable Marriage algorithm with formal notations and illustrations.
- (c) Introduce tools, prerequisites and features of the framework.
- (d) Provide an overview of the working framework and hardware.

## **2. Implementation**

- (a) Configure the environment for simulation and experiment.
- (b) Introduce implemented features as designed in the first objective.
- (c) Implement the algorithms while testing it during build.

## **3. Result & Analysis**

- (a) Conduct tests to ensure the environment is ready for experiments.
- (b) Execute simulation, test Migrate Large.
- (c) Execute simulation, test Migrate Small.
- (d) Execute experiment, test Migrate Large.
- (e) Execute experiment, test Migrate Small.

### **3.2 Mapping the load balancing problem to Stable marriage**

System administrators today are on call 24/7. The need to ensure systems are up and running at all time has introduced variety of issues that did not exist until only a few years back. The job of a system administrator has been ever evolving, new languages, new software, new versions of an OS, going from only having computers at work to almost everyone owning a computer in just a few years. This again introduced a series of things to consider. People working from home, customers trying to access information on company webpages, the complexity kept and keeps growing.

Today is a day and age where there is no fine line between automation and manual labour anymore - we are somewhere in between. Virtualization is a concept which has been around for a long time however only in recent years became a useful tool with the growth of automated systems. With growing demand new and greater ideas formed the systems we have today. One particular system is the cloud system, which is probably one of the most complex systems available today. The extreme demand for cloud solutions has become a big business, now this is where system administrators play a major role.

With almost every infrastructure trying to adapt this new cloud system customers cannot simply accept a "Sorry, our system is not working today"

anymore. A few years ago this was a valid reason and universally accepted, however as the technology evolves, humans have more expectations and they expect systems to be working at all times as well. How can system administrators assure that the already most complex system is available *and* efficient at all time?

The clue would be to not add more pressure on system administrator's work with more manual labour but introduce a more scalable and balanced system, where no decision made by a system administrator would have to exclude important variables to the system.

Thus in this chapter we focus on the problem statement and connecting it to the Stable marriage algorithm through the approach with design and implementation .

### **3.3 Design**

Figures, drawings and images are a few ways of visualizing any particular problem. By visualizing problems, one automatically narrows down the path to a solution. To help visualize the solution a combination of models and figures as close to the actual implementation is provided as a prototype. The complexity of the Stable Marriage algorithm is best described with the support of figures and graphs. The algorithm has a set of requirements, which are expected from the work environment as well as the simulation and each requirement will be identified and explained.

#### **3.3.1 Modelling**

In a complex environment it is desirable that the process of any given project is disassembled into pieces with in-depth details to provide a larger understanding. This also gives a better understanding of every feature, the functionalities and the policies set for a framework. These functionalities and policies will be described through a series of models to support the visual overview of the framework before diving in the implementation chapter to look at the underlying technologies.

A model can be many different things, it may be graphs, diagrams, tables and images. Stable Marriage is a algorithm which can be related to real life models.

Diagrams will act as models to visualize the chain of process each function of the framework is built on. The mind mapping diagrams will be used to display the process of planning the necessary requirements for a functioning framework, while the flow diagrams represent the data which the flow of the system goes through.

The design phase will also provide an overview of a functioning framework as well as the simulation. These models will represent the practical parts of the framework, as well as what is expected final result after the implementation of the framework. It is however difficult to predict

if the final result may match what the models in the design phase show. To ensure that the models and end result are similar to some extent the models will be designed with a basic case which will reflect the result in any case.

### 3.3.2 Load balancing Schemes

Stable marriage is, as explained in section 2.8 the technique of finding the right partner out of preference. However, the same way a selection of partner can have different approaches through either dating or being set up, the algorithm can have more than one way or scheme to find the perfect balance for a system. All the schemes try to balance the load by migrating VMs from over-utilized servers to underutilized servers. The difference between the schemes are the amount of VMs and data transferred and that the selection process is by mutual consent.

In this case the different approaches to find a suitable partner will be divided into two different schemes. These can be viewed as two different *sub schemes*, where the weight and volume of a server is the most attractive quality of a preferred partner. The schemes are straightforward, to avoid any misconception or doubt the schemes are simply named *Large* and *Small*.

Stable Marriage is the principle-underlying algorithm at all times, the point is to create a stable and even data centre. However, as mentioned above the same way a marriage approach can have different ways, a data centre may also vary in shape and size because of its dynamic nature. The large and small scheme will hence be applied to the Stable Matching algorithm as schemes, to make sure that all volumes, small, medium or large servers and infrastructures are scalable in every situation possible.

In the design phase, to provide an understanding of how the Stable Marriage algorithm works, a demonstration will be given in a step-wise model to display the way the implementation may take place and what the design is thought to be. The model will only demonstrate how the algorithm works, and not the schemes in this case.

## 3.4 Implementation

The implementation phase is the technical part of the project. In this phase all the concepts, models and functions described in the design phase are developed to be implemented to an actual framework. Programming and coding are not the only part this phase consist of. The environment the framework is supposed to be constructed under plays a major role as well. That is why the configuration, setup of virtual environment and as well as the physical environment is also to be covered under this chapter.

### 3.4.1 Environment

The project will be implemented in a real working environment, which will try to imitate a real data centre. The physical environment, the hardware and utilities are in a server room at the Oslo and Akershus University College. The college provides all the physical utilities and bandwidth.

The problem statement revolves around load balance in cloud data centers, which implies that the framework needs to be based on a virtualized platform. The servers will run Ubuntu 16.04 LTS as native and the virtualized environment will be using QEMU/KVM and libvirt as a solution for virtualized networks and hosting of VMs. These are all open source tools and easily accessible, in terms of cost and efficiency these tools will not be hard on the budget and resolving problems that might occur may be easier to solve.

### 3.4.2 Framework

The framework itself is divided into two parts. The first part is the configuration of the virtual environment, spawning, deploying, migration and general management of the VMs. This also includes creating custom image for the TinyCore VMs. The first part of the framework is crucial, unless the virtual environment is configured and the volume needed is generated, the implementation of the algorithm is pointless. The second part is the implementation of the algorithm and the schemes. This is done in python version 2.7.

### 3.4.3 Data Collection & Comparison

The aim of creating such a framework is to provide some sort of results. To form a result, some type of data is needed. The problem statement is basically asking for improvement of current load balancing techniques in data centers. This means that the outcome of the project should be *better than* the current situation of any given data centre, to call it a success. The data that will be extracted from this framework will focus on four outcome:

#### 1. Situation Mapping:

- (a) Average CPU load
- (b) Imbalance before migration
- (c) Imbalance after migration
- (d) Gain of migration (better/worse - exit if no gain)

As mentioned earlier the algorithm will be implemented with two varying schemes where either larger VMs are migrated or smaller ones. After collecting data from the various experiments the framework will go through, graphs of how the results turned out will be displayed. These graphs will be generated for each individual test and in the end comparisons of both the schemes will be given with analysis of the data represented.

### 3.5 Result & Analysis

The experiments will be separated into two categories. The first category will be the configuration experiment and the second category will be the scheme experiment. The configuration experiment will test to see if the virtual configuration and its functionalities work and are robust enough to be able to handle the scheme experiments. The scheme testing will base upon the migration of large or small VMs. The experiments for the schemes can be divided into two categories again, simulation vs. real working experiments.

For the sake of readability, each experiment will follow with a *Analysis* section. This is to help the reader understand the experiment and the results extracted. The best possible way to display the graphs will be decided at the time the results are extracted. Depending on the results, it is hard to design a picture of which graph would best suit the results

#### 3.5.1 Configuration Experiments

The configuration experiments test the environment. The virtual environment, as mentioned, will be built using QEMU, KVM and Libvirt. QEMU/KVM will be used to emulate machines and Virsh the command line of libvirt to delegate tasks. Before implementing the algorithm the environment will be tested with a few simple tasks:

##### 1. Task

- (a) Install QEMU, KVM, Libvirt
- (b) Spawn VMs, delete, pause and onto VM
- (c) Migrate VM back and forth between two nodes
- (d) Monitor the memory of the system

### **3.5.2 Scheme experiments**

Since both the Large and small migration schemes will have the same underlying structure and code the same experiments will be conducted for both of the schemes. The data which will be extracted can be compared to find which is more efficient or if both generate results which show a better gain in any given situation.

The tasks that the schemes must perform are:

#### **1. Task**

- (a) Execute scheme large
- (b) Read the imbalance ratio
- (c) Execute scheme small
- (d) Read the imbalance ratio

### **3.6 Expected Results**

Stable Marriage has never been implemented in computer science before. As pioneers, it is hard to predict if the complexity of the idea works the same way after implementation. However, the experiments will show the change of the situation in a data centre based on the imbalance of the centre before the schemes and algorithm were implemented, as well as after.

Each scheme based on the experiments will help indicate whether there are benefits of migrating larger or smaller VMs or if both show benefits. Due to a completely new territory, it is also not unreasonable to expect technical difficulties with implementation, configuration, problems with migrations and even coding errors and bugs.

### **3.7 Appraising Properties**

The vast amount of research and studies on topic of load balance and consolidation in cloud data centers highlight that there is a lot of room for improvement. The problem statement and introduction introduces current problems at traditional data centres. Scaling in data centers has been a problem the recent years due to the dynamic nature of such an environment. To address these issues either load balancing or consolidation is chosen.

This project will implement the concept of stable marriage to gain load balance in an attempt to improve the scaling problem. This will be done with a distributed approach, which means that instead of having one central manager in the data centre where everything can fail, each machine will be an individual with the possibility to plan and execute what is best for the situation. Scaling is an adjustment problem, as mentioned earlier the vast turnover in centers leads to problems.

Stable marriage is a completely new concept and has never been implemented in computer science earlier. The concept has been implemented in an abstract real-world problem by Nobel peace prize winners Alvin Roth and Lloyd Shapley and generated positive results. This contribution holds a lot of promise in terms of results, once it is implemented.

### **3.8 Discussion & Conclusion**

The data extracted from the experiments can tell a lot. Extracted data will consist of three primary values and these will be written to a file and consecutively extracted to draw points for the discussion.

The discussion chapter will summarize the effects of the project and what the outcome of the results were. The chapter will also take a brief look upon all the chapters throughout the project to see how the result were compared to what the expectations and the approach were.

The chapter will conclude with a summary of what the contribution of the project was.

### **3.9 Challenge Prediction**

It is no secret to man that project work can be a variety of emotions. The calm start, irritation, happiness, anger and confusion. This also reflects the normal day of a system administrator. Computer systems are usually reliable, but unfortunate events happen.

There are expectations from the framework, and it has been stressed many times earlier in the project that this is a completely new method to implement. Many unforeseen challenges may arise, it could be anything from problems with implementation, bugs in the code, missing components, problems with the virtual environment and configuration to name a few.

The biggest constraints can be problems with the hardware and its facilities. The small amount of hardware available might not be able to deal with the complex implementation and larger testing. Even if the manipulated amount of CPU and memory allocated to the virtual environment may be able to be steady, the actual hardware may not. This can effect the main problem, which is scalability in a distributed and dynamic environment.

It is better to be prepared for the worst and work for the best, this is the best possible way to deal with any unforeseen problems and that is the motivation of this project.

## Chapter 4

# Result I - Design

The design chapter will present the prototype of the framework before indulging into the technical details in the Implementation and Experiment section. The project is separated in two sets, not only is the project simulated but it is also implemented in a working environment. This chapter will also outline the design of the Stable Marriage algorithm which is implemented in both sets. A deeper insight into the idea for the project will be presented.

### 4.1 Model

Various studies have tried to implement different approaches, within consolidation and load balancing. In the background chapter some of the studies and approaches presented also show that it is common practise to use inspiration from other fields, especially from the nature, to try to solve different problems in the computer science field. There is however no study or implementation of the Stable Marriage algorithm, which shows that there is still plenty of room for introducing methods which can improve the average day of a system administrator and preferably improve the current problems of cloud computing and data centres.

### 4.2 Overview of a functioning framework

As the algorithm implemented will be based on a real life inspiration, it is important to understand that the outcome can end in two different cases. Just as each relationship does not end in marriage neither will the decision of the PMs. Each PM can be viewed as individuals making their own “life choices”.

Figure 4.1 and 4.2 enhances the different outcome the algorithm can opt for and how the framework is set up to work around the execution of the algorithm. Note that the environment later implemented is not in actual data centres however, for the sake of showing that the main goal is to achieve load balance in a distributed cloud data centre, the intention is

to create a framework that may work in any given scenario and setup.

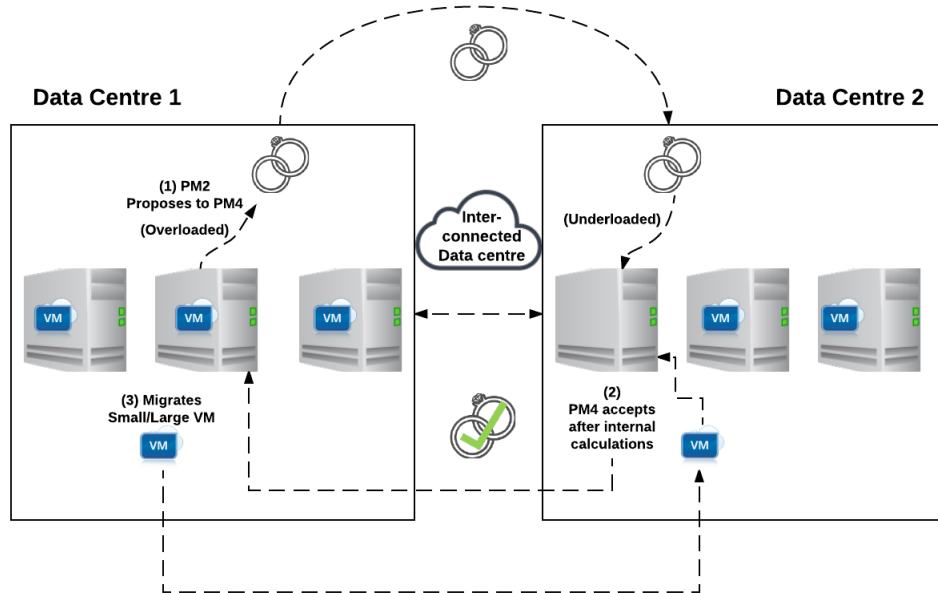


Figure 4.1: Proposal accepted

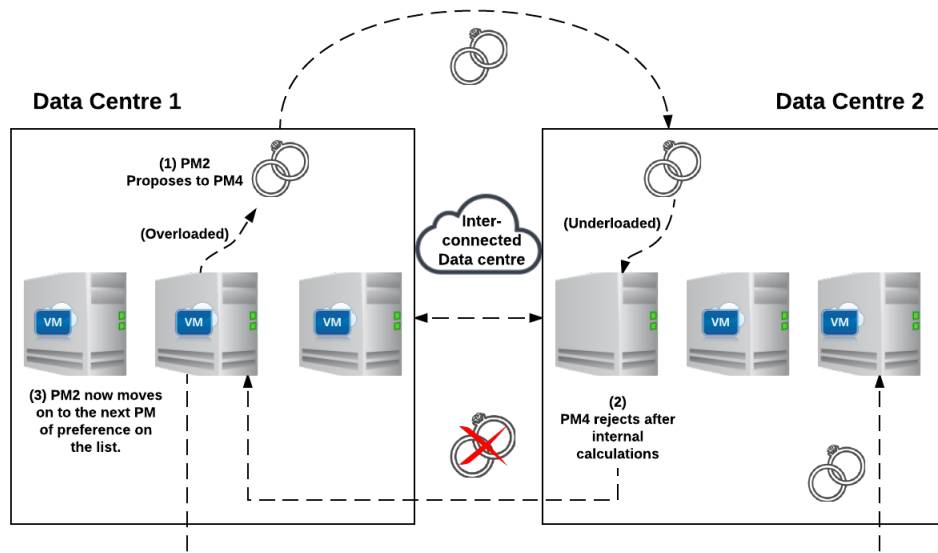


Figure 4.2: Proposal Rejected

The basic framework for both scenarios are the same, it is a data centre

consisting of PMs with different weight. However, as explained in the section above, based on the calculations of the underloaded server in the second scenario the proposal is rejected and the PM moves on to the next best on their list, this process is supposed to be a continuous process, unless the target load for each PM is achieved then the process stops entirely.

### 4.3 Formal Notations

In order to understand the algorithms it is highly important to have a better understanding of the formal notations used in the algorithms. The notations below shows the formal notation defined for the stable marriage algorithm.

$M_i$ : Total used memory of server  $i$ .  $m_j$ : Memory allocated to VM number  $j$ , i.e,  $VM_j$

$$M_i = \sum_{VM_j \in M_i} m_j$$

This means the total used memory of server  $i$  is equal to the the sum of memory of the VMs that are allocated to that server.

Similarly,

$C_i$ : Total used CPU load of server  $i$ .  $c_j$ : CPU assigned  $VM_j$

$$C_i = \sum_{VM_j \in M_i} c_j$$

There are some constraints. Each server  $i$  has a maximum CPU capacity  $M_i^{max}$  and maximum memory capacity

$M_i^{max}$ .

$$M_i \leq M_i^{max}$$

$$C_i \leq C_i^{max}$$

When it comes to consolidation, most algorithms take into account the bottleneck resource as a sole criterion for achieving better consolidation decisions. Similarly, when it comes to load balancing, one can take into account the most imbalanced resource as a criterion. For the sake of simplicity, let us suppose that the CPU is the most imbalanced resource in our data center.

Average load, is the average CPU load,

$$\bar{C} = \sum_{PM_i} C_i / N = \sum_{i=1}^N C_i / N$$

Average capacity  $C^{max}$

$$C^{max} = \sum_{PM_i} C_i^{max} / N$$

Let  $T_i$  be the the target load at  $PM_i$  where there is no imbalance is:

$$T_i = \frac{C_i^{max}}{\bar{C}_{max}} \bar{C}$$

If all the machines have the same capacity, this would reduce to:

$$T(CPU)_i = \bar{C} = \sum_{PM_i} C_i / N$$

Imbalance of a machine  $PM_i$  in terms of CPU load, is the deviation of the load of machine  $PM_i$  from the target CPU load:

$$I(CPU)_i = |T(CPU)_i - C_i|$$

## 4.4 Bin Packing with Stable Marriage

Bin packing is the famous way of packing an  $n$  amount of objects on to least possible bins as possible. In this case, the servers, which the project is implemented on, are called *bins*. This terminology works perfectly with the stable marriage algorithm, as the bins in that particular form would be the humans, where they want to look for a partner/bin which creates least constraints. A constraint can be defined by many different definitions, but for a bin some normal constraints would be the height of the box, its width and depth.

Section 4.3 pointed out some of the constraints the Stable Marriage algorithm will focus on, these constraints were VCPU and memory. Since the aim of the algorithm is to even the weight of the data centre by dividing the weight of the VMs into an equal setting, the bins should never be overfilled or under filled. Best fit algorithm is one of the algorithms that bin packing allows. This particular algorithm can help minimize the number of live migrations, as it focuses on finding the best possible match at all times.

This is why two schemes have been worked out, at all times, the schemes will adjust to the weight of the data centre and find the best match according to the current weight. In this situation either larger VMs, larger in terms of their VCPU, RAM and disk allocation or smaller VMs will be migrated to bins that fit accordingly.

The demonstration below presents the selection process and shows the idea behind the process of the algorithm.

### 4.4.1 Stable Marriage Animation

With a known set of servers divided into two groups overutilized (men) and underutilized (women). The goal is to find a perfect match for the overutilized servers. The matchmaking is based on three values, the average CPU, and the imbalance before and after migration (calculated before the eventual migration) and the profit of such a marriage.

The first figures below will demonstrates the expected outcome of implementing the Stable Marriage algorithm. As this approach is mainly

centralized the PMs know the allocated values of each other, this means that each PM, both over and underutilized has a list of preferred men and women they want to propose to or receive a proposal.

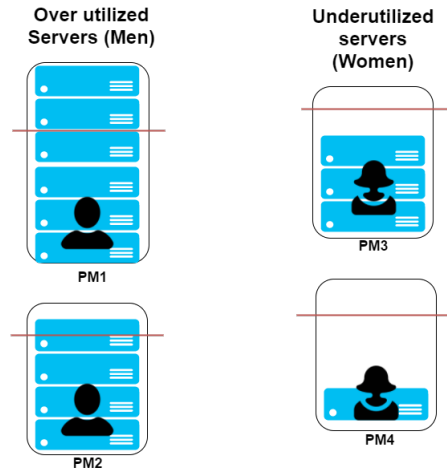


Figure 4.3: Set of over/under utilized servers

PM1 has reached full capacity as marked by the red line. The red line represents the average capacity that each PM can handle. Consider that each group of men can only handle four or six full servers, in this case PM1 has then reached its full capacity and so has PM2. They need to migrate the load to a underloaded PM of preference, so they can balance the load equally. Hence, the overloaded server PM1 proposes to his first choice, PM3.

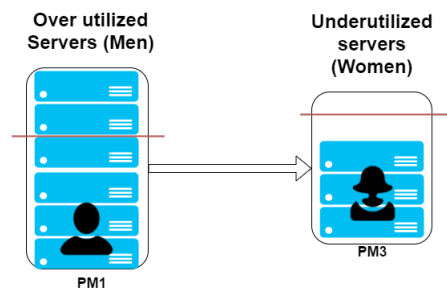


Figure 4.4: PM1 proposes to PM3

However, PM3, the female set of servers have their own capability to calculate the advantage/disadvantage of such a marriage. If the underutilized PM calculates a higher imbalance than before the marriage, she sees this as a disadvantage and rejects the proposal. This is also a great way to avoid the proposing party of getting underutilized in the future.

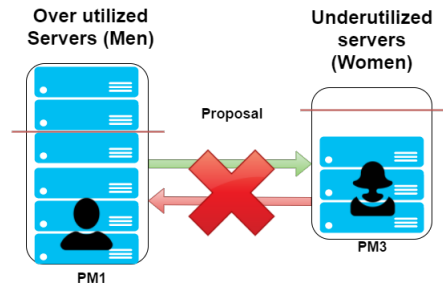


Figure 4.5: PM3 rejects PM1 seeing no benefit to this marriage.

After being rejected, PM1 goes on to his next best choice that is PM4. PM4 then calculates the imbalance before and after marriage to check if it improves after a potential migration. In this case imbalance factor improves, and PM4 accepts the proposal. The migration can now take place.

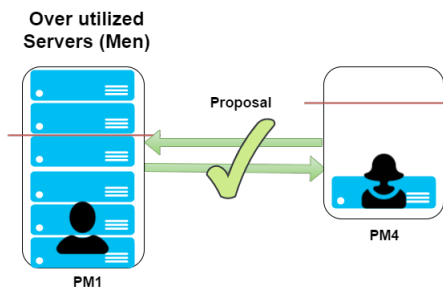


Figure 4.6: PM4 accepts PM1's proposal

Since PM4 has the same amount of capacity to accept load 4/6 is now full. Which means the server is not over-utilized and the load has been balanced between the married PMs.

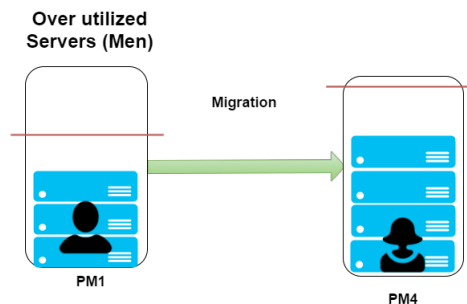


Figure 4.7: Migration successful

This particular animation doesn't have any scheme implemented. The animation gives an idea of how the algorithm is supposed to work. The

schemes will only make a difference in terms of the size that is being migrated. The figure below gives a small insight to how the sizes may differ on each PM and how the migration process may look inside each server.

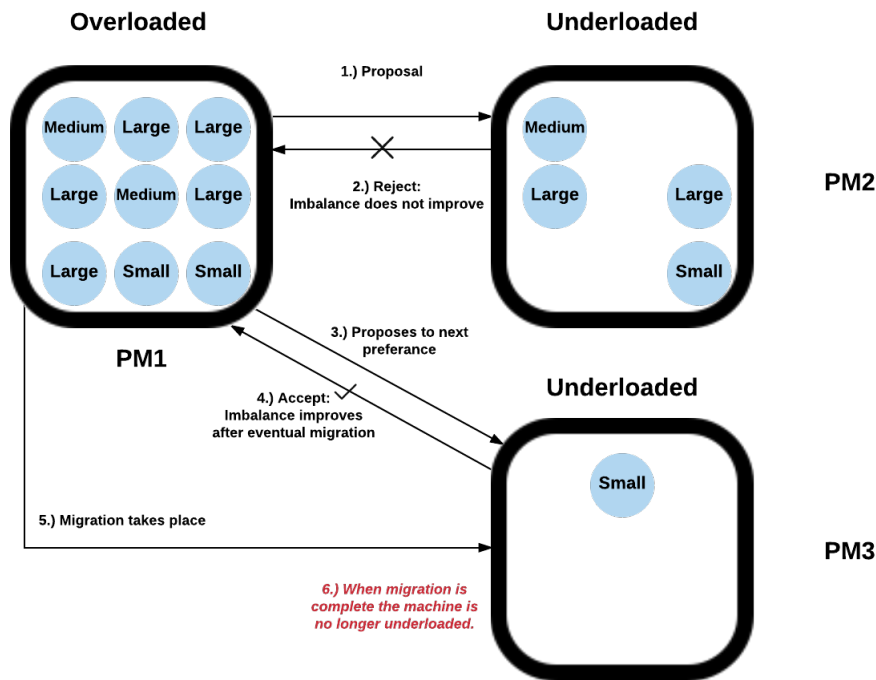


Figure 4.8: PM with various flavours

#### 4.4.2 Libvirt Live Migration

Live migration was introduced earlier in 2.4. Live migration is the process of moving a running VM from one host to another in its current state. Unnecessary amount of live migrations is not good either, this leads to a lot of downtime in total, while the migration is taking place. To avoid migrating unnecessary the Stable Marriage algorithm which focuses on the stability part, will also lead to less migrations, due to the aim of the algorithm to avoid re-allocations.

#### 4.4.3 VMs

To recreate a data centre with intention of it being as close to reality as possible, the VMs play a part in realizing this aspect. As introduced in section 2.9.0.0.7 (TinyCore), the VMs are linux-based with the help of TinyCore. TinyCore offers the possibility to manipulate and customise images as per requirements.

To be able to simulate a real data centre some precautions are important,

to be able to spawn as many VMs as possible TinyCore was a flexible choice, TinyCore can run with varying flavours starting from as little as 46 MB RAM. TinyCore also offers the possibility to manipulate and create custom images or run the default image. One simply has to boot an image file (.iso) and it will run any pre-configured settings and packages.

The project requires that the VMs are doing some sort of work which use CPU and memory to create a realistic effect. Hence all the VMs that will be spawned will perform a SSH-server start up, create a root user, install and configure Python whenever booted.

#### 4.4.4 Node communication

The problem statement introduced *communication* and *message exchange* as a way of mapping the system view for an individual node on search for its best match. The aim for this is to spread awareness between PMs and find the best possible match.

To be able to implement a way to make the PMs communicate together, a socket using TCP protocol will be set up on the available PMs. However, the setup of a socket will depend entirely on the hardware and the results and then see if it is operationally possible to implement this feature.

### 4.5 Schemes

A data centre has varieties of PMs, which consists of different weights. To be able to test the algorithm to see it fit for dynamic environments, also known as scaling, it is only fair that each experiment has different configurations to mirror the vast differences of workload in a centre.

First the weight of each bin in the data centre is calculated and distributed amongst the peers, secondly the total CPU load is calculated and then the average CPU of the centre. This is to ensure that the weight of each bin should never violate the average CPU load, to ensure that a target load is set for each bin to follow, so load balance can be the result. Later the imbalance of the bins (both over and underloaded) are calculated to find the status of the centre.

#### 4.5.0.1 Imbalance

The term *imbalance* means lack of balance or refers to state that is out of proportion. Especially in load balancing where this term is central and a term which defines the state of a data centre. To be able to balance a whole centre, each node needs to be in its own balance. This improves the state thoroughly.

Imbalance will be a central value in this project as well. In this project the imbalance will be defined through a value which indicates the state before and after migration. This is a value which can be used to

compare the two states, to see improvements and eventually *how much* of a improvement.

Each PM will have a max target load which means that when the target load is overlooked, the server is overloaded. The imbalance will define how far off this individual node is from the target load. In terms of balance, as long as the load is underneath the target it is balanced. In total for the data centre, as long as each node stays under its target load it is then in complete balance. Which means there is no imbalance overall. In this project, when referring to imbalance, it means imbalance per node.

#### 4.5.1 Scaling the percentile

There are three particular sizes that the VMs are actually categorized by; small, medium and large. Each VM is allocated with an ID, VCPU, RAM and disk. A data centre does not only have homogeneous VMs. As previously mentioned many studies focus on the concept of scaling in terms of amount where homogeneous nodes are used. It is important to reflect a real centre, with heterogeneous nodes. This is why this project has been designed to present different flavours of VMs.

The chart below draws a picture of the different allocations a VM will supposedly have for the experiments.

##### 4.5.1.1 VM Size index

Size:	Small	Medium	Large
Name:	Small ++	Medium ++	Large ++
VCPU:	2	4	8
RAM:	2048	4096	8192
Disk:	20	40	80

Figure 4.9: VM Chart

Not only can the sizes vary, but instead of manually having to add each VM to each bin in an arbitrary way, a method to scale the *actual amount* of the different sizes of VMs it will be possible to tune the percentage of how many of each category is wanted. If 100% is the maximum for a data centre, then it will be possible to manipulate the data centre to have for instance 20% of small VMs, 20% of medium VMs and 20% of larger VMs. Again, this project aims to provide a scaling solution which fits for any given size or volume.

#### 4.5.2 Stable Marriage Algorithm - Migrate Large & Migrate Small

The two different schemes that will be implemented are *Migrate Large* and *Migrate small*. These two schemes do the almost same thing, but with

different values. As the chart 4.9 implies, there are different allocations. Based on these values a calculation is done, where the value of the CPU is extracted to find out whether the VM in the bin is small or large. Based on this, the program continues to execute the same way for both schemes. The figure below is the allocated value to the designated VMs.

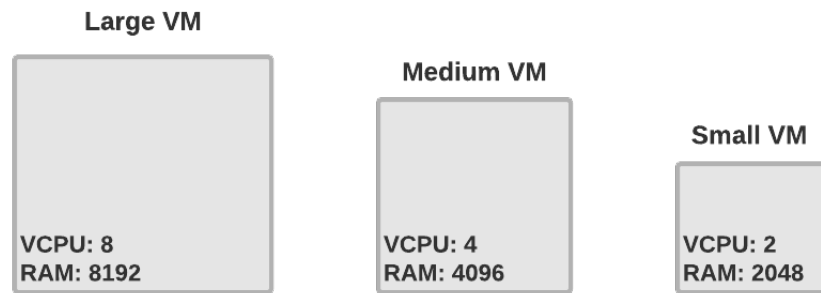


Figure 4.10: VMs with their allocated values

### 4.5.3 Distance Vector

There is a quote that says "*Matches made in heaven*". However it doesn't mean that each perfect match finds its way directly to its better half on the first try. The term *Stable* in Stable Marriage symbolizes the perfect match *out* of all the other possibilities.

In a centre consisting of many thousand VMs it might seem like an impossible task, however since each node is aware of its neighbours details it is easier to start with the one with largest imbalance to find the one with the perfect imbalance for itself.

## Chapter 5

# Result II – Implementation

This chapter will present the implementation of the framework. To provide better understanding of how the framework is put together and how the Stable Marriage algorithm has been implemented, detailed descriptions will be provided throughout the chapter. Each experiment accompanies some analysis to create an understanding of the situation at the given point.

The figure below is a diagram of the work flow, a breakdown of how the implementation chapter is presented.

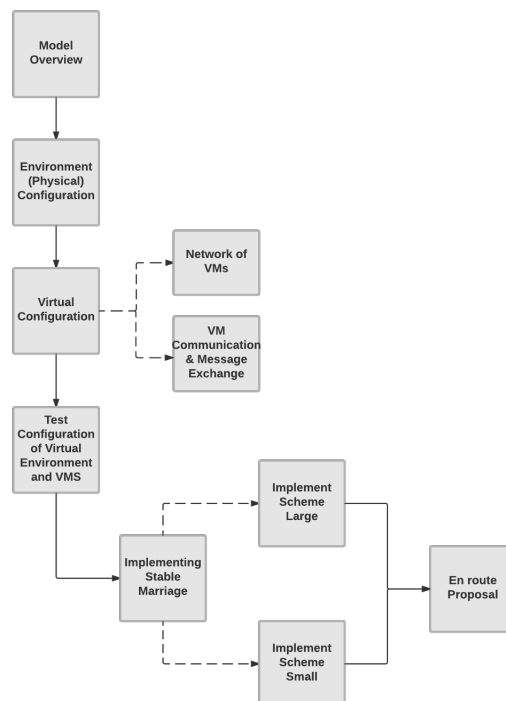


Figure 5.1: Chapter overview

A note to the reader, the code may be presented as pseudocode. For the

actual code, the reader is advised to peek at the Appendix (8).

## 5.1 Model Overview

Figure 5.2 below is a model which gives an overview of the structure in which the project will be implemented. This is a figure which shows how the different components from entirely different worlds are paired together. The bottom layer is the physical hardware consisting of PM1-PM3 or Lab01-Lab03 which are the assigned name on the OS. This layer is controlled by the hypervisor KVM, which is in control of the virtual environment, also the network of VMs which are later spawned in layer 3.

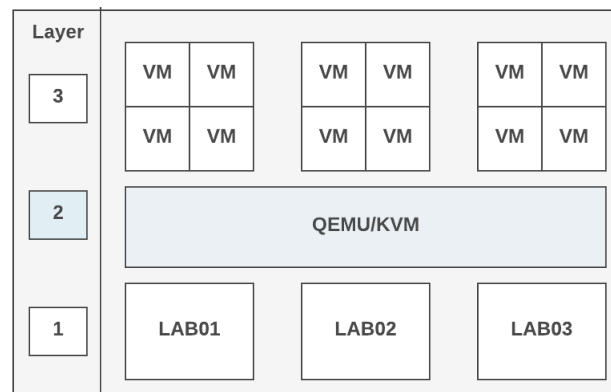


Figure 5.2: Design

## 5.2 Environment Configuration

Evidently, a framework is built with several services and components, which are necessary for an environment to work. To set up a virtual environment for this project several physical and virtual technologies were necessary.

The physical servers in this project are stored in a server room at Oslo & Akershus College University. There are three dedicated servers for this project, as seen in figure 5.4. The setup consists of a dedicated gateway to connect to the outside. All of the PMs are inter-connected through a dedicated switch.

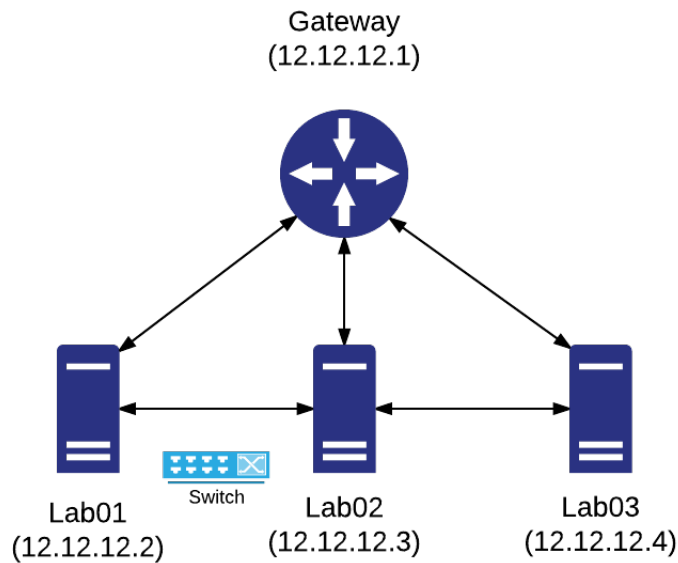


Figure 5.3: Overview of the Physical lab structure

Each server is allocated with same specifications:

Hardware:	Design:
Processor	Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz
Architecture	x86_64
Memory	2048 MB
CPU	2
Operating System	Ubuntu 16.04.2 LTS (Xenial)
Virtual	QEMU/KVM, Libvirt

Figure 5.4: Physical attributes

These are the details for the physical hardware which are dedicated for the virtual implementation. The PMs run Linux, Ubuntu which is easier to work with especially with QEMU and KVM for virtualization of the environment. The installation of these packages are straightforward, however before installing any of these technologies require a check to see if the CPU allows hardware virtualization.

### 5.3 Virtual Configuration

To virtualize the environment it is important to check if the hardware allows this. A simple command as seen below can identify and check if the hardware allows virtualization.

---

```
1 egrep '(vmx/sum)' --color=always /proc/cpuinfo
```

---

If the command responds with a pre-defined answer, then the hardware is able to process and handle virtualization, if nothing happens after typing the command, then the process must be stopped.

The next step is to configure the virtual network. This network will also ensure that when migrating VMs from one host to another, this happens within the same virtualized network. Figure 5.5 below shows how the PMs are connected and how the VMs reside inside the PMs. The VMs are attached to a virtual bridge by birth. This is actually a virtual switch, however it is called a bridge and used with KVM/QEMU hypervisors to be able to use live migration for instance.

To connect the PMs together, a physical switch is connected between the PMs.

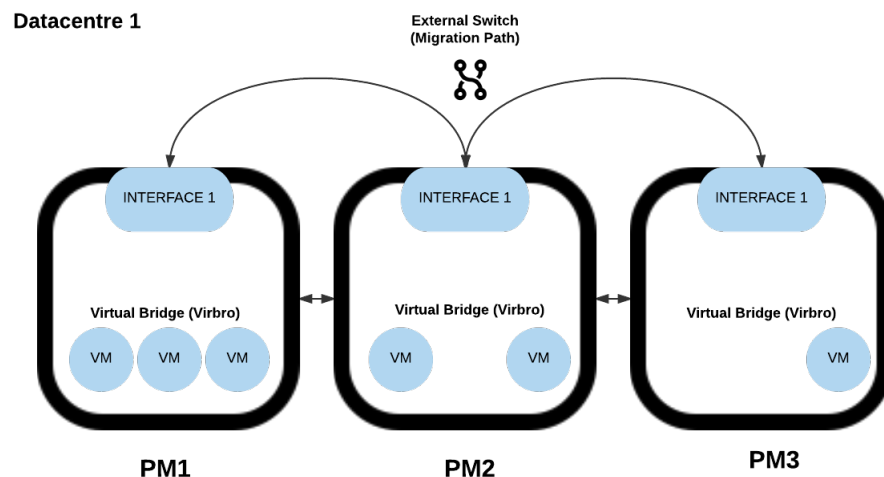


Figure 5.5: Physical Lab details

### 5.3.1 Network of VMs

Creating and deploying several hundred or thousand VMs requires some order. In a giant network braiding several components together can be done by uniquely distributing an ID. Each device connected to a network needs an IP address and a MAC address. This also applies to VMs, hence using ISC DHCP, all three PMs work as DHCP servers.

To provide each VM with an IP and MAC-address a script which can define up to 100 VMs has been implemented. The network range of 192.168.1.0/24 has been reserved for the VMs. This means that the first VM that will be spawned will be addressed to an IP starting from 192.168.1.101,

where the last spawned VM will be given 192.168.1.200. The virtual bridge is configured as:

---

```
1  #Typing the ifconfig commando, lists the virtual bridge in the
2  configuration:
3
4  virbr0    Link encap:Ethernet  HWaddr 52:54:00:f1:46:a3
5            inet addr:192.168.122.1 Bcast:192.168.122.255
6            Mask:255.255.255.0
7            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
8            RX packets:4745 errors:0 dropped:0 overruns:0 frame:0
9            TX packets:4934 errors:0 dropped:0 overruns:0 carrier:0
10           collisions:0 txqueuelen:1000
11           RX bytes:519650 (519.6 KB)  TX bytes:2345192 (2.3 MB)
```

---

The pseudocode underneath demonstrates what the ISC DHCP script does:

---

```
1  def randomMac():
2      <Define a variable mac which uses the random class and
3      to generate MAC addresses>
4      <return the mac adr>
5
6  with open(networkfile, "a")
7      <iterate through the file with for-loop>
8      <Distribute each mac address and fixed IP to
9      a host>
10     <write to file & close>
```

---

This is how the first VM will be assigned an IP and MAC address:

---

```
1  <host mac="00:16:3E:4F:FD:4A" name="vm1" ip="192.168.1.101" />
```

---

The benefit of having local DHCP server on each instance is that once the VM is migrated to host destination, there won't be a need to ask for a new IP address. The new host won't need to assign a new IP, but can ask the previous host to use the existing IP and MAC address.

### 5.3.2 VM Communication & Message Exchange

One of the biggest benefits of having a distributed system is the possibility to implement communication between the nodes. This can spread an awareness of neighbours weight and node information. The idea behind Stable Marriage is to be able to find a best possible match for the node. This leads to a faster calculation time and a effective way of avoiding unnecessary amount of migrations.

Using the TCP/IP protocol a socket for communication between client and servers can be set up. The sockets can be configured to act as server and listen in the background for incoming messages. Sockets can also connect to other applications as a client, and after both ends of a TCP/IP socket are connected, the communication goes both direction.

To implement a socket, one first needs to setup a socket, assign it to a server address, for instance localhost and a port number. Further one can setup a listen and accept method, which waits for incoming connection. Contrary to the server socket setup, the client setup uses a connect method to attach the socket directly to the remote address. After the connection is established, data can be sent through the socket.

Beneath is a shot pseudocode snippet of one way of communication between server & client on sending a message:

---

```
1  #Socket for sending data
2  try:
3      #<Create a message variable for sending messages>
4      message = 'Add some message here or details from a PM'
5      <Print and finish>
6
7      #Search for response
8      <Variables to store the response message>
9
10     #Start a while loop to check if the message received is
11     the same as sent by check the length.
12     <Compare amount received vs. expected>
13
14     <Print and finish>
15
16  final:
17     #<Close Socket>
18     sock.close()
19
20  ##Pack he message to a specific server:
```

```

21
22 while True:
23     sock.sendall(str.encode(message), server_address)

```

---

### 5.3.3 Deployment of a VM

Before implementing the algorithm, a small test is performed to see if VMs are able to spawn and run with libvirt. This is to test if it is possible to deploy and assign VMs with the flavours and configurations that are required by the algorithm.

The code snippet beneath shows how a VM is manually spawned with libvirt, as well as which network it gets assigned to along with IP and MAC address, the choice of flavour is completely arbitrary and does not have any effect. A test migration will also be performed to see if the connection between the PMs are established:

---

```

1  #Spawning a VM, this is a one-liner without any
2  breaks to spawn a VM:
3
4  virt-install --virt-type "kvm" --name "vm1" --cdrom "my.iso"
5  --network "bridge=virbr0,mac=52:54:00:7A:28:7E"
6  --nodisk --vnc --noautoconsole --memory 2048 --vcpu 2
7
8
9  #Output looks like this:
10 Starting install...
11 Creating domain...
12
13 #virsh list --all:
14   Id      Name                                     State
15   -----
16  1        vm1                                     running

```

---

To test if migration with libvirt is working as well as to check the connection establishment between the PMs, a small migration from PM1 (Lab01) to PM2 (Lab02) was performed, underneath is a snippet of the migration process.

---

```

1  #Migrate VM1 previously spawned from Host to new Host:
2  disha@lab01:~$ virsh migrate --live vm1 qemu+ssh://12.12.12.3/system

```

```

3
4  #On PM2 the "watch" command is running, which allows monitoring of the
5  process:
6  disha@lab02:~$ watch -n 0.5 'virsh list --all'
7
8  Id      Name                               State
9  -----
10  1       vm1                               paused
11
12
13  (After 0.5 sec)
14  #disha@lab02:~$ virsh list --all
15
16  Id      Name                               State
17  -----
18  1       vm1                               running

```

One of the benefits of using TinyCore VMs is that it provides the possibility to be able to sign into each individual VM where the user is free to use the VM as whatever preferred reason.

Each VM that is spawned under the virbr0 network, also gets its designated virtual interface. This is created and assigned while the VM is operating, and migrates along during a migration. This interface is also destroyed once the VM is destroyed.

```

1  #ifconfig
2
3  vnet1  Link encap:Ethernet  HWaddr fe:54:00:7a:28:8b
4         inet6 addr: fe80::fc54:ff:fe7a:288b/64 Scope:Link
5         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
6         RX packets:543 errors:0 dropped:0 overruns:0 frame:0
7         TX packets:137076 errors:0 dropped:0 overruns:0 carrier:0
8         collisions:0 txqueuelen:1000
9         RX bytes:68724 (68.7 KB)  TX bytes:7312991 (7.3 MB)

```

## 5.4 Stable Marriage

The physical and virtual environment is tested and all setup. The next step in the process of completing a framework is developing and implementing the main algorithm, the Stable Marriage algorithm.

Stable Marriage is, as mentioned previously, an approach to find the perfect match for gaining load balance. Each node is considered an individual with preferences and demands. These are taken into consideration, to be able to find the perfect balance for each individual node.

The flow diagram below gives an insight on how the Stable Marriage Algorithm is made and which methods are implemented to make the algorithm work:

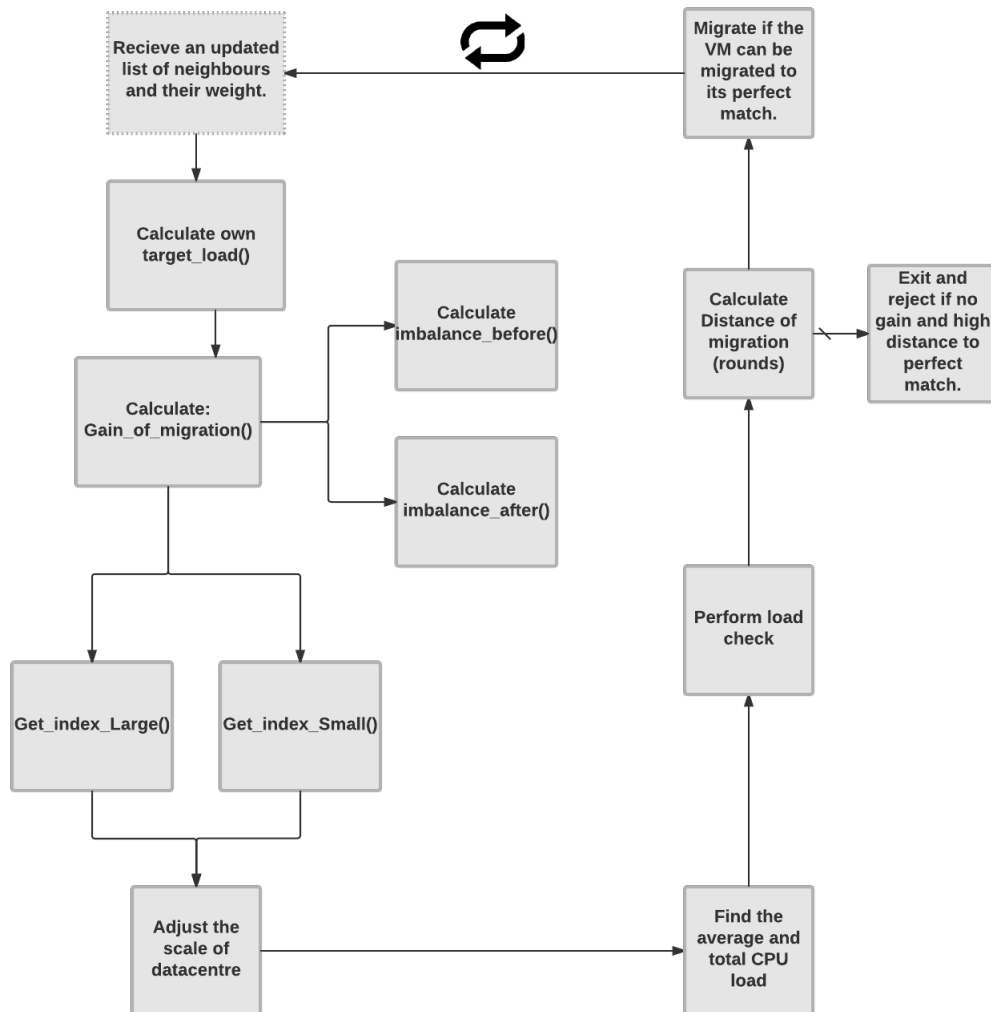


Figure 5.6: Flow Diagram of the Stable Marriage Implementation

Note, the code is available for the reader in the appendix section 8, this section provides a rough translation of the most important methods of the implementation.

The first process is to define the flavours of the VMs that will be used, this example a snippet of how the small VM is defined is presented, medium and large are not presented below, but are implemented the same way:

---

```
1 VM_list = []
2 small_vm = {
3     'name': 'small%d ',
4     'cpu': 2,
5     'mem': 2048,
6     'disk': 20,
7     'n': 1
8 }
```

---

The next step in this process is to find and define the weight or load of the server:

---

```
1 def Get_Bin_Load(bin):
2     load=0
3     <for-loop to iterate and count VMs inside bin>
4     <define load>
5     <load is defined as CPU index of VM>
6     return load
```

---

After the load of the bin is defined, it is necessary for some calculations to take place. One of the most important calculations of this algorithm is the imbalance and the gain of a migration, as well as the CPU load. In the pseudocode below, the imbalance and CPU calculation is presented.

---

```
1 total_cpu_load=0
2
3 for i in range(N):
4     total_cpu_load=total_cpu_load+VM_list[i]['cpu']
5
6
7 average_cpu_load=total_cpu_load*1.0/K
```

---

## Imbalance and gain of migration:

---

```
1 def Gain_of_Migration_Couple(Over_Bin, Under_Bin, Big_VM, average_cpu_load):
2     #Calculate imbalance BEFORE an eventual proposal/migration
3     <get load of overloaded server>
4     <subtract the weight from the average CPU load to find
5     the imbalance before of overbalanced PM>
6
7
8     <Get load of underloaded>
9     <subtract the weight from the average CPU load to find the
10    imbalance before of underbalanced PM>
11    <print the Imbalance before of underutilized server>
12
13    #Calculate the total imbalance BEFORE a marriage:
14    <Total imbalance is = imbalance of overloaded bin +
15    imbalance of underloaded bin>
16
17
18    ###AFTER###
19
20    #Calculate imbalance AFTER an eventual proposal/migration
21    <Overloaded>
22    <Get load of overbalanced server, subtract the largest VMs CPU
23    and again subtract the average CPU load>
24    <print the Imbalance after of overutilized server>
25
26
27    <Underloaded>
28    <Get load of underbalanced server, add the largest VMs CPU and
29    subtract the average CPU load>
30    <print the Imbalance after of underutilized server>
31
32    #Calculate the total imbalance AFTER a marriage:
33    <Total imbalance is = imbalance of overloaded bin -
34    imbalance of underloaded bin>
35
36    #Calculate the gain
37    <gain = total imbalance before - the total imbalance after>
```

38 <Positive result shows a gain, went from worse to better>

---

This method calculates the imbalance of each bin by using the load of overbalanced PMs and subtracting that from the average CPU load. Then the underloaded bins weight is subtracted from the average cpu to find the before imbalance for the underbalanced server. To find the total imbalance, that is the total then added of the number from the imbalance of the overbalanced node + the under balanced node.

The process to find the imbalance after is Imbalance underbalanced = get the load of underbalanced PM and add the CPU load of the largest VM, subtract the average cpu load. The biggest VM is chosen because this is the one VM which creates the largest imbalance in any situation.

The benefit of a migration is to so if there is an improvement of the imbalance, if it is a beneficial marriage. If the result show a positive value, this marriage is going to be stable.

Stable Marriage allows the framework to restrict that if there is no more "gain"/nothing more to balance, the job is done. The algorithm will then exit. It also restricts overloaded servers to become underloaded, which means that PMs may also decline a proposal if the overloaded server becomes underloaded. This would create an unnecessary loop and the whole purpose of load balancing would be pointless if it was not considered. Hence, it was highly important to implement a solution, which considers the state of the machines at all times.

---

```
1  #Define gain/no-gain as a variable
2
3  while (nogain==False):
4      <for-loop to iterate through the under/over bin>
5          <print iteration of the gain calculated by using the info of
6          under/overbin, average cpu load and amount of servers.>
7          <defines under vs. overloaded bin>
8          <if the largest VM in the bin crosses the average cpu>
9              <set nogain = True>
10         <exit if>
11     <exit for>
```

---

In the "no\_more\_gain" method if there is no beneficial proposal that reduces the imbalance or the proposals will increase the imbalance, the algorithm will stop whenever there are no possibilities to reduce further imbalance.

This means that a node can never become overbalanced again or underbalanced to take more VMs on board. This is one important

implementation, as the point of the Stable Marriage algorithm is to *Stabilize* the system, this algorithm contributes to the stability factor.

## 5.5 Schemes

Next, the main methods which distinctly creates the possibility to scale are the two schemes introduced earlier. The implementation of how the algorithm finds the largest vs. smallest index to differentiate between the sizes of the VMs on a PM.

Stable Marriage is not a complex algorithm, which is exactly what is needed in a another complex day for a system administrator. The point is to keep the system in balance, and make it possible to have a dynamic system which is fault tolerant. The schemes that are implemented are basically a *max* or *min* at code level. This is one of the benefits of an abstract idea, which is represented below.

### 5.5.1 Migrate Large First

Here the simple method of finding the largest VM, that loops through the list of VMs to find the index of the largest VM. This is done by first going through each bin and VMs inside it, if the index cpu of the bin is larger than max, then the index of the largest VM is returned.

---

```
1 def Get_Index_Largest(bin):
2     max=0
3     <set index to max>
4     <for-loop to iterate through VMs inside the server>
5         <if the VMs index cpu is larger than max>
6             <set the max to be the largest VM>
7         <index is now set as max>
8     <exit loop and return max index>
```

---

The only difference between the two implementation of the schemes in terms of code, is that the index changes from searching for the largest VM to the smallest based on the CPU.

### 5.5.2 Migrate Small First

Here the simple method of finding the smallest VM, that loops through the list of VMs to find the index of the smallest VM. This is done by first going through each bin and VMs inside it, if the index cpu of the bin is smaller than min, then the index of the smallest VM is returned.

```
1 def <Get the smallest bin>:  
2   min=10000000000  
3   <set index to min>  
4   <for-loop to iterate through VMs inside the server>  
5     <if the VMs index cpu is less than max>  
6       <set the min to be the smallest VM>  
7     <index is now set as min>  
8   <exit loop and return min index>
```

---

## 5.6 Summary

The aim of the project is to see if it is possible to load balance with stable marriage. This implementation chapter provided the necessary information to technically be able to set up the framework, before performing experiments. A small test to check if the environment was done, as well as testing the spawning of a VM and migrating it.

The next chapter will provide the results of the experiment, which is a stepping stone for the Stable Marriage algorithm for any future work on this.

## Chapter 6

# Result III – Experiment & Analysis

This chapter presents the results extracted from the experiments. Each experiment is unique in its own way and is presented with graphs accordingly. The methods which were implemented in the previous chapter will be put to test.

### 6.1 Testing

Testing is one of the most crucial parts. There are many things that can halt a progress such as technical difficulties, errors, and wrong output. Often these questions could be easily resolved if each section was tested before shipping the product. The testing method for this project can fall under the *Unit Testing* method. Each method has been tested as whole during the progress.

### 6.2 Simulation Experiments

After testing the framework in different ways as presented in the beginning of this chapter, the framework is now ready run tests. Most of the experiments generated *very* positive results. Each experiment with different configuration was tested at least 10 times to see if the average of each test had a positive outcome, in total 12 tests were run with different configurations for the simulation. The more positive data generated, the more confidence there is to the fact that the algorithm has a positive effect and works the way it is supposed to. However, it is highly important to take *error margin* into account while generating data.

To strengthen the positive results a confidence interval was made for the total average taken from the average imbalance before and after a migration from all the 12 tests. A value of 95% for confidence interval was chosen, this means that around 5% of the interval is reserved for error margins.

Figure 6.1 and 6.2 below shows the average confidence level of the mean from all of the 12 simulated samples conducted. Meaning the samples are 95% reliable, *however* scientifically speaking in statistics there are some parameters that determine whether the results displayed are of significance or not, this is determined by a *p-value*. In statistical hypothesis testing the p-value is the probability for a given model, that when the *null hypothesis* - a general statement that there is no relationship between two measured phenomena or association among groups - is true the summary, such as the mean, would be the same or more than the actual results [42]. The p-value is of significance if the value is less than 0.5, in this case the p-value resulted in  $p > 0.5 = 0,629$  for figure 6.1 and  $p = 0,655$  for the 6.2. Just above the significance value.

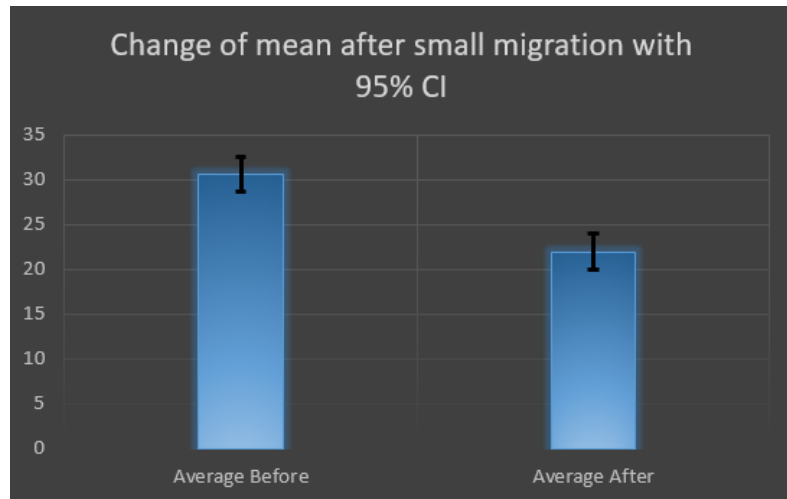


Figure 6.1: Mean of CI for Small Migration scheme

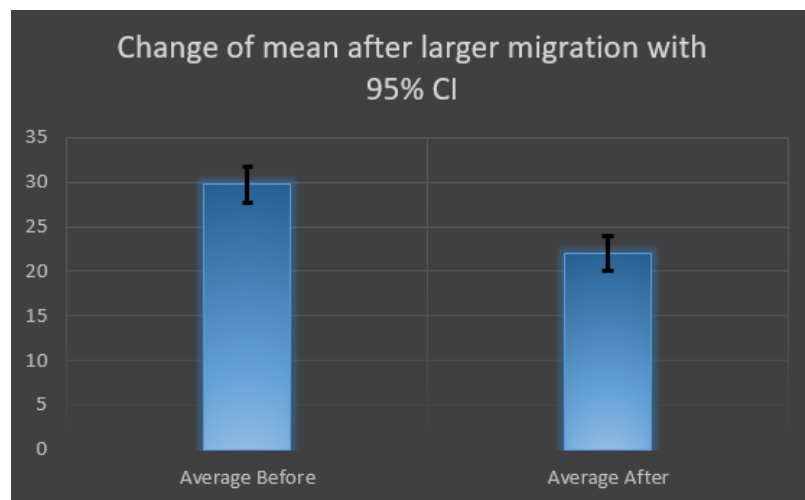


Figure 6.2: Mean of CI for Large Migration scheme

Figure 6.3 shows the number of Experiments from E1-E6. For each

experiment conducted, a box plot graph is provided. A Boxplot graph is the way to see the spread of all the different data points collected. This also provides additional information instead of just providing the average numbers of the experiments. The “whiskers” shows the spread of the data range from the lowest data point to the highest data point.

Experiment	Number of PMs	Number of VMs	Average CPU
E1	10	60	N/A
E2	20	125	N/A
E3	5	5000	3600
E4	100	10.000	720
E5	3	2500	2500
E6	15	20.000	6400

Figure 6.3: Table of Experiment details

## 6.3 Simulation

The first part of the experiments were conducted in a simulated environment. This helped find flaws and bugs which could prevent further problems with the real implementation. There are 6 experiments, and each experiment is presented with an analysis to highlight what the results actually show.

Each graph is the average result taken from each experiment. For example, if experiment 1 had five migrations, the average imbalance before and after is displayed on the graph. This is to show the average improvement.

### 6.3.1 Summary

If the reader would like to skip reading the graphs, the next sections are all a visual display of the results extracted from the experiments. Each graph shows the average imbalance before and after a migration. Each experiment was a new experience in terms of knowledge. Many improvements were made during the progress to find the perfect solution for the algorithm.

The reader is advised to take a look at especially the results from 6.3.7 and 6.3.9. These show a great improvement, where least possible physical resources have been used.

To make it easier for the reader to follow the experiments, the results of the experiments are graphically presented with Boxplot. Each graph is also described in the analysis.

### 6.3.2 Small scale migration - I

The first experiment was a test as well as an experiment. The aim of this test was to see if migration was possible and if it was working. The aim of this experiment was to migrate 60 small vs. 60 large VMs to balance a data centre consisting of 10 PMs.

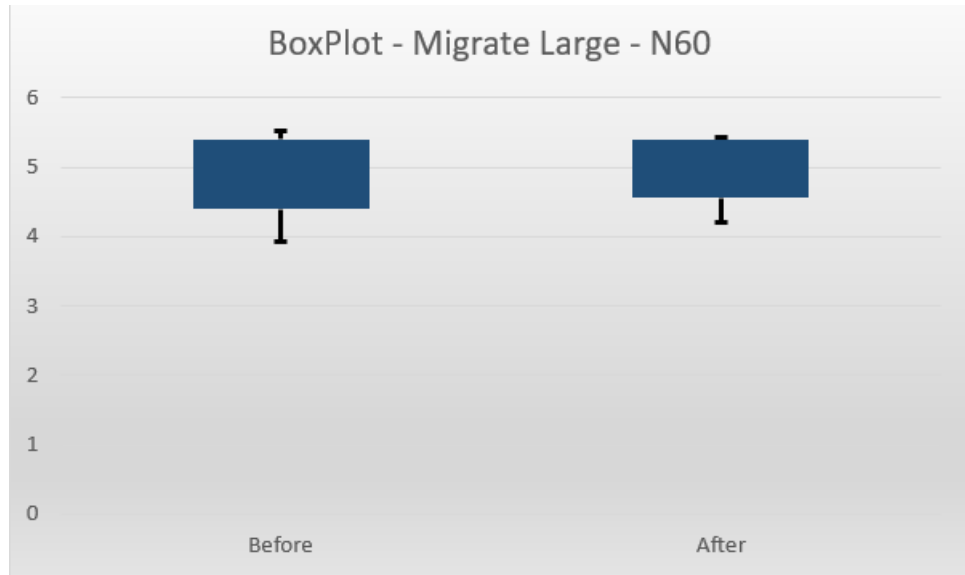


Figure 6.4: Migration of 60 Large VMs

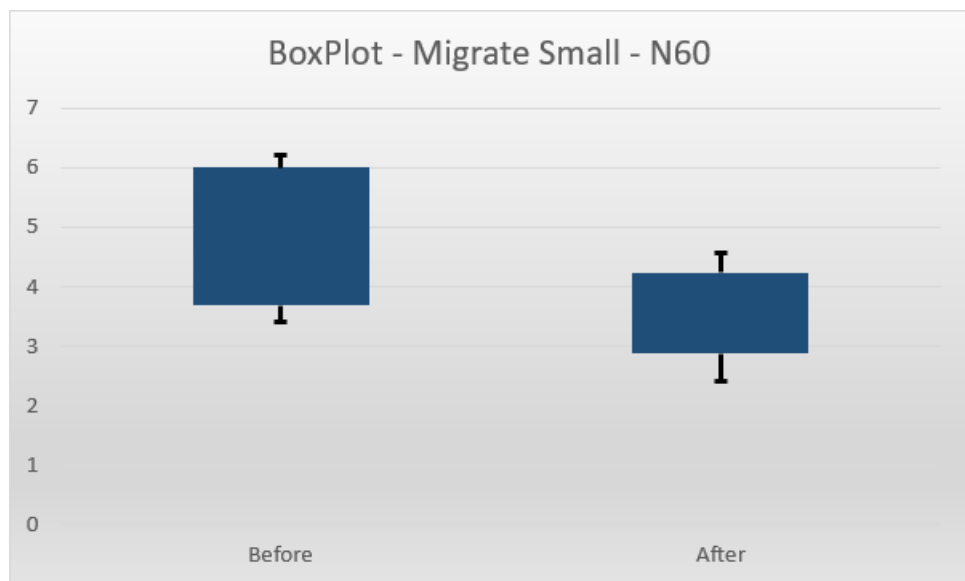


Figure 6.5: Migration of 60 Small VMs

### 6.3.3 Analysis

There were high hopes for the first experiment to turn out positive, the smaller scheme seemed to do much better in terms of improving the imbalance of the center compared to the larger one.

The graphs display the average imbalance for each bin before a migration and after a migration. The aim is to lower the average imbalance of each bin, which leads to a better environment for the VMs. Low CPU and memory utilization in other words.

The graphs presented above display the Stable Marriage algorithm in action with different schemes. The first noticeable improvement is of the Migrate small scheme, the average imbalance improved alot. The graphs show the average imbalance improvement before and after the migration.

In this test the algorithm extracted 5 migrations, each of the migrations provided a value before and after the migration which was the imbalance ratio. The graph above displays the average taken from the 5 migrations. Figure 6.6 below provides an example of how the imbalance was extracted and plotted, resulting in the graph from the average values.

<b>Small</b>					<b>Large</b>				
Imbalance	Before	After			Imbalance	Before	After		
Test 1	6,2	4,56			Test 1	5,4	4,2		
Test 2	4,6	4,24			Test 2	5,52	4,88		
Test 3	3,4	2,88			Test 3	3,92	5,4		
Test 4	6	3,2			Test 4	4,4	5,44		
Test 5	3,68	2,4			Test 5	4,64	4,56		
<b>Imbalance avg. B Imbalance avg. After</b>					<b>Imbalance avg. Imbalance avg. After</b>				
4,776 3,456					4,776 4,896				

Figure 6.6: Imbalance example

### 6.3.4 Small scale migration - II

The aim for the second simulation was to see how scaling up with a few more VMs and PMs would go. In this experiment 125 large and small VMs were migrated inside a 20 PM data centre.

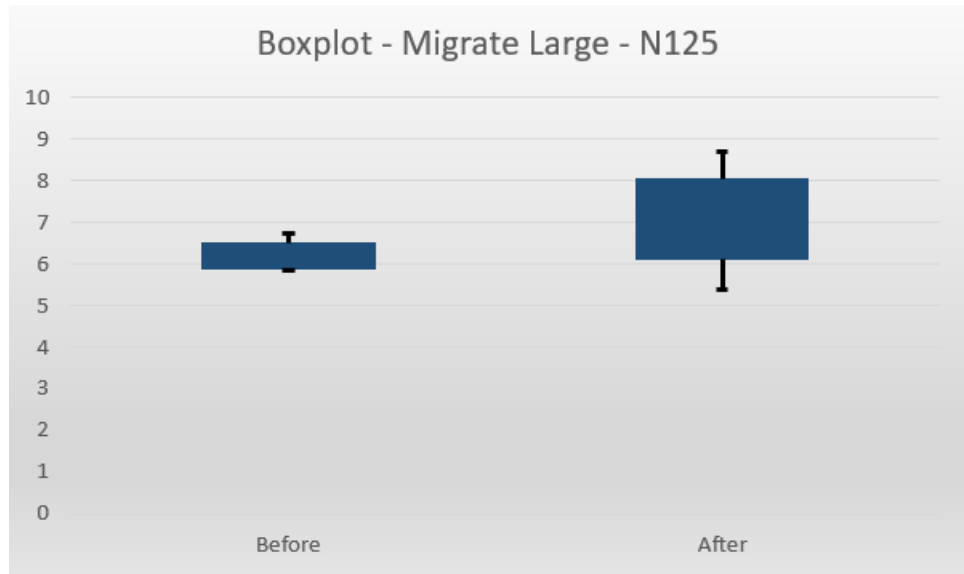


Figure 6.7: Migration of 125 Large VMs

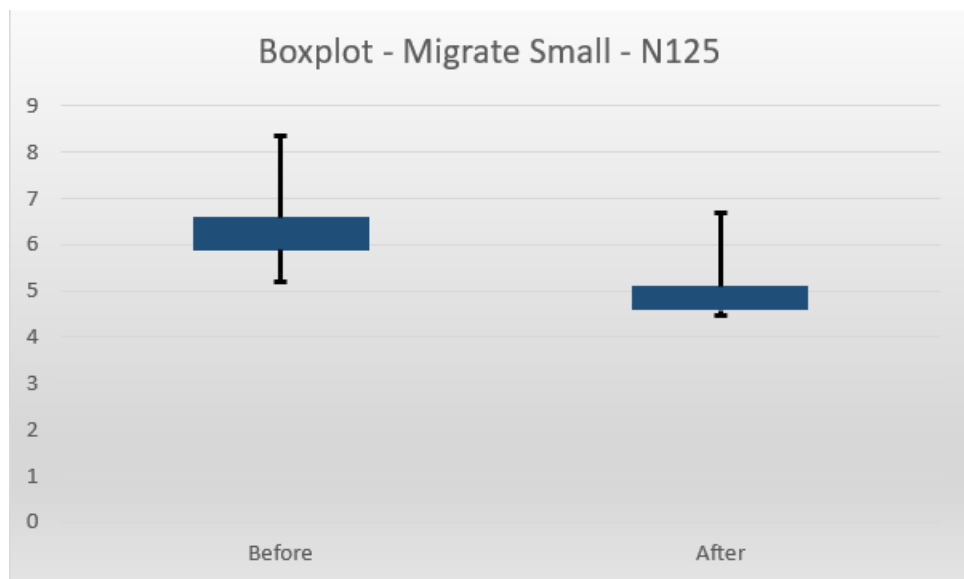


Figure 6.8: Migration of 125 Small VMs

### 6.3.5 Analysis

At this point the CPU average was not monitored for data collection, however the algorithm won't allow migration if it isn't a perfect match, and the CPU and memory are a part of that requirement. This is the one and only experiment which did not improve the imbalance factor. By looking at the *IQR -inter quartile range* IQR Before: 0,64, IQR After: 1,94 it is clear that there has been an increase in the imbalance of the system. The positive thing is, that such a decision will not be able to happen anymore. This was discovered fast enough to prevent future experiments to get the same results.

### 6.3.6 Workaround

After the first two experiment, issues with how the calculation was done started to show. The minute the number of VMs on a PM crossed above 100 the algorithm was not efficient enough for migration of the heavier VMs. The CPU average was crossed and the average imbalance was not improving. This was quickly corrected, as there was some spelling error in the script which showed the incorrect imbalance.

This experiment was included because this bug led to another vital policy included in the schemes. Earlier in the Design section 4.5.1, scaling with percentage was introduced. This is a method which would allow the user to produce different percentage of heterogeneous nodes. The average CPU load was also being monitored.

Each experiment from this test onwards has been modified with different weight. If a complete data centre is 100% then the number of VMs and their weight can be manually manipulated.

### 6.3.7 Medium Scale migration - I

For this experiment scaling was implemented. 2500 VMs were migrated on only 3 PMs. 2500 pose a scale of 100 %. This means that the amount of flavours on the VMs can be modified.

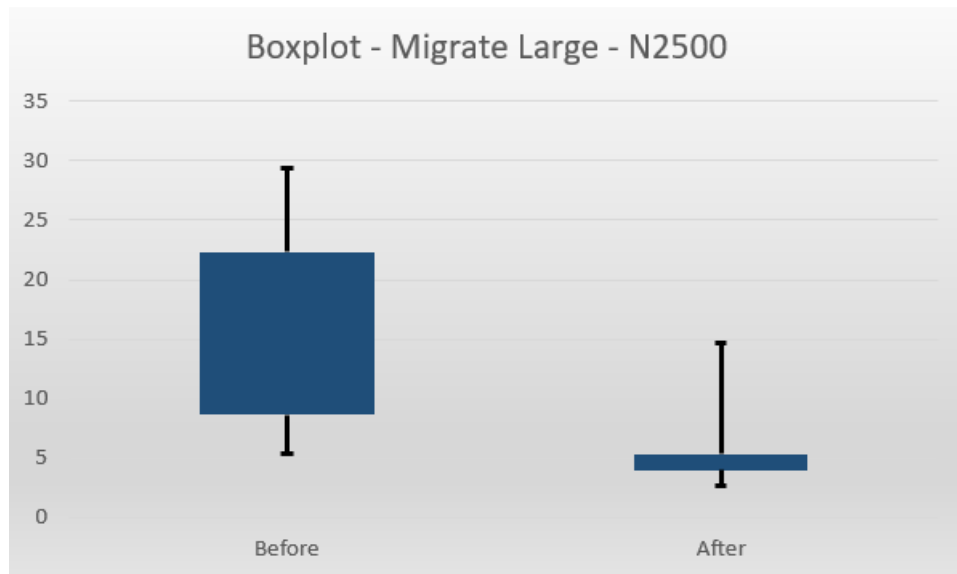


Figure 6.9: Migration of 2500 Large VMs

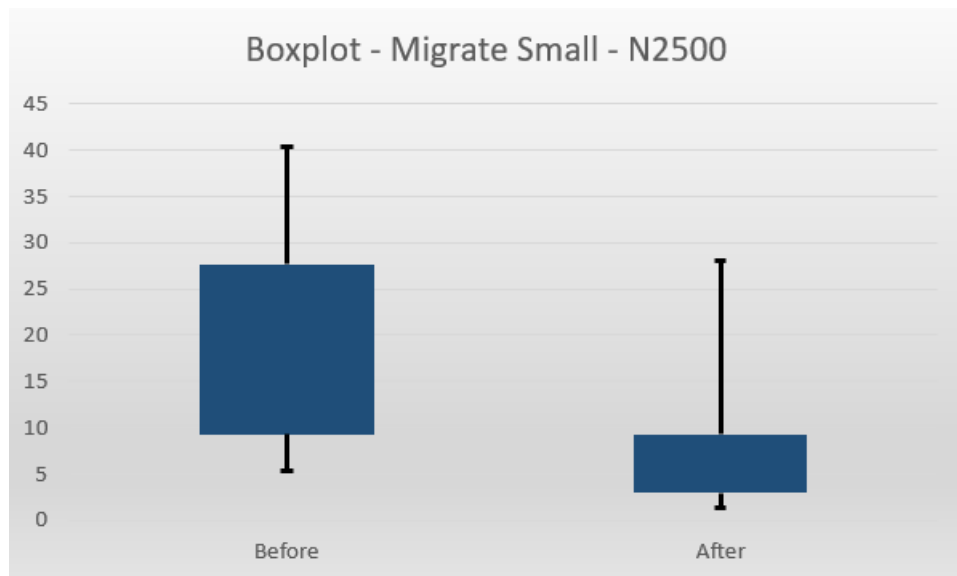


Figure 6.10: Migration of 2500 Small VMs

### 6.3.8 Analysis

This particular experiment consisted of the following sizes and flavours for each scheme:

Small T1			Large T1		
Small = 50%	N=2500		Small = 50%	N=2500	
Medium = 50%	K=3		Medium = 50%	K=3	
Large = 0%	CPU L = 2500		Large = 0%	CPU L = 2500	

Figure 6.11: Flavours of VMs

One of the goals of this experiment was to check if migration would be possible with no large flavoured VM and if the migration went to the next largest flavour - medium. In this case the medium flavour becomes the largest and is considered as a large for the algorithm.

Results still show positive gain and the boxplot graph show a large improvement. The wide boxes symbolise a wide range of number, this means that the distance from the top whisker to bottom whisker all represent the improvement the migration does on the after box. Another important reason why the boxes appear larger than what has been presented is because the amount of average test for each experiment was changed from 5 to 10 to find a larger variety. Hence, the graphs from this experiment onwards are larger, with a larger variety of results.

### 6.3.9 Medium Scale Migration - II

To turn the scale up a bit a simulation of 5000 VMs migrating within 5 PMs was tested. In this particular test the flavours were adjusted to create a data centre consisting of 60% small VMs.

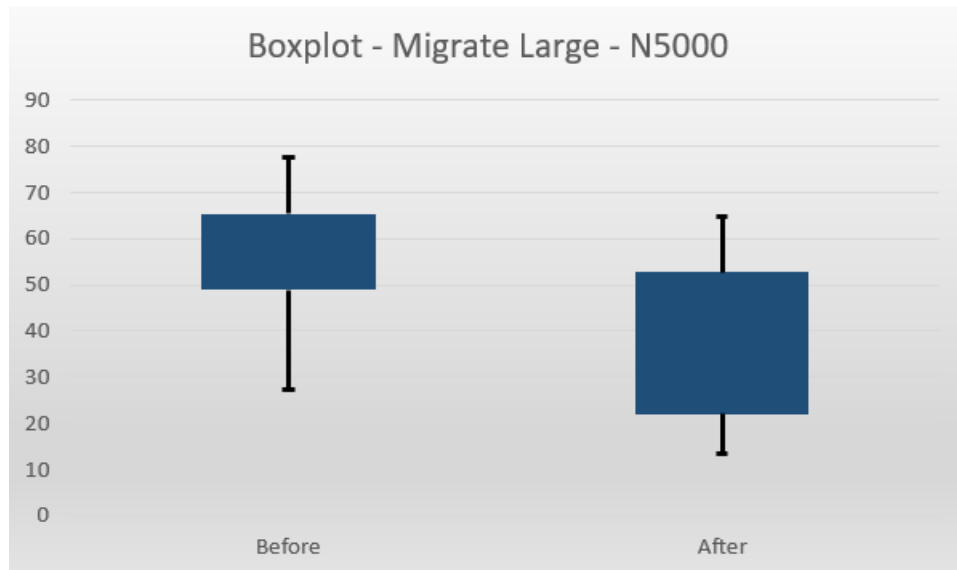


Figure 6.12: Migration of 5000 Large VMs

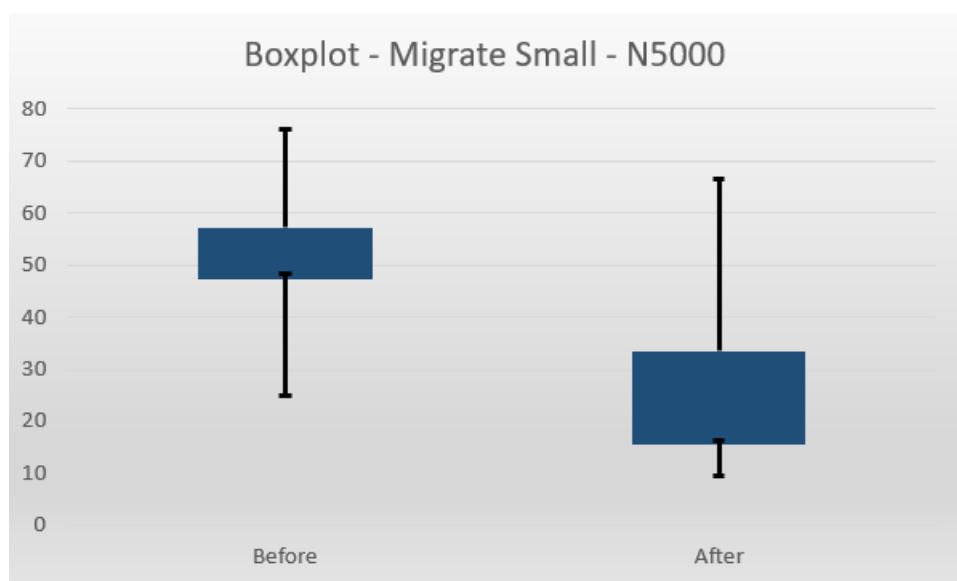


Figure 6.13: Migration of 5000 Small VMs

### 6.3.10 Analysis

After finding a stable point the aim of the experiments was to scale and create simulations with as many different flavours as possible. This was to monitor if the algorithm worked in any given case. The flavours of this experiment:

Small T2		Large T2	
Small = 60%	N=5000	Small = 60%	N=5000
Medium = 20%	K=5	Medium = 20%	K=5
Large = 20%	CPU L = 3600	Large = 20%	CPU L = 3600

Figure 6.14: Flavours of VMs

The box graph for the larger scheme is much wider and fuller than the one before. This is simply because the average number of test results were somewhere between 50-70, while the average numbers after the migration had a varying number from 20-55. This means that larger values before migration improved to much better and lower results in the after box. For instance, imbalance for test 3 was 75,2 for the migrate small box, and the imbalance after was 27,2.

### 6.3.11 Large Scale Migration - I

Moving on with the flavour testing, the next in cue is the test which experiment with the larger flavour. 8 VCPUs each for each and every large node as 10.000 VMs migrate between 100 servers.

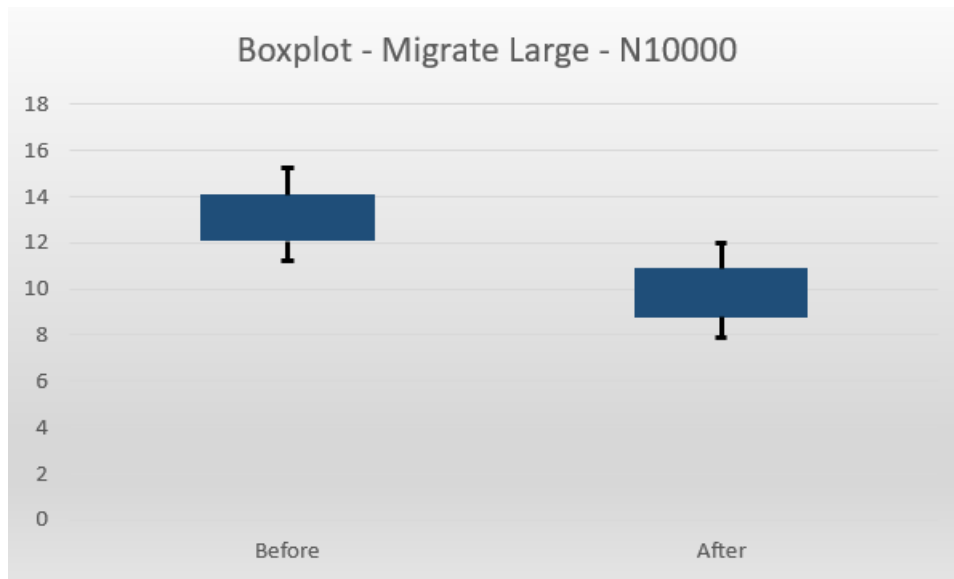


Figure 6.15: Migration of 10.000 Large VMs

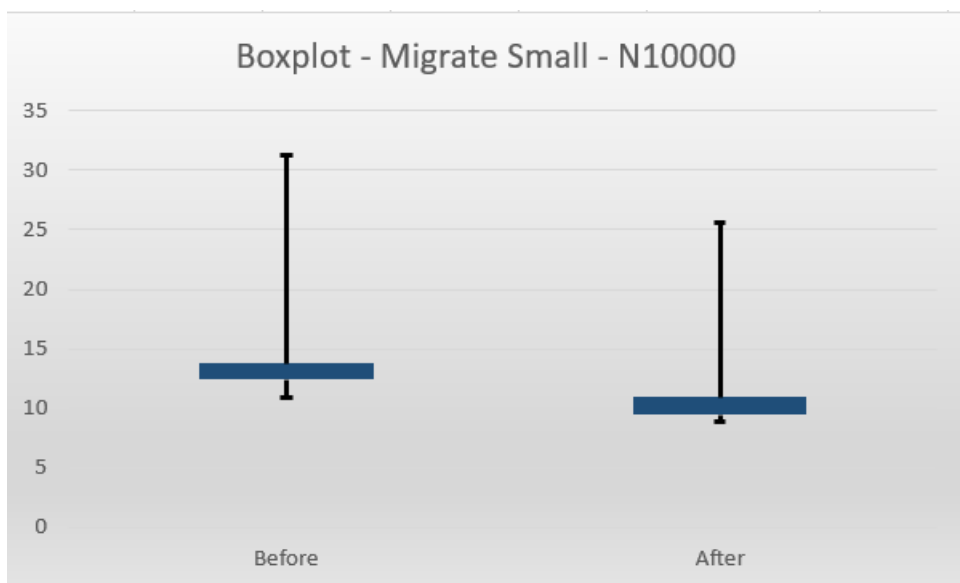


Figure 6.16: Migration of 10.000 Small VMs

### 6.3.12 Analysis

One of the most important and interesting thing about this particular experiment is the size of the boxes. The fact that the after-boxes are in the lower ranges of the imbalance ratio is positive, however, the sizes of the boxes point at something important. This shows that after a certain amount of VMs, there will be a difference in the imbalance, BUT the scale will not be far behind the before-box. For instance, take the scheme for migrating 10.000 large VMs. The imbalance ratio for one of the tests before showed a ratio of 10,88 and the after ratio showed 8,88.

It is no secret that the larger the number of VMs will be the longer and difficult it will be to migrate. However, testing this scheme, these were the flavours implemented:

Small T3		Large T3	
Small = 80%	N=10000	Small = 80%	N=10000
Medium = 0%	K=100	Medium = 0%	K=100
Large = 20%	CPU L = 720	Large = 20%	CPU L = 720

Figure 6.17: Flavours of VMs

The CPU load was also very low on this test compared to the others, this is important to address, because in this particular test the amount of PMs were quite kind, on the other tests the VMs have been tested with fewer PMs to see how much they affect the algorithm in performing. In this test, the aim was to create as heavy nodes as possible and see the migration results.

### 6.3.13 Large Scale Migration - II

One of the final tests consisted of 20.000 VMs with the least possible amount of server, 15. This was to test how the CPU load would rise compared to the previous test with a kind amount of servers.

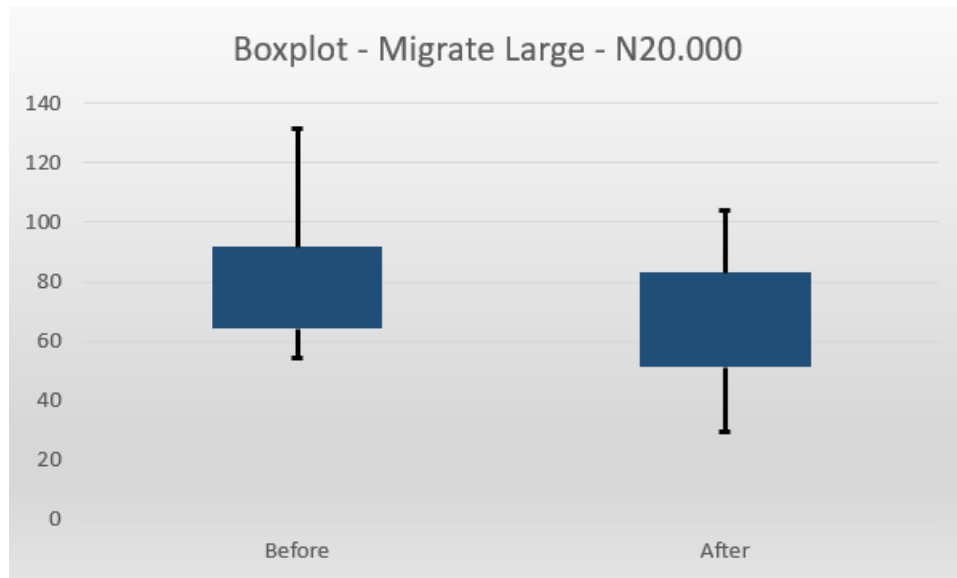


Figure 6.18: Migration of 20.000 Large VMs

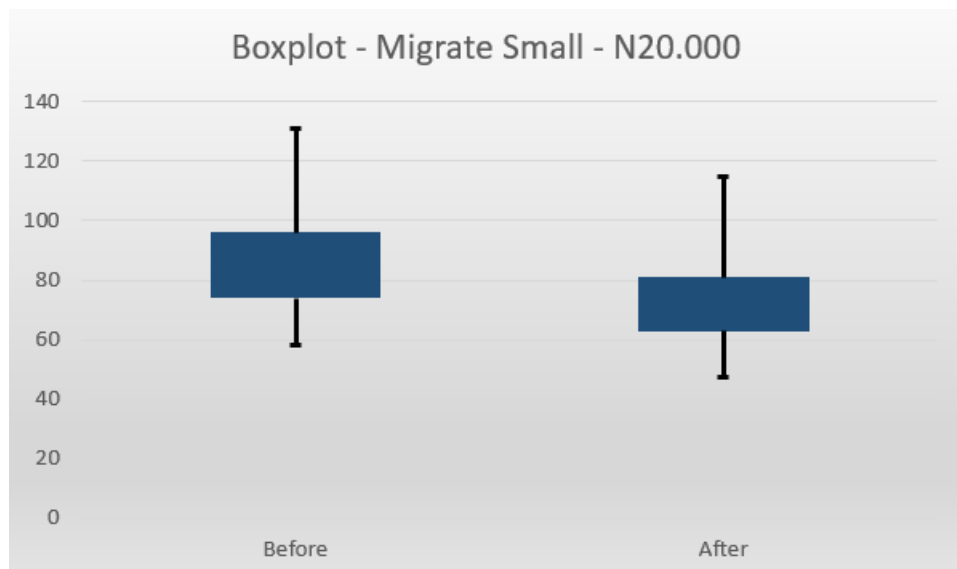


Figure 6.19: Migration of 20.000 Small VMs

### 6.3.14 Analysis

To the best surprise, the algorithm was still able to migrate and balance the data centre. Unsurprisingly on the other hand the CPU load was intense. The graphs also show the improvement of the imbalance before and after migration. Very similar to the last test, even though this experiment was tested with fewer servers. The flavours used for this experiment:

Small T4		Large T4	
Small = 40%	N=20000	Small = 40%	N=20000
Medium = 20%	K=15	Medium = 20%	K=15
Large = 40%	CPU L = 6400	Large = 40%	CPU L = 6400

Figure 6.20: Flavours of VMs

## 6.4 Real Experiment

Unlike the simulation, the real experiment required a lot of failure and trying. The results from the real experiment will not be presented in boxplot graph, but in column chart. Problems with configuring the third PM resulted in only two PMs to experiment on. In some cases this led to manually spawning the VMs to get results.

The same way each experiment was presented for each scheme, so will the experiments from this experiment. In the results below, some snippets as well as graphs will display the process of migration from one host to the other.

### 6.4.1 Migrating Small Scale

The aim of this experiment is to see how migration can take place in a real virtualized environment with different flavours of VMs. The figure below displays the results extracted from the experiment:

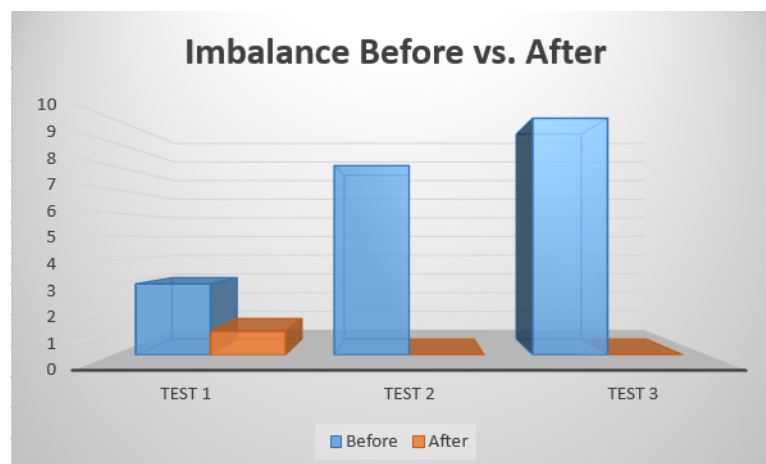


Figure 6.21: Small Imbalance Before vs. After

The output after the migration show the VMs up and running at PM2, with following command:

```
1 root@lab02:virsh list --all
2
3 Id      Name      State
4 -----
5 4       vm3       running
6 5       vm4       running
7 6       vm5       running
8 7       vm10      running
9
10
```

```

11
12 This output for Test1 is generated by the script to inform the user of the \\
13 size of the VMs migrating:
14
15 moving small {'mem': 2048, 'disk': 20, 'name': 'small\\%d', 'cpu': 2}
16 From: 1
17 To: 2
18 moving small {'mem': 2048, 'disk': 20, 'name': 'small\\%d', 'cpu': 2}
19 From: 1
20 To: 2

```

---

Please note that the VM names are not in any particular order, but arbitrary due to an earlier test with IP assignment.

### 6.4.2 Analysis

Figure 6.21 displays the results extracted from 3 rounds of migration between host PM 1 and PM2. Test1 had an imbalance of 3.0 before migration and the imbalance after went down to 1.0. The test had one overloaded server with five small flavoured VMs (6.3). The destination host only had two VMs, and space for more. PM1 sent over 2 VMs to PM2.

Test 2 consisted of seven VMs in total, where only two small VMs resided with five medium VMs. The destination host consisted of four VMs, but all small flavoured. This resulted in 3 migrations in total from PM1 til PM2. In this particular migration, two of the VMs were small flavoured and one was medium flavoured.

Right after the migration, the small VMs were up and running quite quickly, using around 0.4 of CPU, while the medium VM spent a quarter of a second more than the small VMs. The imbalance before migration was 8.0, and after it went down flat 0.0. Which is a great result, concluding the bin in complete balance.

The aim for Test 3 was to see if now PM2 was overloaded, does it locate PM1 and continue migration, as well as how many VMs it would migrate if it only was small flavoured. With an imbalance of 10.0 before and 0.0 after migration.

This test consisted of sixteen small VMs on PM2, while on PM1 there were two small and one large VM. The acceptance of the proposal ended with four migrations. Now, one would think that this results in PM2 still being overloaded with 12 VMs, however, PM1 has one large VM, which evens the imbalance out for both bins. The average CPU load was only 22.

### 6.4.3 Migrating Large Scale

Figure 6.22 displays the results extracted from 3 rounds of migration between host PM1 and PM2, based on the large flavour scheme.

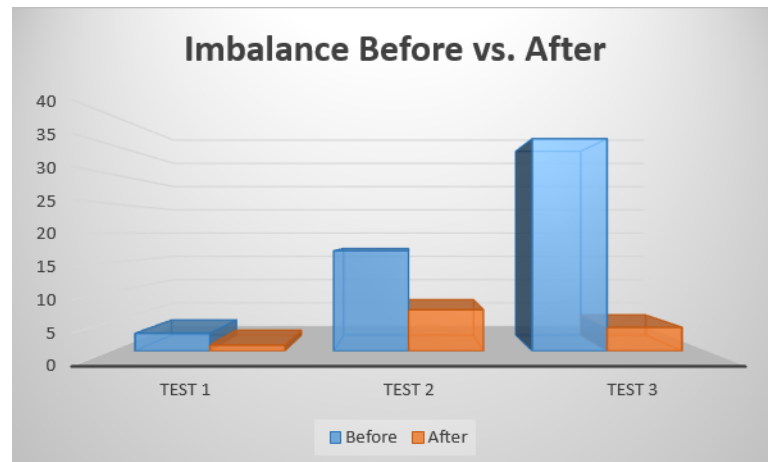


Figure 6.22: Large Imbalance Before vs. After

#### 6.4.4 Analysis

The tests were performed exactly the same way as the previous small flavour experiments were performed. The first Test, had an imbalance of 17.0 before migration and 3.0 after. Already by this point it is obvious that the machine is working hard to perform the migrations.

Test 2 had an imbalance of 17.0 before migration and 7.0 after. Compared to the other columns, it is easy to notice that the two imbalances do not differ much from each other. There is an improvement, however PM1 was heavily loaded with seven large VMs, while PM2 had five VMs where three were small and two were large. The algorithm decided to migrate three of its largest VMs from PM1 to PM2, as this would be the best choice.

One reminder is that the algorithm will not execute on three conditions; if the nodes receiving and giving changes state to under or overloaded, if the weight of the bins are exactly the same and lastly, if there is no gain by migration at all.

Test 3 has an imbalance of 36.0 before and resulted in 4.0 after migration. This test had fourteen large VMs on PM1 and eight VMs on PM 2. Out of the eight VMs, four were small and four were large flavoured. The migration resulted in five large VMs from PM1 to PM2. There was more room on PM2 because of the smaller VMs.

### 6.5 En route Proposal Acceptance

So far the experiments have only provided a long list of imbalances before and after migration. One interesting thing which is not included in those experiments are the many proposals the overloaded servers make in order to find their perfect match.

Earlier under the Approach chapter, a small animation consisting of the approach of how men propose to women was presented. This was also one of the iterations that was considered while designing the algorithm. It would be interesting to see how the men find their better halves and importantly how many it takes before finding the one.

The graphs presented in this chapter show how many *Rounds* it takes to go from the imbalance before to the imbalance after. The number of rounds are arbitrary and depends on the imbalance of the node and available possible matches. To find its best match the node searches for other neighbour nodes to exchange details about a potential match, before finding the perfect match. The graph values will be displayed in descending order from the highest imbalance point before to the lowest after. The results again are experimented in various size scales, with the large vs. small scheme.

### 6.5.1 Five Rejected Proposals - Large

Figure 6.23 shows the current imbalance and imbalance after a migration. The aim for this graph is to show how the algorithm iterates through the list of PMs to find the best match for itself. It looks for a value which is below its current state, until it finds the one perfect match. It took exactly 6 proposals, before the best match was found.

It is important to remember that the imbalance describes the average imbalance of an individual node, and not a data centre. Hence, in this particular test scaling was just as vital as any of the tests before. This centre consisted of 10 PMs and 1000 VMs. There were 40% Large VMs, 20% small and 40% medium VMs.

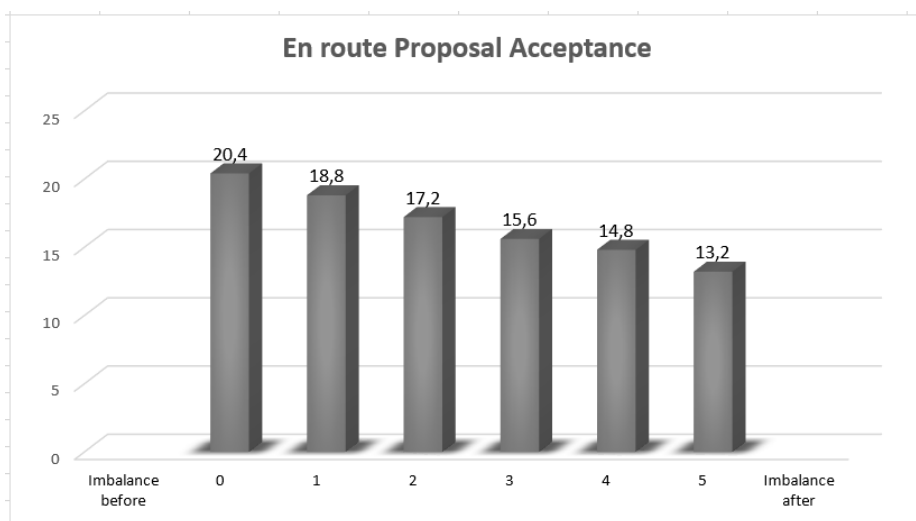


Figure 6.23: Descending Imbalance

### 6.5.2 Three Rejected Proposals

This next centre consisted of 15 PMs and 5000 VMs. There were 80% Large VMs, 20% small and 0% medium VMs. It took three rejected proposals before the final and perfect match was found:

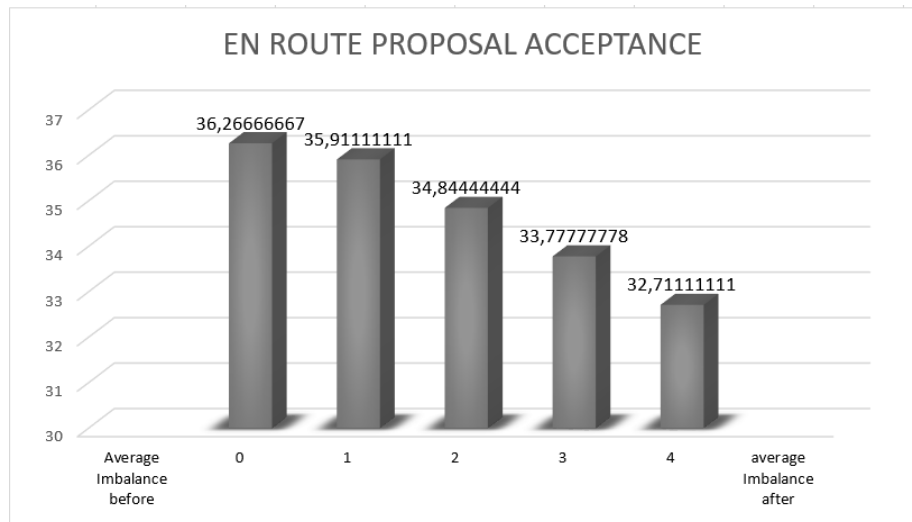


Figure 6.24: Descending Imbalance

### 6.5.3 Thirty three Rejected Proposals

To scale the situation up, the next centre consisted of 20.000 VMs and 70 PMs. The situation of the scale was 40% Large VMs, 40% small and 20% medium VMs.

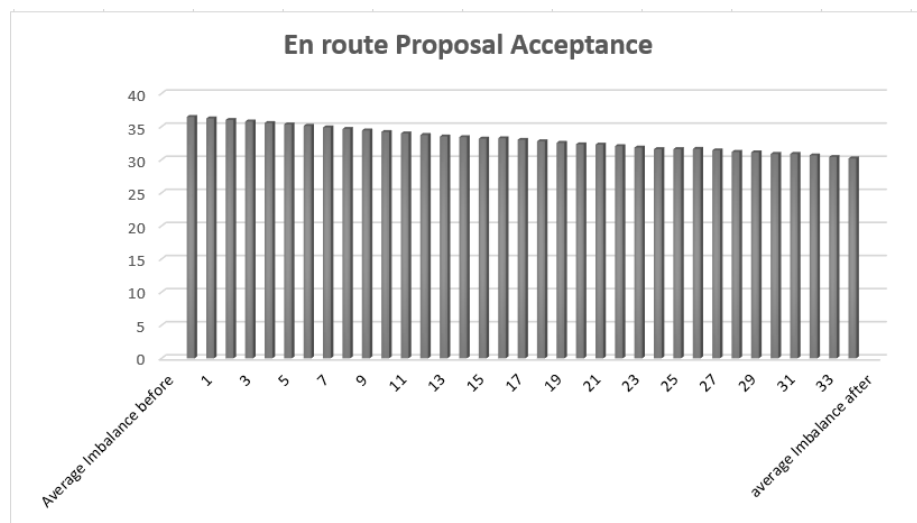


Figure 6.25: Descending Imbalance

#### 6.5.4 Twelve Rejected Proposal

This final large scheme test consisted of 25000 VMs, and 10 PMs. This time there were 40% Large VMs, 40% small and 20% medium VMs. It took the algorithm exactly twelve proposals before the perfect match was found.

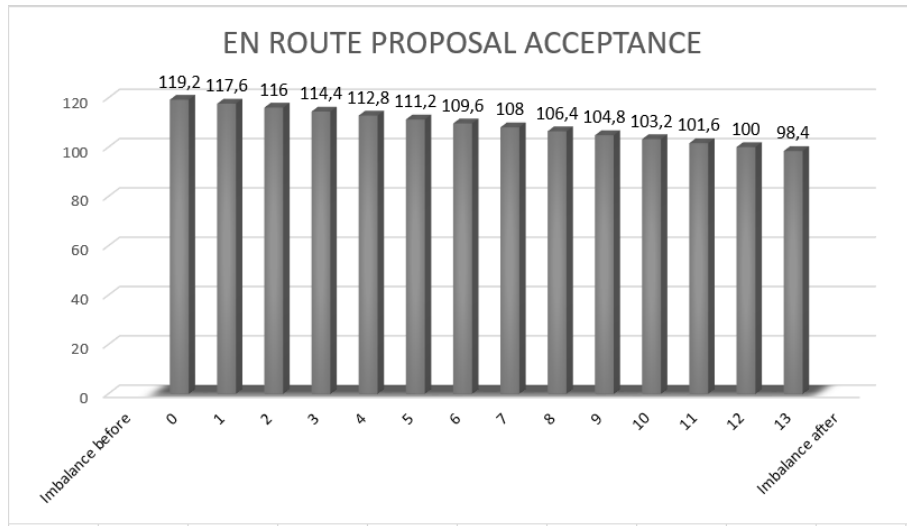


Figure 6.26: Descending Imbalance

#### 6.5.5 Analysis - Scheme Large

The results from lap count and distance to perfect match prove the fact that with more PMs available, the range to select the best partner for one node can vary depending on the amount of PMs as well. This is nothing new, however this means it takes longer for the VM to go through PMs to find its perfect match, and that slows down the process of migration.

#### 6.5.6 Three Rejected Proposals - Small

To test the smaller scale, a data centre consisting of 1000 VMs and 10 PMs, 20% Large VMs, 60% small and 20% medium VMs. The imbalance before was 21.6 and the imbalance after 5 proposals was 16.4 as the final match. It took three rejected proposals before the perfect imbalance was found.

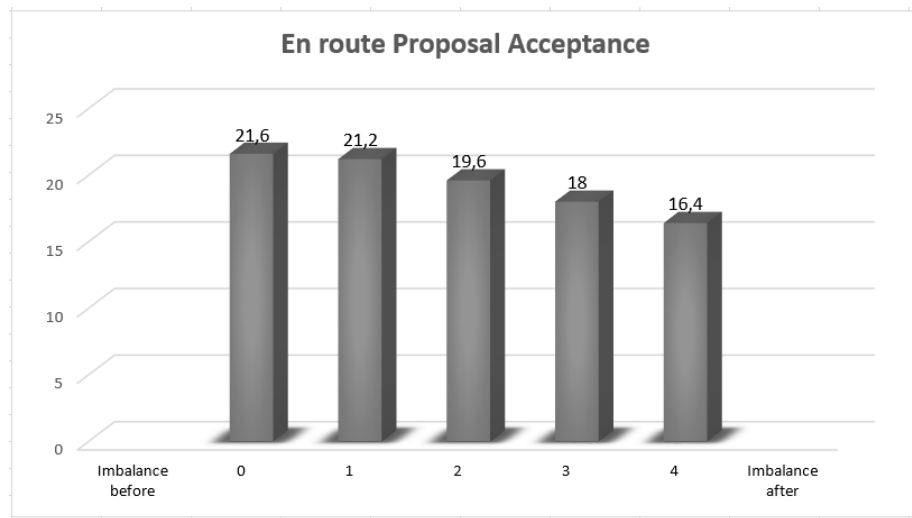


Figure 6.27: Rounds of Proposal

### 6.5.7 Ten Rejected Proposals

The scale was slightly adjusted to host 5000 VMs on top of 10 PMs. 20% Large VMs, 60% small and 20% medium VMs, were hosted on this data centre.

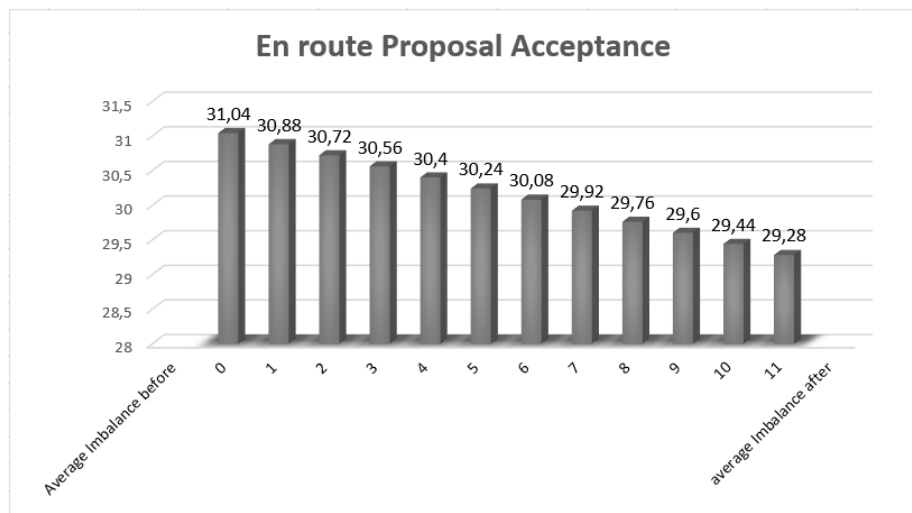


Figure 6.28: Rounds of Proposal

### 6.5.8 Eleven Rejected Proposals

The aim for this test was to check the amount of laps for the slightly scaled up smaller scheme. The data centre consisted of 20000 VMs and 25 PMs, a kinder amount compared to the larger VMs. This time there were 40% Large VMs, 40% small and 20% medium VMs.

One noticeable about this particular experiment is that the graph

doesn't point straight down in descending order. This is because in a larger infrastructure the overloaded PM might run into PMs which are slightly bigger than the previous ones, but still below it's own imbalance. It still won't consider it to be the final match unless this other node doesn't inform him of other nodes with lower imbalances.

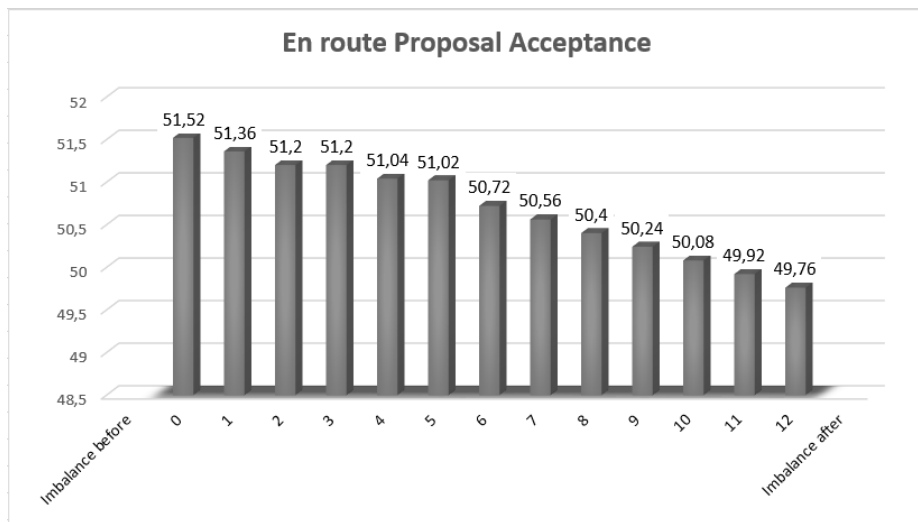


Figure 6.29: Rounds of Proposal

### 6.5.9 Eleven Rejected Proposals

The data centre consisted of 25000 VMs and 25 PMs. This time there were 20% Large VMs, 80% small and 0% medium VMs. In this test it took 11 rejected proposals before finding the perfect node.

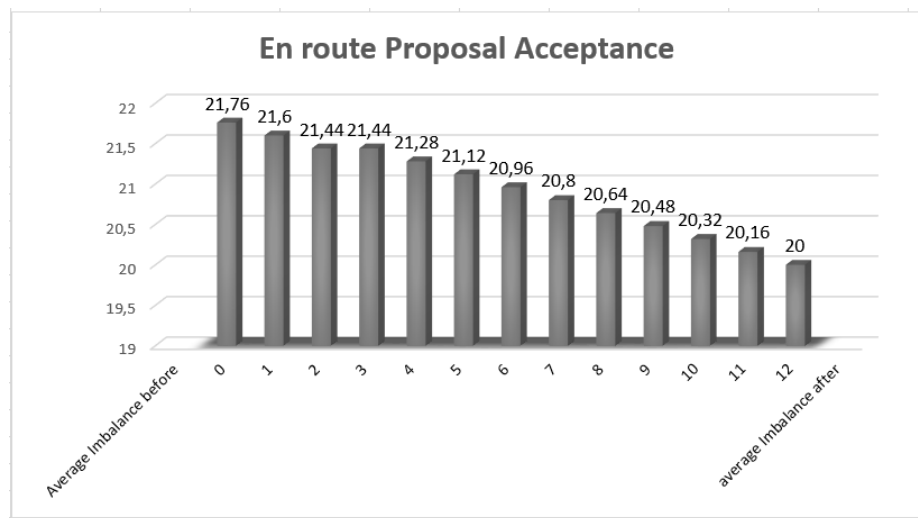


Figure 6.30: Rounds of proposal

#### **6.5.10 Analysis - Scheme Small**

Comparing the results from the larger scheme to the small scheme it is evident that the amount of PMs effect the range of selection and the time of selection. In terms of improvement, the imbalance goes down quite considerably. The larger VMs have a larger effect due to their weight and load, compared to smaller VMs effect.

## Chapter 7

# Discussion

This chapter presents the discussion which follows the end of the project. This is to reflect upon the results and see how well the solution addressed the problem statement presented in the introduction chapter and what problems were introduced along the way.

### 7.1 Background

A large amount of research have tried to optimize the situation in a cloud data center before. During the course of the thesis, it was discovered that this, however was the first thesis to contribute with the Stable Marriage algorithm in computer science.

The main idea in this project was to implement the Stable Marriage algorithm in a data center to load balance. The idea behind Stable Marriage was to find a suitable match for each virtual machine on top of physical machines in a data center until the load of a center would be even. By going forth with a distributed approach where all the nodes would communicate through message exchanges, the current situation of a center would be known at all times. This would make the partner selection easier, as each node in the system is aware of its neighbours.

The wish to work on something that could combine several fields led to this project. The project used an algorithm inspired from the field of economics, which was inspired by a real-world problem. If an idea so abstract and simple could generate good results in economics, maybe it could work well in computer science as well.

The key point in previous research on load balancing show inspired algorithms is a popular research topic for cloud data centers. It should be as well, many of the studies conclude with results that show optimized and effective solution. One particular type of research emerging fast is the type that use some form of communication or gossiping protocols [21, 29, 30]. One of the reason for this trend is that downtime is often a negative term in computing, systems that are fast, reliable and most importantly don't halt other operations are very much desired.

This project aims to use prior research as a base to introduce Stable Marriage in the group of research on distributed peer-to-peer algorithms.

## 7.2 Design & Implementation

Stable Marriage is an abstract idea, a practical problem which was easy to imagine but hard to restrict on the paper. With one study in economics and no real connection to computer science, a lot was left to the imagination. In one way this was a good thing, the complete freedom to design the algorithm in any way possible. The decision fell on finding problems with the traditional cloud data centers and look at which areas other research focus on. This is when designing a problem to address issues with scaling became interesting.

### 7.2.1 Proposals

In the Stable Matching algorithm the experiment was based on a group of couples where in the first experiment, the women propose. In the end the paper concluded with the fact that if women proposed to men, the match was better. In this project however, the Stable Marriage algorithm men propose. This decision was made solely because in terms of servers, there were two possibilities which create a load balancing problem, that being the weight of the servers. In this case the overloaded servers became men and underloaded servers became women.

The concept was designed to avoid being too complex, easy to implement yet effective. Hence the design was based on how humans tend to look for partners and make decisions, as well as implementing the stability factor, where no re-allocations would be allowed. Initially an idea to make it possible for both parties to propose was also introduced, but this would slow down the calculations and be ineffective, hence it was taken out of the design.

One of the problems which can be a drawback is the many laps the algorithm need to make in order to find its perfect match. Chapter 6 introduced experiments with how many rounds it took for the imbalance to go down. In some cases it took 13 rejected proposals before one perfect match was found. This could slow down the process a little. However, one of the biggest achievements of this algorithm is that once the perfect match for each PM is found, there won't be any need for the PM to find a new or better PM afterwards. The restrictions in the algorithm is implemented to make it impossible for a PM to change its state after a migration, the state of a PM can not go back to being overloaded or underloaded. This results in a balanced system, which is the aim of the project.

### 7.2.2 Schemes & Live migration

Another significant difference between the original idea and the implementation in this project is that the algorithm has been divided into two sets. Each set or scheme, as it has been introduced in this project, had to consider either the highest amount of load or the lowest amount of load in each bin, load for a VM is the VCPU value. Originally, there was no such scheme and the plan was to implement only the algorithm to see if it works and has its benefits. However, during the process it became interesting to see whether migrating a larger VM or smaller VM would have any effect on the situation and hence the decision to design two different schemes were taken. This was especially interesting as scaling was one of the central topics for this algorithm in particular.

The schemes are not complex at all. In terms of design and development, the Migrate Large scheme iterates through the list of VMs inside each bin to find the largest VM and based on this, finds a partner most suitable for an eventual migration. The same goes for the Migrate small scheme, only except that it considers the smaller VMs. The medium VMs are neutral, hence any particular scheme to migrate medium servers weren't made. If there are no large VMs (8 VCPUs), the medium VM is considered largest compared to the inferior VMs.

By implementing schemes, it was easier to see if the decision to either migrate a larger VM or smaller VM would make any difference on the system, if there was any gain, if it was better to migrate large or small, or even both. In the end, after implementation and experiments, the results showed that scientifically speaking there was not a large difference between the results of either. Which is positive, because it meant that both the schemes did a good job on improving current centre state and in total this meant that by implementing the Stable Marriage algorithm meant improvement either way. However, the CPU time showed a huge difference for the larger nodes compared to the smaller ones.

The larger nodes had high CPU time usage and the smaller nodes had lower time. The average CPU time for the smaller nodes were around 10.5 seconds, while for the largest VMs the average was around 58.7 seconds. It is hard to state any difference of CPU usage in this particular case, because all the VMs are running the exact same applications and programs. Hence their CPU usage will still be the same, as there is no CPU intensive jobs for the VMs. The CPU usage for both the larger and smaller nodes have been between 0.4 or 0.7 % at max depending on the process.

Another thing to consider for such a project is the migration time. This was not an area of focus in this particular project, however the time of migration was monitored to see the effect of migration. The experiments from the migrate large schemes showed a migration time between 5-6 seconds, from the time it was moving from host A to host B and up and running again. While the time of migration for the smaller scheme was around 3-4 seconds. There may not be a large difference, especially in the

constrained environment the real experiments were conducted, but if this would be to try in a larger environment, it can mean a lot of time, depending on the hardware as well.

### 7.2.3 Migrate Large Vs. Migrate Small

Schemes, which were one of the most vital features of the algorithm required little to no complexity in terms of implementation. The experiments were fun to conduct because the interest to see which scheme performed better than the other generated a lot of enthusiasm.

While all of the results generated have shown improvement, it is hard to pick if one scheme has performed better than the other. The scales have been arbitrary, but have been divided into larger and smaller infrastructure to give the closest feeling to a real data centre as possible.

Two things which can be said and addressed about the schemes are that, the benefit of either scheme depends on the preferred outcome. If migration time is one constraint, then the smaller scheme is much faster and easier for smaller businesses to implement in smaller infrastructures. If time is not a constraint and the weight is, a larger scheme would benefit. In terms of time, the latter scheme would also require a longer down time if it is an intensive VM.

Based on the results, a combination of both can work well in a dynamic data centre. It isn't necessary to only implement one or the other. One goal for any future work on this can be implementing a *Medium Scheme*. In this project, heterogeneous nodes were only small or large, in reality there might be a something in-between which can improve the state of a data centre even further.

## 7.3 Results & Analysis

Due to the unfortunate events of the physical hardware not being robust enough to handle various experiments in larger proportions, the original plan to conduct several experiments had to be cut short. With only two PMs, it seemed unreasonable to spend a fortune of time on the issues, and smarter to find solutions that could verify the Stable Marriage algorithm.

The problems with the hardware led to experiments with fewer number of VMs instead of driving the only two available servers to its death bed. Hence, the number of VMs compared to the simulation was dramatically dropped. The results were still very good compared to the expectations.

One of the benefits of testing the algorithm so many times in a simulated environment made it easier to catch and understand bugs and errors which could prevent further problems. As the analysis from the simulations tell, the first few experiments were ground breaking for further development of the algorithm. CPU, memory and scale were three

constraints that we wanted to address. There were also restrictions which were added to make the system more robust.

This project consisted of many graphs. Compared to other methods of collecting data, this method seemed like the best way to display the change of values over time. The results were one of the most important sections for this project in particular, being a stepping stone for any future implementation of Stable Marriage.

### **7.3.1 Dedicated vs. Non-dedicated links**

One of the drawbacks with the experiments performed in this project was the missing dedicated link to perform the migrations on. The internet is a shared resource and the network at Oslo and Akershus University College is used by many students. A Dedicated line is a form of communication dedicated for a special application, and in this project the PMs could have had the benefit of having such a link.

Migration time was not a direct focus in this project and with few hardware utilities setting up a dedicated link was not in focus. However, in larger networks and data centres, it could be very beneficial to implement a dedicated link to reduce the migration time, as well as the time of the selection process. For any future work on Stable Marriage, it is advised to test and implement the environment with dedicated links, to ensure faster convergence.

### **7.3.2 Managing Scaling in cloud computing**

Scaling is and always will be an important factor. The growing demand of cloud services, small and large businesses rapidly working to use this new technology, servers are working harder than ever before.

The results from this project shows that it is possible to implement different approaches with heterogeneous nodes to improve the imbalance problem of a data centre, including different amounts of the flavoured VMs. Many of the studies presented in the background chapter 2 have addressed these problems using different communication protocols which have shown great results. This is one more contribution to this area of research proving that it is possible to consider scaling, CPU, memory and performance to optimize current problems in cloud data centers.

Load balancing and consolidation has often been set up against each other. The main goals for research on consolidation has been improving the energy consumption and minimize the resource utilization. The current global status has led companies to resort to chose consolidation because green computing is a oncoming trend and reduces company expenses. In the long run, hardware is not made to be jam-packed and running on high load utilization. Load balancing focuses on optimizing performance, avoid exhaustion and flexible in terms of scaling.

For any future work, a balance between load balancing and consolidation to provide for very dynamic centers could be an option. Many of the research in load balancing is also very flexible and able to be implemented for a consolidation environment.

## 7.4 Future Work

The aim for the framework was to be able to perform live migrations based on Stable Marriage in a cloud data centre. This required a very stable and robust environment which could provide a good alternative for modern day load balancing issues.

There are many unforeseen problems that can occur during a time span of a project. Setting up the physical lab was one of them. Testing the real environment didn't seem feasible as the physical servers didn't allow for heavy jobs to be processed. The basic infrastructure and configuration was done without any problems and the virtual environment was configured quite well. The problems occurred with the third available server, which was never configured due to technical problems from the start.

To avoid spending an unnecessary amount of time on errors and trial, the implementation was implemented and tested with fewer VMs than usually. This was done by spawning several VMs with libvirt with different sizes and volumes and later tested following the Stable Marriage approach. This is obviously not an optimal way to solve things, and the first and foremost goal for any future work on this project would be to upgrade the physical hardware. The machines should be able to run many intensive jobs.

For further improvement of the algorithm, there are a few things which can be worked on. In this project CPU was one of the main constraints, which could be better monitored. Memory was also a part of the focus, however this can be much better advanced. While assigning VMs with a certain amount of memory and watching VMs migrate, one interesting thing to point at, is that the reserved amount of memory at the destination host, which was allocated but never entirely used by the VM. Hence the results may show a larger usage of memory than actual.

In defiance of practical issues, simulations and manual tests did reveal to have good effect. Improving traditional data centres consists of many bits and pieces which need to work together. It is evident that research in this area is very much needed and that there still are many interesting approaches left to discover. It would be particularly interesting to see any future implementation of Stable Marriage completely automated and even implemented in a consolidation project to compare the results to load balancing.

## Chapter 8

# Conclusion

The aim of this project was to see if by implementing the Stable Marriage algorithm load balancing could be possible in a cloud data centre. To the best of our knowledge, this is the first attempt in the literature to apply the latter algorithm in the context of load balancing in data centre.

To address the problem statement a framework which was supposed to improve scaling problems in traditional data centres was introduced. The framework was able to show that by implementing the Stable Marriage algorithm along with its schemes we were able to improve the imbalance of a cloud data centre. The imbalance varied depending on the size of a centre and no result has been negative with this implementation.

The main contribution of this thesis is the Stable Marriage algorithm, which migrates VMs based on their weight to a host of its own preference, with no need for re-allocation and unnecessary amount of live migrations. This is however a new approach and more research could benefit and improve current results.



# Bibliography

- [1] Klaithem Al Nuaimi et al. "A survey of load balancing in cloud computing: Challenges and algorithms." In: *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*. IEEE. 2012, pp. 137–142.
- [2] Md Firoj Ali and Rafiqul Zaman Khan. "The study on Load Balancing strategies in distributed computing system." In: *International Journal of Computer Science and Engineering Survey* (2012).
- [3] Kenneth Salem Ashraf Aboulnaga Cristiana Amza. "Virtualization and Databases: State of the Art and Research Challenges". URL: <https://cs.uwaterloo.ca/~ashraf/pubs/icde07dbvirtTutorial.pdf>.
- [4] Donato Barbagallo et al. "A bio-inspired algorithm for energy optimization in a self-organizing data center." In: *Self-Organizing Architectures*. Springer, 2010, pp. 127–151.
- [5] Christopher Clark et al. "Live migration of virtual machines." In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association. 2005, pp. 273–286.
- [6] Ph.D. Daniel A. Menascé. "Virtualization: Concepts, Concepts, Applications, Applications, and Performance Performance Modeling". URL: <http://cs.gmu.edu/~menasce/papers/menasce-cmg05-virt-slides.pdf>.
- [7] Marco Dorigo et al. *Ant Colony Optimization and Swarm Intelligence: 6th International Conference, ANTS 2008, Brussels, Belgium, September 22-24, 2008, Proceedings*. Vol. 5217. Springer, 2008.
- [8] Eugen Feller, Louis Rilling, and Christine Morin. "Energy-aware ant colony based workload placement in clouds." In: *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*. IEEE Computer Society. 2011, pp. 26–33.
- [9] Saurabh Goyal. "Centralized vs Decentralized vs Distributed". URL: <https://medium.com/@bbc4468/centralized-vs-decentralized-vs-distributed-41d92d463868>.
- [10] Thomas Hage. "The CERES project-A Cloud Energy Reduction System." In: (2014).
- [11] James Hamblin. "Math for Liberal Studies: Bin-Packing Algorithms". URL: <https://www.youtube.com/watch?v=vUxhAmfXs2o>.

- [12] Hung-Chang Hsiao et al. "Load rebalancing for distributed file systems in clouds." In: *IEEE transactions on parallel and distributed systems* 24.5 (2013), pp. 951–962.
- [13] David S Johnson. "Near-optimal bin packing algorithms". URL: <https://dspace.mit.edu/handle/1721.1/57819#files-area>.
- [14] Christine Mayap Kamga, Giang Son Tran, and Laurent Broto. "Extended scheduler for efficient frequency scaling in virtualized systems." In: *ACM SIGOPS Operating Systems Review* 46.2 (2012), pp. 28–35.
- [15] Dervis Karaboga. *An idea based on honey bee swarm for numerical optimization*. Tech. rep. Technical report-tr06, Erciyes university, engineering faculty, computer engineering department, 2005.
- [16] Libvirt. "What is libvirt". URL: [http://wiki.libvirt.org/page/FAQ#What\\_is\\_libvirt.3F](http://wiki.libvirt.org/page/FAQ#What_is_libvirt.3F).
- [17] Alvin Roth Lloyd Shapley. "Stable matching: Theory, evidence, and practical design". URL: [http://www.nobelprize.org/nobel\\_prizes/economic-sciences/laureates/2012/popular-economicsciences2012.pdf](http://www.nobelprize.org/nobel_prizes/economic-sciences/laureates/2012/popular-economicsciences2012.pdf).
- [18] What is Margaret Rouse. "Cloud computing". URL: <http://searchcloudcomputing.techtarget.com/definition/cloud-computing>.
- [19] What is Margaret Rouse. "SPI model (SaaS, PaaS, IaaS)". URL: <http://searchcloudcomputing.techtarget.com/definition/SPI-model>.
- [20] Moreno Marzolla, Ozalp Babaoglu, and Fabio Panzieri. "Server consolidation in clouds through gossiping." In: *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a*. IEEE. 2011, pp. 1–6.
- [21] Carlo Mastroianni, Michela Meo, and Giuseppe Papuzzo. "Self-economy in cloud data centers: Statistical assignment and migration of virtual machines." In: *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 407–418.
- [22] Mayank Mishra et al. "Dynamic resource management using virtual machine migrations." In: *Communications Magazine, IEEE* 50.9 (2012), pp. 34–40.
- [23] Libvirt N. "What is libvirt - FAQ". URL: <https://wiki.libvirt.org/page/Networking>.
- [24] NIST. "Guide to Security for Full Virtualization Technologies". URL: <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-125.pdf>.
- [25] Lucas Nussbaum et al. "Linux-based virtualization for HPC clusters." In: *Montreal Linux Symposium*. 2009.
- [26] Martin Randles, David Lamb, and A Taleb-Bendiab. "A comparative study into distributed load balancing algorithms for cloud computing." In: *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*. IEEE. 2010, pp. 551–556.

- [27] Ananth Rao et al. "Load balancing in structured P2P systems." In: *International Workshop on Peer-to-Peer Systems*. Springer. 2003, pp. 68–79.
- [28] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. "Virtualization: A survey on concepts, taxonomy and associated security issues." In: *Computer and Network Technology (ICCNT), 2010 Second International Conference on*. IEEE. 2010, pp. 222–226.
- [29] Mina Sedaghat, Francisco Hernández-Rodríguez, and Erik Elmroth. "Autonomic resource allocation for cloud data centers: A peer to peer approach." In: *Cloud and Autonomic Computing (ICCAC), 2014 International Conference on*. IEEE. 2014, pp. 131–140.
- [30] Mina Sedaghat et al. "Divide the task, multiply the outcome: Cooperative VM consolidation." In: *2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2014, pp. 300–305.
- [31] Richa Singh. "Load balancing in Distributed Systems". URL: <https://www.slideshare.net/RichaSingh59/load-balancing-in-distributed-systems>.
- [32] Fredrik Meyn Ung. "Towards efficient and cost-effective live migrations of virtual machines." In: (2015).
- [33] UNL. "Bin Packing algorithms". URL: <http://www.math.unl.edu/~s-sjessie1/203Handouts/Bin%5C%20Packing.pdf>.
- [34] VMware. "Understanding full virtualization, paravirtualization and hardware assist". URL: <http://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html>.
- [35] Wiki. "Bin Packing". URL: [https://en.wikipedia.org/wiki/Bin\\_packing\\_problem](https://en.wikipedia.org/wiki/Bin_packing_problem).
- [36] Wiki. "NP-Hardness". URL: <https://en.wikipedia.org/wiki/NP-hardness>.
- [37] wiki. "Centralisation". URL: <https://en.wikipedia.org/wiki/Centralisation>.
- [38] wiki. "Decentralized system". URL: [https://en.wikipedia.org/wiki/Decentralised\\_system](https://en.wikipedia.org/wiki/Decentralised_system).
- [39] wiki. "Distributed Computing". URL: [https://en.wikipedia.org/wiki/Distributed\\_computing](https://en.wikipedia.org/wiki/Distributed_computing).
- [40] wiki. "Swarm Intelligence". URL: [https://en.wikipedia.org/wiki/Swarm\\_intelligence](https://en.wikipedia.org/wiki/Swarm_intelligence).
- [41] Wikipedia. "Features KVM-QEMU". URL: <http://wiki.qemu.org/Features/KVM>.
- [42] Wikipedia. "P-value". URL: <https://en.wikipedia.org/wiki/P-value>.
- [43] Wolfram. "NP-Hard problem". URL: <http://mathworld.wolfram.com/NP-HardProblem.html>.
- [44] Xin-She Yang. *Nature-inspired optimization algorithms*. Elsevier, 2014.



# Appendices



## Appendix A

# Working Environment

### A.1 create isc hosts dhcp.py

---

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import random
5  import pprint
6  import os
7
8  networkfile="isc_dhcp_hosts"
9
10 def randomMAC():
11     # mac = [ 0x00, 0x16, 0x3e,
12     mac = [ 0x52, 0x54, 0x00,
13             random.randint(0x00, 0x7f),
14             random.randint(0x00, 0xff),
15             random.randint(0x00, 0xff) ]
16     return ':'.join(map(lambda x: "%02x" % x, mac))
17
18 with open(networkfile, "a") as f:
19
20     for i in range(1,101):
21         mac = randomMAC()
22         hostname = "vm"+str(i)
23         host_octet = str(i+100)
24         f.write("host "+hostname+" {\n")
25         f.write("\thardware ethernet "+mac.upper()+";\n")
26         f.write("\tfixed-address 192.168.1."+host_octet+";\n")
27         f.write("}\n")
28     f.close()
```

---

## A.2 vm-net

---

```
1 <network>
2   <name>vms</name>
3   <bridge name="virbr0" />
4   <forward mode="nat"/>
5     <ip address="192.168.1.1" netmask="255.255.255.0">
6       <dhcp>
7         <range start="192.168.1.210" end="192.168.1.250" />
8         <host mac="00:16:3e:4f:fd:4a" name="vm1" ip="192.168.1.101" />
9         <host mac="00:16:3e:6c:48:ef" name="vm2" ip="192.168.1.102" />
10        <host mac="00:16:3e:5c:b1:50" name="vm3" ip="192.168.1.103" />
11        <.....>
12        <host mac="00:16:3e:28:c7:06" name="vm100" ip="192.168.1.200" />
13      </dhcp>
14    </ip>
15 </network>
```

---

## A.3 deploy\_large.py

---

```
1 import random
2 import math
3
4
5 def Get_Bin_Load(bin):
6     load=0
7     for i in range(len(bin['VMS'])):
8         #print "here bin['VMS'][0]", bin['VMS'][0]
9         #print "here bin['VMS'][0]['cpu']", bin['VMS'][0]['cpu']
10        load=load+bin['VMS'][i]['cpu']
11    return load
12
13
14
15 def Gain_of_Migration_Couple(Over_Bin, Under_Bin, Big_VM, average_cpu_load):
16
17     ##### Before #####
18     Imbalance_Over_Bin_Before=math.fabs(Get_Bin_Load(Over_Bin)-average_cpu_load)
19     #print "Imbalance_Over_Bin_Before ", Imbalance_Over_Bin_Before
20     Imbalance_Under_Bin_Before=math.fabs(Get_Bin_Load(Under_Bin)-average_cpu_load)
21     #print "Imbalance_Under_Bin_Before ", Imbalance_Under_Bin_Before
22
23     Total_Imbalance_Before=Imbalance_Over_Bin_Before+Imbalance_Under_Bin_Before
24     #print "Total_Imbalance_Before", Total_Imbalance_Before
25
26     #imbalance after eventual migration
27     Imbalance_Over_Bin_After=math.fabs(Get_Bin_Load(Over_Bin)-Big_VM['cpu']-average_cpu_load)
28     #print "Imbalance_Over_Bin_After", Imbalance_Over_Bin_After
29
30
31     ##### AFTER #####
32     Imbalance_Under_Bin_After=math.fabs(Get_Bin_Load(Under_Bin)+Big_VM['cpu']-average_cpu_load)
33     #print "Imbalance_Under_Bin_After", Imbalance_Under_Bin_After
34
35     Total_Imbalance_After=Imbalance_Over_Bin_After+Imbalance_Under_Bin_After
36
37     ##### Gain (difference) #####
38
39     Gain=Total_Imbalance_Before-Total_Imbalance_After
40     return Gain
41
42 def Get_Index_Largest(bin):
43     max=0
44     index_max=0
45     for i in range(len(bin['VMS'])):
46         if bin['VMS'][i]['cpu']>max:
```

```

47         max=bin['VMS'][i]['cpu']
48         index_max=i
49
50
51     return index_max
52
53
54 VM_list = []
55
56 small_vm = {
57     'name': 'small%d',
58     'cpu': 2,
59     'mem': 2048,
60     'disk': 20,
61     'n': 1
62 }
63
64 med_vm = {
65     'name': 'medium%d',
66     'cpu': 4,
67     'mem': 4096,
68     'disk': 40,
69     'n': 1
70 }
71
72 large_vm = {
73     'name': 'large%d',
74     'cpu': 8,
75     'mem': 8192,
76     'disk': 80,
77     'n': 1
78 }
79
80
81 Bins=[]
82
83 #2 Physical Machines
84 bin_0 = {
85     'id':0,
86     'cpu': 48, # 4.0 overcommit with cpu
87     'mem': 240000,
88     'disk': 2000,
89     'VMS': [],
90 }
91
92
93 bin_1 = {
94     'id':1,
95     'cpu': 48, # 4.0 overcommit with cpu

```

```

96     'mem': 240000,
97     'disk': 2000,
98     'VMS': [],
99 }
100
101
102 # 2 Bins
103
104 Bins.append(bin_0)
105 Bins.append(bin_1)
106
107 K=len(Bins)
108
109 #Populate BIN 1
110 Bins[0]['VMS'].append(small_vm)
111 Bins[0]['VMS'].append(large_vm)
112 Bins[0]['VMS'].append(large_vm)
113 Bins[0]['VMS'].append(large_vm)
114 Bins[0]['VMS'].append(large_vm)
115
116
117 #Populate BIN 2
118 Bins[1]['VMS'].append(small_vm)
119 Bins[1]['VMS'].append(small_vm)
120 Bins[1]['VMS'].append(large_vm)
121 Bins[1]['VMS'].append(med_vm)
122
123
124 #All VMS2
125 VM_list=Bins[1]['VMS']+Bins[0]['VMS']
126 total_cpu_load=0
127
128
129 for i in range(len(VM_list)):
130     total_cpu_load=total_cpu_load+VM_list[i]['cpu']
131
132
133 average_cpu_load=total_cpu_load*1.0/K
134
135 print "average cpu load", average_cpu_load
136
137
138 Bins_Under_Utilized=[]
139 Bins_Over_Utilized=[]
140
141
142 total_load_check=0
143 for i in range(K):
144     if Get_Bin_Load(Bins[i])>=average_cpu_load:

```

```

145         print "got it over"
146         Bins_Over_Utilized.append(Bins[i])
147     else:
148         print "got it under"
149         Bins_Under_Utilized.append(Bins[i])
150
151     total_load_check=total_load_check+Get_Bin_Load(Bins[i])
152
153
154     for i in range(K):
155         index_largest= int(Get_Index_Largest(Bins[i]))
156         #print "index largest",index_largest
157         #print "type of VM for Bin number ", i, " is ", Bins[i]['VMS'][index_largest]['name']
158
159
160
161
162     #print "here", Bins_Over_Utilized
163     Over_Bin=Bins_Over_Utilized[0]
164     Under_Bin=Bins_Under_Utilized[0]
165
166     Big_VM=Over_Bin['VMS'][Get_Index_Largest(Over_Bin)]
167
168
169
170
171     Bins_Before=Bins[:]
172     Imbalance_before=0
173
174     for bin in Bins:
175         Imbalance_before=Imbalance_before+math.fabs(Get_Bin_Load(bin)-average_cpu_load)
176     print "Average Imbalance before", Imbalance_before*1.0/K
177
178
179     Bins_After=Bins[:]
180
181
182
183     min=1000000000000
184
185
186     NO_more_gain=False
187     while (NO_more_gain==False):
188         NO_more_gain=False
189         for under_bin in Bins_Under_Utilized:
190             min_best_to_balance=min
191             best_over_bin_to_balance=Bins_Over_Utilized[0]
192             distance_to_balance=-1
193             for over_bin in Bins_Over_Utilized:

```

```

194         #print "Over bin er: ", over_bin
195         #print over_bin['VMS']
196         largeVM=over_bin['VMS'][Get_Index_Largest(over_bin)]
197
198         distance_to_balance=average_cpu_load-(Get_Bin_Load(under_bin)+largeVM['cpu'])
199         if (distance_to_balance<=0):
200             NO_more_gain=True
201         if (distance_to_balance>0):
202             #print "getting into loop"
203             #difference to the average
204
205             if (distance_to_balance<min_best_to_balance):
206                 #print "before it ----"
207                 min_best_to_balance=distance_to_balance
208                 best_over_bin_to_balance= over_bin
209                 #print "inside best", distance_to_balance
210
211         if (distance_to_balance!=min):
212             Bins_Under_Utilized.remove(under_bin)
213             Bins_Over_Utilized.remove(best_over_bin_to_balance)
214             print "moving large", largeVM
215             print "from: ", under_bin['id']
216             print "to: ", best_over_bin_to_balance['id']
217             under_bin['VMS'].append(largeVM)
218             best_over_bin_to_balance['VMS'].pop(Get_Index_Largest(best_over_bin_to_balance))
219             Bins_Under_Utilized.append(under_bin)
220             Bins_Over_Utilized.append(best_over_bin_to_balance)
221
222
223
224
225     Bins=Bins_Under_Utilized+Bins_Over_Utilized
226     Imbalance_after=0
227
228     for bin in Bins:
229         Imbalance_after=Imbalance_after+math.fabs(Get_Bin_Load(bin)-average_cpu_load)
230     print "average Imbalance after", Imbalance_after*1.0/K

```

---

## A.4 deploy\_small.py

---

```
1 import random
2 import math
3
4
5 def Get_Bin_Load(bin):
6     load=0
7     for i in range(len(bin['VMS'])):
8         load=load+bin['VMS'][i]['cpu']
9     return load
10
11
12 def Gain_of_Migration_Couple(Over_Bin, Under_Bin, Big_VM, average_cpu_load):
13
14     ##### Before #####
15     Imbalance_Over_Bin_Before=math.fabs(Get_Bin_Load(Over_Bin)-average_cpu_load)
16     #print "Imbalance_Over_Bin_Before ", Imbalance_Over_Bin_Before
17     Imbalance_Under_Bin_Before=math.fabs(Get_Bin_Load(Under_Bin)-average_cpu_load)
18     #print "Imbalance_Under_Bin_Before ", Imbalance_Under_Bin_Before
19
20     Total_Imbalance_Before=Imbalance_Over_Bin_Before+Imbalance_Under_Bin_Before
21     #print "Total_Imbalance_Before", Total_Imbalance_Before
22
23
24     #imbalance after eventual migration
25     Imbalance_Over_Bin_After=math.fabs(Get_Bin_Load(Over_Bin)-Big_VM['cpu']-average_cpu_load)
26     ##### AFTER #####
27
28
29     Imbalance_Under_Bin_After=math.fabs(Get_Bin_Load(Under_Bin)+Big_VM['cpu']-average_cpu_load)
30     #print "Imbalance_Under_Bin_After", Imbalance_Under_Bin_After
31
32     Total_Imbalance_After=Imbalance_Over_Bin_After+Imbalance_Under_Bin_After
33
34     ##### Gain (difference) #####
35     Gain=Total_Imbalance_Before-Total_Imbalance_After
36     return Gain
37
38 def Get_Index_Largest(bin):
39     max=0
40     index_max=0
41     for i in range(len(bin['VMS'])):
42         if bin['VMS'][i]['cpu']>max:
43             max=bin['VMS'][i]['cpu']
44             index_max=i
45     return index_max
46
```

```

47
48 def Get_Index_Smallest(bin):
49     min=10000000000
50     index_min=0
51     for i in range(len(bin['VMS'])):
52         if bin['VMS'][i]['cpu']<min:
53             min=bin['VMS'][i]['cpu']
54             index_min=i
55     return index_min
56
57 VM_list = []
58
59 small_vm = {
60     'name': 'small%d',
61     'cpu': 2,
62     'mem': 2048,
63     'disk': 20,
64     'n': 1
65 }
66
67 med_vm = {
68     'name': 'medium%d',
69     'cpu': 4,
70     'mem': 4096,
71     'disk': 40,
72     'n': 1
73 }
74
75 large_vm = {
76     'name': 'large%d',
77     'cpu': 8,
78     'mem': 8192,
79     'disk': 80,
80     'n': 1
81 }
82
83
84 Bins=[]
85
86 #2 Physical Machines
87 bin_0 = {
88     'id':0,
89     'cpu': 48, # 4.0 overcommit with cpu
90     'mem': 240000,
91     'disk': 2000,
92     'VMS': [],
93 }
94
95

```

```

96  bin_1 = {
97      'id':1,
98      'cpu': 48, # 4.0 overcommit with cpu
99      'mem': 240000,
100     'disk': 2000,
101     'VMS': [],
102 }
103
104
105 Bins.append(bin_0)
106 Bins.append(bin_1)
107
108 K=len(Bins)
109
110
111 #Populate BIN 1
112 Bins[0]['VMS'].append(small_vm)
113 Bins[0]['VMS'].append(large_vm)
114 Bins[0]['VMS'].append(large_vm)
115 Bins[0]['VMS'].append(large_vm)
116 Bins[0]['VMS'].append(large_vm)
117
118
119 #Populate BIN 2
120 Bins[1]['VMS'].append(small_vm)
121 Bins[1]['VMS'].append(small_vm)
122 Bins[1]['VMS'].append(large_vm)
123 Bins[1]['VMS'].append(med_vm)
124
125 VM_list=Bins[1]['VMS']+Bins[0]['VMS']
126
127 total_cpu_load=0
128
129
130 for i in range(len(VM_list)):
131     total_cpu_load=total_cpu_load+VM_list[i]['cpu']
132
133
134 average_cpu_load=total_cpu_load*1.0/K
135
136 print "average cpu load", average_cpu_load
137
138
139 Bins_Under_Utilized=[]
140 Bins_Over_Utilized=[]
141
142 total_load_check=0
143 for i in range(K):
144     if Get_Bin_Load(Bins[i])>average_cpu_load:

```

```

145         Bins_Over_Utilized.append(Bins[i])
146     else:
147         Bins_Under_Utilized.append(Bins[i])
148
149     total_load_check=total_load_check+Get_Bin_Load(Bins[i])
150
151
152
153     for i in range(K):
154         index_largest= int(Get_Index_Largest(Bins[i]))
155         #print "index largest",index_largest
156         #print "type of VM for Bin number ", i, " is ", Bins[i]['VMS'][index_largest]['name']
157
158
159
160     Over_Bin=Bins_Over_Utilized[0]
161     Under_Bin=Bins_Under_Utilized[0]
162
163     Big_VM=Over_Bin['VMS'][Get_Index_Largest(Over_Bin)]
164
165
166
167
168     Bins_Before=Bins[:]
169
170
171     Imbalance_before=0
172
173     for bin in Bins:
174         Imbalance_before=Imbalance_before+math.fabs(Get_Bin_Load(bin)-average_cpu_load)
175     print "Average Imbalance before", Imbalance_before*1.0/K
176
177     Bins_After=Bins[:]
178
179
180
181     min=1000000000000
182
183
184     NO_more_gain=False
185     while (NO_more_gain==False):
186         NO_more_gain=False
187         for under_bin in Bins_Under_Utilized:
188             min_best_to_balance=min
189             best_over_bin_to_balance=Bins_Over_Utilized[0]
190             distance_to_balance=-1
191             for over_bin in Bins_Over_Utilized:
192                 largeVM=over_bin['VMS'][Get_Index_Smallest(over_bin)]
193                 #should not get underload

```

```

194
195     distance_to_balance=average_cpu_load-(Get_Bin_Load(under_bin)+largeVM['cpu'])
196     if (distance_to_balance<=0):
197         NO_more_gain=True
198     if (distance_to_balance>0):
199         if (distance_to_balance<min_best_to_balance):
200             #print "before it ----"
201             min_best_to_balance=distance_to_balance
202             best_over_bin_to_balance= over_bin
203             #print "inside best", distance_to_balance
204
205     if (distance_to_balance!=min):
206         Bins_Under_Utilized.remove(under_bin)
207         Bins_Over_Utilized.remove(best_over_bin_to_balance)
208         print "moving large", largeVM
209         print "from: ", under_bin['id']
210         print "to: ", best_over_bin_to_balance['id']
211
212         under_bin['VMS'].append(largeVM)
213         best_over_bin_to_balance['VMS'].pop(Get_Index_Smallest(best_over_bin_to_balance))
214         Bins_Under_Utilized.append(under_bin)
215         Bins_Over_Utilized.append(best_over_bin_to_balance)
216
217
218     Bins_Under_Utilized[0], Big_VM, average_cpu_load)
219
220
221
222     Bins=Bins_Under_Utilized+Bins_Over_Utilized
223     Imbalance_after=0
224
225     for bin in Bins:
226         Imbalance_after=Imbalance_after+math.fabs(Get_Bin_Load(bin)-average_cpu_load)
227     print "average Imbalance after", Imbalance_after*1.0/K

```

---

## Appendix B

# Artificial Simulation

### B.1 Many to Many Move Smallest.py

---

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  import random
4  import math
5
6
7  def Get_Bin_Load(bin):
8      load=0
9      for i in range(len(bin['VMS'])):
10         load=load+bin['VMS'][i]['cpu']
11     return load
12
13
14  def Gain_of_Migration_Couple(Over_Bin, Under_Bin, Big_VM, average_cpu_load):
15
16     ##### Before #####
17     Imbalance_Over_Bin_Before=math.fabs(Get_Bin_Load(Over_Bin)-average_cpu_load)
18     Imbalance_Under_Bin_Before=math.fabs(Get_Bin_Load(Under_Bin)-average_cpu_load)
19     Total_Imbalance_Before=Imbalance_Over_Bin_Before+Imbalance_Under_Bin_Before
20
21     #imbalance after eventual migration
22     Imbalance_Over_Bin_After=math.fabs(Get_Bin_Load(Over_Bin)-Big_VM['cpu']-average_cpu_load)
23
24     ##### AFTER #####
25     Imbalance_Under_Bin_After=math.fabs(Get_Bin_Load(Under_Bin)+Big_VM['cpu']-average_cpu_load)
26     Total_Imbalance_After=Imbalance_Over_Bin_After+Imbalance_Under_Bin_After
27
28     ##### Gain (difference) #####
29     Gain=Total_Imbalance_Before-Total_Imbalance_After
30     return Gain
31
32  def Get_Index_Largest(bin):
```

```

33     max=0
34     index_max=0
35     for i in range(len(bin['VMS'])):
36         if bin['VMS'][i]['cpu']>max:
37             max=bin['VMS'][i]['cpu']
38             index_max=i
39     return index_max
40
41
42 def Get_Index_Smallest(bin):
43     min=10000000000
44     index_min=0
45     for i in range(len(bin['VMS'])):
46         if bin['VMS'][i]['cpu']<min:
47             min=bin['VMS'][i]['cpu']
48             index_min=i
49
50
51     return index_min
52
53 N = 20000
54 N_large=N*4/10
55 N_small=N*4/10
56 N_med=N*2/10
57
58 #N=60
59 K=15
60 #K=10
61
62 VM_list = []
63 for i in range(N_small):
64     small_vm = {
65         'name': 'small%d' % i,
66         'cpu': 2,
67         'mem': 2048,
68         'disk': 20,
69         'n': 1
70     }
71     VM_list.append(small_vm)
72
73 for i in range(N_med):
74     med_vm = {
75         'name': 'medium%d' % i,
76         'cpu': 4,
77         'mem': 4096,
78         'disk': 40,
79         'n': 1
80     }
81     VM_list.append(med_vm)

```

```

82
83
84 for i in range(N_large):
85     large_vm = {
86         'name': 'large%d' % i,
87         'cpu': 8,
88         'mem': 8192,
89         'disk': 80,
90         'n': 1
91     }
92     VM_list.append(large_vm)
93
94
95 Bins=[]
96 for i in range(K):
97
98     bin = {
99         'id':i,
100         'cpu': 48*4, # 4.0 overcommit with cpu
101         'mem': 240000,
102         'disk': 2000,
103         'VMS': [],
104     }
105     Bins.append(bin)
106
107
108 Bins=[]
109 for i in range(K):
110
111     bin = {
112         'id':i,
113         'cpu': 48*4, # 4.0 overcommit with cpu
114         'mem': 240000,
115         'disk': 2000,
116         'VMS': [],
117     }
118     Bins.append(bin)
119
120 random.shuffle(VM_list)
121
122
123 for i in range(N):
124     index_bin=i%K
125     Bins[index_bin]['VMS'].append(VM_list[i])
126
127
128 total_cpu_load=0
129
130

```

```

131 for i in range(N):
132     total_cpu_load=total_cpu_load+VM_list[i]['cpu']
133
134
135 average_cpu_load=total_cpu_load*1.0/K
136
137 print "average cpu load", average_cpu_load
138
139
140 Bins_Under_Utilized=[]
141 Bins_Over_Utilized=[]
142
143
144 total_load_check=0
145 for i in range(K):
146     if Get_Bin_Load(Bins[i])>average_cpu_load:
147         Bins_Over_Utilized.append(Bins[i])
148     else:
149         Bins_Under_Utilized.append(Bins[i])
150
151     total_load_check=total_load_check+Get_Bin_Load(Bins[i])
152
153 for i in range(K):
154     index_largest= int(Get_Index_Largest(Bins[i]))
155     #print "index largest",index_largest
156     #print "type of VM for Bin number ", i, " is ", Bins[i]['VMS'][index_largest]['name']
157
158
159 Over_Bin=Bins_Over_Utilized[0]
160 Under_Bin=Bins_Under_Utilized[0]
161
162 Big_VM=Over_Bin['VMS'][Get_Index_Largest(Over_Bin)]
163
164
165 Bins_Before=Bins[:]
166
167
168 Imbalance_before=0
169
170 for bin in Bins:
171     Imbalance_before=Imbalance_before+math.fabs(Get_Bin_Load(bin)-average_cpu_load)
172 print "Average Imbalance before", Imbalance_before*1.0/K
173
174
175 Bins_After=Bins[:]
176
177 min=1000000000000
178
179

```

```

180 NO_more_gain=False
181 while (NO_more_gain==False):
182     NO_more_gain=False
183     for under_bin in Bins_Under_Utilized:
184         min_best_to_balance=min
185         best_over_bin_to_balance=Bins_Over_Utilized[0]
186         distance_to_balance=-1
187         for over_bin in Bins_Over_Utilized:
188             largeVM=over_bin['VMS'][Get_Index_Smallest(over_bin)]
189             #should not get underload
190
191             distance_to_balance=average_cpu_load-(Get_Bin_Load(under_bin)+largeVM['cpu'])
192             if (distance_to_balance<=0):
193                 NO_more_gain=True
194             if (distance_to_balance>0):
195                 #print "getting into loop"
196             #difference to the average
197
198
199             #print "outside", distance_to_balance
200
201             if (distance_to_balance<min_best_to_balance):
202                 #print "before it ----"
203                 min_best_to_balance=distance_to_balance
204                 best_over_bin_to_balance= over_bin
205                 #print "inside best", distance_to_balance
206
207             if (distance_to_balance!=min):
208                 #print "found a match"
209                 Bins_Under_Utilized.remove(under_bin)
210                 Bins_Over_Utilized.remove(best_over_bin_to_balance)
211                 under_bin['VMS'].append(largeVM)
212                 best_over_bin_to_balance['VMS'].pop(Get_Index_Smallest(best_over_bin_to_balance))
213                 Bins_Under_Utilized.append(under_bin)
214                 Bins_Over_Utilized.append(best_over_bin_to_balance)
215                 #NO_more_gain=True
216
217 Bins=Bins_Under_Utilized+Bins_Over_Utilized
218 Imbalance_after=0
219
220 for bin in Bins:
221     Imbalance_after=Imbalance_after+math.fabs(Get_Bin_Load(bin)-average_cpu_load)
222 print "average Imbalance after", Imbalance_after*1.0/K

```

---

## B.2 Many to Many Move Largest.py

---

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import random
5  import math
6
7  def Get_Bin_Load(bin):
8      load=0
9      for i in range(len(bin['VMS'])):
10         load=load+bin['VMS'][i]['cpu']
11
12     return load
13
14
15 def Gain_of_Migration_Couple(Over_Bin, Under_Bin, Big_VM, average_cpu_load):
16
17     ##### Before #####
18     Imbalance_Over_Bin_Before=math.fabs(Get_Bin_Load(Over_Bin)-average_cpu_load)
19     Imbalance_Under_Bin_Before=math.fabs(Get_Bin_Load(Under_Bin)-average_cpu_load)
20
21     Total_Imbalance_Before=Imbalance_Over_Bin_Before+Imbalance_Under_Bin_Before
22     #imbalance after eventual migration
23     Imbalance_Over_Bin_After=math.fabs(Get_Bin_Load(Over_Bin)-Big_VM['cpu']-average_cpu_load)
24
25     ##### AFTER #####
26     Imbalance_Under_Bin_After=math.fabs(Get_Bin_Load(Under_Bin)+Big_VM['cpu']-average_cpu_load)
27     Total_Imbalance_After=Imbalance_Over_Bin_After+Imbalance_Under_Bin_After
28
29     ##### Gain (difference) #####
30
31     Gain=Total_Imbalance_Before-Total_Imbalance_After
32     #Positive Gain means we went from Worse to Better
33     return Gain
34
35 def Get_Index_Largest(bin):
36     max=0
37     index_max=0
38     for i in range(len(bin['VMS'])):
39         if bin['VMS'][i]['cpu']>max:
40             max=bin['VMS'][i]['cpu']
41             index_max=i
42
43     return index_max
44
45 N = 20000
46 N_large=N*4/10
```

```

47 N_small=N*4/10
48 N_med=N*2/10
49
50 #N=60
51 K=15
52 # number of bins
53 #K=10
54
55 VM_list = []
56 for i in range(N_small):
57     small_vm = {
58         'name': 'small%d' % i,
59         'cpu': 2,
60         'mem': 2048,
61         'disk': 20,
62         'n': 1
63     }
64     VM_list.append(small_vm)
65
66 for i in range(N_med):
67     med_vm = {
68         'name': 'medium%d' % i,
69         'cpu': 4,
70         'mem': 4096,
71         'disk': 40,
72         'n': 1
73     }
74     VM_list.append(med_vm)
75
76 for i in range(N_large):
77     large_vm = {
78         'name': 'large%d' % i,
79         'cpu': 8,
80         'mem': 8192,
81         'disk': 80,
82         'n': 1
83     }
84     VM_list.append(large_vm)
85
86
87 Bins=[]
88 for i in range(K):
89
90     bin = {
91         'id':i,
92         'cpu': 48*4, # 4.0 overcommit with cpu
93         'mem': 240000,
94         'disk': 2000,
95         'VMS': [],

```

```

96     }
97     Bins.append(bin)
98
99     #N should be bigger than K
100    random.shuffle(VM_list)
101
102    for i in range(N):
103        index_bin=i%K
104        Bins[index_bin]['VMS'].append(VM_list[i])
105        #print "Bin", Bins[index_bin]
106
107    total_cpu_load=0
108
109
110    for i in range(N):
111        total_cpu_load=total_cpu_load+VM_list[i]['cpu']
112
113
114    average_cpu_load=total_cpu_load*1.0/K
115
116    print "average cpu load", average_cpu_load
117
118
119    Bins_Under_Utilized=[]
120    Bins_Over_Utilized=[]
121
122
123
124    total_load_check=0
125    for i in range(K):
126        if Get_Bin_Load(Bins[i])>average_cpu_load:
127            Bins_Over_Utilized.append(Bins[i])
128        else:
129            Bins_Under_Utilized.append(Bins[i])
130
131    total_load_check=total_load_check+Get_Bin_Load(Bins[i])
132
133
134    for i in range(K):
135        index_largest= int(Get_Index_Largest(Bins[i]))
136        #print "index largest",index_largest
137        #print "type of VM for Bin number ", i, " is ", Bins[i]['VMS'][index_largest]['name']
138
139
140
141    Over_Bin=Bins_Over_Utilized[0]
142    Under_Bin=Bins_Under_Utilized[0]
143
144    Big_VM=Over_Bin['VMS'][Get_Index_Largest(Over_Bin)]

```

```

145
146 Bins_Before=Bins[:]
147
148 Imbalance_before=0
149
150 for bin in Bins:
151     Imbalance_before=Imbalance_before+math.fabs(Get_Bin_Load(bin)-average_cpu_load)
152 print "Average Imbalance before", Imbalance_before*1.0/K
153
154
155 Bins_After=Bins[:]
156
157 min=1000000000000
158
159 NO_more_gain=False
160 while (NO_more_gain==False):
161     NO_more_gain=False
162     for under_bin in Bins_Under_Utilized:
163         min_best_to_balance=min
164         best_over_bin_to_balance=Bins_Over_Utilized[0]
165         distance_to_balance=-1
166         for over_bin in Bins_Over_Utilized:
167             #print "Over bin er: ", over_bin
168             #print over_bin['VMS']
169             largeVM=over_bin['VMS'][Get_Index_Largest(over_bin)]
170
171             #should not get underloaded
172
173             distance_to_balance=average_cpu_load-(Get_Bin_Load(under_bin)+largeVM['cpu'])
174             if (distance_to_balance<=0):
175                 NO_more_gain=True
176             if (distance_to_balance>0):
177                 #print "getting into loop"
178             #difference to the average
179
180
181             if (distance_to_balance<min_best_to_balance):
182                 #print "before it ----"
183                 min_best_to_balance=distance_to_balance
184                 best_over_bin_to_balance= over_bin
185                 #print "inside best", distance_to_balance
186
187         if (distance_to_balance!=min):
188             #print "found a match"
189             Bins_Under_Utilized.remove(under_bin)
190             Bins_Over_Utilized.remove(best_over_bin_to_balance)
191             under_bin['VMS'].append(largeVM)
192             best_over_bin_to_balance['VMS'].pop(Get_Index_Largest(best_over_bin_to_balance))
193             Bins_Under_Utilized.append(under_bin)

```

```
194         Bins_Over_Utilized.append(best_over_bin_to_balance)
195         #NO_more_gain=True
196
197     Bins=Bins_Under_Utilized+Bins_Over_Utilized
198     Imbalance_after=0
199
200     for bin in Bins:
201         Imbalance_after=Imbalance_after+math.fabs(Get_Bin_Load(bin)-average_cpu_load)
202     print "average Imbalance after", Imbalance_after*1.0/K
```

---