A Biomorphic model for automated cloud adaptation

Web server scaling based on cellular differentiation as a case of adaptive behaviour

Gyorgi Stoykov

Master's Thesis Spring 2015

A Biomorphic model for automated cloud adaptation

Gyorgi Stoykov

May 18, 2015

Preface

This work is dedicated to:

- **Tanja Turundzieva** - My loved mother for bringing me to this world and providing her unconditional love. Thank you for being always there for me. I love you.
- **Gligor Stojkov** - My supportive father that has been my strong support at all times. Thank you for all the smart advices that you have given me (even those which I didn't listen to). I love you.
- **Aleksandar & Andona-Marija** - My smaller brother and sister who I love more than everything in this world. I will be there always for you, you can always count on your older brother.
- **Baba Mare & Dedo Done** - My grandmother and grandfather that have taught me the most important things that matter in life. Thank you for raising me up and for teaching me how to ski and ride a bicycle. Sometimes small things matter the most in life.
- **Baba Bote & Dedo Gogo** - My grandmother for being a great professor that taught me the value of academics and a person that had strong influence in my life decisions. My grandfather with whom I share the same name with and who I wish was alive to be proud of his grandson.
- **Vujce Sase** - My uncle who is and always will be my idol. Thank you for teaching me how to swim and how to treat ladies.
- **Teta Zana** - My aunt for giving me life advices that always made me think. Thank you for making all the school projects with me.
- **Gjorgi Stankoski** - My stepfather for being the person you can rely on when it matters.
- **Makso, Dejan, Stefan & Martin** - My best friends which are my family. Thank you for your support and love through all these years.

I want you to know that without you I wouldn't be what I am today

Abstract

Cloud computing is one of the most discussed areas in computer science in the last years. Although there is an extensive amount of research covering this field, the field of bio-inspired cloud computing is underinvestigated when compared to the general research area. This study tries to find answers on how a biomorphic model can be implemented in the cloud in order to achieve adaptive cloud behaviour.

The process of cellular differentiation where cells transform from one type to another, is chosen to be the foundation model for a developed technical model. We define analogies to the cloud where stem cells are blank servers and web servers are cells with a specific function. With a combination of configuration management, version control and cloud deployment systems, an imitation of this biological process is applied in the cloud. The use of automated cloud scaling as a case of adaptive behaviour is the main goal of the research.

Two different approaches have been developed for mapping the biological model to the cloud. The first approach consists of a prototype where the signal detection and node activation is being triggered by using the concept of random generated timers. The second approach is based on the concept of random seeds which are used to coordinate the transformation procedure. The project results were able to adapt the cloud based on current needs, with each prototype having its advantages and disadvantages over the other.

Contents

1 Introduction **1**

2 Background **5**

2.1 Configuration management and deployment technologies . 5

2.1.1 Puppet 5

2.1.2 MLN & ALTO 7

2.1.3 Git - Version Control System 8

2.2 Cloud computing and virtualization 8

2.2.1 Virtualization 9

2.2.2 Cloud types, platforms and providers 12

2.2.3 Cloud computing from a business standpoint 14

2.3 Relevant work in bio-inspired cloud computing 16

2.3.1 Biomorphic characteristics 16

2.3.2 Artificial Immune Systems 16

2.3.3 Artificial Hormone Systems 20

2.3.4 Particle Swarm Optimization 21

2.3.5 Ant Colony Optimization 21

3 Approach **25**

3.1 Objectives 25

3.2 Design stage 27

3.2.1 Biological model 27

3.2.2 Technical model 27

3.2.3 Two algorithms for prototypes 28

3.2.4 Underlying infrastructure 29

3.3 Implementation and experimentation stage 29

3.3.1 Necessary tools to build the model 29

3.3.2 Infrastructure deployment script 30

3.3.3 Prototypes of the two algorithms 31

3.3.4 Testing the correct functioning of the setup 32

3.4 Measurement, analysis and comparison stage 32

3.4.1 Measurement and plotting scripts 33

3.4.2 Experiments 34

3.4.3 Data analysis and comparison 34

4 Result I - Design and models 39
4.1 Biological model 39
4.2 Technical model and Prototype Designs 42
4.2.1 Infrastructure design 44
4.2.2 Networks & Domains 44
4.2.3 Server Deployment and Configuration 46
4.2.4 Deployment script - deploy.py 47
4.2.5 Puppet .pp file generator - createpp.py 49
4.2.6 Signalling and sensor part of prototype 49
4.2.7 Scaling down - reset.py 58
4.2.8 Monitoring and plot script - monitor.py + plot.py 58

5 Result II - Implementation and Experiments 61
5.1 MLN skeleton 61
5.2 Puppet skeleton 63
5.3 Deployment Framework 65
5.4 HAProxy and PHP 66
5.5 Testing and experiments 67

6 Results III - Measurements and Analysis 69
6.1 Timing Algorithm 70
6.1.1 CPU and memory performance in timer interval 1-5 minutes 70
6.1.2 CPU and memory performance in timer interval 5-10 minutes 71
6.2 Random Seed Algorithm 72
6.3 Scaling down 74

7 Discussion 77
7.1 Project evolution 77
7.2 Algorithm comparison and proposed improvements 78
7.2.1 Timing algorithm 79
7.2.2 Random seed algorithm 79
7.2.3 Improvements 80
7.3 Future work 80

8 Conclusion 83

Appendices 91

A Puppet and MLN files 93
A.1 MLN - build.mln 93
A.2 Puppet 95
A.2.1 Functional .pp 95
A.2.2 Stem .pp 96

B Frameworks and configuration files **97**
B.1 deploy.py 97
B.2 createpp.py 99
B.3 signal.py 99
B.4 listen-1.py 100
B.5 listen-2.py 101
B.6 reset.py 102
B.7 monitor.py 104
B.8 haproxy.cfg 105

List of Figures

2.1 Puppet configuration run 6
2.2 Illustration of a full virtualization architecture 10
2.3 Illustration of a partial virtualization architecture 11
2.4 The three cloud service layers and the different services inside of each layer 15
2.5 Layers and processes in the vertebrate immune system . . . 18
2.6 Layers and processes in a network-based artificial immune system 19

3.1 Overview of planned tasks in the design stage 36
3.2 Overview of planned tasks in the implementation stage . . . 37
3.3 Overview of planned tasks in the analysis and comparison stage 38

4.1 A graphical illustration of the process of chemical diffusion or cell signalling 43
4.2 Chemical signals in the environment provoke changes in the gene expressions resulting in cell differentiation 43
4.3 Cellular division plays an important role in the tissue repairing process 44
4.4 OpenStack Network Design 46
4.5 Illustration of a completely automated deployment process from start to finish 47
4.6 Design for the automatic deployment script - deploy.py . . . 48
4.7 Random Scenarios for algorithm one 56
4.8 Random Scenarios for algorithm two 56
4.9 The logic difference in the prototype algorithms 57
4.10 Biological model vs Technical model 57

6.1 CPU and memory performance in a scenario where the first timers are distributed in a time interval between 1 and 5 minutes 71
6.2 CPU and memory performance in a scenario where the first timers are distributed in a time interval between 5 and 10 minutes 72
6.3 CPU and memory performance from the random seed algorithm which spawns one virtual machine instantly, and one in every third minute afterwards 74

6.4 A view of the system's average resource parameters during the process of infrastructure downscaling 75

List of Tables

2.1 Overview and comparison between different cloud solutions 13

4.1 Proposed analogies for the biological and technical terms . . 42
4.2 Elements in the design of the underlying infrastructure . . . 45

Acknowledgements

I would like to thank the following people and organisations of the support that they have given me:

- **Paal E. Engelstad** - For being a true mentor and my support not only in the academic field, but also personally. I greatly appreciate the teaching positions that I have been offered. It is and always will be a pleasure to work together with Paal.
- **Anis Yazidi** - For being my thesis supervisor, being dedicated and available at all times. Without his contribution and feedback, the quality of this thesis wouldn't be on this level.
- **Kyrre Begnum** - For helping me always when I get stuck with his extensive technical knowledge and for teaching me so many things about system administration. I am thankful for the chance to attend the LISA conference and I hope I will attend it again some day in the future.
- **Hårek Haugerud** - For allowing me to teach the security course and all the things I have learned about IT security. I also appreciate the opportunity for the Trondheim trip and I hope I can attend another trip like this in the future.
- **University of Oslo and Oslo college of applied sciences** - for providing me the opportunity to take part of one of the best system and network administration degrees in the world.
- **All of my peer students** - that took the programme together with me. Thank you for being such good friends and making Norway feel as my second home.

Chapter 1

Introduction

A few decades ago, system administrators were taking care of infrastructures that had been constructed in a centralized manner. Depending on the size of the infrastructure, the number of servers varied, but each physical server was only running one operating system. Using some of today's popular data center virtualization technologies, a single hardware node can run more than 1024 virtual machines in parallel [75]. With the current expansion of cloud based computing and its promising business aspect, there is no doubt that cloud computing is the future in IT [45]. Most companies that are following the current trends already have their infrastructure running in the cloud. Some companies that are usually larger in scale build their own private clouds, while other choose to host their infrastructure inside a public cloud using cloud service providers. One of the benefits of using a cloud infrastructure is having a distributed system across different locations to improve performance. Other benefits include scalability, redundancy and decreased operation costs when compared to traditional infrastructures [39].

The increased amount of VMs compared to physical servers made traditional system administration tools inefficient, especially in the case of large infrastructures. A common example for this, would be executing commands through SSH to deploy new software packages and to modify server configuration. In the case of a small company where this has to be done on a couple of machines, completing these tasks is not time consuming. On the other hand, if the company owns hundreds or thousands of servers which are managed by a couple of system administrators, this becomes a problem. One solution to this problem would be to develop scripts that automate these daily tasks. The drawback of using this approach is that when something has to be changed inside of the server configuration or something additionally has to be installed, the automation script has to be continuously modified by the system administrator to be able to complete the different necessary tasks. To overcome these challenges, the use of configuration management systems can be implemented in the infrastructure to manage server configuration. Moreover, configuration management systems do not have the ability to manage tasks like creation of new VMs,

managing hardware resources, shuttuing down or powering on existing VMs, but they can be combined with deployment systems and scripts to achieve an infrastructure where administration tasks are automated and efficient.

The goal of this paper, is to explore how one can design and implement a reactive system situated in the cloud which will demonstrate adaptive behaviour by utilizing different large-scale system administration tools which include MLN [5] as a VM deployment tool, OpenStack [61] as a cloud platform, Git [34] as a software deployment tool and Puppet [42] as a configuration management system. Configuration management will be used to ensure a safe transition at runtime from the current to the new configuration [54] [30]. Adaptive behaviour is a broad term and can be demonstrated within different spheres of informatics, ranging from software development [63] [77] to database [38] [44], network [69] [16] and system [1] administration. In principal, adaptation means that the system reacts and changes itself based on certain parameters. These system changes can be observed in the application code, in the database, in the operating system configuration or in the hardware specifications depending on how the model for adaptation is constructed. This project will use automated scaling as a case of adaptive behaviour. Most projects in the area of cloud auto-scale are based on a centralized model, where a controller is responsible for scaling the virtual machines based on the systems needs[1] [10]. In some cases, traditional centralized management approaches may not be optimal because of the single point of failure, dynamic requirements of applications and the dynamic resources provided in cloud environments. On the other hand, there are few approaches which are built upon a decentralized model, and even fewer are bio-inspired which led to a different approach in this thesis, by designing and building a decentralized model where the nodes have the ability to regulate themselves, thus having an impact on the whole system.

In order to design adaptation models, one could try to search for adaptation examples and sources of inspiration beyond of the area of computer science. One such source which contains countless examples is nature itself. For example, one of the most primitive adaptation mechanisms in nature is cellular differentiation which also falls under the area of bio-inspired computing and is the fundamental inspiration to our approach. In essence, a biomorphic design is just a design extracted from biological systems and processes. This term was first invented by a zoologist named Desmond Morris and it was popularized by Richard Dawkins in one of his most popular books [15]. A characteristic that is common among all biomorphic models is that these systems are all self-organizing and dynamic. In such models, solutions to problems arise as a result from interaction between the individual components in the system, rather than from applying an external mechanism or alghorithm [43]. Computer programs and systems that have a biomorphic design are not controlled by a centralized entity and each element inside of the system is responsible for regulating itself. In the field of bio-inspired computing there are many models directly inspired

by nature, but to the best of our knowlegde biomorphism in a virtualized cloud environment is a model that was not applied before. An alternative term that is often used in the relevant field is biomimetric. Unlike biomorphic, this term refers to mimicking a certain biological behaviour rather than being an analogy. This thesis tries to find answers on how we could build a biologically inspired model into the cloud and what analogy can we make with natural models like cellular differentiation? Cellular differentiation is simply a process in which a less specialized cell develops to possess a more distinct form and function in developmental biology [63]. If we look at the cloud as the organism environment can we look at the servers as individual cells? Different types of cells have different responsibilities inside of the organism and so do servers inside of a system. At the same time, there are cells of the same type form tissues and organs and there are cells without a function that can become anything like stem cells. Just as servers with the same function are combined in clusters, there are servers which have no configuration and are ready to be provisioned and configured to become servers of a specific role. The role of a cell is defined by it's DNA and so is the server role defined by it's configuration files.

Problem Statement *How to design and implement a biomorphic model for achieving adaptive system behaviour in the cloud.*

Chapter 2

Background

This chapter includes introduction to the multiple technologies that will be used in the later chapters of this project, as well as a review of relevant work and a presentation of several concepts in the field of bio-inspired computing.

2.1 Configuration management and deployment technologies

Configuring servers for a specific role and maintaining the state of those servers is a daily system administrators responsibility. In the recent period there has been a significant rise in popularity of virtualization and cloud computing, mainly due to the significant diminished performance overhead of virtualization over the past few years [56]. Working with big amount of servers very often requires automation and system administrators use additional tools in order to automate daily tasks that require human interaction. Automation in big environments allows administrators to save time and to be more efficient. One type of automation tools that are used in big environments are configuration management utilities. System administrators use configuration management utilities mainly in order to apply or change configuration files among servers in the company's infrastructure and verify the correct functioning of services once configuration changes happen. One of the most popular tools used for configuration management is Puppet [42] together with CFEngine [3] and Chef [33]. For the purpose of this project, Puppet will be implemented as a configuration management system within the biomorphic model.

2.1.1 Puppet

In most environments, Puppet works in a client-server architecture which is often the preferred way of building a configuration management system. Alternatively, Puppet can be run as a standalone software, using only the agent software combined with local puppet configuration files. In a client-server model, the server is called a "master" and the clients are

called "agents". This software supports managing agents for both Unix-based platforms including packages for different BSD and Linux operating systems, as well as Windows platforms. On the other hand, operating a puppet master on Windows servers is not supported and no package is available for installation.

Puppet uses its own declarative language which is used to make statements about the state of the configuration. An example of such declaration would be declaring that a certain package should be present on the system or that a service should be running. Unlike traditional custom developed configuration tools which are procedural (i.e. they describe in what order and how things should be done), Puppet requires users to describe the desired state of the system and takes the responsibility of how to achieve this state including the related details. By using a tool called Facter, the Puppet agent reports information to the master. This information includes the operating system that the agent is running, the IP address, hardware information and other useful facts. By knowing such facts, Puppet can make decisions on how to achieve the desired outcome. For example, by knowing the remote operating system, Puppet can choose the appropriate package manager and repositories that are used by that OS to install a package. If the system is running a Red-Hat based system, Puppet will find the appropriate package name and install it using the 'yum' package manager and if the system is running a Ubuntu based OS it will execute the same procedure with using the 'aptitude' package manager instead.

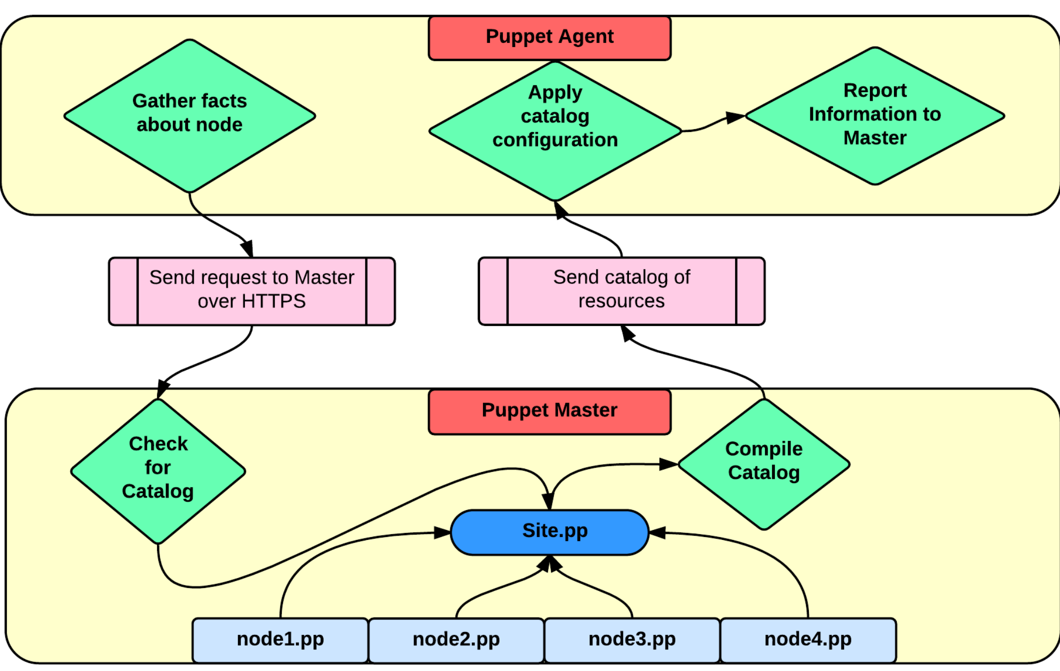

Figure 2.1: Puppet configuration run

Applying of configuration with Puppet is illustrated in Figure 2.1 and the whole process is called a Puppet configuration run. All configuration information is stored on the master side where Puppet runs as a deamon waiting for requests. For each node that runs a puppet agent, there are specific files that contain all configuration information located on the Puppet master and referred to as 'manifests'. These files have a '.pp' extension and can contain classes which have predefined packages and configurations, commands which should be executed on the specific agent, different types of checks for states of services and installed packages, as well as other system administration tasks. All of these files are included into one central file called 'site.pp'. One could also add node configurations directly to this file as well, instead of a separate manifest file. The requests for configuration initiated by the agent are executed automatically every 30 minutes by a deamon and through an encrypted connection using SSL, but can also be requested manually if needed. Each time a request is executed from the agent side, the catalog for that specific node will be served to the client. The agent will apply the configuration from the catalog and report the outcome of it to the master. If the agent has no configuration available in the catalog, or if the configuration has already been applied, Puppet will not do anything. This is a key feature of Puppet and is called idempotency [40].

2.1.2 MLN & ALTO

The deployment of all the servers will be done in HiOA's private cloud named ALTO [58]. The cloud is running OpenStack which is currently one of the most used open-source cloud computing platforms. OpenStack will be responsible for the virtualization layer, together with assigning the storage,memory,computational and networking properties of the provisioned nodes. The cloud provides a Web based GUI from which the hardware specification and deployment of the nodes can be finished. The drawback of using the GUI is that it requires human interaction and it is not suitable for automatic deployment and provisioning of new nodes. To overcome the need of human interaction and automate the deployment process, OpenStack can be managed through MLN.

MLN [5] is a tool used for distributed infrastructure deployment and it provides the capability for managing virtual machines in the cloud. The tool supports cloud VM management for both Amazon EC2 and OpenStack. MLN uses templates with pre-configured parameters, where one could specify a variety of things from hardware specifications to startup commands. The structure of the template is object-oriented alike and it allows creation and use of variables and super-classes. This provides the ability to deploy a large amount of virtual machines in the cloud at once which for the current operalization of this project is not needed, but could be used for future work when expanding the developed frameworks to provide additional functionalities.

2.1.3 Git - Version Control System

When working everyday with codes and data, multiple changes are done as the software is developed. Most often in big environments many people tend to work on the same project at the same time. While developing the software, it is likely that the code is frequently going to have logical errors and will not work as intended. This is why it is of crucial importance to keep track of changes that have been made to the software, so that one could reverse back to a previous software version if needed. With using the main features of Version Control Systems, one could keep track of changes, save software versions, revert back to previous versions and use the VCS to deploy a specific version of software. The most popular VCS is called "Git" and will be used for this project.

Git is a distributed VCS which has the same software version database spread over many nodes providing redundancy and allowing multiple users to work at the same time. As with the above mentioned tools, Git can be easily obtained as well, using the standard package manager for any Linux distribution. There are many platforms offering public Git repositories which are used for storing and managing software codes. The most notable one is named "Github" and will be used in the proposed bio-inspired model to deploy a specific software version once the nodes have been provisioned by MLN and configured by Puppet. Moreover, implementing Git in this project will allow an easy transition to updated or new versions of frameworks in future work based on this project.

2.2 Cloud computing and virtualization

One of the most popular expressions used in relation to today's technology is the phrase 'Cloud Computing', appearing more than 110 million [1] times in websites on the Internet. Although the origin of the term is not clearly defined, the earliest occurrence of this term used in nowadays connotation can be traced back to 1996 in one of Compaq's internal documents [62]. A number of factors contributed to the popularization of this phrase, but it is certain that some of the most important powers that started this popularization were Google who used it for the first time in an industry conference [27] and Amazon by introducing its elastic compute cloud project [66] in 2006.

There are many different definitions on cloud computing, but generally the cloud can be viewed as a pool of combined physical and virtual resources that are available for leasing by cloud users. Using cloud computing provides users with several benefits, some of which include elasticity, scalability, availability, pricing flexibility, reliability and on-demand services. Elasticity and scalability refer to the ability of the resource pool to grow and shrink based on users needs. The integration and combina-

[1] Statistic extracted using Google to search for "cloud computing" on the Internet

tion of countless virtual and physical computing nodes, storage blocks and networks completely removes the single point of failure and offers redundancy and quality of service at all times while these features are available whenever the users need them. Moreover, the users only pay for the time in which they actually use the resources avoiding the need of infrastructure and operations related investments.

One of the most commonly used definitions that approximately describes cloud computing has been coined by Mell and Grance in their paper "Effectively and securely using the cloud computing paradigm" [48] and was later published by the National Institute of Standards and Technology of the USA (NIST), stating that :

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

2.2.1 Virtualization

Virtualization as a term is often being interchangeably used with cloud computing, but although cloud computing is based on virtualization, a differentiation between these two terms has to be made. Virtualization refers to the ability of sharing hardware i.e. computing and storage resources among many different operating systems and it was pioneered by IBM in the mid 1960's. In virtualization, a hardware machine often referred to as "host" provides, controls and allocates the hardware resources that are being shared by virtual machines called "guests" by using a virtual machine manager named hypervisor [14] [31]. Basic server virtualization as a concept can be seen as limited and centralized since only one particular hardware machine provides the computing resources to all VM operating systems, while cloud computing additionally allows the capability of distributing these resources through the network.

Virtualization Types

There are different classifications of virtualization depending on whether a system (server and desktop), infrastructure(network and storage) or software(application) virtualization is being discussed. The most general and commonly used categorization includes three types of virtualization, namely full virtualization, partial virtualization and para-virtualization [9].

Full virtualization

In full virtualization, the host is not running any particular operating system. Instead, the hypervisor is replacing the most important functionalities of the OS. The benefits of using this type of virtualization and hypervisor

which is also called *native* or *Type 1* [80] [11] [28] are demonstrated with increased performance due to the removal of the OS layer between the hardware and the virtual machine. The most common examples for this type of hypervisor include KVM [41] [28], Microsoft Hyper-V [50], Xen [23] and VMWare ESCi [37].

With using this type of virtualization, the result is that the guest operating system is unmodified and unaware of the virtualization and the actual hardware underneath. Additionally, there are security advantages as well when using this type of virtualization compared to others, mainly due to its isolated nature. A graphical illustration of how a full virtualization architecture resembles is presented in Figure 2.2.

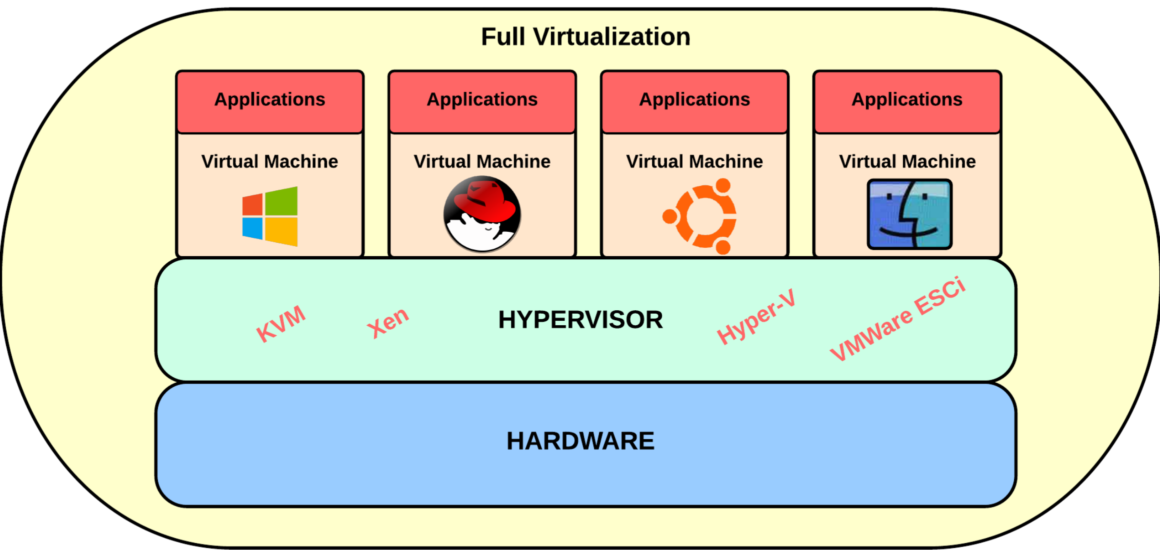

Figure 2.2: Illustration of a full virtualization architecture

Partial virtualization

Partial or frequently called OS-based virtualization is inferior to full virtualization and mostly used on personal computers. On the other hand, sometimes it can be the preferable option over full virtualization due to the fact that the deployment can be carried out in a simple manner and often it is efficient enough for small environments or testing purposes. As the name suggests, it runs on top of an operating system running on the host machine, but it uses a different type of hypervisor. This virtualization and hypervisor type is named *hosted* or *Type 2* and it is running as an application which makes it highly dependent on the processes happening inside the operating system. For example, if there is a process that causes a system corruption or error in the host operating system, it will in most cases have a negative effect on the virtual machines as well. Some applications

using this type of hypervisors include Oracle VirtualBox [57], VMWare Workstation [36] and Microsoft Virtual PC [51]. A similar graphical representation to the previous figure is shown for partial virtualization as well in Figure 2.3.

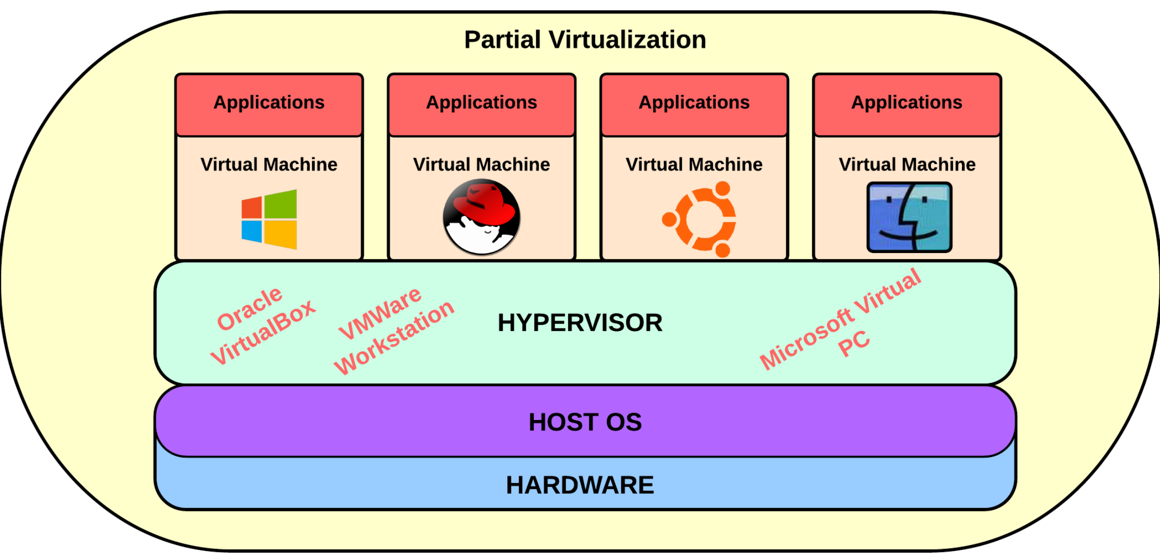

Figure 2.3: Illustration of a partial virtualization architecture

Paravirtualization

Paravirtualization is a concept that has mutual characteristics with both full and partial virtualization. In this type of virtualization, the virtual(guest) machines are running an operating system with a modified kernel. These modifications replace non-virtualizable instructions with hypercalls and make the guest operating system aware that it is virtualized. This improves communication with the hypervisor which in turn results with improved efficiency and performance. Although paravirtualization is running on a host operating system, its performance is superior to partial virtualization and depending on the workload it can be superior to full virtualization as well [74]. The drawback of using this type of virtualization is that OS kernel modifications can include support and maintenance system problems which is not desirable in production environments. Moreover, it is only applicable to modifiable host operating systems which limits the choice of operating systems to open-source options only. An example of a modified OS combined with a hypervisor that allows this type of virtualization by virtualizing the processor and memory with using custom guest OS device drivers is the Xen project [23].

2.2.2 Cloud types, platforms and providers

Generally, there are three kinds of cloud computing that are used when discussing the deployment model of a cloud system [12] [45] [17] [2]. A small overview of the differences between these types, platforms and providers is presented in this subsection.

In ***Public*** clouds, the whole infrastructure is provided by cloud service providers through the Internet and they are also responsible for securing, maintaining and upgrading the hardware part of the cloud infrastructure. Furthermore, cloud service providers allocate individual customer resources and provision their virtual machines while some also offer other application-level service solutions. The biggest and most popular cloud provider companies at the current moment are Amazon with Amazon Web Services [65], VMWare with vCloud [73], Microsoft with Microsoft Azure [49], Google with the Google Cloud Plaform [26] and Rackspace with their managed cloud services [35]. Aside from the general benefits of cloud computing, this kind of cloud usually offers a more affordable and inexpensive infrastructure solution when compared to other cloud types. Due to the flexible hourly pricing and the eliminated maintenance costs, it is usually the preferred option for small to medium sized businesses. Public clouds are also frequently used for teaching purposes in academic environments and in online training centers.

Unlike Public clouds, ***Private*** clouds have an infrastructure that is usually owned, protected and upgraded by the company that uses them. This cloud type is mostly deployed in organizations of large size and in some cases in universities. The advantage of using this type of cloud over the Public cloud is bigger control over the infrastructure and a high level of security, due to the limited cloud usage and access for other businesses and individuals. In universities, building a private cloud data center in universities gives researchers the ability to use the combined node cluster capabilities in order to make calculations and projects that require high computing resources for a low price when compared to Public clouds. When building a private cloud, there are two options to choose from. One option is to choose an open-source cloud platform and the other option is to buy a company cloud platform. A popular commercial option is VMWare vSphere(vCloud Suite) [67] while open-source options include platforms such as OpenStack, Eucaliptus, Apache CloudStack, Nimbus and OpenNebula [78] [76] [79] [64].

In ***Hybrid*** clouds, automation and orchestration is used to combine a private cloud and a public cloud leased from a cloud service provider. This type of cloud provides the advantage of having unlimited on-demand resources delivered by the cloud provider, while still having a secure and controlled local infrastructure. The private part of the hybrid cloud is mostly used for running internal and critical applications,local virtual machines and services. On the other hand, the public part provides resources

for dealing with high and temporary workloads without the need of having to upgrade and invest into the local infrastructure. Although having a hybrid cloud is a preferred solution for most big companies, building and maintaining the cloud is costly, thus making it a less desirable solution from small sized businesses.

A basic overview between available cloud solutions is presented in Table 2.1, where the cloud options are compared based on different criteria. The type of cloud infrastructure is compared along with the option for hybrid support which in most platforms is supported, but in some cases it is being offered with certain limitations. In the selected cloud solutions, the commercial options are usually more limited than the open source solutions when it comes to the choice of the underlying hypervisor. The provisioning type is divided into best-effort and immediate. This classification is discussed in "Virtual infrastructure management in private and hybrid clouds" [67] where the authors Sotomayor et al. refer to best-effort as a provisioning model where resources are provisioned as soon as they are available, while in the immediate type, the resources are provisioned when required or they are not being provisioned at all.

	Amazon AWS	OpenStack	Eucalyptus	Open Nebula	VMWare vCloud
Type	Public	Private	Private	Private	Private
Commercial or Open Source	Commercial	Open Source	Open Source	Open Source	Commercial
Hybrid Cloud Support	No	Yes	Can use Amazon's APIs	Yes	Only if both clouds use vSphere
Hypervisor	Xen	Xen, KVM, VMWare ESXi,Hyper-V	Xen, KVM, VMWare	Xen, KVM, VMWare	VMWare ESXi
Provisioning type	Best-effort	Immediate	Immediate	Immediate	Immediate
Development language	Perl,C++, Java	Python	Java, C	C++, C, Ruby, Java, Bash	Unavailable

Table 2.1: Overview and comparison between different cloud solutions

2.2.3 Cloud computing from a business standpoint

From a customers perspective, different cloud computing models are being offered by the cloud computing industry that refer to the different layers of the cloud architecture. These models are very frequently used in research related to cloud technologies as well, mainly when describing the layer of cloud services that the project is examining or the underlying cloud infrastructure. The cloud services are grouped into three different models [45] [2] [48], namely Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a service (IaaS). Each of these models is dependent and built on top of another model, respectively.

Software as a Service runs on top of the other cloud services and refers to the application part of the model. This cloud service offers end users to run and use cloud-located applications on their local machines without the need of installing the software locally. This is achieved by running the software in the client browser, thus making the application independent of the local platform and infrastructure. Some common examples for SaaS for personal use include software such as Google App and Google Docs, Gmail, Twitter, Facebook or even online browser games. Additionally, SaaS is commonly used on an enterprise level with applications such as NetSuite which is a customer relationship management application and TurboTax which is a tax advisory software. Most of the functionalities that these applications provide are free to use which has been one of the reasons why small and medium sized businesses have started replacing their local applications with cloud ones in order to reduce operating costs.

Platform as a Service is mostly used within information technology companies that work with the development of software or websites. This model offers a development and deployment environment that is being rented out to companies and customers that use it to avoid the cost and complexity of purchasing, building and managing the underlying software and hardware layers. For example, assuming that a person wants to develop a website and a web-based application using some commercial software, he could choose between two available options. The traditional choice would be to purchase hardware along with an operating system and a development platform. This requires a lot of resources and time spent on setting up the environment before starting the actual software development. Instead, the person could rent out a platform which includes these technologies from a PaaS provider, therefore eliminating the time and financial resources spent on setting up the system in the previous case. Microsoft's Azure cloud services, Google's App Engine, Rackspace's Cloud Sites and Amazon's Relational Database Services are a part of the currently most used PaaS solutions.

Infrastructure as a Service is the ground layer in the cloud computing model and refers to the leasing of virtualized storage and computing resources provided by cloud service providers. This part of the model offers

customers the freedom to design and deploy their own infrastructure by providing the hardware part of the cloud. It is the most used and popular model of cloud computing services used by companies of all sizes, since it provides scalable and on-demand resources with a flexible pricing model. As the basis for the above mentioned cloud services, it is the core and most important layer in cloud computing. Amazon's EC2 computing platform along its S3 storage service and Rackspace's Cloud Servers platform are some of the most favored cloud providers used for providing IaaS solutions. To sum up, all of the above mentioned services and their representative layers are illustrated in Figure 2.4.

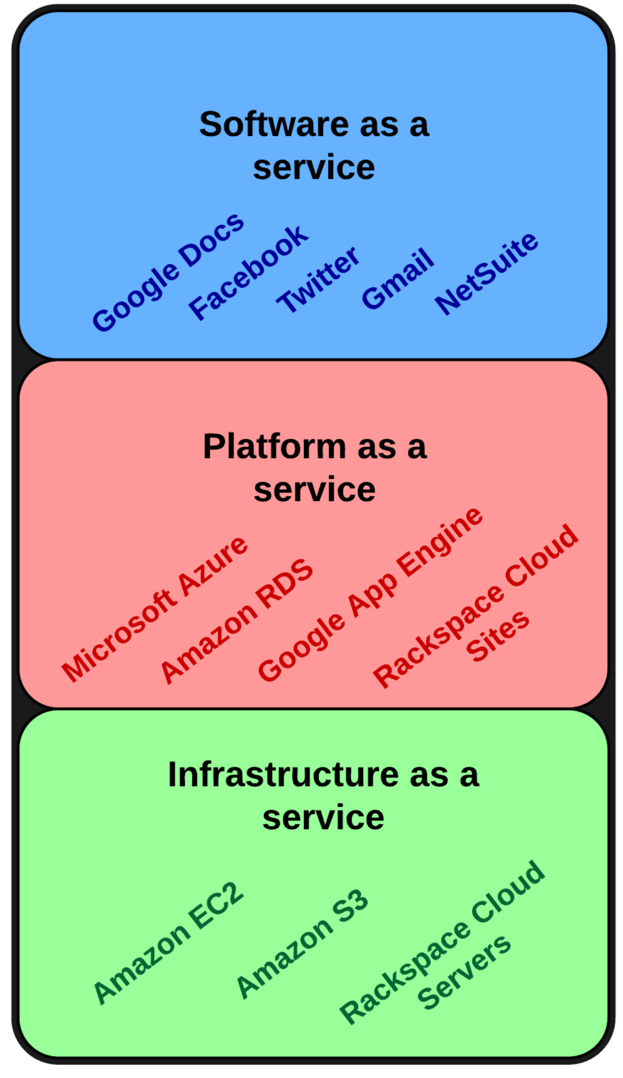

Figure 2.4: The three cloud service layers and the different services inside of each layer

2.3 Relevant work in bio-inspired cloud computing

The extensive research in bio-inspired computing covers a broad area of different fields, some of which include artificial immune systems, artifical hormone systems, genetic algorithms, cellular automata, sensor networks and emergent systems. The most typical approach related to self-organizing distributed systems is based on swarm intelligence [6] and include techniques like ant colony optimization [19]. This section contains review of previously conducted research based on concepts that are included in these areas and related to our model and bio-inspired cloud computing.

2.3.1 Biomorphic characteristics

Several characteristics on behaviour of biomorphic systems are classified in previous research by Wang and Suda [77] some of which include birth and death, adaptation, evolution, interaction based on local information, collective interaction and autonomous action. Evolution and adaptation refers to the ability of the individual components to be able to evolve over time and adapt with changing information or changing the desired tasks. The components are being added and removed(born and die) into a specific group triggered by an event. Additionally, the components react based on local individual information rather than global system information. Autonomous action attributes to individual elements controlling themselves rather than being controlled by a master. The characteristic behaviour of the system is based on interaction of many similar units.

According to a study carried out by IBM [32], self-managed systems and their functions can be classified as:

- 'Self-Optimizing' - systems which monitor and adjust resources.
- 'Self-Healing' - systems which detect and react to disruptions.
- 'Self-Configuring' - systems which adapt to changing environments.
- 'Self-Protecting' - systems which can discover and secure against potential threats.

2.3.2 Artificial Immune Systems

The computer virus as a biological analogy is one of the most common computer related expressions in the everyday language. This expression has gained popularity when Robert T. Morris launched an attack and made the newspaper headlines in 1988 [70]. With the attack he has infected approximately 6000 computer terminals connected to the Internet which at that time was around 10 percent of the total size. In the early ages of the Internet, this was an unencountered problem in the field of computer networks, but in biology this problem was already solved with the use of the immune system. In order to understand how this system been applied in

computer networks and systems, one first has to explore how the immune system works.

There are multiple layers of defense in which the immune system is divided. The first layer has a task of preventing infectious agents named *pathogens* from penetrating into the system. An example for this layer would be the skin, because it is impenetrable by most pathogens and bodily secretions with antibiotic characteristics. The second layer is called the innate immune system and this layer can detect cells that come from the outside and are different from the body cells. The intruders that successfully enter the system are called *antigens*. The immune system reacts so that it triggers an immune response that involves different types of defending cells. The defending cells can either devour or destroy the intruders and there is a special type of these cells called APCs (*antigen presenting cells*) that have the ability to keep samples of the intruder and pass them to the adaptive immune system which serves as the last and strongest layer of defense. One main difference between the cells from the adaptive and innate immune system is that the adaptive cells called *lymphocytes* have targeted attacks. Lymphocytes only attack the antigens that they are compatible with. When the antigen presenting cells pass one of these antigens to the lymphocytes and a match is being found, the lymphocytes will clone and attack the antigen, as well as any cells that have been infected with these antigens. Additionally, the adaptive system cells have the ability to remember the antigen once they have forced it out, so that a more effective defense is being executed in future attacks from the same type of intruder. A basic representation of the whole process inside a vertebrate immune system is displayed in Figure 2.5.

It is interesting that the whole process is completely decentralized and is a result of combined simultaneous action of many independent cells. So, how can this be applied into computer security, specifically in computer networks ?

The first layer of security are the firewalls. They have the task of rejecting attacks originating from the outside network, targeted on the internal network. Moreover, they have the responsibility of denying access to the local networks, thus keeping intruders away from the internal network and the local machines. If an intruder manages to get inside the internal system and starts malicious activity, these actions will be flagged by an intrusion detection system which is the second layer of security. The suspicious activity becomes flagged independently of whether the attack is known or unknown. As a last step, the activity is being examined and if there is a match between a known attack, actions are being taken by the intrusion prevention system which is the last and most sophisticated layer of security. The IPS has the task of blocking or sanitizing the incoming known attack. Additionally, if the attack is unknown, the system adapts and learns how to combat the attack effectively. If the unknown attack has been rejected, the defense mechanism is being memorized for future defenses.

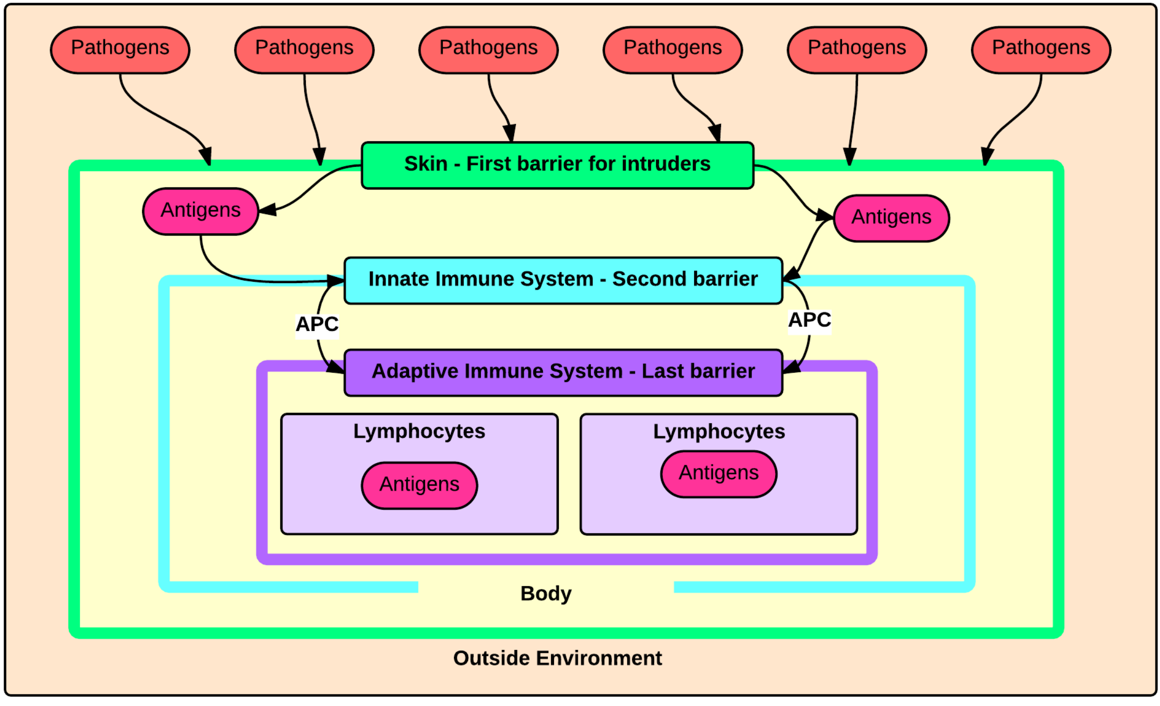

Figure 2.5: Layers and processes in the vertebrate immune system

When applying the concept of AIS, it is often combined with different algorithms so that the functionality and efficiency of the intrusion detection system are expanded. For example, in the paper "Evolution induced secondary immunity: An artificial immune system based intrusion detection system" [13], the authors present a technique for network anomaly detection by combining the concept of AIS with a genetic algorithm to create an unpredictable system with improved detection capability. They use the analogy of a memory cells (a type of lymphocytes) contained in most immune systems with the ability to protect from anomalies. Additionally, unlike the typical approach of having only a single primary immune response that is often presented in most AIS researches, their approach uses a secondary immune response that has been evolved from the previous one.

While most of the intrusion detection systems based on artificial immune systems are built upon network detection, there are also some projects that have host intrusion detection as their main detection mechanism. One such project has been developed by Ou, Wang, and Ou [59] in which an agent-based IDS is proposed. The logic of their system originates from the danger theory of the human immune system. The danger theory was first introduced by Matzinger [46] and suggests that the human immune system only responds when cell damage that would be prevented

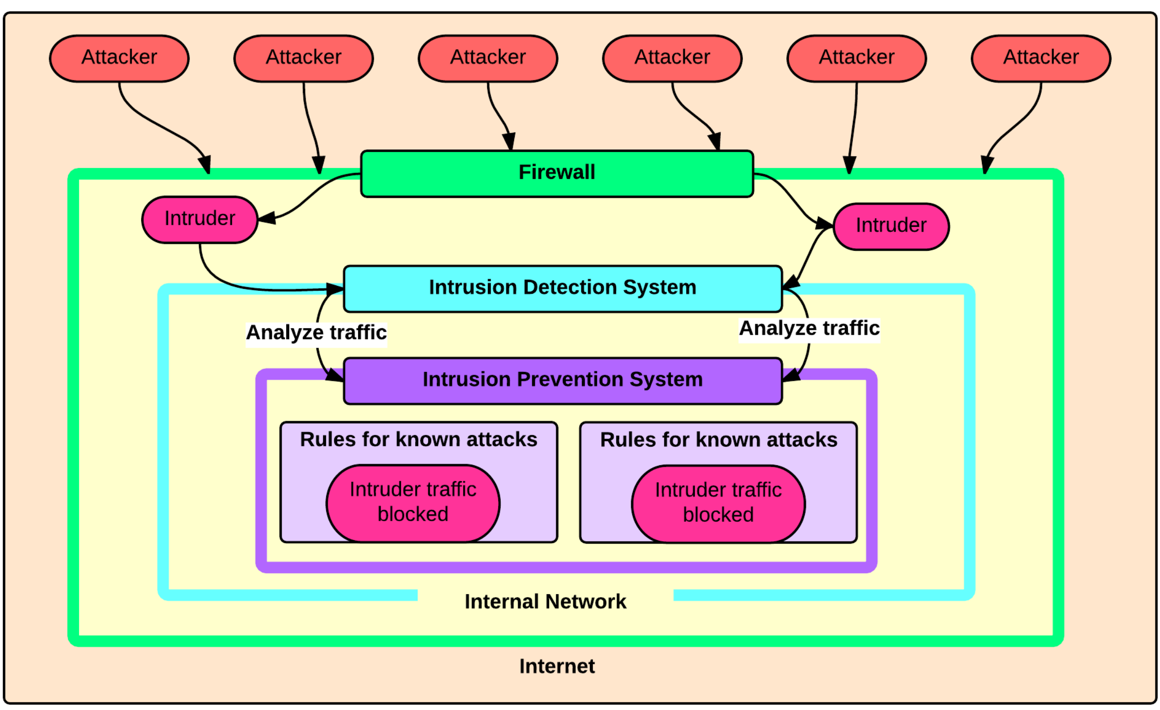

Figure 2.6: Layers and processes in a network-based artificial immune system

otherwise is detected. This immune system theory is fundamentally different from the theory explained in Figure 2.5, because it divides the threat for the body in damaging(dangerous) and non-damaging(safe) rather than foreign(dangerous) and self(safe).

2.3.3 Artificial Hormone Systems

Similar to the artificial immune system, related approaches are based on an artificial hormone system that in most cases is inspired by the human hormone system. There is a variety of known responsibilities that the human hormone system has, some of which include regulating and balancing the processes related to growth, reproduction and digestion inside the body. This is achieved by secreting hormones that flow through the circulatory (blood) system in order to reach all cells inside the body. When a hormone interacts with a cell, it can only trigger specific actions in those cells that have that specific type of cell receptors. In the field of bio-inspired computing, signals that are being sent inside the system are representing the artificial hormones, while the computational nodes are representing the artificial cells. In some AHS based projects [8] [71] [7], the models that are being presented have the mentioned characteristics and functions of self managing systems included in subsection 2.3.1.

A diversity of types of hormones are presented in a number of approaches with each approach having a different goal. Brinkschulte and Pacher divide the hormones that are represented as small messages in three different types depending of their function. In their paper "An Agressive Strategy for an Artificial Hormone System to Minimize the Task Allocation Time" [7], the types of the hormones that are used and combined are the accelerator, suppressor and eager value hormones. In essence, the eager value type is used to determine the suitability of a processing element(artificial cell) for executing a specific task. The suppressor type is used to lower the suitability and is subtracted from the eager value, while the accelerator type is used to prioritize a computing task and is added on the eager value. As the title suggests, in order to improve time allocation a classic and an aggressive strategy is offered, with the aggressive strategy having a significantly better execution time for task allocation.

Trumler, Thiemann, and Ungerer [71] propose a similar approach based on the artificial hormone system with a goal of managing and organizing networked systems. Their approach uses the artificial hormone system as a method for communication between internetworked nodes while the implementation and testing has been done in a simulated Java environment where three resource constraints are used for optimization.

2.3.4 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a method for optimization that has a goal of finding a global best solution by iteratively improving a set of local solutions in regard to a specific value that is being calculated by the fitness function [20]. This method can be used for optimizing multiple parameters and has a population-based approach with the population being referred to as the 'swarm' and the candidate solutions being referred to as 'particles'. Highly nonlinear and mixed integer optimization problems can be solved effectively using PSO, but the downside is that it comes with an expensive computational cost [29]. On the other hand, PSO is commonly used in low computational applications due to its simplicity of implementation [60].

A modified version of a typical particle swarm optimization algorithm is presented in "A Novel Bio-Inspired Load Balancing of Virtualmachines in Cloud Environment" [18]. The authors Domanal, Guddeti, et al. constructed the algorithm for consistently allocating incoming jobs on available VMs in the cloud. As part of their experiment, they have compared their algorithm with two other scheduling algorithms, namely a modified throttled load balancing algorithm and a round robin load balancing algorithm. Their modified PSO algorithms has shown better results in both allocation of the load on available VMs, as well as in average response times.

Pandey et al. [60] created a PSO-based scheduling heuristic in order to achieve minimization of cost for executing application workflows in the cloud. They have chosen to use the Amazon Web Service [65] platform to execute their experiments. By comparing their PSO method to a BRS or "Best Resource Selection" method, they explored that using PSO would lead to reducing the total cost by three times. Moreover, they have used particle swarm optimization in a load balancer that distributes computing tasks to available resources.

2.3.5 Ant Colony Optimization

ACO or Ant Colony Optimization [22] [24] [12] [21] is a concept that is being applied to optimization related issues, most commonly to graph problems. In nature, ant formed groups are searching for the shortest option to their source of food while walking randomly through a path that is usually represented in science by a graph. On their way, they are placing information in the form of pheromone, so that they can later use it to find the optimal direction to their destination in other words, their food source. Some of the goals of the developed algorithms contained in the research include improving routing information, minimizing resource wastage, minimizing power consumption, balancing load and workload consolidation.

In the field of computer networks, the first two pioneering researches [69] [16] on ACO based routing have been developed at the same time. In "Ants and reinforcement learning: A case study in routing in dynamic net-

works," the authors Subramanian, Druschel, and Chen present a so called 'uniform ant' algorithm in which ants travel through all paths in the nest with equal probability. The second research by Di Caro and Dorigo provided an algorithm called AntNet where they introduce two different types of ants, namely forward and backward ants. The forward ant follows the pheromone information and records the time from the source to the destination node. Once it reaches it destination, it starts moving back to the source and it becomes a backward ant. The backward ant updates the pheromone tables on the way to the source, by increasing them for a time-based amount specified by the time it took the forward ant to reach to destination. The AntNet algorithm influenced a number of other relevant works and has been the basis for other algorithms in the ACO based routing field [47].

Minimizing power consumption and resource wastage is the goal of a research conducted by Gao et al. [24] in their paper "A multi-objective ant colony system algorithm for virtual machine placement in cloud computing." They propose an ant colony for VM allocation in the cloud with two limitations, namely CPU and memory while the pheromone is used as an analogy to relate to the preference for the server that is going to host the virtual machine.

In their paper "Ant system for service deployment in private and public clouds" [12], Csorba, Meling, and Heegaard investigate how applying of ant colony optimization can be used for balancing load by mapping virtual machines onto physical machines. Different services are grouped in different nests meaning that each service actually starts from a separate nest. Ants are leaving these nests to find other possible locations in which services could be hosted. The algorithm continues to work even after a possible mapping is found, because a better mapping could be found which would lead to a potential reconfiguration of the system.

Feller, Rilling, and Morin [22] present a nature inspired approach in "Energy-aware ant colony based workload placement in clouds" using ant colony optimization for solving dynamic workload placements in energy-aware Infrastructure-as-a-Service(IaaS) cloud computing environments. The optimization algorithms is applied to optimize the multi-dimensional bin-packing where the physical machines act as bins and workloads act as items that should be packed. After the ants have finished building a solution, the pheromone trail is being updated. The update is being done based on MIN-MAX Ant System (MMAS) [68] where only an ant with the best solution can deposit pheromone in the path.

Additional relevant work and inspiration is contained in the Anthill project [4] which was created by the University of Bologna in cooperation with a researcher from the Norwegian University of Science and Technology. In this project, they have developed a middle-ware for peer-to-peer systems which uses a similar ant colony approach. The network of

peer nodes are referred to as nests, while the society of adaptive agents is referred to as ants that travel across the nest to complete user requests. Anthill could be used to create a variety of classes of peer-to-peer services that manifest resilience, adaptation and self-organizing properties. Two frameworks have emerged from this project, namely Messor [53] which is used as a load-balancing application and BISON [52] which is a conceptual framework and an acronym for "Biology Inspired techniques for Self Organization in dynamic Networks".

Chapter 3

Approach

This chapter will outline the methodology and the necessary steps that will be taken in order to find answers to the given problem statement : ***How to design and implement a biomorphic model for achieving adaptive system behaviour in the cloud.*** When developing a model based on a biological representation it is important that clearly summarized goals are presented, so that it is easy for other researchers with a related problem to understand and build upon the presented design and prototype model. The approach consists of several phases and attempts to describe how the technologies and design will be combined and used in this project. An overview of the key features and aspects of the project that are discussed in detail contained in this chapter are as follows:

- Exploring the fundamental biological model
- Design of two technical models based on the biological model
- Elements in the testing cloud environment
- Software deployment tools and scripting languages
- Performance constraints
- Experimenting with the proposed models
- Implementing and comparing the proposed models
- Expected results

3.1 Objectives

As briefly described by the problem statement and the introduction section, this study attempts to explore how a system that has reactive and adaptive characteristics could be implemented in the cloud. The system will be based on a biological model and will combine the concepts and technologies described in the background section to demonstrate self-management, self-optimization, self-protection and scalability characteristics in an automated and efficient manner. Traditional cloud scaling methods are based

on a centralized system where decisions are made at a single node based on gathered information across the system.

In this approach, an alternative solution will be provided, which offers a model where nodes are unaware of other nodes and make decisions autonomously while still having an impact on the whole system. In such a model, there will be no central controlling element and therefore a single point of failure will be avoided. Furthermore, the assumption is that this kind of system will react and adapt faster since the decisions are made only based on local data, unlike centralized systems where data used for decision making originates from multiple nodes. Alternative approaches could use a different biological model as a foundation for the logic of the technical model. There are many appliances of biological concepts in cloud computing and informatics in general, some of which are discussed in section 2.3.

One of the goals of this project is to provide a practical implementation of the presented prototypes so that it is easy for companies or individuals to spawn an identical environment. Additionally, it will be designed so that it is easily modifiable and adjustable to the company's or individuals needs and infrastructure. This is a challenging task because of the complexity of the environment and requires a utilization of different tools. Some of these tools include configuration management systems, version control systems, cloud platforms, scripting and infrastructure deployment tools discussed in sections 2.1 and 2.2. There is a variety of potential benefits of using such a system, but the main benefit would be efficiency due to the scaling nature of the system which will in turn results to reduced financial and operating costs. Furthermore, the model of the system and it's prototypes are based on complete automation requiring as little human interaction as possible. Automation reduces some of common tasks that system administrators do, therefore allowing them to focus their efforts on improving some other parts of the infrastructure, rather that having to expand or shrink the system based on specific requirements.

To find a solution to our problem, the project work is divided into the following three stages :

I Design stage.

II Implementation and experimentation stage.

III Measurement, analysis and comparison stage.

All three stages contain both theoretical and practical tasks. The first two consist primarily of practical work, while the last stage will have a more analytical approach. Since this is an exploratory and investigative thesis, the focus will be more on the practical work rather than deep analyzes, such as comparing the proposed results with other existing models for cloud scaling. The proposed technologies and a detailed description of the planned tasks in each of these stages is presented in the following subsections.

3.2 Design stage

The design stage which is also illustrated in Figure 3.1 is the first part of the project work and is organized into the following tasks :

1. Explore and choose a biological model.
2. Create an adaptive technical model for scaling of individual nodes, along with a text description, pseudocode and illustrations.
3. Create two algorithms for communication and actions between individual elements inside the system.
4. Design of the underlying technical infrastructure.

3.2.1 Biological model

The first task in the design stage will consist of examining available biological models that could be used to demonstrate adaptive cloud behaviour. This process should also outline the reasons that trigger certain actions in the biological system. Additionally, a detailed view on how individual elements communicate between each other and how the individual changes affect the biological system would contribute to designing a better technical model.

An important factor when making the decision about what biological system to use, is that previous research in the bio-inspired system administration area has not yet implemented a similar approach in the cloud. After a suitable model has been found, it has to be explored in detail so that analogies between the biological and technical part can be created. It is crucial that the analogies related to the individual biological terms are well explained and connected to the technological terms, since most people interested in such a study would come from a technical or computing background.

3.2.2 Technical model

The design of a technical model has a goal of providing an overview of all functionalities and characteristics of the system. The logic and structure of the model will be described with text, images and pseudocode to provide the reader with a good understanding of the system without having to dive into and understand the developed code.

Since the focus of this thesis will be centered on scaling as a case of an adaptive behaviour, the workload will be distributed throughout the system on all available nodes. The type of servers used to handle the incoming workload will be web servers, while the distribution will be done by a load balancing software located on a separate server. As previously mentioned, the technical model should have self-optimizing characteristics

which refers to monitoring and adjusting available resources. This will be achieved through developing a framework that will constantly gather resource information so that the system is aware of the availability of its local resources at all times. Additionally, the self-configuring and self-protecting characteristics of the system refer to the automated node configuration and the individual security from external threats, respectively.

3.2.3 Two algorithms for prototypes

Two similar algorithms will be designed as part of the technical model which will have the responsibility of controlling the whole system. From these algorithms, two prototypes will be developed and responsible for defining the communication methods between individual elements, as well as defining what kind of processes trigger certain actions inside the system. Both algorithms will be based on the biological model, but they will be different in the way that they trigger their respective scaling methods. The first algorithm, should trigger several methods that will expand or shrink the system based on current needs, each time expanding the system by at least one element. On the other hand, the other algorithm should expand or shrink the system by a block of multiple elements. Both models for these algorithms should consider the same three constraints on all the individual nodes:

- CPU and system load
- RAM memory usage
- Web server response time

These parameters should be measured at specific time intervals which should be identical in the two models so that at a later stage, the effectiveness of both models can be compared in a efficient way. Once these values are detected that they are above or below a certain threshold, numerous different mechanisms need to be activated that will trigger actions such as:

- Deciding if an expansion or shrinkage is required.
- Starting a communication process.
- Determining a specific size and time for expansion or shrinkage.
- Adding or removing elements from the system.
- Confirming that the system is in an adapted state.

3.2.4 Underlying infrastructure

Once the technical model is designed, a plan for the structure of the underlying infrastructure has to be created. This plan should include the amount of virtual machines that is going to be used inside of the system when the system is being set up. Additionally, the computing capabilities, memory and storage sizes have to be defined so that a large amount of VMs can be spawned at once without occupying a huge amount of cloud resources.

Depending on the created model, a number of networks will have to be created, so that elements of a different type are separated from each other. The idea is that different elements are connected in different domains and networks, thus belonging to a dissimilar environment. Furthermore, a limitation with a maximum amount of servers for each network has to be specified because the resources of the cloud are limited and being shared by other students and professors for research purposes. With this approach the traffic is going to be simulated and forwarded to the system, and the system model will be designed to be adaptive,scaling and responsive, hence no unused resources will be held by idle virtual machines.

3.3 Implementation and experimentation stage

The implementation and experimentation stage displayed in Figure 3.2 consists of multiple tasks, some of which include:

- Determining the necessary tools to build the models.
- Building an infrastructure deployment script.
- Implementing prototypes of the two algorithms.
- Testing the correct functioning of the setup.

3.3.1 Necessary tools to build the model

To build a testing and experimentation environment, different tools have to be combined in order to achieve the desired objectives, thus finding an answer to our problem statement. The procedure of picking the desired tools and the appropriate software includes a review of available technologies along with their functionalities and limitations.

To be able to automate the tasks inside of the infrastructure, a scripting language has to be used. While it is possible to automate installation and configuration tasks inside the script, it is not efficient and requires a lot of changes inside of the code in case there is a change in the design. Therefore, the use of a configuration management system has to be also included inside of the technical model which will manage system configuration on the fly. As previously mentioned, one of the objectives of this project is to

be able to reproduce the same environment for future research or practical purposes. This requires the use of an infrastructure deployment tool, which will spawn a number of virtual machines, configure networks and storage information, and manage access control inside of the OpenStack platform. Once the underlying infrastructure is set up, the software deployment of the prototypes and their updates can be managed by a version control system. In the end, for testing and obtaining results, the system has to be stressed with simulated web traffic which can by done by using a benchmarking tool.

A detailed description of the used technologies and their characteristics is contained in section 2.1 and 2.2. After reviewing the available tools that could be implemented, the following tools have been chosen:

- **MLN** - as an infrastructure deployment tool
- **Puppet** - as a configuration management system
- **Git** - as a version control system and software deployment tool
- **Python** - as a scripting language
- **Httperf** - as a web traffic simulation and benchmarking tool

3.3.2 Infrastructure deployment script

This part of the implementation phase consists of building a framework which will be based on the infrastructure design that has been created in the previous stage. With the use of the deployment framework, the tasks of configuring the network interfaces, assigning the computing and storage resources, installing a specific operating system type and defining a hostname for each individual VM should be automated. Moreover, the framework should be dynamic, meaning that it should be able to accept multiple user-specified parameters before starting the process of VM spawning.

To achieve this, the framework will be built using Python and it will utilize MLN in order to manage OpenStack. Additionally, the framework will use, create and modify MLN skeletons based on entered parameters. The proposed user parameters that will be offered by the deployment framework are:
(**red** is for mandatory parameters, **blue** is optional parameters)

- **project** - This parameter refers to the name of the project. This is a mandatory parameter, since there needs to be a distinction between different testing environments.
- **VM-number** - This parameters specifies the total amount of VMs that will be spawned. This is a mandatory parameter because different MLN templates are created based on this parameter.

- **hostname** - This parameter is optional and refers to the basic hostname that will be used for all virtual machines. The hostname naming convention consist of "basic name + vm number". If the hostname is not specified, the basis of the hostname will be the same as the project name.

- **keypair** - This parameter is used to deploy the virtual machines along with specific security keys that are placed in OpenStack. By default, the keypair from the MLN skeleton will be used, thus making it an optional parameter. The idea of using this option is useful when rebuilding the VMs with a different keypair in case of denied access.

- **OS image** - This parameter refers to the operating system image that will be installed on the virtual machines. An ubuntu 12.04 image will be configured in the MLN skeleton by default. If the user wishes to use another image instead, he can specify the image, therefore making the deployment support any Linux based operating system.

- **OpenStack flavor** - This parameter specifies the size of the computing and storage resources provided for each virtual machine by OpenStack. Since the number of the VMs can vary, the smallest size for the VM will be configured in the MLN skeleton. This will help avoiding over-saturating the cloud performances. This parameter is left optional, since the deployment framework can be used to deploy the VMs inside of a cloud with larger performances.

It would be useful, if the user of the framework is provided with a help menu along with a explanation of the available parameters. When the deployment framework is run, the whole infrastructure with all of the virtual machines based on the entered parameters should be simultaneously deployed inside of OpenStack. The logic of the framework and its structure will be presented in the results section with the help of illustrations, text and pseudocode. A reader who wants to understand the code of the framework in depth can refer to the Appendix section B for code details.

3.3.3 Prototypes of the two algorithms

Two identical underlying infrastructures will be deployed using the previously developed deployment framework. Each control and communication prototype will be implemented in a separate environment. The frameworks will be based on the two algorithms modeled in the design section. The goal of providing different environments is that it would provide as accurate results as possible since each prototype will be run in its isolated environment of the same type. Note that these prototypes are playing a fundamental role in the proposed adaptable system. They are the ones who will be responsible for all reactions inside the system.

Once the prototypes have been developed, the representative puppet server for that environment will be configured so that all virtual machines can obtain their appropriate software and configuration from the puppet master. This means that when the whole model has been set up and configured, the implementation and execution of the prototype scripts will be fully automated, requiring no human interaction at all. At this point of the project, the testing of the setup can begin.

3.3.4 Testing the correct functioning of the setup

Before running the actual experiments and gathering the data, it is of crucial importance that the all of the frameworks work as intended. This requires continuous testing of all the methods inside the scripts as well as their provided outcomes and actions. All of the tests will be executed in a real life environment rather than being simulated, so that one can be sure that same outcome is provided in the experiment as well. Additionally, the whole process of deployment and configuration along with the correct functioning of all virtual machines has to be confirmed.

This testing period is planned to be spread over the course of three to four weeks, so that the chance for any bugs inside the software is minimized. Although there are tests that might develop during the implementation and design stage, some tests can be planned in advance. Some of the planned tests are summed up in the following list:

- Test T_1 - Deploy a range of virtual machines (Testing the provisioning process).
- Test T_2 - Run connection tests through the local network (Testing of the underlying infrastructure).
- Test T_3 - Inspect the location of all installed packages along with the custom scripts (Testing of the version control system).
- Test T_4 - Analyze the specific configuration on each virtual machine (Testing the configuration management system).
- Test T_5 - Testing the control and communication mechanisms of the developed prototypes.
- Test T_6 - Testing the measurement and analysis tools.

3.4 Measurement, analysis and comparison stage

The last stage of work for this project starts with gathering empirical data that will be obtained by running a set of experiments. The measurement of several key parameters will be accomplished by using a separately developed extraction script. In the end, the gathered data will be plotted into several charts for an illustrated analysis and comparison. Some of the

these arranged tasks that will be done in this last stage of the project work are shown in Figure 3.3 and described below:

- Building and implementing a data measurement script.
- Building and implementing a data plotting script.
- Executing the experiments.
- Analyzing the obtained data and figures.
- Comparing the data between the two models.

3.4.1 Measurement and plotting scripts

Beside confirming and testing the correct functioning of the created system, this is one of the most significant tasks in this project. Having a continuous overview of all the important parameters will provide a foundation for analysis and building a strong conclusion. Since the implemented prototypes will make their decisions based on local data, it will be interesting to observe how these actions will affect the system as a whole.

Measurment script

The measurement script will run on each node and will fetch the same parameters on which the local elements base their decisions, namely the CPU and system load, memory usage and server response times. Consequently, it will calculate the average of the last ten web request response times. Once this is finished, the script should store each of the extracted parameters along with a timestamp inside of a local file. In the case of having less than 10 requests, the response time average will be calculated based on the available response time entries. The local file will have the name of the hostname for that node with a .log extension. This process will be repeated in a 30 seconds interval until canceled. To assure that the data is accurate across the system, the NTP Linux package will be implemented on all nodes to synchronize the time, thus having an identical timestamp. The proposed output structure for the parameters delimited by a space in the log file are as follows:

Proposed parameter file structure

```
CPU-SYSTEM-LOAD AVAILABLE-MEMORY RESPONSE-TIME-AVG
```

Example parameter file output

```
0.8 80% 1.23
```

Plotting script

Once the parameter files are created, these files will be transfered to a central location where additional calculations will be made. The plotting script will extract data from all available nodes across the system and combine the data to calculate the averages on a global level, in other words the system averages. The same procedure will be done for both of the proposed models. The calculated parameters will then be plotted on various charts for comparison between the two prototypes.

3.4.2 Experiments

The experimenting in this project will be done simultaneously in both environments. All virtual machines will run an Ubuntu 12.04 OS, but in the case some other researcher wants to redo the experiment, he is encouraged to use any other Linux operating system since the configuration management system will take care of the package installations and configuration of the servers. As previously described, the system will be stressed using simulated traffic. The traffic will be generated by a benchmarking tool called Httperf [1]. The software that will be used as a webserver to handle all traffic will be Apache [2]. It is worth noting that the same amount of traffic will be generated and targeted to both environments, so that the results are as precise as possible. At the time in which the traffic generations starts, the measurement scripts will be activated as well. Once the experiment finishes, the data plotting and analysis will follow.

3.4.3 Data analysis and comparison

The last tasks in this phase will consist of analysing and comparing the obtained data sets. The analysis part will try to look inside the data to find answers concerning the proposed models, specifically which models offer better performances and efficiency. There are two performance factors which will be considered when analyzing the data.

The first factor that will be considered is how the parameters adjust when expanding the system infrastructure. For example, in the case of an overloaded system and a system expansion, the data analysis would provide answers to which system performs better, considering the memory or CPU parameters. The superior system would be the one which adapts faster and lowers the load off the virtual machines more quickly, resulting in better performance.

The second factor that will be considered is how efficient the prototype and the system is overall. Since the amount of web traffic load would differ in size, the system might spawn additional server instances that might not be needed in the near future. Unused machines will be turned off by the

[1] httperf - http://www.hpl.hp.com/research/linux/httperf/
[2] Apache - http://www.apache.org/

shrinking process, but this process often takes some time to finish. To sum up, the time the virtual machines run in an idle state contributes to the inefficiency of the system.

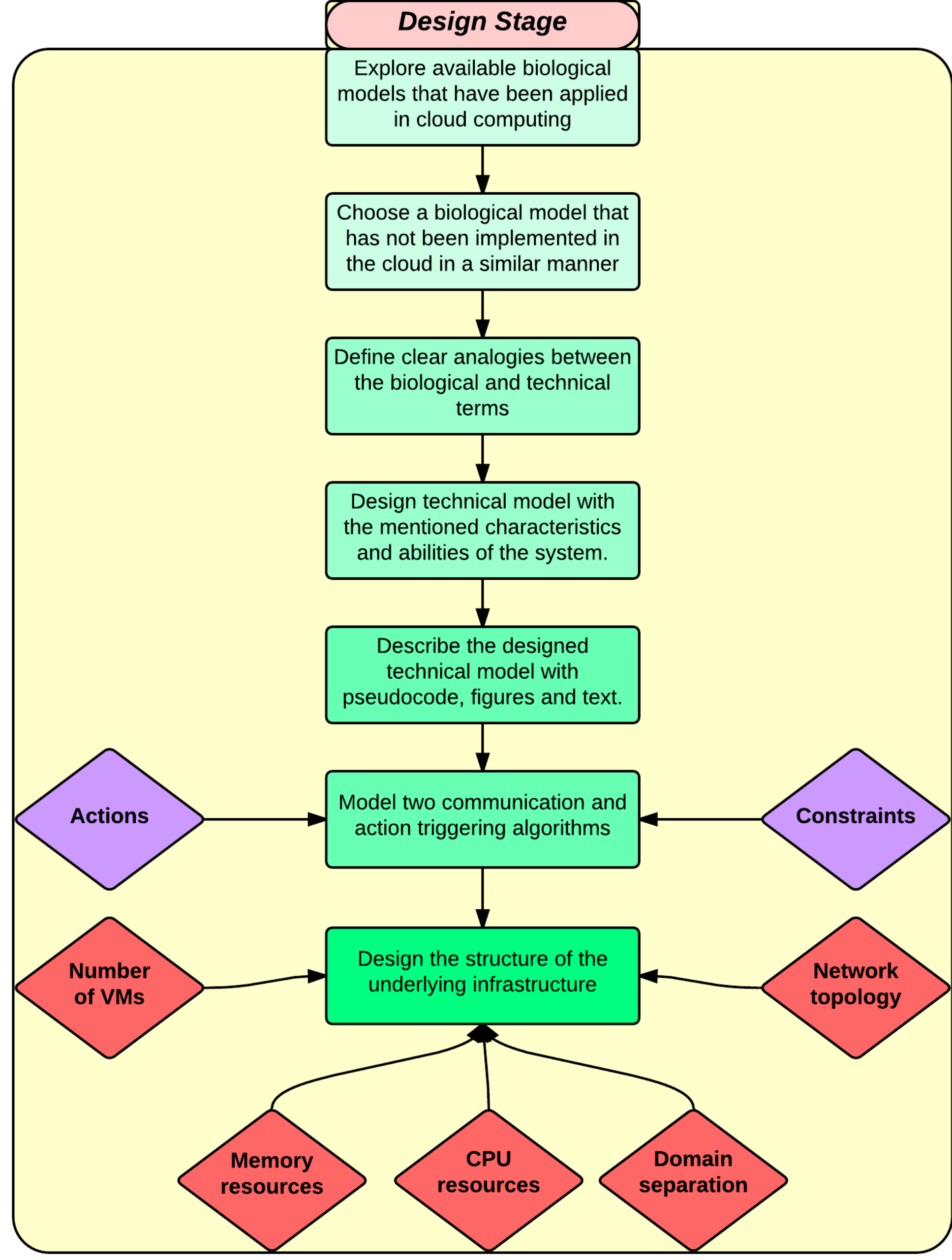

Figure 3.1: Overview of planned tasks in the design stage

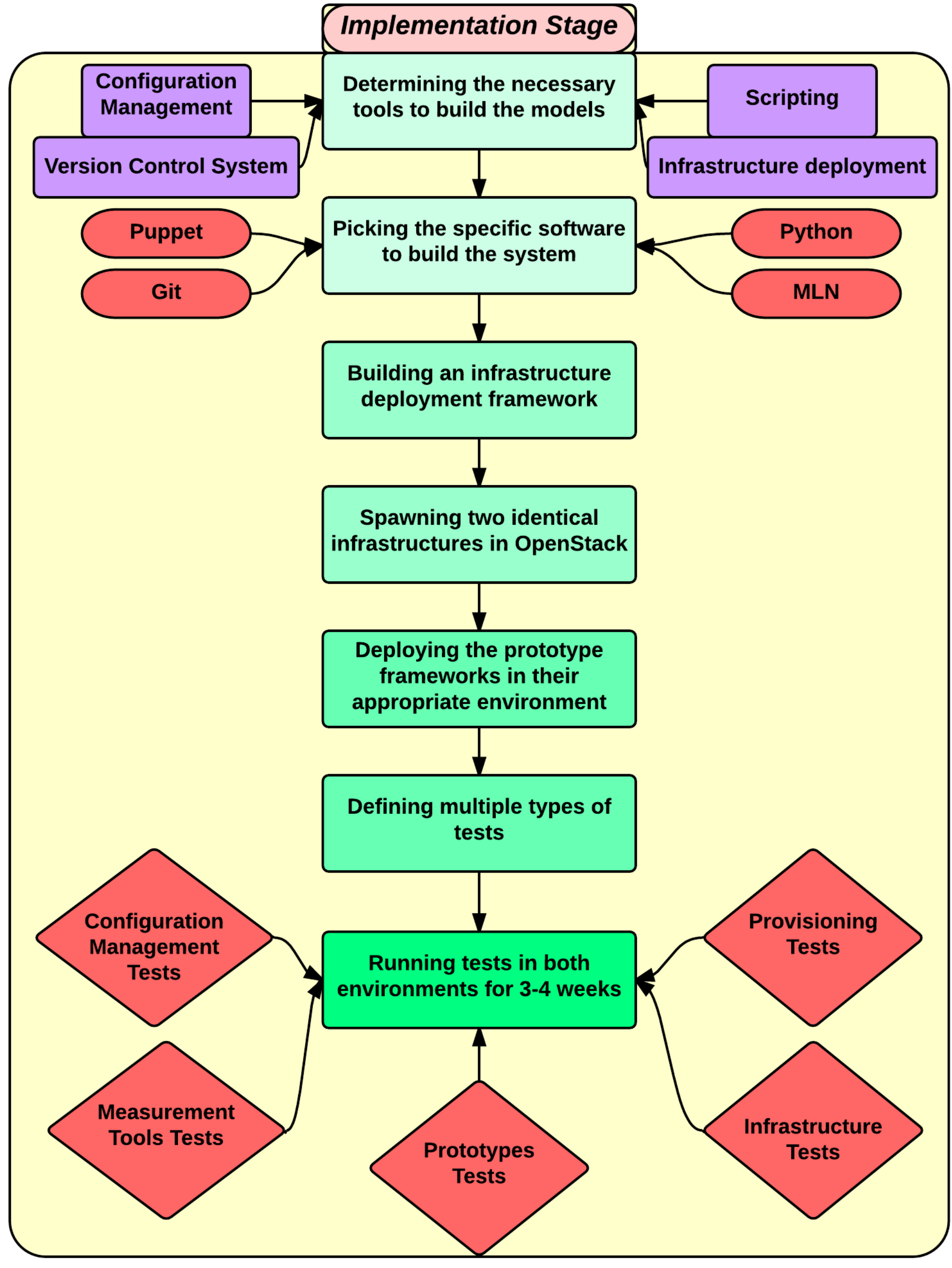

Figure 3.2: Overview of planned tasks in the implementation stage

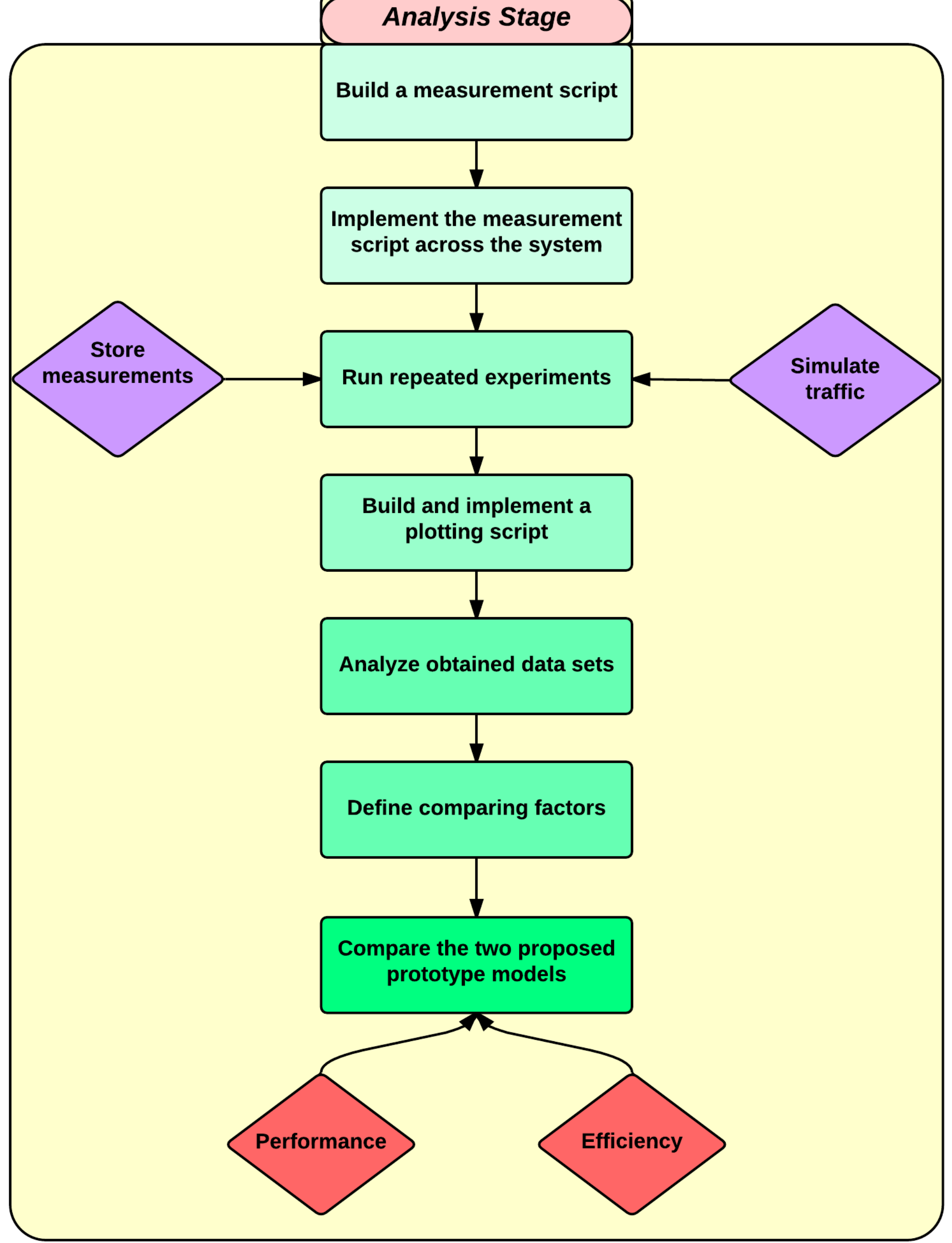

Figure 3.3: Overview of planned tasks in the analysis and comparison stage

Chapter 4

Result I - Design and models

This chapter contains the results from the completed tasks described in the design stage of the approach section. The subsections below present a fundamental description of the biological model and a detailed description of the technical models and prototypes. Since the project work is done in a working environment rather than being simulated, the design for the underlying infrastructure that was used is presented in the end. The results for the implementation and analysis stage are contained in the Result chapters 2 & 3 respectively.

4.1 Biological model

After reviewing multiple biological concepts that have been applied in the area of bio-inspired computing, the process of ***cellular differentiation*** has been chosen to be the foundation for the proposed technical model of the system. To the best of our knowledge, this is a concept that has not been implemented for achieving adaptive cloud behaviour by now. An interested reader is encouraged to read the review of other related bio-inspired approaches discussed in section 2.3.

In essence, the process of cellular differentiation is a process in which the cell transforms from one type to another. During this process, the cell changes its shape, metabolic activity, signal responsiveness and size by modifying its gene expressions, but not the DNA itself. The result of the transformation is usually a newly transformed cell which is able to perform more specific tasks inside the system. The cell types that are able to differentiate into any type of cells are often referred to as stem cells [55] while the area of the biology which explores the gene expression modifications is called epigenetics [72]. Please note that stem cells are not the only type of cells that can differentiate, but rather one cell type that has this ability.

In humans, the stem cells are known to be found in a variety of different tissues and organs, some of which include the skin, heart, blood vessels, liver, bone marrow, brain, teeth and other locations. They are often seen as

undifferentiated cells with no specific function, except for their main role of repairing and maintaining the tissue where they are located. Even then, the process of cellular differentiation and cellular division needs to occur in order for the tissue to be repaired. Although these two processes are different from each other, they are still closely related.

A basic representation of the process of cellular differentiation is described in three phases below. Understanding some of the following terms will provide an easier understanding of the whole process that is also illustrated in figures 4.1 and 4.2.

- **Cellular gene expressions** - The genes inside of cells can be activated or deactivated based on the presence of a particular chemical concentration. The result of this process is the production of specific chemicals inside the cell. The cells state is defined by its gene expressions, which can be either expressed(activated) or repressed(deactivated). Different gene expressions have a different effect on the cell functioning. For example, when a gene responsible for tumor suppressing is deactivated, it can lead to uncontrolled cell division and growth which results in some type of cancer.

- **Cellular signaling** - By producing different proteins, cells diffuse chemicals into the nearby environment. The chemicals in the environment might have an effect on the cell function and structure, because the cells have the ability to sense chemicals on their walls and the close habitat. Whether any changes are triggered inside the cell depend on multiple factors like the quantity, quality and the type of the chemical signal. One could see this as a process where cells are using their ability of chemical diffusion into the surrounding environment as a way of communicating with other cells. Although cell signaling is a complex topic, fundamentally it can be divided into two basic types of communication used in bio-inspired computing [25]. The first type of communication which is often referred to as diffusion, occurs when cells spread the chemicals over a certain range in any direction. The second type of communication is targeted communication and it is often referred to as emission. This type consists of cells communicating directly to other cells through their cellular membranes.

- **Cellular division** - The process of cellular division is a process in which the cell replicates and divides itself into two daughter cells. In most cases the daughter cells will have the same characteristics and function. The case of division where daughter cells do not share the same characteristics is called asymmetric division. The different behaviour of the daughter cells is mainly stimulated by different chemicals on the cellular membranes, as well as the difference in their chemical compositions [25].

Phase 1

This is the first phase of cellular differentiation where the cells emit signals. These signals can be triggered based on a diversity of reasons. For example in the case of a damaged tissue, cells that are near the tissue or part of it will send replicate signals into the environment which will result in an enhanced process of cellular division. Another example can be observed when cells become aware of the lack of space in the close environment, consequently beginning to send inhibit signals to nearby cells that are used to stop cell growth. When a cell death happens, chemical diffusion is being ceased, therefore inducing nearby cells to regenerate the dead cell. Cellular signaling is illustrated in Figure 4.1 where cells of a specific function inside the tissue diffuse chemical signals that reach the stem cell environment.

Phase 2

Now that the chemical signals are spread throughout the environment, nearby stem cells will sense these chemicals on their membranes (illustrated on the left side in Figure 4.2). Some of these stem cells will enter the process of cellular diffusion, while others will retain their current structure and function. If the chemical persists in the environment, more cells will start their transformation process. On the other hand, if the chemical is absent the remaining stem cells will not transform anymore.

Phase 3

The stem cells will carry out the process of cellular differentiation by altering the gene expressions using the received chemical signals. Based on the required cell type, specific genes will be either expressed or repressed, thus providing a different protein and chemical output. Once this process is finished, the result will be modified gene expressions and a changed cell type (illustrated on the right side in Figure 4.2). In example, the stem cell would have transformed into a muscle cell, bone cell or any other type of cell with a specific function. The last step for the tissue repairing process which is a common characteristic in self-healing systems is cellular division. This process, also shown in Figure 4.3 begins after the process of cellular differentiation has finished.

Bio-Tech relationships

The main question that remains to be answered at this point is how this process of cellular differentiation can be applied to our technical model and what analogies could we make. Looking at the mutual characteristics of the individual elements, it is observable that the cells and servers attributes inside of the particular system are very much alike. For example, cells that have an identical responsibility and function inside the biological system are very likely to have the same inside structure. Additionally, such cells with a same role usually tend to be grouped inside a system in order to finish the necessary tasks. In large scale environment, servers practically

do the same. A server which is running only an operating system without any additional software has no role inside the system until it has been provisioned, configured and activated. Furthermore, servers of the same type (i.e. webservers) often have an identical configuration and are grouped together, which can be seen in techniques such as server clustering and load balancing. The communication between cells is accomplished by sending chemicals in the environment, while between servers it is accomplished through the network in the form of TCP or UDP packets. The following table contains some analogies that are going to be used when referring to the system features throughout the remaining chapters:

Biological term	**Technical term**
Cellular Structure and Gene Expressions	Server OS and Configuration
Cellular Signaling	TCP Signals
Cells of a Specific Function	Webservers and other server types.
Stem Cells	Blank Servers
Cell Tissues	Server Clusters
Cellular Differentiation	Server Configuration and Function change
Cellular Environment	Virtual Networks and Domains

Table 4.1: Proposed analogies for the biological and technical terms

4.2 Technical model and Prototype Designs

After exploring the biological system and defining analogies, the next step would be to create a cloud environment similar to the cellular environment presented in the previous section. The following element properties have been of high importance, when defining the underlying infrastructure model and the prototype:

- Servers should act autonomously, no central controller is needed.
- Decisions are based on local resource parameters.
- Configuration change happens on the fly without any human interaction.
- Servers should inform the environment about necessary actions with the use of network signals.
- Each type of server belongs in its own environment(network and domain).

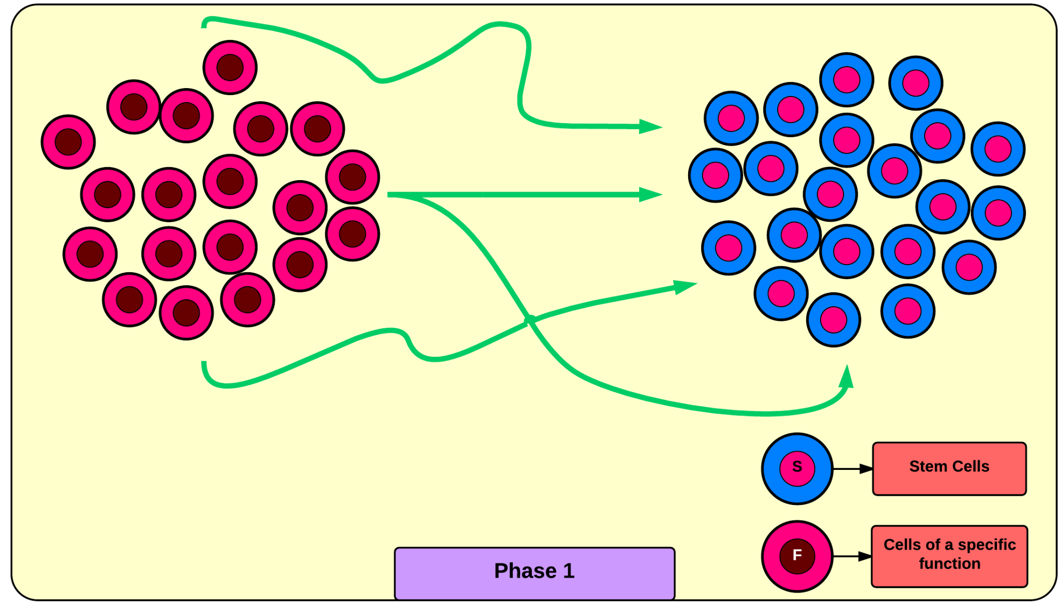

Figure 4.1: A graphical illustration of the process of chemical diffusion or cell signalling

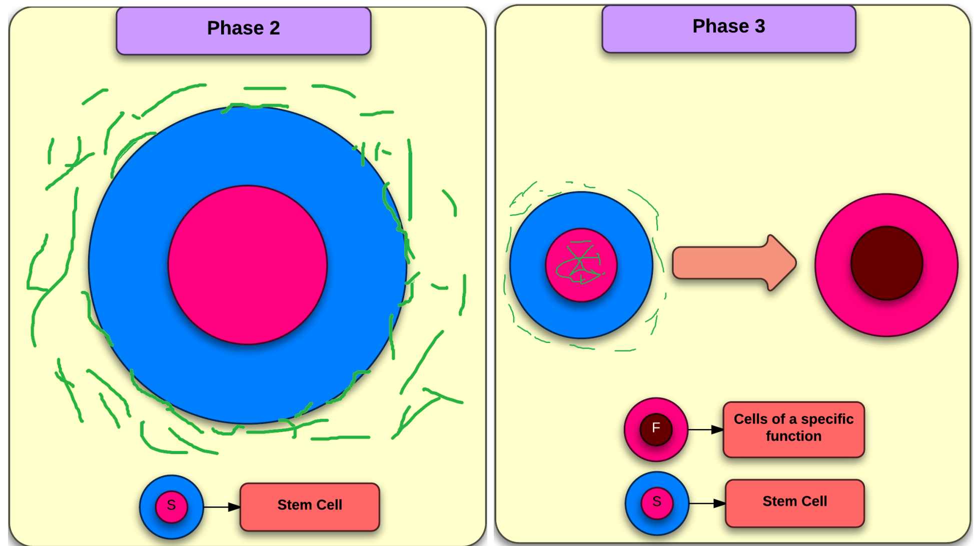

Figure 4.2: Chemical signals in the environment provoke changes in the gene expressions resulting in cell differentiation

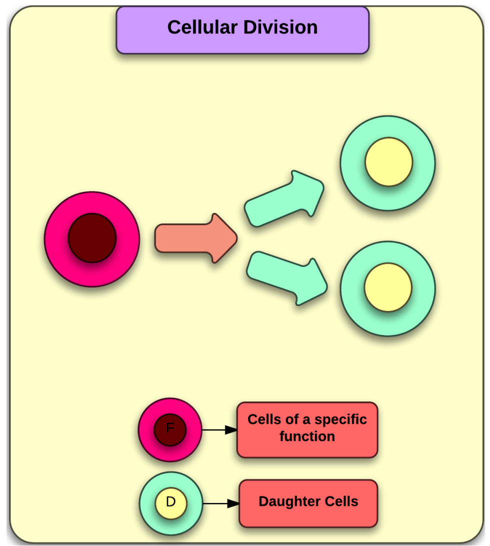

Figure 4.3: Cellular division plays an important role in the tissue repairing process

4.2.1 Infrastructure design

Based on the above mentioned properties, several different types of servers have been defined, but only two types will actually imitate cells. The first type of server will be blank servers that are running only an operating system, which will act as stem cells. These stem cells should transform into cells of a particular function, which in this case is web servers. How this process is accomplished and how decisions are made is described into the prototypes subsection. This subsection will cover the infrastructure layout, the deployment design along with the required servers that have been set up together with the networks and domains. The initial infrastructure design that was used for deploying a completely new infrastructure inside of OpenStack consists of the elements which are contained within Table 4.2.

4.2.2 Networks & Domains

While it is important to have connectivity between all nodes inside the system, it is also important to isolate some of the traffic in different environments. For this purpose, two networks and domains have been created inside of OpenStack. Each network represents the surrounding habitat around the cell or the server. Servers will communicate through the different networks depending on which type of server they need to reach, and for what purpose. Both networks will have a router that will act as a gateway for Internet access to all nodes connected to these networks. A graphical representation of the networks along with the subnet configuration is shown in Figure 4.4. Some servers need to be connected to both networks, i.e. the servers represented in the figure refer to the puppet master and the MLN/Monitoring server. The webservers and the blank servers will be connected to their respective networks.

Element	Role	Description
HA Proxy	Load Balancer	Distributes traffic to available webservers
Puppet Master	Configuration Management	Initial configuration and continuous reconfiguration
MLN Server	Provisioning	Node creation, OS installation and Network Interface configuration
Gateway Server	Connectivity	Used as a gateway for accessing all internal servers.
Storage Server	Monitoring	Used to gather all data measurements accross the system
Networks	Connectivity	Two Virtual Networks have been created in OpenStack.
Domains	Isolation	Each domain has its own network in a separated isolated environment.
Apache Web servers	Web Service	Servers that simulate functional cells and serve web traffic
Blank servers	Scaling	Servers that simulate stem cells, ready for automated configuration to fill systems needs.

Table 4.2: Elements in the design of the underlying infrastructure

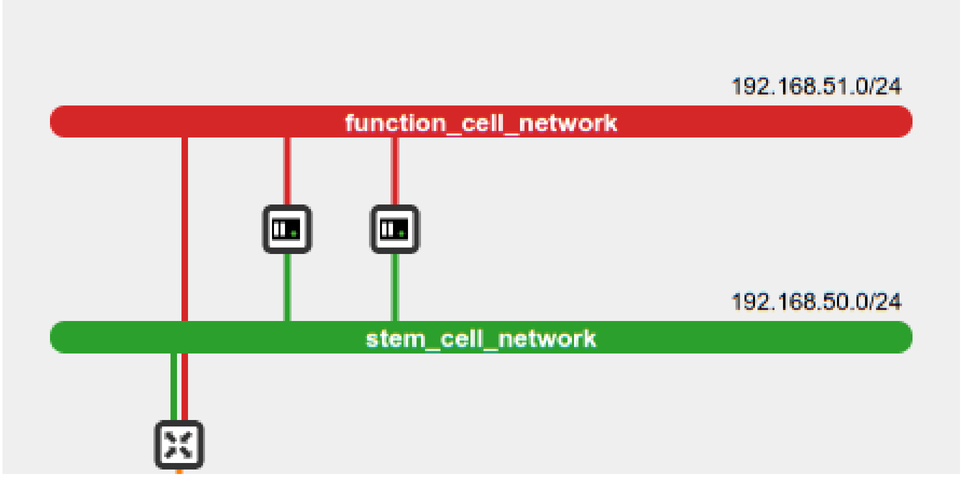

Figure 4.4: OpenStack Network Design

4.2.3 Server Deployment and Configuration

In this design, the servers are going to be deployed both manually and automatically, depending on the type of servers. The initial setup requires that a working instance of a Puppet master, Monitoring server, as well as a MLN server are configured before automatically provisioning and deploying the rest of the infrastructure. All servers including the cell type servers will be running a Ubuntu 12.04 operating system along with the prerequisite software packages. Once the system is in place, the automatic deployment of the infrastructure will be carried out.

The complete process of deploying a server from start to finish is illustrated in Figure 4.5. Before the server is created and provisioned, MLN and Puppet skeletons have to be built. These skeletons will be used by two scripts called ***'deploy.py'*** and ***'createpp.py'*** which will generate the actual files by combining the user defined parameters and the skeletons. The MLN skeleton contains information about the OpenStack resource configuration i.e. memory size, network devices and CPU cores. Additionally, in this file the configuration for security keypairs and post-installation scripts is also contained. There will be two Puppet skeleton that will be different for different types of servers. The puppet .pp files contain information about necessary packages, services and files that have to be applied on all nodes of that particular type. Both the MLN and Puppet skeletons along with a detailed description are presented in the implementation results section.

The next process requires execution of MLN commands which will be implemented in the MLN deployment script. MLN will use the generated file to communicate with the OpenStack interface and create the servers. Once the servers have been defined, the operating system will be installed and when this has finished, the servers will be powered on.

Post-installation commands will be executed after the booting has finished to update the software repository, install the puppet package and to configure the domain resolution for the puppet master. The puppet agent software will then contact the puppet master and request the particular configuration catalog for that host. Resolving the information from this catalog will results in configuration being applied, packages being installed and services being started. As a last step, the communication and control prototype frameworks will be started on the appropriate server types.

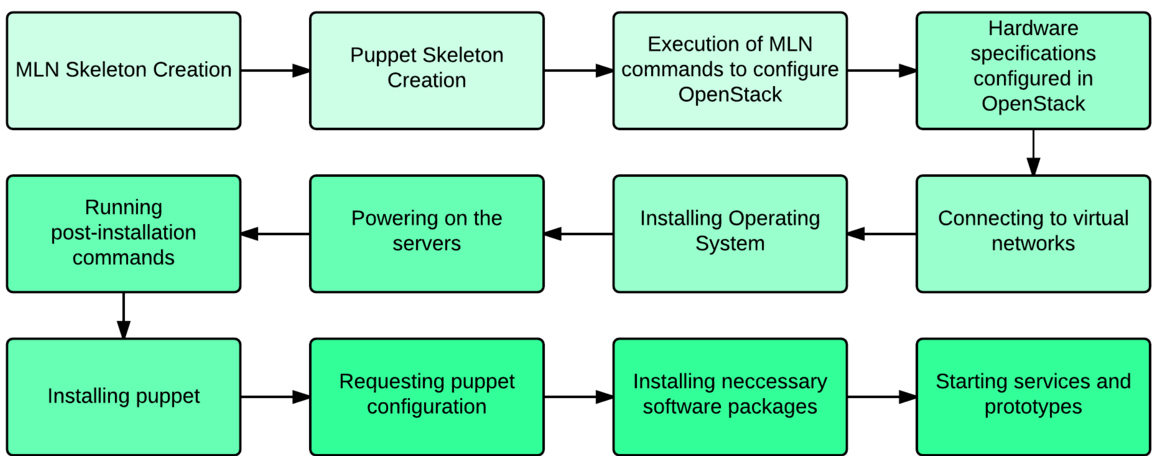

Figure 4.5: Illustration of a completely automated deployment process from start to finish

4.2.4 Deployment script - deploy.py

As outlined in section 3.3.2, the deployment scripts will use user entered parameters to accomplish the automatic deployment process described above. This script will have three main methods, namely CreateHosts, BuildTemplate and StartDeployment.

The first(CreateHosts) method will take the user input for the number of stem cells and functional cells and it will generate a .mln template file with server definitions. This file is not a skeleton file, but rather a temporary file that will be created, only for the hostname and individual server definitions. MLN templates have a object oriented structure, where a node class can inherit the main variables and characteristics from a main class. The main class and all the other configuration is located in the mentioned skeleton file. The second (BuildTemplate) method, will create the main .mln building file by combining the previously generated server file and a modified version of the skeleton. This main .mln file will be stored locally and used by the third function(StartDeployment) which will execute MLN build and start commands based on this template. A preliminary design of the three functions, their output and the logic explained with pseudocode is shown in the code block below and in Figure 4.6.

deploy.py pseudocode

```
1  Definition CreateHosts ( takes # stemcells,# funccells,hostname,projectname):
2     total_cells = stemcells + funccells
3     if hostname is empty:
4        hostname = projectname
5     else:
6        hostname = hostname (make it lowercase)
7
8     Create cells.mln file
9     Open cells file for input
10    Write stem cell info into cells.mln
11    Write functional cell info into cells.mln
12    Close and save cells.mln
13
14 Definition BuildTemplate(takes project,keypair,flavor and os_image)
15    open and read skeleton.mln
16    For items in skeleton:
17       If project > replace project
18       If keypair > replace keypair
19       If flavor > replace flavor
20       If os_image > replace os_image
21    write new info to build.mln
22    close skeleton.mln
23    append cells.mln to build.mln
24
25 Definition StartDeployment():
26    call bash command (mln build -f build.mln)
27    call bash command (mln start -f build.mln)
```

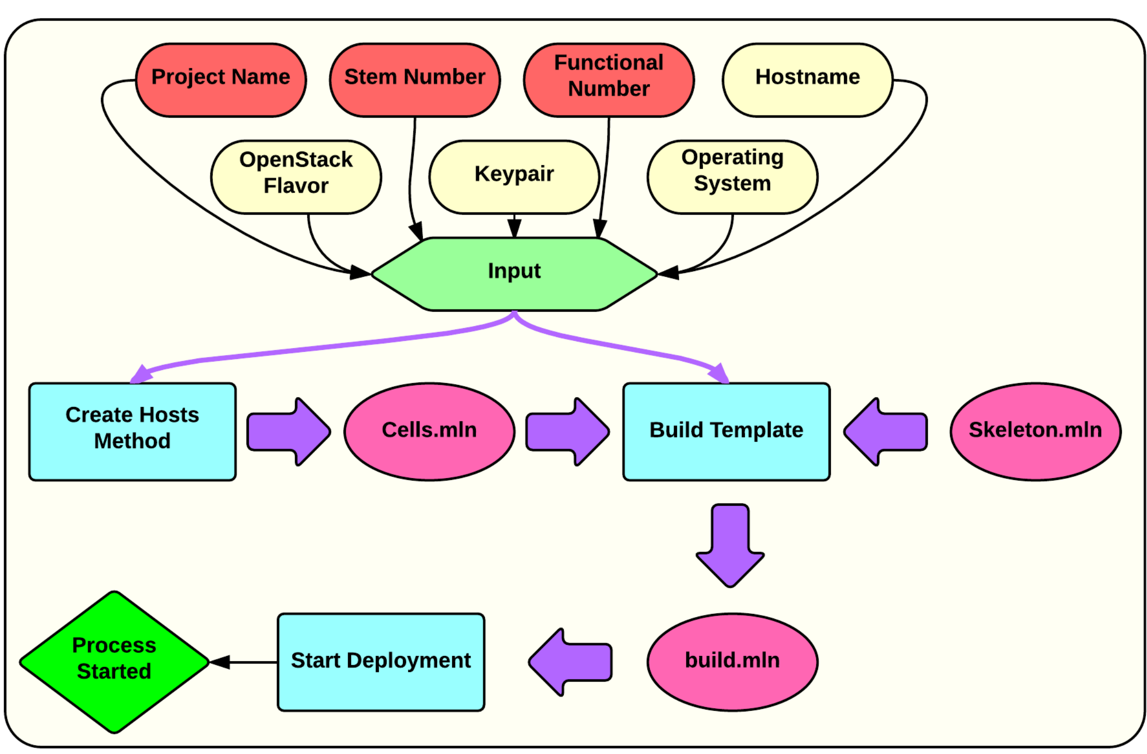

Figure 4.6: Design for the automatic deployment script - deploy.py

4.2.5 Puppet .pp file generator - createpp.py

The second script that is designed as part of the deployment and provisioning process will be used to generate the puppet .pp configuration files for both the stem cell servers and the functional type of servers. This is a small script with two identical methods, which based on the skeleton for each server type, creates the .pp files for all servers with that function inside the system. The methods inside of the script will utilize the sed Linux tool in order to replace each occurrence of the hostname and additional information from the skeleton contents to create a distinctive file for each host. This script will not take any user entered parameters and the proposed method names are createStemPP and createFuncPP. In case there is need for configuration changes, this can be done directly inside the skeleton .pp files, since creating a script which will automatically create and modify configurations is out of the scope of this study. Assuming that there is a need for ten .pp files of each type, the logic of the script is described with the pseudocode below:

createpp.py pseudocode

```
1  import linux os module
2
3  Definition createStemPP():
4     for server in range(1,11):
5        "cat" skeleton file and pipe that output to sed
6         use "sed" to replace each occurance of hostname+1_stem to hostname+1_stem
7         Save that filename with the name
8  hostname_stem+1.pp
9
10
11 Definition createFuncPP():
12    for server in range(1,11):
13       "cat" skeleton file and pipe that output to sed
14        use "sed" to replace each occurance of hostname_func to hostname+1_func
15        Save that filename with the name
16 hostname+1_func.pp
17
```

4.2.6 Signalling and sensor part of prototype

As part of the details defined in the puppet configuration files, the installation and execution of git commands is completed. Depending on the type of the cell or server, either the sensor and reset framework or the signalling framework will be installed and activated. This procedure will also be fully automated and will not require any human entered parameters. On the servers that are functional cells or web servers, a signalling framework called 'signal.py' will be deployed. The stem cell servers will have a framework called 'listen.py', which will act as a sensor for incoming environmental signals. The focus of the complete prototype model is to imitate the cell behaviour as much as possible.

Signalling part - 'signal.py'

The signalling framework consists of several different methods. The basic separation of methods within this framework can be done between the

monitoring methods and the decision making methods. The three monitoring methods will also be implemented in the monitoring scripts and will be called FetchMemory, FetchSystemLoad and FetchResponseTime. The decision making methods will be SendSignal and Actions.

The FetchMemory function will utilize the Linux utility called free to obtain memory information. This program fetches data from the running memory information located in "/proc/meminfo". The method will calculate the data based on the output of the free utility to obtain percentage results for the free and used memory on the local system. The parameter that is of importance in this function is to obtain the available free memory in percentage on the local host.

The FetchSystemLoad function will obtain it's information from "/proc/loadavg" where system CPU average is continuously calculated over the course of 1, 5 and 15 minutes. The output inside of this file is represented by decimal numbers, which can be transferred to percentages as well. These decimal integers can go above 1.0 (i.e. 1.5), meaning that the CPU has been completely loaded and there are processes in the waiting queue waiting to be finished. The .5 or the additional 50% in this number, refer to the amount of CPU load that this waiting queue of processes requires. This function will fetch all three averages, but only the average for the course of the last minute will be of importance.

The last parameter will be obtained from a custom created log file. As part of the puppet configuration for functional cells, a modified Apache Webserver version will log the response time for each request in seconds and microseconds. The logs will be stored under '/var/log/apache2/response_time.log'. The first column inside the log points to the amount of seconds, while the second column points to the amount of microseconds in which that request has been completed. If the server webpage is a plain HTML index, the number of requests has to be drastically increased, meaning that it will be really hard to stress the servers. This is because the amount of requests will flood the virtual networks and cause a network problem rather than stress the individual servers. An additional web page algorithm has to be implemented on each of the webservers so that the CPU is stressed. This will be done in order to avoid over-saturating the load balancer as well as the networks with the generated web requests. This method will calculate the average of the last ten requests and report the calculated average as output. In the case when there are less than 10 requests, the method will calculate the average for the amount of request times that are available. These three methods are outlined by the pseudocode below:

signal.py Monitoring Functions pseudocode

```
1  import linux os module
2
3  Definition FetchMemory():
4      memory_in_use = run free | grep Mem | use the awk expression to calculate the used memory
5      | convert to percentages
```

```
      free_memory = run free | grep Mem | use the awk expression to calculate the free memory
      | convert to percentages

      free_memory = output_from_os_firstcommand_module.readlines()
      memory_in_use = output_from_os_secondcommand_module.readlines()
      return free_memory

   Definition FetchSystemLoad():
      data = run linux command "cat /proc/loadavg"
      data = data.readlines()
      avg_values = strip the data for spaces
      avg_values = split the data into individual columns with space being the delimiter

      avg_1_min = avg_values[0]
      avg_5_min = avg_values[1]
      avg_15_min = avg_values[2]
      return avg_1_min

   Definition FetchResponseTime():
      data = run linux command "tail -n 10 /var/log/apache2/response_time.log"
      data = data.readlines()
      request_size = length of output

      for line in output:
         seconds.append(integer(first element in the split output))
         microseconds.append(integer(first element in the split output))

         seconds_avg = sum(seconds)/request_size
         microseconds_avg = sum(microseconds)/request_size
         return seconds_avg
```

The two remaining methods, will be responsible for making a decision on whether to send a signal to the network containing of stem cells. The method called Actions will be responsible for calling the previously presented methods to obtain the resource information. Based on the available information it will decide if the SendSignal method will be executed, or if no further actions will be taken. The requirement that needs to be met in order for a signal to be sent is as follows:

- System load for the last minute should be above 85
- Available free memory should be below 15
- Response times should exceed 1 second.

Only if these three prerequisites have been met, the SendSignal method will be called. This method scans the stem cell network using nmap for all servers that have an open port 10000. Every server that is an available stem cell will have TCP port 10000 open. Using the netcat utility an activation signal is sent through this port number. The Actions function will be constantly run in a while loop in an interval of 10 seconds or 6 times per minute.

signal.py Decision methods pseudocode

```
   Definition Actions(util,memory,response_time):
      if util >= 0.85 and memory <=15 and response_time > 1:
         SendSignal()

   Definition SendSignal():
      Scan stem cell network
```

```
7      Sort available stem cell servers in list
8      for server in list:
9         Open TCP socket on 10000
10        Send signal to all servers in the stem cell network
11        Close communication
12
13   Definition Main():
14     while 1:
15        Actions(FetchSystemLoad(),FetchMemory(),FetchResponseTime())
16        sleep 10
17   Main()
```

Sensor part - listen.py

This framework plays a crucial difference between the two prototype models and the expected results. In the two identical infrastructures that will be deployed to obtain the results, a different version of the listen.py will be deployed. There are some methods which will be used in both versions, but the main difference is the way that the cell or server gets activated or changes its type. This difference and the logic of both algorithms is illustrated in 4.9.

First of all, since the interface IP address will vary all the time across different nodes, a network that detects the correct IP and interface has to be created. This method will implement the use of the Linux ifconfig command to obtain the interface configuration, the information will then be transfered to the Linux tools cut and awk to obtain only the IP part of the output. Then, a regex expression will be run on the obtained two IP addresses, so that the the sensor gets activated only for the IP on the interface from the stem cell network.

The second method which is the foundation of the change management of the system, is a method that will alter the hostname and domain part of the server. Before changing the hostname, the method will reset the hostname. For example, in the case of the sensor running on a server named server9 that has a hostname of server9.stem, the method will first restore its hostname to its original state - server9 without the extension. Once the change has been completed it will add the new extension server1.func and it will run a puppet command. The puppet command will request certificate information, and since puppet is configured to resolve configurations based on hostnames, a new configuration will be given to that server. This process is basically simulating the process of gene expression changes in the biological model.

The third method is the method which will listen to the TCP port. This method has to use the exact same port 10000, which is also specified in the signal.py framework. If that TCP port is taken for any reasons, another port has to be redefined both at the listen.py and sensor.py. Since the frameworks are deployed using a git repository, it should be no problem to change the code and redistribute it to every node.

listen.py Basic shared methods pseudocode

```
1   import regex and os module
2
3   Definition GetIPInterface():
4      regex = "IP regex to macth 192.168.51.* for example"
5      compile regex
6
7      eth0 = ifconfig eth0 | grep for addr | cut the left part of : | awk the IP address
8      eth1 = ifconfig eth0 | grep for addr | cut the left part of : | awk the IP address
9
10     for interface in [eth0,eth1]:w
11        result = regex.search(interface)
12        if result:
13           mainIP = result.group()
14
15     return mainIP
16
17  def SetHostname(type):
18     if type =='reset':
19        Run Linux ("HOSTNAME=`hostname | cut -d. -f1`; echo $HOSTNAME;
20        echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
21     elif type=='stem':
22        Run Linux ("HOSTNAME=`cat /etc/hostname`; HOSTNAME=$HOSTNAME.stem;
23        echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
24     elif type=='func':
25        Run Linux ("HOSTNAME=`cat /etc/hostname`; HOSTNAME=$HOSTNAME.func;
26        echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
27     else:
28        pass
29
30
```

Version 1 - Timing algorithm

The first algorithm (Top part of Figure 4.9) will activate the server by using sleep timers. The assumption is that when the system is overloaded, all or most of the servers will start sending messages depending on how evenly distributed the traffic load is. At the moment when each server receives a signal, a timer will be started. The timer will be a randomly generated integer, between one and ten which refers to one or ten minutes.

The goal of using such an approach is to avoid activating all servers at the same time, while still being certain that at least one server will get activated at all times. Once the timer has finished, the framework will check whether the signal persists in the network. When a system is overloaded, it might be the case that a huge spike in traffic occurs lasting only for a short period of time. In the case where the signal is still present in the network after the timer expires, it is clear that a system expansion needs to start. If the sensor doesn't receive a signal in a 20 seconds period, the framework will return to its initial state.

The amount of the machines that are activated at once using this algorithm is unpredictable, since it depends on the numbers generated across all stem cell servers. For example, if two servers get the same timer and when the timer has finished the signal is still detected, both servers will get activated in the same time frame. This can happen for three or even four

or more servers, but the probability of this happening decreases drastically with the amount of servers.

In formal terms, this can be described by resorting to a binomial distribution. Let n be the number of stem cells, p the probability of picking a timer between 1 and 10 minutes ($p = 1/10$ in this case), and x the number of machines that would pick the same timer. Then the probability that x machines will pick a particular timer t_i is given by the following formula:

$$\binom{n}{x} p^x (1-p)^{n-x} \tag{4.1}$$

listen.py Version 1 pseudocode

```
1  import sockets
2
3  Def ListenToInterface():
4     IP = GetIPInterface()
5     port = 10000
6     open TCP socket (IP,port)
7     if message = 1;
8        timer = rand.int(1,11)
9     sleep timer
10    close socket
11    CheckSignal(IP,Port):
12
13 Def CheckSignal(IP,Port):
14    time.start()
15    while 1:
16       if time.end - time.start <= 20:
17          open TCP socket
18          if message == 1:
19             SetHostname(reset)
20             SetHostname(func)
21             puppet agent --test
22             return
23          else:
24             sleep (1)
25       else:
26          break
27    ListenToInterface()
```

Version 2 - Random Seed algorithm

The second version of the algorithm (Bottom part of Figure 4.9) will use a python pseudo-random number generator. In this concept, every node will get the same seed planted inside the function. When a number is chosen, this number will be identical across all nodes. The logic in this case is that when a server receives a signal, the method for generating a random seed number will be called. The provided number will be compared to the hostname number. Only in the case where the hostname number matches the generated number an activation will start. Once this process has finished, the server will stop listening for any signals in the next three minutes. Three minutes is a proposed time frame in which the load is expected to go down. Using this approach, the number of the activated server will always be one

in every three minutes.

listen.py Version 2 pseudocode

```
1  import sockets
2  import random
3  random.seed = (5000)
4
5  Def ListenToInterface():
6     IP = GetIPInterface()
7     port = 10000
8     open TCP socket (IP,port)
9     if message = 1;
10       timer = rand.int(1,11)
11    sleep timer
12    close socket
13    CheckSignal(IP,Port):
14
15 Def CheckSignal(IP,Port):
16    number = random.seed(1,11)
17    hostname_number = 'hostname | cut -f1 -d. | tail -c 2"
18
19    if number == hostname_number:
20       SetHostname(reset)
21       SetHostname(func)
22       puppet agent --test
23       return
24    else:
25       sleep (180)
26       ListenToInterface()
```

The activation results from both algorithms are unexpected, because there might be different outcomes. Lets assume two random scenarios, where the two algorithms are run. One scenario where three VMs will bring the load in its acceptable limits (Figure 4.7, 4.8 on left) and another one where only one VM will do the same (Figures 4.7, 4.8 on right). In the first case, one timer on the VM rolled 2 minutes and got activated, while the next two VMs rolled 4 minutes and they got activated around that period. The load went down, signalling stopped and the total amount of time spent to bring the load down was 5 minutes. Since the second algorithm has a 3 minutes time span between spawning, the process took 7 minutes, thus being inferior to the version 1. On the other hand, in the second case it is the other algorithm who provided better performance by random chance. This is because the smallest number generated by random chance was 5 minutes for the first algorithm while the second algorithm activated the stem cell server instantly. Please note that these are only a few example scenarios that can result even in different outcomes, since everything depends on the random activation of the Version 1 timers.

Essentially, the outcome of the two versions of these prototypes is the same. The whole process tries to simulate the process of cell differentiation to the greatest degree. A short analogy between the two models, namely the biological model and the technical model is displayed in Figure 4.10.

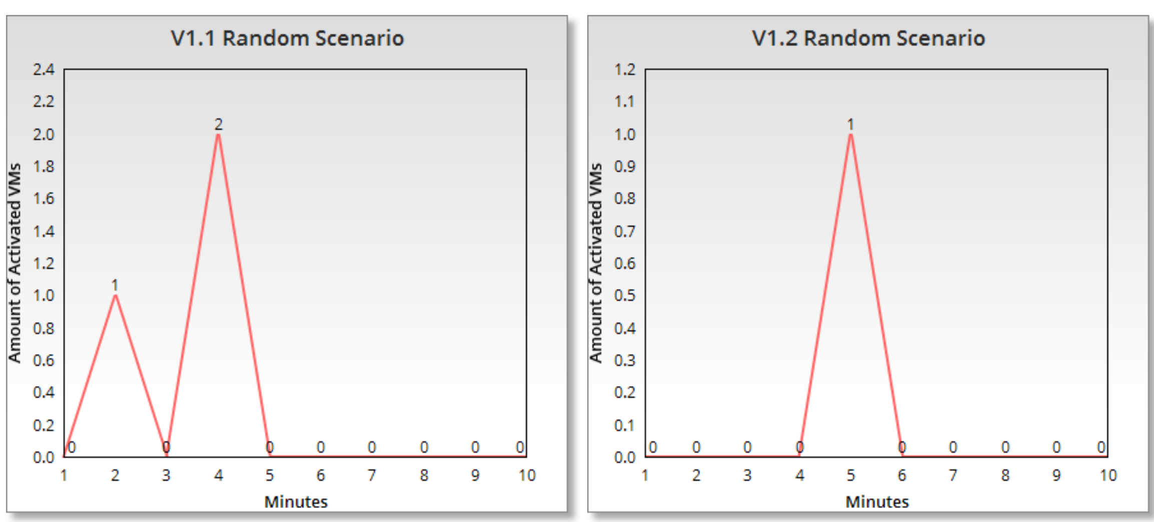

Figure 4.7: Random Scenarios for algorithm one

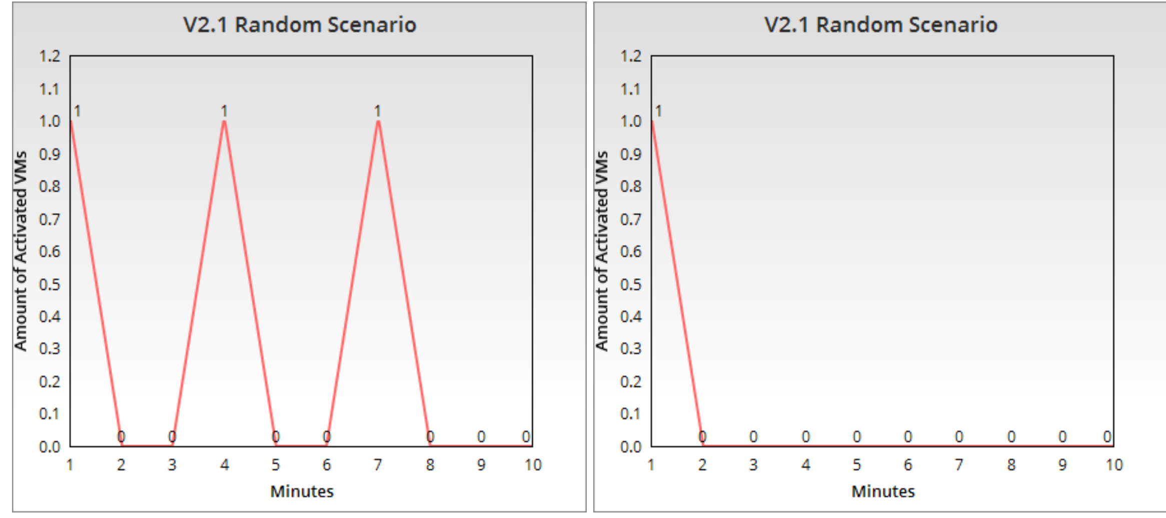

Figure 4.8: Random Scenarios for algorithm two

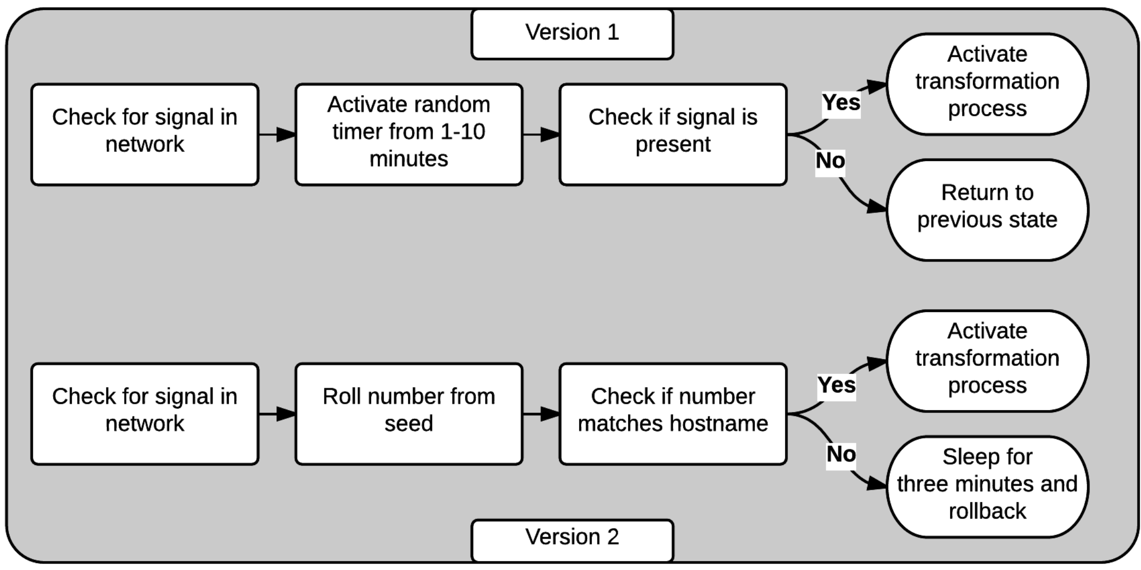

Figure 4.9: The logic difference in the prototype algorithms

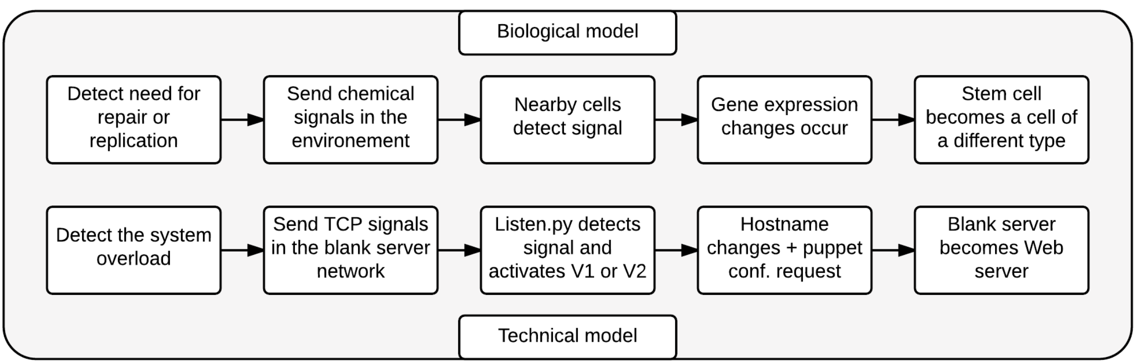

Figure 4.10: Biological model vs Technical model

4.2.7 Scaling down - reset.py

All of the above described frameworks will be implemented to achieve an expansion of the systems size and infrastructure. On the other hand, since the system has to demonstrate adaptability, there needs to be a controlling mechanism that will shrink the system when virtual machines are running in an idle state. For this purpose, a framework called reset.py will be running on all of the functional cells or webservers. The script will implement similar monitoring functions to the ones presented in the signalling framework, but the requirements will be different:

- System load for the last minute should be below 10%.
- Available free memory should be above 80%.
- Response times should be 0 seconds.

The checking for these parameters will be done in two minute intervals. If after five consecutive checks or 10 minutes, these requirements are met, a hostname change will occur again. This hostname change will be used to return the webserver to the pool of blank servers. The assumption is that our system is under constant load which will require more resources than the ones mentioned as requirements above. As nodes get removed from the webserver network and the load balancer configuration, the system load, memory and response times will rise across the other webservers that are still present in that domain.

reset.py shrinking algorithm pseudocode

```
1  import * from signal.py
2  counter = 0
3
4  Definition Decision(util,memory,response_time,counter):
5     if util <= 0.10 and memory >= 80 and response_time == 0:
6        counter = counter +1
7        if counter >= 5:
8           Remove()
9     else:
10       counter = 0
11
12 Definition Remove():
13       SetHostname(reset)
14       SetHostname(stem)
15       puppet agent --test
16       return
17
18 Definition Main():
19    while 1:
20       counter = Decision(FetchSystemLoad(),FetchMemory(),FetchResponseTime(),counter)
21       sleep 120
```

4.2.8 Monitoring and plot script - monitor.py + plot.py

The monitoring script will be deployed on all web servers. The goal of this script is to gather accurate local data and save these measurements to a central location. This central location is a previously set up storage server.

The measurements script will use the FetchSystemLoad, FetchMemory and FetchResponseTime functions and store their output along with a timestamp into a file. Once the experimenting finishes and all measurement files have been transferred to the storage server, system wide calculations can be done.

monitor.py algorithm pseudocode

```
1  import * from signal.py
2  import time
3
4  Definition Monitoring():
5     sys = FetchSystemLoad()
6     mem = FetchMemory()
7     rt = FetchResponseTime()
8     get timestamp
9     get hostname
10
11    SSH into Storage server:
12    open file (measurement_hostname)
13    file.write(sys mem rt timestamp hostname)
14    file.close
15    ssh.close
16
17 Definition Main()
18    while 1:
19       Monitoring()
20       sleep 60
```

The plotting script will be located on the storage server and will be used to prepare the data for plotting. The responsibility of this framework is to open all the obtained files and based on the stored data calculate the average response times, memory usage and system load for the whole system during the experimenting stage. The timestamp will be used inside a dictionary and the values will be stored inside of that timestamp. For each time stamp separate values will be combined to calculate the system values.

plot.py algorithm pseudocode

```
1  import os
2  timestamps_mem = {}
3  timestamps_sys = {}
4  timestamps_rt = {}
5
6  Definition SumUp():
7     files = os.listdir('/path-to-files')
8     for file_s in files:
9        file.open(file_s)
10       file_output=file.readlines():
11       for line in file_output:
12          timestamp_sys[line[3]] = line[0]
13          timestamp_mem[line[3]] = line[1]
14          timestamp_rt[line[3]] = line[2]
15
16    sum(timestamp_sys.values())
17    sum(timestamp_mem.values())
18    sum(timestamp_rt===.values())
```

Chapter 5

Result II - Implementation and Experiments

This chapter provides a brief description of the work that has been done to prepare the environment for running the experiments. In the first section of this chapter, the skeletons described in the previous chapter along with their structure are explained in detail. The skeletons have been built and improved throughout the whole implementation process in order to refine their function inside the system. During the experiments and the building of the system, the tests discussed in section 3.3.4 have been run interchangeably. At last, this chapter presents the developed scripts and covers the type of experiments that have been done throughout this stage before obtaining the results.

The first servers that have been set up manually are the Puppet and the MLN server which run both on the same virtual machine. The storage server for the monitoring logs along with the HAProxy have been set up on separate instances. These servers have been configured by using SSH and configuring the system through the command line interface, while the networks and the router have been created using the OpenStack Horizon WebGUI.

5.1 MLN skeleton

The MLN templates will be created based on the skeleton described below, combined with user input given to the deploy.py script. The MLN skeleton file is structured in different blocks for different purposes, namely:

- Blue blocks are used for creation of MLN projects and nodes. The node will be visible in the cloud GUI as "hostname.projectname", but if a hostname parameter is not given, the node name would be the same as the project name, i.e. "projectname1.projectname". If a parameter is given as hostname to the deploy.py script, then "projectname1" will be replaced with the given parameter. The

created hosts will inherit the properties of two superclasses named funccell and stemcell. In the skeleton structure below, only the func cell is included since the stem cell class is identical to the func class except for the parameter given to init_hostname.sh. In the stem class, the "stem" keyword is given instead of "init_hostname.sh func".

- The blocks marked with red lines will contain OpenStack configurarion. The keypair specification is used for configuring preconfigured SSH keys that are deployed on the new node, the image parameter specifies the OS image which will be installed and the flavor contains hardware specifications and quotas. In this case the flavor "tiny" is used which refers to 1 VCPU, 2 GB of storage and 512 MB RAM. All of these mentioned parameters will be replaced by the deployment script if there is any user input for that specific variable.

- The user data block is marked with violet color. This block contains post-install commands which will be executed after successful OS installation. In this case, the package repository information will be updated to the latest package information. Additionally, since no DNS server is present, one has to manually create an entry for the nameserver and the puppet master node. To create a distinction between the different types of nodes (stem or func), a bash script that alters the internal hostname along with other options is obtained and executed. In the end, the puppet agent package will get installed and a command to request configuration from the puppet master will be executed.

- The network block marked as **black** will contain cloud network configuration. Inside of this block, it is specified to what virtual network the newly provisioned node should be placed along with the way the node should obtain it's IP. Because of rebuilding and testing purposes, it is preferable to use DHCP over static addressing.

skeleton.mln

```
1  global {
2      project projectname
3  }
4
5  superclass funccell {
6      openstack {
7          keypair puppet_key
8          image ubuntu-12.04
9          flavor m1.tiny
10         user_data {
11             echo "192.168.50.2 puppet puppet.stemcell" >> /etc/hosts
12             echo "192.168.51.2 puppet puppet.funccell" >> /etc/hosts
13             ifconfig eth1 up
14             dhclient eth1
15             echo "nameserver 8.8.8.8" > /etc/resolvconf/resolv.conf.d/head
16             echo "nameserver 8.8.8.8" > /etc/resolv.conf
17             apt-get update
18             wget http://opas3n.com/init_hostname.sh
19             bash init_hostname.sh func
20             apt-get install puppet -y
21             puppet agent --test puppet
22         }
```

```
        }
        network eth0 {
                net stem_cell_network
                address dhcp
        }
        network eth1 {
                net function_cell_network
                address dhcp
        }
}

```

When building the complete build.mln file which is contained in the appendix section, the deploy.py script appends the server information to the modified skeleton. Each server that will be automatically created has its own block. For example, if there are 2 stem cells and 2 functional cells, the blocks would look like below:

cells.mln

```
host server1 {
    superclass stemcell
}

host server2 {
    superclass stemcell
}

host server3 {
    superclass funccell
}

host server4 {
    superclass funccell
}
```

5.2 Puppet skeleton

A shortened version of the puppet skeleton is contained in the code block below, while the complete and used files are contained in the appendix. The version below is used to describe how certain actions can be achieved. This .pp file has a block-like syntax similar to MLN. To fulfill the needs described in the approach section, the code in the blocks below configures these options :

- The violet block specifies the hostname for which this configuration will be applied.
- The blue blocks install the required packages.
- The green block makes sure that the web service is running after the packages have been installed.
- The red blocks will execute commands after the packages and services have been configured. The software will be pulled from the public Github repository and the configuration files will be places in the intended places.

server1_func.pp

```
node 'server1.func'{
$php_packages = ["php5","libapache2-mod-php5","php5-mcrypt"]
	package { 'apache2':
		ensure => present,
	}

	package {$php_packages:
		ensure => "present",
	}

	package { 'git':
		ensure => "present",
	}->

	exec { "git clone":
		cwd => "/tmp",
		require => [Package["git"], Package["apache2"]],
		command => "git clone https://github.com/opas3n/Netlab.git",
		path => "/usr/bin:/bin",
	}

	exec { "Fix interface":
		cwd => "/tmp",
		require => [Exec["git clone"]],
		command => "cp /tmp/Netlab/interfaces /etc/network/interfaces",
		path => "/usr/bin:/bin",
	}

	exec { "apache-conf":
		cwd => "/tmp",
		require => [Exec["git clone"]],
		command => "cp /tmp/Netlab/default /etc/apache2/sites-available/default",
		path => "/usr/bin:/bin",
	}->
	service { 'apache2':
		ensure => running,
		enable => true,
	}

	exec { "apply-index":
		cwd => "/tmp",
		require => [Exec["git clone"]],
		command => "cp /tmp/Netlab/index.php /var/www/index.php",
		path => "/usr/bin:/bin",
	}

	exec { "run-signal":
		cwd => "/tmp",
		require => [Exec["git clone"]],
		command => "python signal.py",
		path => "/usr/bin:/bin",
	}
	exec { "run-monitor":
		cwd => "/tmp",
		require => [Exec["git clone"]],
		command => "python monitor.py",
		path => "/usr/bin:/bin",
	}
	exec { "run-reset":
		cwd => "/tmp",
		require => [Exec["git clone"]],
		command => "python reset.py",
		path => "/usr/bin:/bin",
	}
}
```

The presented pp file will transform a blank server to a web server by completing the tasks contained in the code block above. On the other hand, the reverse process, or the process of resetting a server, requires that all of the above packages, as well as the software obtained from the git repository are not present on the system. In puppet, this can be easily accomplished by using the same syntax as above and replacing the "ensure => present" phrase to "ensure => absent". This syntax is valid for both software packages, directories, as well as individual files located on the local system.

In the current set up, puppet has been configured to automatically sign a certificate for each request that it receives from a server that belongs to the .stem or .func domain. This is done to avoid the need of human interaction in the process of authentication, but can be a major security issue because every server has the ability authenticate as another instance. Since this is a testing environment for running experiments, having such a solution is acceptable, but if the frameworks are implemented in a working production environment, changes have to be made.

5.3 Deployment Framework

The deployment framework has a built in help menu that is used to offer the user a description about individual parameters that the framework uses. Every parameter has one big letter case argument and an alternative word argument that can be used. The help menu is given by running the '–help' extension and the provided output is shown below:

deploy.py help menu

```
root@puppet:~# ./deploy.py --help
usage: deploy.py [-h] -P PROJECT -S SNUMBER -F FNUMBER [-H HOSTNAME]
                 [-K KEYPAIR] [-O OS]
                 [-T m1.tiny,m1.small,m1.medium,m1.large,m1.xlarge]

optional arguments:
  -h, --help            show this help message and exit
  -P PROJECT, --project PROJECT
                        Specify the project name for the MLN project
  -S SNUMBER, --snumber SNUMBER
                        Specify the number of stem cells/blank servers
  -F FNUMBER, --fnumber FNUMBER
                        Specify the number of functional cells/web servers
  -H HOSTNAME, --hostname HOSTNAME
                        This is the basis of the hostname for the deployed
                        nodes, if not entered,the basis will be the same as
                        the project name
  -K KEYPAIR, --keypair KEYPAIR
                        Specify the keypair which will be used to access the
                        servers
  -O OS, --os OS        Enter this parameter in case you want to deploy a
                        different OS, the exact name of the OS image has to be
                        entered
  -T {m1.tiny,m1.small,m1.medium,m1.large,m1.xlarge},
  --flavor {m1.tiny,m1.small,m1.medium,m1.large,m1.xlarge}
                        Choose one of the available choices if you want to use
                        a different OpenStack flavor
```

Once the framework has finished deploying the infrastructure, the individual servers are displayed in the OpenStack Horizon GUI. The next step was to configure the HAProxy and to create a small php script that will stress the CPU.

5.4 HAProxy and PHP

The HAProxy has been configured to balance the web traffic load between the available web servers. This is achieved by implementing a custom checking mechanism, so that the server can check what webservers are available and what servers are unavailable. A simple html file called check.html with the string "Itworks" is deployed on all web servers. The HAProxy will scan all given IPs configured inside the main configuration file and compare the given result. All webservers that are hosting this website and that reply within a given period will be considered as active. Furthermore, a statistics frontend for the load balancing has been enabled on port 2000, so that one could have an overview of the available nodes and additional statistics data. A truncated version of the HAProxy configuration file is shown below, which configures the web server backend, the HAProxy frontend and the statistics page. The full configuration file is contained in the appendix section.

haproxy.cfg

```
1  listen stats :2000
2      mode http
3      stats enable
4      stats hide-version
5      stats realm Haproxy Statistics
6      stats uri /
7
8  frontend web
9          bind *:80
10         default_backend back
11
12 backend back
13         server server1 192.168.51.6:80 check
14         server server2 192.168.51.7:80 check
15         server server3 192.168.51.8:80 check
16         server server4 192.168.51.9:80 check
17         server server5 192.168.51.10:80 check
18         server server6 192.168.51.11:80 check
19         server server7 192.168.51.12:80 check
20         server server8 192.168.51.14:80 check
21         server server9 192.168.51.13:80 check
22         server server10 192.168.51.5:80 check
23
24         http-check expect string Itworks
25         option httpchk GET /check.html
```

After configuring the HAProxy load balancer, only the active webservers will respond to the web requests. When targeting certain amounts of web traffic to the servers, it is observable that the CPU and memory resources are not easily depleted. A small PHP script that is deployed and served on all servers as a web page will stress out the system, so that the

amount of requests can be lowered while still occupying the same quantity of resources. The script will create arrays of random numbers, shuffle and sort the numbers and print the output to the web page. It is created to be dynamic so that with only a few changes inside the code (i.e. increasing the size of the array or the shuffle size), more resources are required to display the web page. The code of the script, as well as a sample output provided inside the web page are displayed in the code blocks below. Note that each time a webpage is served by the web server, different random numbers are being displayed.

index.php

```
<?php
$rand_num = array();
for ($i=0; $i<200; $i++) {
   $rand_num[]= mt_rand(1, 200);
}
for ($j=0;$j<=500;$j++) {
   shuffle($rand_num);
   sort($rand_num);
}
print_r($rand_num);
?>
```

index.php sample output

```
Array ( [0] => 5 [1] => 6 [2] => 8 [3] => 9 [4] => 9 [5] => 12 [6] => 13 [7] => 15 [8] => 17 [9]
=> 17 [10] => 17 [11] => 17 [12] => 18 [13] => 19 [14] => 19 [15] => 19 [16] => 19 [17] => 20 [18]
=> 20 [19] => 21 [20] => 21 [21] => 26 [22] => 28 [23] => 28 [24] => 29 [25] => 29 [26] => 32 [27]
=> 34 [28] => 34 [29] => 37 [30] => 38 [31] => 38 [32] => 38 [33] => 39 [34] => 42 [35] => 43 [36]
=> 43 [37] => 44 [38] => 44 [39] => 47 [40] => 48 [41] => 48 [42] => 49 [43] => 50 [44] => 53 [45]
=> 54 [46] => 55 [47] => 55 [48] => 55 [49] => 55 [50] => 56 [51] => 56 [52] => 56 [53] => 58 [54]
=> 58 [55] => 60 [56] => 61 [57] => 62 [58] => 64 [59] => 70 [60] => 71 [61] => 73 [62] => 75 [63]
=> 76 [64] => 77 [65] => 77 [66] => 77 [67] => 79 [68] => 79 [69] => 80 [70] => 80 [71] => 83 [72]
=> 84 [73] => 85 [74] => 86 [75] => 89 [76] => 89 [77] => 92 [78] => 92 [79] => 92 [80] => 92 [81]
=> 92 [82] => 94 [83] => 95 [84] => 95 [85] => 97 [86] => 97 [87] => 98 [88] => 99 [89] => 99 [90]
=> 100 [91] => 102 [92] => 102 [93] => 104 [94] => 104 [95] => 105 [96] => 105 [97] => 106 [98] =>
106 [99] => 107 [100] => 108 [101] => 109 [102] => 111 [103] => 115 [104] => 115 [105] => 115
[106] => 117 [107] => 117 [108] => 117 [109] => 121 [110] => 121 [111] => 121 [112] => 121 [113]
=> 122 [114] => 122 [115] => 122 [116] => 124 [117] => 124 [118] => 125 [119] => 126 [120] => 126
[121] => 127 [122] => 127 [123] => 128 [124] => 129 [125] => 133 [126] => 135 [127] => 136 [128]
=> 137 [129] => 137 [130] => 138 [131] => 138 [132] => 139 [133] => 140 [134] => 140 [135] => 142
[136] => 144 [137] => 145 [138] => 145 [139] => 146 [140] => 149 [141] => 150 [142] => 151 [143]
=> 151 [144] => 152 [145] => 152 [146] => 152 [147] => 152 [148] => 153 [149] => 153 [150] => 154
[151] => 154 [152] => 155 [153] => 155 [154] => 155 [155] => 156 [156] => 156 [157] => 157 [158]
=> 158 [159] => 158 [160] => 159 [161] => 159 [162] => 159 [163] => 160 [164] => 160 [165] => 162
[166] => 163 [167] => 164 [168] => 165 [169] => 165 [170] => 166 [171] => 166 [172] => 168 [173]
=> 172 [174] => 173 [175] => 173 [176] => 175 [177] => 175 [178] => 176 [179] => 176 [180] => 176
[181] => 176 [182] => 178 [183] => 179 [184] => 179 [185] => 181 [186] => 182 [187] => 182 [188]
=> 182 [189] => 183 [190] => 186 [191] => 189 [192] => 189 [193] => 190 [194] => 191 [195] => 194
[196] => 194 [197] => 198 [198] => 198 [199] => 200 )
```

5.5 Testing and experiments

After the successful infrastructure deployment and the configuration of the required servers and tools, the next part of these results consists of running the experiments. Before obtaining the actual results, all of the previously defined tests have been completed in order to confirm the correct functioning of the system and the developed frameworks. Some

of the tasks that have been part od the testing process for the individual servers and the system include:

- Test T_1 - Multiple different environment with different proportions have been set up and rebuilt from scratch with the help of the deployment framework.
- Test T_2 - Pinging between individual node and using SSH to check accessibility has been completed throughout the topology.
- Test T_3 and T_4 - Puppet certificate requests and git commands have been executed to check configuration results. Additionally, listing of all present configuration files and inspecting their contents has been done on all newly spawned or transformed servers.
- Test T_5 and T_6- Running the frameworks manually, printing statements and testing the Linux commands inside the scripts has been done continuously throughout the development and implementation process. Furthermore, testing the expected output and structure provided by the measurement tool has been accomplished.

The work that has been completed as part of the experimenting process includes fine-tuning the .php script and the quantity of web requests to saturate the individual web servers in order to reach a signal triggering and system expansion. After the system has expanded and the load on the individual servers is decreased, the system runs in an optimal state. If the web traffic load is reduced to a point where web servers run below the specific parameters, the webservers will start the process of resetting and consequently the process of system shrinking. The expansion and shrinking experiments have been done repeatedly for a larger period of time, and multiple common behaviours of the system are chosen to be presented in the next chapter for analysis and comparison.

Chapter 6

Results III - Measurements and Analysis

The experimenting process for enlarging and decreasing the size of the system was repeated in the course of several months and the obtained results from this time period are presented in this chapter. The results have been obtained by the developed measurement script by calculating the local performance average and by combining these acquired data sets from each individual server to compile system wide results.

Most experiments were conducted with a total number of ten cells, out of which three were functional cells or webservers, and seven were stem cells or blank servers. Only a few test experiments were carried out starting with one webserver and nine blank servers. Please note that as the amount of web servers changes, the quantity of the generated web traffic has to be modified so that the system can be stressed out within its limits in order to trigger the signalling process. Since most systems that could use this type of scaling and adaptability would have at least three webservers, the presented results are based on a topology where in the beginning of the experiment the load was balanced between three servers.

After the experimenting process was finished and the performance charts were plotted, it was observable that the Random Seed algorithm prototype was performing almost identical between all experiments. On the other hand, the Timing algorithm prototype provided different performance outcomes due to the randomness of the built in timer method. Although the performance results for this prototype were varying between experiments, they can be grouped in two types of sets depending on the timer distribution. The first section in this chapter presents the two most common timing algorithm results, while the next one presents the identical random seed algorithm outcome.

6.1 Timing Algorithm

From the measured results, one could observe that the main difference between the two most common result types lies in the distribution of the timers, therefore the two different scenarios below are based on a case where the timer interval lies between 1 & 5 minutes and 5 & 10 minutes, respectively.

6.1.1 CPU and memory performance in timer interval 1-5 minutes

In this first case where the smallest generated timers lie between one and five minutes, the total amount of VMs that have been spawned to lower the system load is three, thus the final amount of web servers present in the system was six (three functional cells + three transformed stem cells). In the beginning of the experiment, the CPU load (displayed on the left side of Figure 6.1) was 0.06 or 6% before the traffic generation began. The generated traffic was gradually increased between the first two minutes of the experiment in order to reach the system's saturation point. As it is noticeable on Figure 6.1, the signalling process that occurs when the load is above 85 % started prior to the second minute, where the system load reaches 96%. Assuming that the signal reached the other VMs around the second minute, we can define that this is the point where all VMs in the stem cell network (blank servers) started generating their respective timers.

The smallest generated timer was two minutes which meant that the VM is going to sleep for two minutes (between 2 and 4 minutes on the chart). After this period is over the VM is going to check if the signal still persists in the network. As it is displayed in the chart, the signal was still present in the network, since the system load is above 85% and the available memory is below 15%. This resulted in the first virtual machine being transformed from a blank server to a web server. As one can see in the fifth minute, the system load decreased for 14% bringing to total system load to 86%. The second timer that was chosen by the next VM was four minutes which means that the signal presence will be checked at the sixth minute in the chart in Figure 6.1. Different from the previous case where only one VM had generated a specific timer, in this case two blank servers have chosen the same timer, resulting in two activations at the same time.

After the transformation has been completed, the system load was instantly decreased to 55% and the available memory was increased to 44%. The timers generated by the remaining blank virtual machines in this case are distributed between five and ten minutes and the process of activation will not be completed for these VMs, but rather they will return to their previous state. The system load for the remaining minutes of the experiment varies roughly between 55% and 47% while the available memory between 44% and 52%.

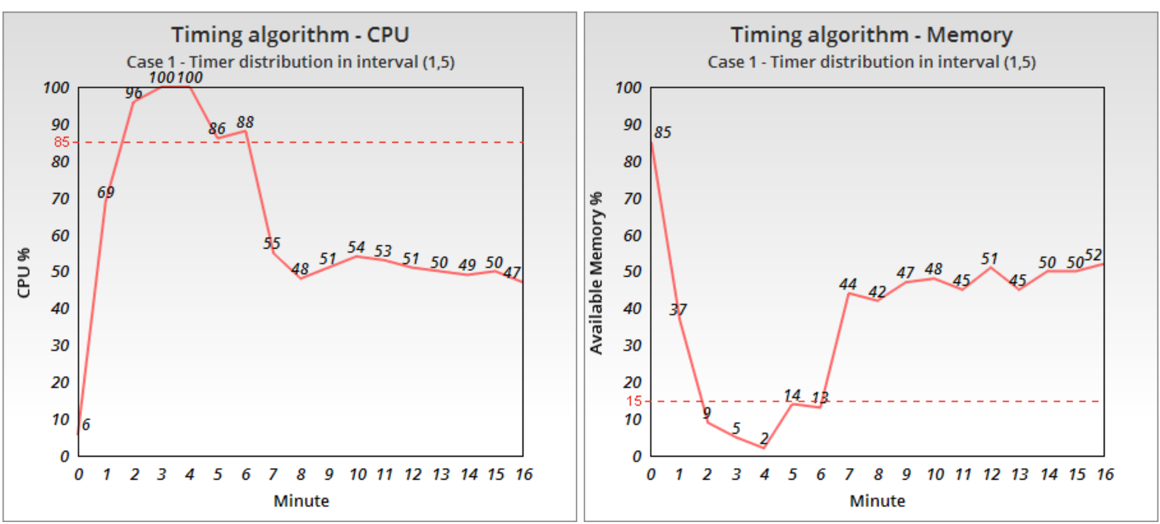

Figure 6.1: CPU and memory performance in a scenario where the first timers are distributed in a time interval between 1 and 5 minutes

One could conclude that the transformation process lasted for 4-5 minutes and this was the time it took the prototype to bring the system within its acceptable limits. In this case, the prototype lowered the system load by approximately 45-55% and increased the available memory 42%-50%. Even though only two virtual machines were needed to bring the load down and increase the available memory, the prototype generated activated one additional VM, thus increasing the available system resources by a larger amount.

6.1.2 CPU and memory performance in timer interval 5-10 minutes

Unlike the previous scenario, in this type the generated timers that were chosen by the seven stem cell servers were in an interval between five and ten minutes. As it is shown in Figure 6.2, the traffic generation started at the same period as in the other case. On the other hand, the system load was at a 100% and the memory resources were exhausted for a longer period of time. While the signal has been triggered at the same time, one can see that the virtual machines were activated at a later stage.

The first virtual machine was activated at the seventh minute mark or five minutes after the signal has been dispatched to the stem cell network. This virtual machines lowered the CPU load across all individual web servers for 13%-14% on average which is observable on the left side of Figure 6.2. The first transformation has also increased the pool of available memory for 10%-12%. Although the activation of the first blank server has resulted into additional resources for use, it did not bring the system to its desirable acceptable limits. This can be noticed in both charts displayed between the seventh minute and ninth minute.

The second timer has been activated after seven minutes or at the ninth minute mark in the chart. The changes of this activation can be spot in the rapid increase of available free memory and the decreased system load in the tenth minute in Figure 6.2. For the remaining period of the experiment, the average CPU load across the system was between 68%-71% while the free memory was between 31%-33%. In this case, only two virtual machines were transformed to bring the system to its desirable state, thus the total amount of active web servers in the end of the experiments was five. Looking at these results, it is evident that the two VMs brought the total system load down for around 29%-31%, while the total memory that has been freed is roughly between 30%-32%. When compared to the results from the previous case, the total transformation process took 5-7 minutes which is drastically longer due to the fact that the amount of transformed virtual machines was lower.

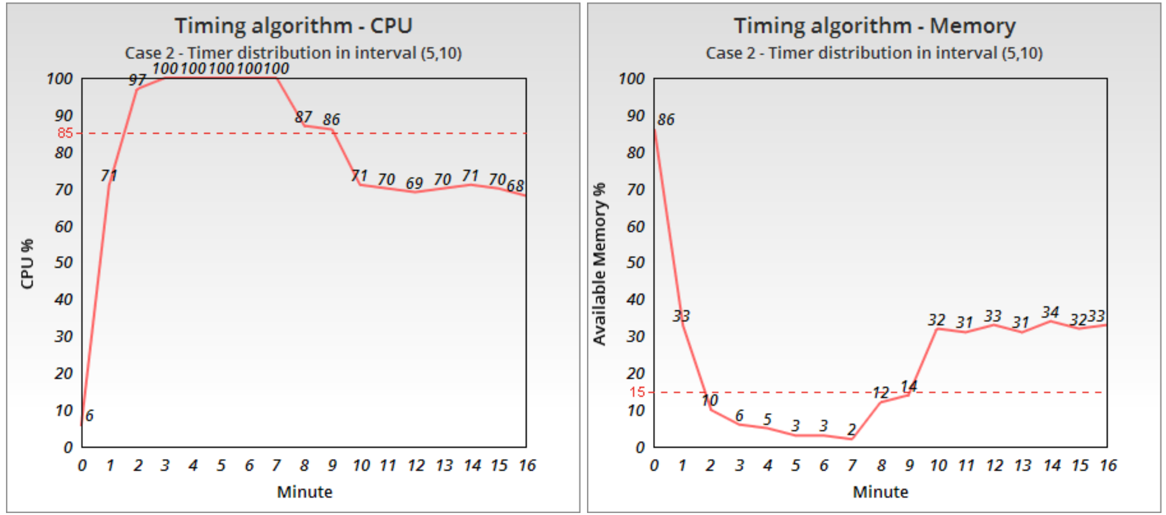

Figure 6.2: CPU and memory performance in a scenario where the first timers are distributed in a time interval between 5 and 10 minutes

6.2 Random Seed Algorithm

The random seed algorithm can be seen as a predictable algorithm when compared to the timing algorithm. Although the CPU and memory performance improvements are not foreseeable, it is certain that only one virtual machine will always be activated when a signal is received. This behaviour is demonstrated in the average performance data gathered from all of the servers which is illustrated in Figure 6.3.

The traffic generation has been completed in the same way as in the two cases that have been presented above. It is interesting that in this case, due to nonspecific reasons the peak of 100% was reached earlier, but this will not affect the ongoing data since there is no difference in the time at

which the signal is being sent. In both cases the load in the second minute is above 85% and the signal dispatch is executed around the same period of time. The main distinction between the two algorithms can be noticed right at the beginning of the experiment.

After the signal has been sent, all of the virtual machines have picked a random number based on the same random seed that has been planted inside the method of the python framework. Only the one virtual machine which hostname number matched the generated number will start the transformation process. At the third minute on Figure 6.3 which is also the first minute after the instant transformation process has finished, a reduction of 15% in CPU usage and a 11% growth of the available memory is manifested.

Three minutes after the transformation process has been triggered, the sensor functionality on every individual blank server will check again for a signal presence in the stem cell network. This process occurs in the fifth minute when an additional second virtual machine gets activated by repeating the same process of seed number generation due to the signal existence in the network emitted by the stressed functional cells or web servers. After the process of transformation which imitates cell differentiation, the load and available resources of the system have been brought to the acceptable limits.

The system will remain in its allowed state varying between 68%-73% in system load and 29%-34% in available memory from the sixth minute until the end of the experiment. The total time that has been spent to adapt the system to the current needs excluding the first two minutes where the traffic was generated and sent, but rather focusing on the algorithm and transformation processes was 3-4 minutes.

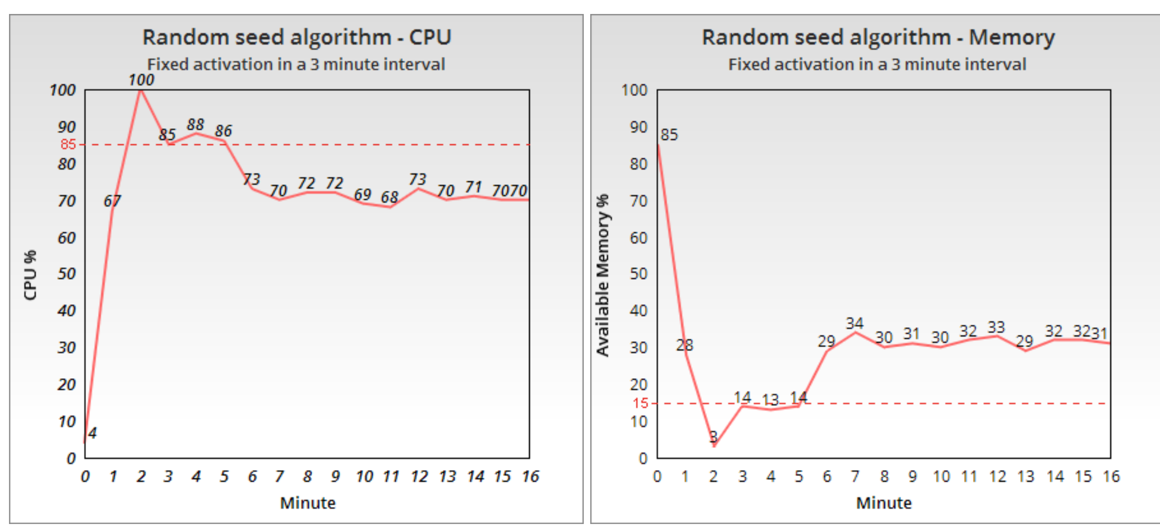

Figure 6.3: CPU and memory performance from the random seed algorithm which spawns one virtual machine instantly, and one in every third minute afterwards

6.3 Scaling down

After the experimentation stage with expansion of the system has finished, experiments with reducing the systems size have been repeated in order to analyze the shrinking behaviour of the infrastructure. The experiments consist of lowering the generated web traffic to a certain degree where only one webserver is enough to handle all of the load, while all other servers are being reset and returned back to the stem cell group. This procedure is handled by the reset framework, which has a checking mechanism being run each two minutes. The mechanism will check whether the system load is below or equal to 10% and the available free memory is above or equal to 80%. If the obtained parameters match the criteria after five consecutive checks in a row, the process of reset will start on the virtual machine. This results in a reduced number of available web servers.

In most of the experiments the gathered results were identical and the results of one particular experiment are presented below. The experiments where the obtained results were not as expected are mainly due to the fine tuning of the amount of web traffic that needs to be generated in order to meet our goals. In this sample, the experiment and shrinking of the system continues from the first case of expansion presented above (see Figure 6.1) where the timing algorithm expands the system by three virtual machines. By the end of this experiment there are six available web servers that are serving the incoming requests. After the sixteenth minute which is the last minute displayed in the chart, the web traffic was instantly reduced to bring the system in an idle state.

There is a window of ten minutes where the checks are executed, but

the data in this time frame is insignificant since there are no essential system changes occurring. Every virtual machine runs the reset.py script after it has obtained its puppet configuration catalog which means that they will not execute the checks at the same time. If this was the case, all virtual machines would have turned off simultaneously which is what has happened during the debugging procedure in some of the first experiments.

The illustration displayed in Figure 6.4 displays data that has been gathered ten minutes after the web traffic reduction while the individual data points were gathered in a five second interval for a period of one minute. It was interesting to observe how the number of virtual machines correlates to the average of the system's available resources during the contraction of the infrastructure. As the number of web servers decreases, the CPU load rises only for a small amount, up until the point where the last two web servers are being reset. This is manifested at the green bar at 26:45 on Figure 6.4. A similar effect can be seen with the system's free memory because between 6 and 3 available web servers (26:00 - 26:45, displayed with red bars) the average memory percentage(displayed with blue bars) has decreased for only 6%-9% percent. On the other hand, in the last period for only two additional virtual machine resets, the free memory got reduced by 13%-16%.

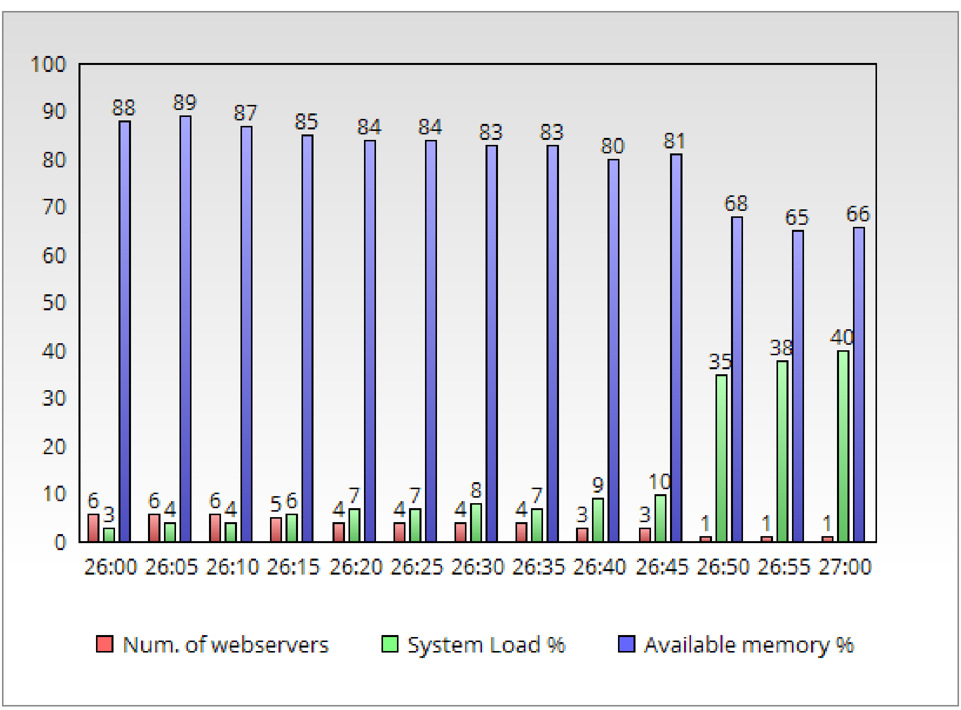

Figure 6.4: A view of the system's average resource parameters during the process of infrastructure downscaling

Chapter 7

Discussion

The different stages of this study are discussed in this chapter, describing the theoretical and practical challenges that have been encoutered during the work on this project. Furthermore, improvements of the developed prototypes are suggested and future work is proposed as an extension of this study.

7.1 Project evolution

In the beginning of this project, there were many alternative approaches that could have been taken. The field of bio-inspired computing can be seen as a underinvestigated researched area when compared to traditional computing and there is a lot of potential for research based on biological models especially in the field of bio-inspired cloud computing. The main motivation for choosing a cellular differentiation model as a foundation for the technical part was that this concept has not yet been applied to achieve web server scaling in the cloud. Unfortunately, the time period that was given to finish the study was limited and not enough to extend the study in order to compare it with some of the other biological models discussed in 2.3. This is one of the things that can be completed in a future study where the focus will be shifted from the model designing to algorithm optimization and an in-depth result comparison.

Achieving adaptive behavior in this project provided some challenges especially because decentralized nature of the biomorphic model. In fact, the technical system had to retain all of the characteristics from the biological model. Some of these characteristics include: autonomous action in which every element configured itself rather than being controlled by a master, interaction based on local information which manifests in the how the decisions are made as well as the addition and removal of individual elements from their respective groups which is an analogy to the process of birth and death. Some of these challenges would have been eliminated with a centralized system, but it would come with the price of a single point of failure and wouldn't belong to any of the discussed bio-inspired areas.

It requires an extensive and long measurement period in order to be able to make strong statements based on the extracted data from researching with cloud related technologies, when compared to traditional computing. One reason for this is the shared underlying infrastructure and the way it affects the virtual machines. Although virtual machines are isolated from each other, stressing out the physical hardware resources might have an effect on the virtual resources. It would be interesting to observe the results of the developed prototypes in a dedicated cloud environment which has no other users, and compare them to the presented results which were obtained from a shared cloud environment provided by HiOA's ALTO cloud. The outcome of such a study would be gained knowledge and experience that can be used to add additional frameworks to the system to regulate the available stem cell server pool, by keeping track of the cloud's active users and the available hardware resources.

During the testing and experimentation period there were some major issues that were faced. One issue was that the complexity of the project includes different technologies that have to work in a synchronised manner in order to provision and configure the virtual machines in the planned time interval. Some minor changes were made to the initial framework models during the implementation stage while other changes and improvements are suggested in the section for future work discussed below. Another issue that was faced during the implementation stage was the version of the software that has been used and a small bug in the MLN software. For example, the Puppet version which was used is 2.7.11 and was the latest version that was available from the software repository for Ubuntu 12.04, but it lacked some important functionalities like domain prefix matching. The bug with the MLN software was encountered when spawning the infrastructure, because MLN could not parse the build files. After some debugging period, this problem was ironically fixed by simply rebooting the server.

7.2 Algorithm comparison and proposed improvements

Based on the results that were presented in chapter 6, one could conclude that both algorithms have weak and strong sides. Choosing a superior algorithm version would be based on the needs of the system. The conducted experiments were completed by gradually increasing the traffic load up until a certain point at which the amount of web traffic stopped increasing. The addition of two virtual machines was enough to bring the load down to its acceptable limits, but if the load was higher then more virtual machines would be needed. The efficiency of the prototype mainly depends on the number of virtual machines that need to be spawned and the acceptable reaction time for this procedure to finish. To clarify the priorities, let us first look at the advantages and disadvantages of both prototypes.

7.2.1 Timing algorithm

One of the advantages of the timing prototype is that using this model multiple virtual machines can be spawned at once. As seen in the first case that was presented, even though two machines were needed to handle the load, a third one was spawned because two VMs had chosen the same timer. If the load kept rising above the web traffic threshold, the system would be able to support even more traffic without the need of signalling or spawning a new virtual machine. This prototype can be seen as predictive and efficient in this case. On the other hand, if the traffic load stayed the same as in the presented case, one virtual machine was unnecessary taking up cloud resources which could have been used in another way, thus making the prototype inefficient. Additionally, as seen in the figures that have been presented for this algorithm, it is of a highly unpredictive nature, since the smallest generated timer can be anything between 1 and 10 minutes. To be precise, the reaction times can be extremely fast with spawning a couple of virtual machines in the first two to three minutes, but it can also be extremely slow taking up to 10 minutes to spawn a single virtual machine.

- **Advantage** - Multiple VM spawning at once, possible system expansion to handle increased future traffic.
- **Disadvantage** - Timer randomness which could lead to long reaction times, possibly spawning additional unnecessary VMs.

7.2.2 Random seed algorithm

The advantages of the random seed algorithm over the timing prototype is that the first reaction time is instant. Additionally, if the smallest timer generated by the timing prototype across all individual nodes in the system is bigger than 5 to 6 minutes, the random seed algorithm will have quicker reaction times for transforming at least 2 virtual machines. On the other hand, when a larger amount of virtual machines needs to be spawned, the time it takes to transform the required stem cell servers to functional servers is long because of the 3 minute built-in sleep timer in the prototype. Depending on the load and the generated timers from the timing algorithm, this random seed algorithm can both be inferior or superior when compared to the timing algorithm.

- **Advantage** - Instant reaction, predictive nature because of the fixed time spawning interval.
- **Disadvantage** - Slower reaction times as the number of needed web server VMs increases. Can only spawn one virtual machine at once in a three minute interval.

7.2.3 Improvements

To overcome the disadvantages of both algorithms, some things can be improved in future versions that could be developed following this study. For the timing algorithm the generated timer could be decreased to be a random number between 1 and 5 minutes. It is worth noting that the range in which the timer is generated has to be adapted to the amount of available stem cell servers. If there are more than 10 stem cell servers, it would be wise to increase the maximum timer to be over 5 minutes since there would be a big chance that two instances are spawned at the same time. The random seed algorithm could be changed, so that the signal detection is done every 90 seconds instead of 180 seconds. This would allow faster reaction times, while still allowing space for the system to adapt to the current load. The second algorithm will not spawn more than one virtual machine at once when there is a signal presence in the network, but with reducing the time frame the reaction times are brought to a minimum, thus improving the overall performance of future prototypes.

7.3 Future work

The initial goal of this study was to explore how one could design and implement an adaptive cloud model based on a biological one. Although this goal has been fulfilled, there are many other things that can be considered affecting the way of how this goal is achieved. It would be interesting to explore the difference between a technical model that is based on an artificial immune system or human hormone system, when compared to the presented technical model which was based on the process of cellular differentiation. Although many of these biological systems share the same characteristics, the designed technical models could have a completely different implementation, therefore providing divergent results.

For the current technical model, there are many things that can be improved and functionalities that can be extended. One example would be to extend the signalling capabilities, so that there are different signal types depending on the incoming web traffic load which would provide a model that has a predictive characteristics. Furthermore, the HAProxy configuration in this project was managed manually due to time constraints. Future models could be extended to provide an automatically manageable load balancing configuration. During the transformation process, the individual server could SSH into the HAProxy server and add or remove itself from the list of available web servers.

Another issue that is of crucial importance is the management of puppet certificates. The signing of puppet certificates for authentication was done in an automated manner for all elements belonging in a particular domain. This is a known security vulnerability which needs to be fixed if the framework is going to be deployed in a production environment. The signing

of the certificates for known hosts can be done in a similar manner by using SSH to execute commands on the Puppet master. The removing of old and signed puppet certificates was also done manually during the experimenting process which proved to be inefficient and time consuming. Both of these puppet certificate issues could be addressed in a separate script or inside of the existing framework designs.

As previously mentioned, the Ubuntu and Puppet versions used in this experiment were not the latest stable versions. The same environment could be set up with newer versions of the software and operating system to explore if there will be any performance improvement and difference in the results. Additionally, more methods could be implemented as part of the monitoring script that would fetch additional system parameters. Having different types of local parameters would provide a better overview on the overall system performance.

A different approach could be also taken, where a completely different operating system is used to host the web servers. For example, with the same available cloud resources instead of running ten Ubuntu servers, one could more virtual machines that use a smaller and less demanding version of Linux i.e. TinyCore Linux. Using a stripped-down Linux version might provide better overall performance when used only for a specific function. As previously noted, a comparison of the results of this study with some of the other biological models discussed, could be also completed in a future study where the focus will be shifted from the model designing to algorithm optimization and an in-depth result comparison. This comparison would bring additional insight into how the current algorithms can be improved. These are only a few things that have been thought of while working on this project and surely there will be a lot more ideas and improvements that will emerge when working on future projects related to this study.

Chapter 8

Conclusion

The primary goal of this study was to investigate how a biomorphic model can be designed and implemented in order to achieve adaptive system behaviour in the cloud.

The biological model which was chosen as a foundation for the designed technical model is cellular differentiation. Cellular differentiation is a process where a particular cell changes from one type to another by altering gene expressions inside the cells structure. These changes can be triggered by releasing chemical signals in the surrounding environment.

We defined multiple analogies between blank servers which act as stem cells, and web servers which act as cells of a specific function. The change in the cells gene expressions results in a change of the cells structure, whereas a puppet configuration run results in a change of the servers configuration. The signalling process in which cells emit chemical signals is also part of the technical model where web servers send TCP signals to the stem cell network. The stem cells have an ability to detect the a particular chemical presence on their membranes and so do the blank servers by running the developed framework to detect TCP signals on their Ethernet ports.

The technical part of the problem statement is addressed by developing two different approaches for adopting the biomorphic model in order to solve the cloud scaling issue. For both prototypes, an identical underlying infrastructure was deployed using the developed deployment framework, along with a number of web servers and blank servers which run a signal and a sensor prototype, respectively.

With the first approach, a prototype was developed which uses a random timer mechanism each time it receives a TCP signal. On the other hand, the second approach was based on a concept where a random number is generated from an identical seed which was planted on each stem cell server in order to coordinate the transformation procedure across the system. The obtained results from the system expansion show that both prototypes can outperform each other in different scenarios. The difference

in the gathered results is mainly based on the amount of the targeted web traffic and the smallest generated timer of the first prototype.

Bibliography

[1] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. "An adaptive hybrid elasticity controller for cloud infrastructures." In: *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE. 2012, pp. 204–212.

[2] Michael Armbrust et al. "A view of cloud computing." In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[3] CFEngine AS. *CFEngine*. 2015. URL: http://cfengine.com/ (visited on 03/14/2015).

[4] Ozalp Babaoglu, Hein Meling, and Alberto Montresor. "Anthill: A framework for the development of agent-based peer-to-peer systems." In: *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*. IEEE. 2002, pp. 15–22.

[5] Kyrre Begnum. *Manage Large Networks - MLN*. 2015. URL: http://mln.sourceforge.net/ (visited on 03/13/2015).

[6] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: from natural to artificial systems*. 1. Oxford university press, 1999.

[7] Uwe Brinkschulte and Mathias Pacher. "An Agressive Strategy for an Artificial Hormone System to Minimize the Task Allocation Time." In: *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2012 15th IEEE International Symposium on*. IEEE. 2012, pp. 188–195.

[8] Uwe Brinkschulte, Mathias Pacher, and Alexander von Renteln. "An artificial hormone system for self-organizing real-time task allocation in organic middleware." In: *Organic Computing*. Springer, 2008, pp. 261–283.

[9] V Chaudhary et al. "A comparison of virtualization technologies for HPC." In: *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*. IEEE. 2008, pp. 861–868.

[10] Trieu C Chieu et al. "Dynamic scaling of web applications in a virtualized cloud computing environment." In: *e-Business Engineering, 2009. ICEBE'09. IEEE International Conference on*. IEEE. 2009, pp. 281–286.

[11] Susanta Nanda Tzi-cker Chiueh and Stony Brook. "A survey on virtualization technologies." In: *RPE Report* (2005), pp. 1–42.

[12] Mate J Csorba, Hein Meling, and Poul E Heegaard. “Ant system for service deployment in private and public clouds.” In: *Proceedings of the 2nd workshop on Bio-inspired algorithms for distributed systems*. ACM. 2010, pp. 19–28.

[13] Divyata Dal et al. “Evolution induced secondary immunity: An artificial immune system based intrusion detection system.” In: *Computer Information Systems and Industrial Management Applications, 2008. CISIM'08. 7th*. IEEE. 2008, pp. 65–70.

[14] Jeff Daniels. “Server virtualization architecture and implementation.” In: *Crossroads* 16.1 (2009), pp. 8–12.

[15] Richard Dawkins. *The blind watchmaker: Why the evidence of evolution reveals a universe without design*. WW Norton & Company, 1996.

[16] Gianni Di Caro and Marco Dorigo. “AntNet: Distributed Stigmergetic Control for Communications Networks.” In: *J. Artif. Intell. Res.(JAIR)* 9 (1998), pp. 317–365.

[17] Tharam Dillon, Chen Wu, and Elizabeth Chang. “Cloud computing: issues and challenges.” In: *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. Ieee. 2010, pp. 27–33.

[18] Shridhar G Domanal, Ram Mohana Reddy Guddeti, et al. “A Novel Bio-Inspired Load Balancing of Virtualmachines in Cloud Environment.” In: *Cloud Computing in Emerging Markets (CCEM), 2014 IEEE International Conference on*. IEEE. 2014, pp. 1–4.

[19] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. “Ant colony optimization.” In: *Computational Intelligence Magazine, IEEE* 1.4 (2006), pp. 28–39.

[20] Russ C Eberhart and James Kennedy. “A new optimizer using particle swarm theory.” In: *Proceedings of the sixth international symposium on micro machine and human science*. Vol. 1. New York, NY. 1995, pp. 39–43.

[21] Patricia Takako Endo et al. “Self-organizing strategies for resource management in Cloud Computing: State-of-the-art and challenges.” In: *Cloud Computing and Communications (LatinCloud), 2nd IEEE Latin American Conference on*. IEEE. 2013, pp. 13–18.

[22] 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.

[23] The Linux Foundation. *The Xen Project, the powerful open source industry standard for virtualization.* 2015. URL: http://www.xenproject.org/ (visited on 03/25/2015).

[24] Yongqiang Gao et al. “A multi-objective ant colony system algorithm for virtual machine placement in cloud computing.” In: *Journal of Computer and System Sciences* 79.8 (2013), pp. 1230–1242.

[25] Selvin George, David Evans, and Steven Marchette. "A biological programming model for self-healing." In: *Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems: in association with 10th ACM Conference on Computer and Communications Security*. ACM. 2003, pp. 72–81.

[26] Google. *Google Cloud Computing, Hosting Services & Cloud Support — Google Cloud Platform*. 2015. URL: https://cloud.google.com/ (visited on 03/25/2015).

[27] Google. *Search Engine Strategies Conference*. 2006. URL: http://www.google.com/press/podium/ses2006.html (visited on 03/15/2015).

[28] Irfan Habib. "Virtualization with kvm." In: *Linux Journal* 2008.166 (2008), p. 8.

[29] Rania Hassan et al. "A comparison of particle swarm optimization and the genetic algorithm." In: *Proceedings of the 1st AIAA multidisciplinary design optimization specialist conference*. 2005, pp. 18–21.

[30] Geir Horn et al. "Analysing the Lifecycle of Future Autonomous Cloud Applications." In: ().

[31] IBM. *Virtualization in Education*. 2007. URL: http://www-07.ibm.com/solutions/in/education/download/Virtualization%20in%20Education.pdf (visited on 03/25/2015).

[32] Autonomic IBM Computing et al. "An architectural blueprint for autonomic computing." In: *IBM White Paper* (2006).

[33] Chef Software Inc. *Chef : IT automation for speed and awesomeness*. 2015. URL: https://www.chef.io/chef/ (visited on 03/14/2015).

[34] GitHub Inc. *GitHub · Build software better, together.* 2015. URL: https://github.com/ (visited on 03/14/2015).

[35] Rackspace US Inc. *Rackspace Managed Cloud Services - More than just infrastructure*. 2015. URL: https://www.rackspace.com/ (visited on 03/29/2015).

[36] VMware Inc. 2015. URL: http://www.vmware.com/products/workstation (visited on 03/25/2015).

[37] VMware Inc. *vSphere ESXi Bare-Metal Hypervisor*. 2015. URL: http://www.vmware.com/products/esxi-and-esx/overview (visited on 03/25/2015).

[38] Zachary G Ives et al. "An adaptive query execution system for data integration." In: *ACM SIGMOD Record*. Vol. 28. 2. ACM. 1999, pp. 299–310.

[39] Derrick Kondo et al. "Cost-benefit analysis of cloud computing versus desktop grids." In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–12.

[40] Spencer Krum et al. *Pro Puppet*. 2nd. Berkely, CA, USA: Apress, 2013. ISBN: 1430260408, 9781430260400.

[41] KVM. *Kernel Based Virtual Machine*. 2015. URL: http://www.linux-kvm.org/page/Main_Page (visited on 03/25/2015).

[42] Puppet Labs. *Puppet Labs: IT Automation Software for System Administrators*. 2015. URL: https://puppetlabs.com/ (visited on 03/14/2015).

[43] Kenneth N Lodding. "The hitchhiker's guide to biomorphic software." In: *Queue* 2.4 (2004), p. 66.

[44] Pramote Luenam and Peng Liu. "The design of an adaptive intrusion tolerant database system." In: *Foundations of Intrusion Tolerant Systems*. IEEE Computer Society. 2003, pp. 14–14.

[45] Sean Marston et al. "Cloud computing—The business perspective." In: *Decision Support Systems* 51.1 (2011), pp. 176–189.

[46] Polly Matzinger. "Tolerance, danger, and the extended family." In: *Annual review of immunology* 12.1 (1994), pp. 991–1045.

[47] Michael Meisel, Vasileios Pappas, and Lixia Zhang. "A taxonomy of biologically inspired research in computer networking." In: *Computer Networks* 54.6 (2010), pp. 901–916.

[48] Peter Mell and Tim Grance. "Effectively and securely using the cloud computing paradigm." In: *NIST, Information Technology Laboratory* (2009), pp. 304–311.

[49] Microsoft. *Microsoft Azure: Cloud Computing Platform & Services*. 2015. URL: https://azure.microsoft.com/en-us/ (visited on 03/29/2015).

[50] Microsoft. *Virtualization for your modern datacenter and hybrid cloud*. 2015. URL: http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx (visited on 03/25/2015).

[51] Microsoft. *Windows Virtual PC*. 2015. URL: http://www.microsoft.com/en-us/download/details.aspx?id=3702 (visited on 03/25/2015).

[52] Alberto Montresor and Ozalp Babaoglu. "Biology-inspired approaches to peer-to-peer computing in bison." In: *Intelligent Systems Design and Applications*. Springer, 2003, pp. 515–522.

[53] Alberto Montresor, Hein Meling, and Özalp Babaoğlu. "Messor: Load-balancing through a swarm of autonomous agents." In: *Agents and Peer-to-Peer Computing*. Springer, 2003, pp. 125–137.

[54] Brice Morin et al. "Models@ run. time to support dynamic adaptation." In: *Computer* 42.10 (2009), pp. 44–51.

[55] US Department of Health National Institutes of Health and Human Services. *Stem Cell Information*. 2015. URL: http://stemcells.nih.gov/info/basics/pages/basics4.aspx (visited on 04/25/2015).

[56] Daniel Nurmi et al. "The eucalyptus open-source cloud-computing system." In: *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*. IEEE. 2009, pp. 124–131.

[57] Oracle. *Oracle VM VirtualBox*. 2015. URL: https://www.virtualbox.org (visited on 03/25/2015).

[58] Oslo and Akershus University College of Applied Sciences. *HiOA is getting its own cloud*. 2015. URL: http://www.hioa.no/eng/News/HiOA-is-getting-its-own-cloud (visited on 03/23/2015).

[59] Chung-Ming Ou, Yao-Tien Wang, and Chung-Ren Ou. "Intrusion detection systems adapted from agent-based artificial immune systems." In: *Fuzzy Systems (FUZZ), 2011 IEEE International Conference on*. IEEE. 2011, pp. 115–122.

[60] Suraj Pandey et al. "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments." In: *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. IEEE. 2010, pp. 400–407.

[61] The OpenStack project. *OpenStack Open Source Cloud Computing Software*. 2015. URL: https://www.openstack.org/ (visited on 03/19/2015).

[62] MIT Technology Review. *Who Coined 'Cloud Computing'?* 2011. URL: http://www.technologyreview.com/news/425970/who-coined-cloud-computing/ (visited on 03/13/2015).

[63] Ichiro Satoh. "Self-Adaptive Resource Allocation in Cloud Applications." In: *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. IEEE Computer Society. 2013, pp. 179–186.

[64] Peter Sempolinski and Douglas Thain. "A comparison and critique of eucalyptus, opennebula and nimbus." In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. Ieee. 2010, pp. 417–426.

[65] Amazon Web Services. *Amazon elastic compute cloud*. 2015. URL: https://aws.amazon.com/ec2/ (visited on 03/08/2015).

[66] Amazon Web Services. *Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta*. 2006. URL: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/ (visited on 03/13/2015).

[67] Borja Sotomayor et al. "Virtual infrastructure management in private and hybrid clouds." In: *Internet computing, IEEE* 13.5 (2009), pp. 14–22.

[68] Thomas Stutzle and Holger Hoos. "MAX-MIN ant system and local search for the traveling salesman problem." In: *Evolutionary Computation, 1997., IEEE International Conference on*. IEEE. 1997, pp. 309–314.

[69] Devika Subramanian, Peter Druschel, and Johnny Chen. "Ants and reinforcement learning: A case study in routing in dynamic networks." In: *IJCAI (2)*. Citeseer. 1997, pp. 832–839.

[70] Michael Wine - The New York Times. *A Youth's Passion for Computers Gone Sour*. 1988. URL: http://www.vmware.com/consolidation/consolidate.html (visited on 03/15/2015).

[71] Wolfgang Trumler, Tobias Thiemann, and Theo Ungerer. "An artificial hormone system for self-organization of networked nodes." In: *Biologically Inspired Cooperative Computing*. Springer, 2006, pp. 85–94.

[72] University of Utah. *Epigenetics*. 2015. URL: http://learn.genetics.utah.edu/content/epigenetics/ (visited on 04/25/2015).

[73] VMWare. *Cloud Computing is vCloud Air by VMware*. 2015. URL: http://vcloud.vmware.com/uk/ (visited on 03/25/2015).

[74] VMWare. *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. 2015. URL: http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf (visited on 04/01/2015).

[75] VMware. *Server Consolidation*. 2015. URL: http://www.vmware.com/consolidation/consolidate.html (visited on 03/08/2015).

[76] Gregor Von Laszewski et al. "Comparison of multiple cloud frameworks." In: *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE. 2012, pp. 734–741.

[77] Michael Wang and Tatsuya Suda. "The bio-networking architecture: A biologically inspired approach to the design of scalable, adaptive, and survivable/available network applications." In: *Applications and the Internet, 2001. Proceedings. 2001 Symposium on*. IEEE. 2001, pp. 43–53.

[78] Matt Weinberger. *Three Open Source-Based Cloud Alternatives to OpenStack*. 2015. URL: http://siliconangle.com/blog/2012/04/24/three-open-source-based-cloud-alternatives-to-openstack/ (visited on 03/25/2015).

[79] Xiaolong Wen et al. "Comparison of open-source cloud management platforms: OpenStack and OpenNebula." In: *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on*. IEEE. 2012, pp. 2457–2461.

[80] Andrew J Younge et al. "Analysis of virtualization technologies for high performance computing environments." In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE. 2011, pp. 9–16.

Appendices

Appendix A

Puppet and MLN files

A.1 MLN - build.mln

build.mln

```
global {
	project final
}

superclass funccell {
	openstack {
		keypair puppet_key
		image ubuntu-12.04
		flavor m1.tiny
		user_data {
			echo "192.168.50.2 puppet puppet.stemcell" >> /etc/hosts
			echo "192.168.51.2 puppet puppet.funccell" >> /etc/hosts
			ifconfig eth1 up
			dhclient eth1
			echo "nameserver 8.8.8.8" > /etc/resolvconf/resolv.conf.d/head
			echo "nameserver 8.8.8.8" > /etc/resolv.conf
			apt-get update
			wget http://opas3n.com/init_hostname.sh
			bash init_hostname.sh func
			apt-get install puppet -y
			puppet agent --test puppet
		}

	}

	network eth0 {
		net stem_cell_network
		address dhcp
	}
	network eth1 {
		net function_cell_network
		address dhcp
	}
}

superclass stemcell {
	openstack {
		keypair puppet_key
		image ubuntu-12.04
		flavor m1.tiny
		user_data {
			echo "192.168.50.2 puppet puppet.stemcell" >> /etc/hosts
			echo "192.168.51.2 puppet puppet.funccell" >> /etc/hosts
			ifconfig eth1 up
			dhclient eth1
			echo "nameserver 8.8.8.8" > /etc/resolvconf/resolv.conf.d/head
```

```
47                      echo "nameserver 8.8.8.8" > /etc/resolv.conf
48                      apt-get update
49                      wget http://opas3n.com/init_hostname.sh
50                      bash init_hostname.sh stem
51                      apt-get install puppet -y
52                      puppet agent --test puppet
53                  }
54
55          }
56
57          network eth0 {
58                  net stem_cell_network
59                  address dhcp
60          }
61          network eth1 {
62                  net function_cell_network
63                  address dhcp
64          }
65  }
66
67  host server1 {
68      superclass stemcell
69  }
70
71  host server2 {
72      superclass stemcell
73  }
74
75  host server3 {
76      superclass stemcell
77  }
78
79  host server4 {
80      superclass stemcell
81  }
82
83  host server5 {
84      superclass stemcell
85  }
86
87  host server6 {
88      superclass stemcell
89  }
90
91  host server7 {
92      superclass stemcell
93  }
94
95  host server8 {
96      superclass funccell
97  }
98
99  host server9 {
100     superclass funccell
101 }
102
103 host server10 {
104     superclass funccell
105 }
```

A.2 Puppet

A.2.1 Functional .pp

server1_func.pp

```
node 'server1.func'{
$php_packages = ["php5","libapache2-mod-php5","php5-mcrypt"]
$apache_default = "filecontent"
	package { 'apache2':
		ensure => present,
	}

	package {$php_packages:
		ensure => "present",
	}

	package { 'git':
		ensure => "present",
	}->

	exec { "git init":
		cwd => "/root",
		require => [Package["git"], Package["apache2"]],
		command => "git init",
		path => "/usr/bin:/bin",
	}

	exec { "git clone":
		cwd => "/tmp",
		require => [Package["git"], Package["apache2"]],
		command => "git clone https://github.com/opas3n/Netlab.git",
		path => "/usr/bin:/bin",
	}

	exec { "Fix interface":
		cwd => "/tmp",
		require => [Exec["git clone"]],
		command => "cp /tmp/Netlab/interfaces /etc/network/interfaces",
		path => "/usr/bin:/bin",
	}

	exec { "apache-conf":
		cwd => "/tmp",
		require => [Exec["git clone"]],
		command => "cp /tmp/Netlab/default /etc/apache2/sites-available/default",
		path => "/usr/bin:/bin",
	}->
	service { 'apache2':
		ensure => running,
		enable => true,
	}

	exec { "apply-index":
		cwd => "/tmp",
		require => [Exec["git clone"]],
		command => "cp /tmp/Netlab/index.php /var/www/index.php",
		path => "/usr/bin:/bin",
	}

	exec { "apply-check":
		cwd => "/tmp",
		require => [Exec["git clone"]],
		command => "cp /tmp/Netlab/check.html /var/www/check.html",
		path => "/usr/bin:/bin",
	}

	exec { "remove-html-index":
```

```
63          cwd => "/tmp",
64          require => [Exec["git clone"]],
65          command => "rm /var/www/index.html",
66          path => "/usr/bin:/bin",
67      }
68
69      exec { "run-signal":
70          cwd => "/tmp",
71          require => [Exec["git clone"]],
72          command => "python signal.py",
73          path => "/usr/bin:/bin",
74      }
75      exec { "run-monitor":
76          cwd => "/tmp",
77          require => [Exec["git clone"]],
78          command => "python monitor.py",
79          path => "/usr/bin:/bin",
80      }
81      exec { "run-reset":
82          cwd => "/tmp",
83          require => [Exec["git clone"]],
84          command => "python reset.py",
85          path => "/usr/bin:/bin",
86      }
87  }
```

A.2.2 Stem .pp

server1_stem.pp

```
1   node 'server1.stem'{
2   $php_packages = ["php5","libapache2-mod-php5","php5-mcrypt"]
3   $apache_default = "filecontent"
4       package { 'apache2':
5           ensure => absent,
6       }
7
8       package {$php_packages:
9           ensure => absent,
10      }
11
12      package { 'git':
13          ensure => absent,
14      }
15
16      file {'/tmp/Netlab':
17          ensure => absent,
18          path => '/tmp/Netlab',
19          recurse => true,
20          purge => true,
21          force => true,
22      }
23
24      package { 'ntp':
25          ensure => present,
26      }
27  }
```

Appendix B

Frameworks and configuration files

B.1 deploy.py

deploy.py

```
#!/usr/bin/python
import argparse
import os
import time

## Parsing all arguments given to the script
parser = argparse.ArgumentParser()
parser.add_argument("-P", "--project", help="Specify the project name for the MLN project",
required=True)
parser.add_argument("-S", "--snumber", help="Specify the number of stem cells/blank servers",
type=int, required=True)
parser.add_argument("-F", "--fnumber", help="Specify the number of functional cells/web servers",
type=int, required=True)
parser.add_argument("-H", "--hostname", help="This is the basis of the hostname for the
deployed nodes, if not entered,the basis will be the same as the project name")
parser.add_argument("-K", "--keypair",
help="Specify the keypair which will be used to access the servers")
parser.add_argument("-O", "--os", help="Enter this parameter in case you
want to deploy a different OS, the exact name of the OS image has to be entered")
parser.add_argument("-T", "--flavor", help="Choose one of the available choices
if you want to use a different OpenStack flavor",
choices=['m1.tiny','m1.small','m1.medium','m1.large','m1.xlarge'])

args = parser.parse_args()

project = args.project
stem_cell_number = args.snumber
func_cell_number = args.fnumber
hostname = args.hostname
keypair = args.keypair
os_image = args.os
flavor = args.flavor

def CreateHosts(stem_cell_number,func_cell_number,hostname):
    # Get total ammount of servers
    total = stem_cell_number + func_cell_number
    # Counters for servers
    count_f=func_cell_number
    count_s=stem_cell_number

    if hostname==None:
        hostname = project.lower()
```

```
43          else:
44              hostname = hostname.lower()
45
46          os.popen("rm cells.mln")
47          w_file = open("cells.mln", 'a')
48
49          for i in range(1,total+1):
50              if count_s > 0:
51                  count_s = count_s - 1
52                  w_file.write("host " + hostname + str(i) + " {\ n")
53                  w_file.write("    superclass stemcell \ n")
54                  w_file.write("}\ n \ n")
55              elif count_f > 0:
56                  count_f = count_f - 1
57                  w_file.write("host " + hostname + str(i) + " {\ n")
58                  w_file.write("    superclass funccell \ n")
59                  w_file.write("}\ n \ n")
60
61          w_file.close()
62
63      def BuildTemplate(project,keypair,flavor,os_image):
64          r_file = open("skeleton.mln",'r')
65          output = r_file.readlines()
66          output_save = []
67          for line in output:
68              if "project" in line:
69                  x = line.replace("projectname",project)
70                  output_save.append(x)
71              elif "keypair" in line:
72                  if keypair != None:
73                      x = line.replace("puppet_key",keypair)
74                      output_save.append(x)
75                  else:
76                      output_save.append(line)
77              elif "flavor" in line:
78                  if flavor != None:
79                      x = line.replace("m1.tiny",flavor)
80                      output_save.append(x)
81                  else:
82                      output_save.append(line)
83              elif "image" in line:
84                  if os_image != None:
85                      x = line.replace("ubuntu-12.04",os_image)
86                      output_save.append(x)
87                  else:
88                      output_save.append(line)
89              else:
90                  output_save.append(line)
91          r_file.close()
92
93
94          w_file = open('build.mln', 'w')
95          for line in output_save:
96              w_file.write(line)
97          w_file.close()
98          os.popen("cat cells.mln >> build.mln")
99
100
101
102     def StartDeployment(project,stem_cell_number,func_cell_number,hostname,keypair,os_image,flavor):
103         CreateHosts(stem_cell_number,func_cell_number,hostname)
104         BuildTemplate(project,keypair,flavor,os_image)
105         os.popen("mln build -f build.mln")
106         os.popen("mln start -p " + project)
107
108     print project, stem_cell_number, func_cell_number, hostname, keypair, os_image, flavor
109     StartDeployment(project,stem_cell_number,func_cell_number,hostname,keypair,os_image,flavor)
```

B.2 createpp.py

createpp.py

```
#! /usr/bin/python
import os

def createStemPP():
	for i in range(1,11):
		sed = "sed -e ’s/server" +str(i)
		sed += "/server" + str(i+1) +"/g;n’"
		output = os.popen("cat server"+ str(i) +"_stem.pp | " + str(sed)
		+ " > server" + str(i+1) + "_stem.pp")
		output = output.readlines()
		sed =""

def createFuncPP():
	for i in range(1,11):
		sed = "sed -e ’s/server" +str(i)
		sed += "/server" + str(i+1) +"/g;n’"
		output = os.popen("cat server"+ str(i) +"_func.pp | "
		+ str(sed) + " > server" + str(i+1) + "_func.pp")
		output = output.readlines()
		sed =""

createStemPP()
createFuncPP()

```

B.3 signal.py

signal.py

```
#! /usr/bin/python
from __future__ import division
import os
import socket
import time

def FetchMemory():
	memory_in_use = os.popen("free | grep Mem | awk ’print $3/$2 * 100.0’")
	free_memory = os.popen("free | grep Mem | awk ’print $4/$2 * 100.0’")
	memory_in_use = memory_in_use.readlines()
	free_memory = free_memory.readlines()

	## Alternative is to use used memory
	#print memory_in_use[0].strip()
	free_mem = free_memory[0].strip()
	return free_mem

def FetchSystemLoad():
	output = os.popen("cat /proc/loadavg")
	output = output.readlines()
	avg_values = output[0].strip().split()
	avg_1_min = avg_values[0]
	avg_5_min = avg_values[1]
	avg_15_min = avg_values[2]
	#print avg_1_min
	#print avg_5_min
	#print avg_15_min
	return float(avg_1_min)


def FetchResponseTime():
	output = os.popen("tail -n 10 /var/log/apache2/response_time.log")
	output = output.readlines()
```

```
        request_size = len(output)
        seconds = []
        microseconds = []
        for line in output:
                seconds.append(int(line.strip().split()[0]))
                microseconds.append(int(line.strip().split()[1]))

        seconds_avg = sum(seconds)/request_size
        microseconds_avg = sum(microseconds)/request_size

        return seconds_avg

def Actions(util,memory,response_time):
        print "Average system load ", util
        print "Free memory in %", memory
        print "Average response time in seconds/microseconds ", response_time

        if float(util) >= 0.85 and float(memory)<=15 and int(response_time)>0:
                SendSignal()

def SendSignal():
        ips = os.popen("nmap -p T:10000 192.168.50.0/24 | grep -B 3 'open '
        | grep report | grep -Eo '[0-9]1,3\.[0-9]1,3\.[0-9]1,3\.[0-9]1,3'
        | egrep -Ev '192.168.51.2|192.168.51.4'")
        all_ip = []
        for ip in ips.readlines():
                all_ip.append(ip.strip())

        for ip in all_ip:
                print ip
                os.popen("echo 'activate' | nc " + ip + " 10000")

def Main():
        while 1:
                Actions(FetchSystemLoad(),FetchMemory(),FetchResponseTime())
                time.sleep(10)
Main()

```

B.4 listen-1.py

listen-1.py

```
#! /usr/bin/python
import os
import time
import re
import random

def GetIPInterface():
        ## Regex matching the correct interface belonging in the stem subnet
        regex_py = "\ d+\.\ d+\.50 \.\ d+"
        regexpy = re.compile(regex_py)

        ## Fetching IP for eth0
        eth0 = os.popen("/sbin/ifconfig eth0 | grep 'inet addr:' | cut -d: -f2 | awk '{ print $1}'")
        eth0 = eth0.readlines()[0]
        eth0 = eth0.strip()

        ## Fetching IP for eth1
        eth1 = os.popen("/sbin/ifconfig eth1 | grep 'inet addr:' | cut -d: -f2 | awk '{ print $1}'")
        eth1 = eth1.readlines()[0]
        eth1 = eth1.strip()

        ## Running regex to get IP of stem cell network
        for interface in [eth0,eth1]:
```

```
24          result = regexpy.search(interface)
25          if result:
26              final_IP = result.group()
27      return final_IP
28
29  print GetIPInterface()
30
31  def ListenInterface():
32      output = os.popen("nc -l 10000")
33      output=output.readlines()[0].strip()
34      if output =="activate":
35          timer = GenerateTimer()
36          timer = timer * 60
37          time.sleep(timer)
38
39          #print "confirmation signal"
40          signal = os.popen("nc -l 10000")
41          signal=signal.readlines()[0].strip()
42          time.sleep(20)
43          if signal=='activate':
44              SetHostname(reset)
45              SetHostname(func)
46              os.popen("puppet agent --test puppet")
47              return
48          else:
49              os.system("killall -9 nc")
50              ListenInterface()
51
52  def SetHostname(hosttype):
53      if hosttype=='reset':
54          output=os.popen("HOSTNAME=`hostname | cut -d. -f1`;
55          echo $HOSTNAME; echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
56      elif hosttype=='stem':
57          output=os.popen("HOSTNAME=`cat /etc/hostname`; HOSTNAME=$HOSTNAME.stem;
58          echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
59      elif hosttype=='func':
60          output=os.popen("HOSTNAME=`cat /etc/hostname`; HOSTNAME=$HOSTNAME.func;
61          echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
62      else:
63          pass
64
65  def GenerateTimer():
66      return random.randint(1,11)
67
68  ListenInterface()
```

B.5 listen-2.py

listen-2.py

```
1   #! /usr/bin/python
2   import os
3   import time
4   import re
5   import random
6   random.seed( 100 )
7
8   def GetIPInterface():
9       ## Regex matching the correct interface belonging in the stem subnet
10      regex_py = "\ d+\.\ d+\.50 \.\ d+"
11      regexpy = re.compile(regex_py)
12
13      ## Fetching IP for eth0
14      eth0 = os.popen("/sbin/ifconfig eth0 | grep 'inet addr:' | cut -d: -f2 | awk '{ print $1}'")
15      eth0 = eth0.readlines()[0]
16      eth0 = eth0.strip()
```

```
17
18          ## Fetching IP for eth1
19          eth1 = os.popen("/sbin/ifconfig eth1 | grep 'inet addr:' | cut -d: -f2 | awk '{ print $1}'")
20          eth1 = eth1.readlines()[0]
21          eth1 = eth1.strip()
22
23          ## Running regex to get IP of stem cell network
24          for interface in [eth0,eth1]:
25                result = regexpy.search(interface)
26                if result:
27                      final_IP = result.group()
28          return final_IP
29
30    print GetIPInterface()
31
32    def SetHostname(hosttype):
33          if hosttype=='reset':
34                output=os.popen("HOSTNAME=`hostname | cut -d. -f1`;
35                echo $HOSTNAME; echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
36          elif hosttype=='stem':
37                output=os.popen("HOSTNAME=`cat /etc/hostname`; HOSTNAME=$HOSTNAME.stem;
38                echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
39          elif hosttype=='func':
40                output=os.popen("HOSTNAME=`cat /etc/hostname`; HOSTNAME=$HOSTNAME.func;
41                echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
42          else:
43                pass
44
45    def ListenInterface():
46          output = os.popen("nc -l 10000")
47          output=output.readlines()[0].strip()
48          if output =="activate":
49                hostnamenumber = os.popen('hostname | cut -f1 -d. | tail -c 2')
50                hostnamenumber = hostnamenumber.readlines()
51                hostnamenumber = hostnamenumber[0].strip()
52                number = random.randint(1,10)
53                if str(number) == str(hostnamenumber):
54                      SetHostname(reset)
55                      SetHostname(func)
56                      os.popen("puppet agent --test puppet")
57                      return
58                else:
59                      os.system("killall -9 nc")
60                      time.sleep(180)
61                      ListenInterface()
62
63    ListenInterface()
```

B.6 reset.py

reset.py

```
1     #! /usr/local/python
2     import time
3     import os
4
5     counter = 0
6     def SetHostname(hosttype):
7           if hosttype=='reset':
8                 output=os.popen("HOSTNAME=`hostname | cut -d. -f1`;
9                 echo $HOSTNAME; echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
10          elif hosttype=='stem':
11                output=os.popen("HOSTNAME=`cat /etc/hostname`; HOSTNAME=$HOSTNAME.stem;
12                echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
13          elif hosttype=='func':
14                output=os.popen("HOSTNAME=`cat /etc/hostname`; HOSTNAME=$HOSTNAME.func;
```

```
15                 echo $HOSTNAME > /etc/hostname; hostname $HOSTNAME;")
16         else:
17                 pass
18 def FetchMemory():
19         memory_in_use = os.popen("free | grep Mem | awk 'print $3/$2 * 100.0'")
20         free_memory = os.popen("free | grep Mem | awk 'print $4/$2 * 100.0'")
21         memory_in_use = memory_in_use.readlines()
22         free_memory = free_memory.readlines()
23
24         ## Alternative is to use used memory
25         #print memory_in_use[0].strip()
26         free_mem = free_memory[0].strip()
27         return free_mem
28
29 def FetchSystemLoad():
30         output = os.popen("cat /proc/loadavg")
31         output = output.readlines()
32         avg_values = output[0].strip().split()
33         avg_1_min = avg_values[0]
34         avg_5_min = avg_values[1]
35         avg_15_min = avg_values[2]
36         #print avg_1_min
37         #print avg_5_min
38         #print avg_15_min
39         return float(avg_1_min)
40
41
42 def FetchResponseTime():
43         output = os.popen("tail -n 10 /var/log/apache2/response_time.log")
44         output = output.readlines()
45         request_size = len(output)
46         seconds = []
47         microseconds = []
48         for line in output:
49                 seconds.append(int(line.strip().split()[0]))
50                 microseconds.append(int(line.strip().split()[1]))
51
52         seconds_avg = sum(seconds)/request_size
53         microseconds_avg = sum(microseconds)/request_size
54
55         return seconds_avg
56
57 def Decision(util,memory,response_time,counter):
58         if float(util) <= 0.10 and float(memory)>=80 and int(response_time)==0:
59                 counter = counter + 1
60                 if counter >= 5:
61                         Remove()
62                 else:
63                         counter = 0
64         return counter
65
66 def Remove():
67         SetHostname("reset")
68         SetHostname("stem")
69         os.popen("puppet agent --test puppet")
70         return
71
72 def Main():
73         while 1:
74                 Decision(FetchSystemLoad(),FetchMemory(),FetchResponseTime(),counter)
75                 time.sleep(120)
76
```

B.7 monitor.py

monitor.py

```
#! /usr/bin/python
from __future__ import division
import os
import paramiko
import time

def FetchMemory():
    memory_in_use = os.popen("free | grep Mem | awk ’print $3/$2 * 100.0’")
    free_memory = os.popen("free | grep Mem | awk ’print $4/$2 * 100.0’")
    memory_in_use = memory_in_use.readlines()
    free_memory = free_memory.readlines()

    ## Alternative is to use used memory
    #print memory_in_use[0].strip()
    free_mem = free_memory[0].strip()
    return free_mem

def FetchSystemLoad():
    output = os.popen("cat /proc/loadavg")
    output = output.readlines()
    avg_values = output[0].strip().split()
    avg_1_min = avg_values[0]
    avg_5_min = avg_values[1]
    avg_15_min = avg_values[2]
    #print avg_1_min
    #print avg_5_min
    #print avg_15_min
    return float(avg_1_min)


def FetchResponseTime():
    output = os.popen("tail -n 10 /var/log/apache2/response_time.log")
    output = output.readlines()
    request_size = len(output)
    seconds = []
    microseconds = []
    for line in output:
        seconds.append(int(line.strip().split()[0]))
        microseconds.append(int(line.strip().split()[1]))

    seconds_avg = sum(seconds)/request_size
    microseconds_avg = sum(microseconds)/request_size

    return seconds_avg

def CreateData():
    hostname = os.popen("hostname")
    hostname = hostname.readlines()[0].strip()
    epoch_time = int(time.time())

    data = str(FetchSystemLoad()) + " " + str(FetchMemory()) + " "
    + str(FetchResponseTime()) + " " + hostname + " " + str(epoch_time)

    ip = "188.226.151.105"
    user = "root"
    passwd = "mypassword"
    cmd = "echo " + data + " >> data.dat"

    s = paramiko.SSHClient()
    s.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    s.connect(ip, 22, user, passwd)
    stdin, stdout, stderr = s.exec_command(cmd)
    result=stdout.read()
    s.close()
    return result
```

```
66 print CreateData()
```

B.8 haproxy.cfg

haproxy.cfg

```
1  # this config needs haproxy-1.1.28 or haproxy-1.2.1
2
3  global
4          log 127.0.0.1   local0
5          log 127.0.0.1   local1 notice
6          #log loghost    local0 info
7          maxconn 4096
8          #chroot /usr/share/haproxy
9          user haproxy
10         group haproxy
11         daemon
12         #debug
13         #quiet
14
15 defaults
16         log     global
17         mode    http
18         option  httplog
19         option  dontlognull
20         retries 3
21         option redispatch
22         maxconn 2000
23         contimeout      5000
24         clitimeout      50000
25         srvtimeout      50000
26
27 listen stats :2000
28     mode http
29     stats enable
30     stats hide-version
31     stats realm Haproxy Statistics
32     stats uri /
33
34 frontend web
35         bind *:80
36         default_backend back
37
38 backend back
39         server server1 192.168.51.6:80 check
40         server server2 192.168.51.7:80 check
41         server server3 192.168.51.8:80 check
42         server server4 192.168.51.9:80 check
43         server server5 192.168.51.10:80 check
44         server server6 192.168.51.11:80 check
45         server server7 192.168.51.12:80 check
46         server server8 192.168.51.14:80 check
47         server server9 192.168.51.13:80 check
48         server server10 192.168.51.5:80 check
49
50
51         http-check expect string Itworks
52         option httpchk GET /check.html
```