UiO : Department of Informatics
University of Oslo

Coordinating vertical and horizontal scaling for achieving differentiated QoS

Orchestrating resource allocation for real-time vs non-real-time QoS services using hybrid controller model

Bilal Ahmad

Master's Thesis Spring 2016

Coordinating vertical and horizontal scaling for achieving differentiated QoS

Bilal Ahmad

May 23, 2016

Acknowledgements

I wish to express my sincere appreciation and gratitude to the following people for their support and help during my work on this thesis:

- My supervisors Anis Yazidi and Hårek Haugerud for their encouragement, support and guidance during the whole thesis period.
- Amir Maqbool Ahmed for providing me great advice and technical support to overcome difficulties during the project.
- To Kyrre Begnum for his inspiring lectures and assignments during the master program.
- To all my fellow master students with whom I share many good memories with, for being supportive and helping me to develop new ideas.
- Finally, I wish to express my sincere appreciation to my family for their unconditional support and motivation.

Abstract

The growth in popularity of cloud computing, along with the rapid development of internet based technologies, has led to a paradigm shift in the way computing resources are provisioned. Computing resources are to a larger extent offered as services, and exposed via the internet in models as pay-as-you-go and on-demand. Cloud service providers are trying to reduce their operating costs while offering their services with higher quality by resorting to the concept of elasticity. However, this is challenging because of heterogeneous applications running on their infrastructure with arbitrary requirements in terms of resources.

This thesis investigates how this problem can be addressed by designing and implementing an autonomic provisioning controller based on elements from control theory to coordinate between real-time and non-real-time applications. Traditional approaches perform elasticity-decisions either based on monitoring the resource usage or merely based on the QoS of, solely, latency-critical applications, particularly web servers. Nevertheless, a broad class of applications have different SLA-constraints that does not fall under the same class of latency-critical applications. For instance batch processing is becoming more popular and posses QoS requirements that are defined in different manner than latency-critical applications. The novelty of this study is the focus on how to coordinate resource allocation between real-time and non-real-time applications. Furthermore, our approach is hybrid as it relies on both resource usage and QoS-requirements in the same time to drive the elasticity decisions. Another novelty of our work is orchestrating both vertical and horizontal scaling under the same framework.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing is an emerging technology and is becoming more popular, due to advantages such as elasticity and infinite computing resources. Companies are increasingly taking advantage of the benefits and moving their infrastructure to the cloud for reduced operational cost. Despite the advantages, cloud technologies poses security and privacy concerns [1].

Cloud technologies has been widely adopted by small and medium-sized enterprises (SMB). It is estimated that 78 % of small businesses in USA will have fully adopted cloud computing by 2020 [2].

Modern virtualization technologies such as Kernel-based Virtual Machine (KVM)[1], VMware[2] and Xen[3] allow virtual machines (VMs) with different operating systems and multiple services to run on a single physical hardware. The technologies provides functionality to consolidate servers to optimize efficiency, reduce power consumption and the environmental impact. Several studies that have been conducted show an average server utilization of resources in data centers ranges between 10 % and 50 % [3, 4, 5, 6, 7].

According to a study (2014) performed by Natural Resources Defense Council the main issue for energy saving is under-utilization of data centers. In addition, data centers has the fastest growing consumption of electricity in United States [8]. It is estimated that Google websearch servers often has a idleness of 30 % over 24 hour period [9]. If we imagine a cluster of 20,000 servers, this would mean the capacity of 6,000 servers is wasted. From a business perspective maximizing server consolidation would cut unnecessary energy and operation costs and increase return on investment (ROI).

[1] http://www.linux-kvm.org/
[2] http://www.vmware.com/products/vsphere/
[3] http://www.xenproject.org/

Since power consumption increases linearly with the number of physical machines, the main goal is to increase the utilization in each of the physical nodes for continued scaling. The concept of horizontal scaling lies on increasing the capacity by connecting multiple hardware or software entites and make them work as a single entity. While vertical scaling consist of adding more resources to a single node in a system. Horizontal scaling of VMs has been widely adopted by cloud providers because of its simplicity, as it does not require any extra support from the hypervisor. Horizontal elasticity is course-grained, which means that a CPU core can dynamically be leased to a VM for a certain amount of time. While vertical elasticity is fine-grained, fractions of a CPU core can be leased for as short as a few seconds [10]. There is little research that has fully addressed vertical elasticity because of the increased complexity [11].

Applications hosted on VMs have different demands when it comes to Quality of service (QoS). Real-time applications are for instance latency-critical and sensitive to unpredictable spikes in user access, even small amount of interference can cause significant Service-level agreement (SLA) violations. While non-real-time applications have less requirements to resources and can derive significant value even the tasks are occasionally disrupted. The problem with cloud computing is that there is no transparency, the cloud providers do not know what kind of applications are running on the infrastructure and their requirements to achieve a preferred QoS.

The aim of this paper is to explore, design and attempt to prototype an autonomic provisioning controller based on elements from *control theory*. Which is a mathematical model based on a feedback system, where states of a system is measured and compared to the desired one and changes are made accordingly. The controller will be used for managing QoS of heterogeneous application types. The focus lies on exposing available resources and increasing server utilization. Application level metrics of interactive and non-interactive applications will be used as an indicator of QoS. The controller will maintain real-time control and use elements from control theory for reactive resource allocation.

The potential gain of this thesis is more knowledge on how one can manage resources in data centres more efficiently. The results of this research may help system administrators to increase server utilization before scaling, since adding physical nodes increases power usage, complexity and costs in terms of maintenance.

1.1 Problem statement

The problem statement for this paper is divided into three main research questions, which will be used as a foundation for the work:

- *How can we create an autonomic provisioning controller based on elements from control theory to increase server utilization, and at the same time expose available resources to ensure QoS of real-time and non-real-time applications?*
- *How to coordinate vertical and horizontal scaling for better resource allocation?*
- *How to benefit from application level metrics in order to efficiently provision resources to interactive and non-interactive applications?*

Explanation of the terms:

Autonomic provisioning controller can be defined as a controller that without user-interaction automatically performs decisions to achieve high performance and resource utilization. By ensuring *QoS*, application level metrics such as response time is evaluated and used in decision-making.

Control theory, refers to dynamically control the behaviour of a system by using a reference value, which is the desired output from the system and comparing it to the actual output value. Control theory is mostly adopted for vertical scaling and this makes prediction attractive for horizontal scaling.

Vertical scaling consist of improving the capacity of a server by increasing the amount of resources allocated to the server. *Horizontal scaling*, refers to increasing the capacity by adding more servers to the pool of resources. Vertical scaling is fine-grained and can be performed within few seconds, horizontal scaling is coarse-grained and needs more time to be performed. Literature focuses either on *vertical* or *horizontal* scaling, and there has been little focus on coordinating them.

Application level metrics, refers to different demands in terms of resources. Interactive applications are latency-critical and have higher demands to resources compared to non-interactive applications. Using metrics such as response time makes it possible to accurately and efficiently provision resources without committing any SLA violations.

Chapter 2

Background

In this chapter a brief introduction will be given of the technologies that will be applied later as a part of this paper. In addition, a thorough review of related research will be presented.

2.1 Cloud computing

Cloud computing can be described as resources and services offered through the Internet [12]. There has been observed an increased popularity in cloud computing within few years due to the many benefits offered. Cloud computing services are delivered from data centres all around the world to facilitate the customers. Some of the benefits has been cost savings in terms of outsourcing hardware installation, operations services and maintenance. The increased reliability where most cloud providers offers an SLA which guarantees 99.99 % availability, and the ability to scale on-demand has been important factors for most enterprises.

2.1.1 Software as a service (SaaS)

Software as a service, or SaaS, also referred as "on-demand software" where applications and services are provided directly to the user. This is mostly done through the browser where the user interact with the software without any need to install, maintain or update the software. SaaS provides benefits such as global accessibility, where the software is accessible from all around the world. Cross device compatibility, the software is accessible and equally formatted throughout different devices such as tablets, phones and computers. There are many examples of different SaaS softwares such as Office 365, Google Docs, Gliffy and Facebook just to mention a few.

2.1.2 Platform as a service (PaaS)

Platform as a service, or PaaS, is a category of cloud computing services, where platforms are provided to users. This could either be development platforms where users can develop applications without the need to setup and maintain a local infrastructure. PaaS services are hosted on the providers servers and maintained by the provider. The user is able to access the services through the web browser and billing is mostly subscription based (pay-per-use). The benefit with PaaS services is that applications are up-to-date, in addition provides features such as scalability and reliability.

2.1.3 Infrastructure as a service (IaaS)

Infrastructure as a service, or IaaS, refers to providing servers, storage devices and network devices such as load balancers and switches to customers. The cloud provider is responsible to maintain the physical servers with electricity, upgrades and air conditioning. IaaS offers highly scalable resources that can be adjusted on-demand. IaaS providers often bill the customers with the pay-per-use model. Some of the IaaS providers are Windows Azure, Rackspace Open Cloud and Amazon AWS, just to mention a few.

2.1.4 Cloud platforms and providers

There are mainly three different types of cloud platforms; public, private and hybrid.

Public cloud is a cloud computing model where service providers manages the hardware of the infrastructure, this includes operation, maintenance and upgrading hardware components. The providers are also responsible of the security and providing isolation between the customers, which is a major challenge due to the fact of having millions of customers and providing an SLA-agreement of 99.99 % uptime [13]. Public clouds provide benefits such as inexpensive set-up and scalability upon demand and pay-as-you-go model. There are number of public cloud providers, the largest and most popular one is Amazon with Amazon Web Services [1]. Microsoft has its own public cloud called Microsoft Azure [2], RackSpace with Managed Cloud Services [3] and Google with Google Cloud Platform [4].

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com
[3] https://www.rackspace.com/cloud
[4] https://cloud.google.com/

Private cloud is another cloud computing model where an organization or company usually manage their own cloud infrastructure. Unlike public clouds, the company usually manage, operate and upgrades the hardware in the infrastructure by them self. This gives them the ability to have control over the cloud environment and manage both the physical- and network security of the private cloud. There are numerous of alternative cloud platforms used in a private cloud, such as OpenStack [5], Eucalyptus [6] and commercial solutions as VMWare vSphere [7].

Hybrid cloud is is a cloud environment where a combination of distinct cloud platforms are used to orchestrate a cluster. This combination provides many benefits as on-demand resources from a public cloud while still having a secure private cloud. This gives the possibility to move workloads between cloud platforms and overall increases flexibility, furthermore a hybrid cloud can also provide redundancy of the infrastructure. One of the tools which gives the possibility to set-up a hybrid cloud is Apache Mesos [8].

2.2 Virtualization

The history of virtualization began in 1960's, when IBM spent a lot of resources in developing time-sharing solutions. Which refers to sharing of resources among many users at the same time, the goal was to increase the efficiency of the computer resources [14]. Later this developed to be a paradigm shift in computer technology and is now known as virtualization.

Virtualization refers to creating a virtual version of a device or resource, such as network resources, hardware platform, storage devices or even an operating system. Explained in easier terms virtualization is software technology which makes it possible to run multiple operating systems and applications at the same time on a single server. Today's data centres use virtualization to create abstraction of the physical hardware and create a pool of resources which are offered to customers in the form of consolidated and scalable VMs.

2.2.1 Types of virtualization

There are mainly three different types of virtualization, these will be briefly explained in the next sections.

[5] https://www.openstack.org/
[6] http://www8.hp.com/us/en/cloud/helion-eucalyptus.html
[7] https://www.vmware.com/cloud-computing/private-cloud
[8] http://mesos.apache.org/

2.2.2 Para-virtualization

Para-virtualization, also referred as PV, is a technique where the guest operating system is modified and is aware of being virtualized. Some of the benefits with PV is that there is no virtualization extensions requirements on the host CPU and this enables virtualization on hardware architectures. The guest OS sends hypercalls directly to the virtual machine manager (VMM), this also includes critical kernel operations. The result of this is increased efficiency and performance compared to full virtualization. The drawback with this type of virtualization is the OS kernel modifications, which could result in maintenance and support problems.

2.2.3 Partial virtualization

Partial virtualization is a virtualization technique where the VMs simulates the underlying hardware environment. This means the entire operating system cannot run in a VM, in the same way as full virtualization. Address space virtualization is one of the forms used where each VM has independent address space. The hypervisor type is classified as *type 2* and with partial virtualization the guest OS runs as an application on the host machine. Partial virtualization is acknowledged to be an important step towards full virtualization.

2.2.4 Full virtualization

Full virtualization is another virtualization technique where the guest operating system runs on top of the VMM, this removes the burden of having an extra layer between hardware and the guest operating system. The user code is executed directly on the hardware. In this type of virtualization compared to para-virtualization, the guest OS is not aware of running in a virtualized environment. This means that it is possible to install most operating systems since they do not need to be modified.

2.2.5 Hypervisors

Hypervisor, also referred as virtual machine monitor (VMM) makes it possible to run multiple operating systems and share the resources of a single physical hardware. The hypervisor is responsible to monitor, allocate the needed resources and isolate the virtual machines from each other. There are multiple popular hypervisors, to mention a few: KVM, Xen, Microsoft's Hyper-V and VMWare ESX/ESXi.

There are essentially two main types of hypervisors, type 1 and type 2, see figure 2.1. The type 1 hypervisor, also called *bare-metal* or *native* runs directly on top of the hardware without any layer between. It monitors, con-

trols the hardware and the guest operating systems running above, some examples are Xen, VMWare ESX, Microsoft Hyper-V and Oracle VM.

Type 2 hypervisor, also known as *hosted* runs on top of traditional operating systems, such as Linux and Microsoft Windows. In difference from type 1, type 2 has one extra layer on top of the hosted operating system, which makes the guest operating system the third layer above the hardware. Some examples of type 2 hypervisors are: Oracle VM VirtualBox, VMWare Server and Workstation and KVM.

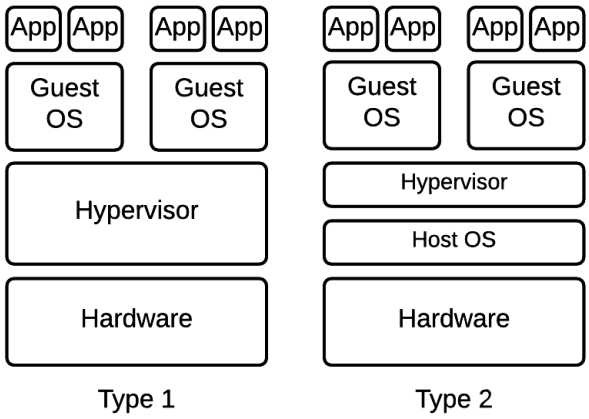

Figure 2.1: Architecture of type 1 and 2 hypervisors

2.3 Xen and KVM

In this section the main focus will be on the open source hypervisors Xen and KVM, a thorough analysis on the two hypervisors will be conducted. Moreover, how their architectures are built up and how they perform CPU scheduling.

2.3.1 Architecture

The KVM hypervisor, as visualized in figure 2.2 was merged with the Linux 2.6.20 kernel in 2007. The result of this was support for virtualization [15] and KVM has been maintained as part of the kernel since. The KVM module requires the CPU of host OS to support Intel VT / AMD-V hardware virtualization extensions. Running the Linux Kernel as a hypervisor in difference from Xen, makes it possible to use existing components such as the memory manager and scheduler. In addition, Xen architecture requires maintenance of the Xen hypervisor and Dom0, which will be explained in details later. As figure 2.2 illustrates, the KVM is a kernel module and runs

on the Linux kernel, while each of the virtual machines runs as a Linux process. A lightly modified Qemu process is used to provide emulation for devices such as BIOS, USB bus, network cards and disk controllers. Quemu is unprivileged and isolated, it uses SELinux for security to provide isolation between processes.

Xen, on the other hand has a different architecture, it runs as *type 1* hypervisor. The terminology in Xen can be explained as the host OS is referred to as Domain 0 (dom0), while the guest OS is referred to as Domain U (domU). The host OS runs above the Xen hypervisor with virtual CPUs (vCPUs) and virtual memory [16]. Even though, it is privileged and has control interface to the hypervisor. Dom0 provides device drivers for the host hardware which consist of different controllers, network card and management tools such as Virsh.

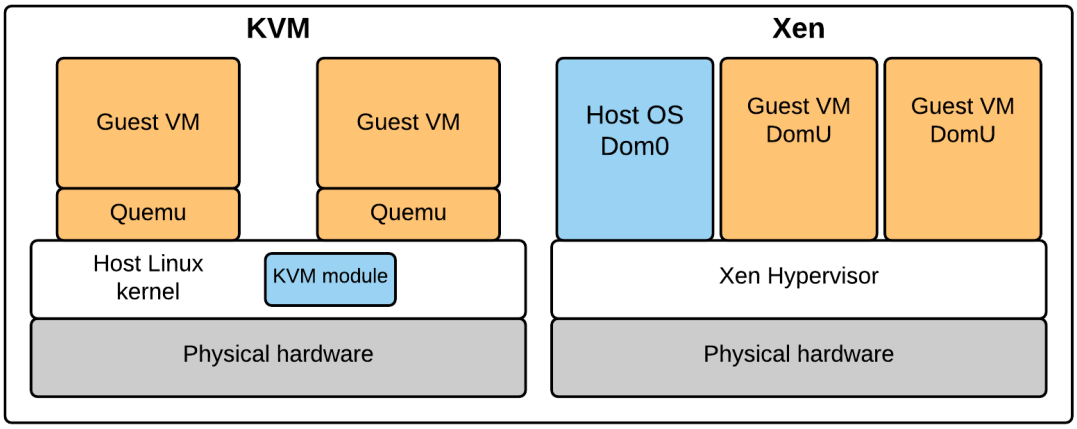

Figure 2.2: Architecture of KVM and Xen

2.3.2 CPU schedulers

The shared resources such as memory and disk can easily be adjusted in run-time, while CPU is fine-grained and requires to be adjusted by a scheduler. Xen has in the past used different CPU schedulers, however since Xen 3.0 the *Credit scheduler* is used by default [17]. This is because of improved scheduling on multiprocessors and better QoS controls [18]. It is also possible to choose among other schedulers such as *Simple Earliest Deadline First* (SEDF) and *Borrowed Virtual Time* (BVT).

In contrast to Xen, KVM leaves the scheduling to the Linux kernel. The current Linux kernel is a multi-tasking kernel and uses *Completely Fair Scheduler* (CFS) as default. The Linux kernel has earlier used the *O(1)* scheduler.

Xen - CPU schedulers

Credit scheduler is a proportional scheduler based on fairness. The scheduler works in most cases in the same as the Linux scheduler, to minimize the waste of CPU cycles and provide fairness to the domains [19]. One drawback with the credit scheduler is that since all of the domains are equally scheduled, the dom0 may in some cases be subjected to low CPU cycles if the domain is not assigned high enough *weight*.

The credit scheduler uses metrics such as *weight*, where each domain is assigned with a weight. If for instance a domain gets a weight of 512 that means it will get twice much CPU power as a domain with 256 (default). *Cap* is optionally assigned which limits the amount of CPU a domain can use. The values are expressed in percentage of one physical CPU, e.g 100 for one 1 physical CPU and 300 for 3 physical CPUs. The default value is 0, which means there is no upper limit [19].

Borrowed Virtual Time (BVT) scheduler is a fairness scheduler based on the concept of virtual time, where CPU is leased out based on *weights*. In difference from credit scheduler, BVT is a work conserving scheduler. This means that if one domain is idle, the second domain gets all the CPU without considering *weight* [20].

Simple Earliest Deadline First (SEDF) scheduler support both work-conserving and non work-conserving modes, which means that each vCPUs is consumed when running, and preserved when not running. SEDF guarantees that resources are allocated based on a domain's *slice* and *period*. A domain will be given resources as long as it is executed for the time given in the slice for each period [21].

KVM - CPU schedulers

The O(1) scheduler and CFS were both introduced by Ingo Molnár [22]. O(1) became a part of the Linux kernel 2.6 prior to 2.6.23. The scheduler is based on achieving fairness, interactivity and performance. The scheduler is preemptive and priority-based, where 0 is the highest priority and 140 is the lowest as visualized in figure 2.3. Real-time tasks falls between 0 and 99, while other tasks falls between 100 and 139 in a so called time-sharing task group. The *nice* value affects the priority of a process. Each of the priorities corresponds to a nice value where the default value is 0, the highest priority value is -20, while lowest priority value is 19 [23]. Processes with higher priority gets more CPU time, while those with low priority receive less CPU time. This calculation of time-slices are performed dynamically

to avoid starvation of processes. The scheduler uses two types of arrays, an active array and an expired array. The active array contains all the processes which has CPU time left, while the expired contains those have used their CPU time-slice. Before a process time-slice is used up a recalculation is performed to find the new priority.

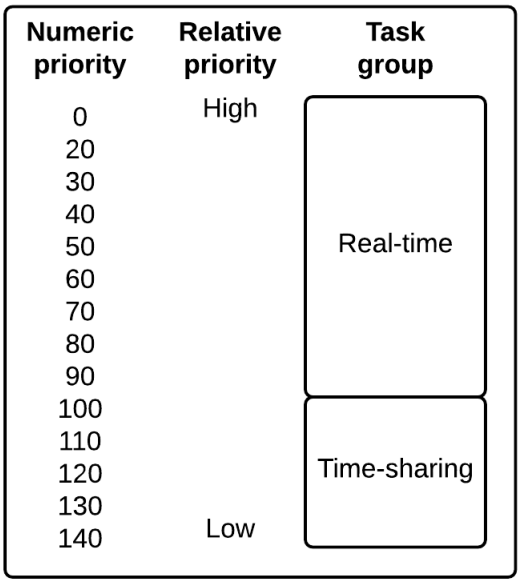

Figure 2.3: O(1) scheduler - priority levels

CFS as the name of the scheduler reveals is completely based on fairness to provide equal CPU time to tasks. CFS has since Linux kernel version 2.6.23 been the default scheduler and replaced O(1) [24]. Ideally the goal of CFS is to provide equally CPU share among the running processes and balance processes between multiple cores in Symmetric Multiprocessing (SMP) systems. The biggest change with CFS compared to the previous schedulers is use of a runnable processes list, the previous schedulers had a implementation of using linked list based queues [25]. CFS uses the concept of *red-black tree* which is mainly self-balancing, by removing and adding entries to maintain balanced. CFS manages meta-information about tasks in *virtual runtime*, which record the amount of CPU time each task has been permitted. The smaller the value, this indicates the higher need for more CPU time.

One of the main features which was introduced with CFS (Linux kernel 2.6.24) is group scheduling where connected processes are split into groups. This provided the possibility to ensure the groups were given the fair amount of CPU time instead of single processes [22]. Because of this CFS brought optimized scheduling to both desktops and servers.

2.4 Vertical and horizontal scaling

In this section vertical and horizontal scaling will be explained, furthermore analysis on vertical scaling capabilities of popular hypervisors, such as Xen, KVM, VMware and Hyper-V.

Vertical and horizontal scaling is a concept in cloud computing where decisions for scaling are based on workload, illustrated in figure 2.4. Vertical scaling typically means to add more resources to an existing VM in the form of CPU, RAM and disk. The use of vertical scaling requires additional investment of physical hardware, support from the hypervisor and can in many cases result in single point of failure. If the VM goes down for any reason, the service provided will not be available for the users. The advantages with vertical scaling is less overhead, since there is only a single VM running. While with horizontal scaling there are multiple VMs with the overhead from the operating systems and different services running on each of them which consumes resources.

On the other hand we have horizontal scaling, which requires less support from the Hypervisor in comparison to vertical scaling. With horizontal scaling the concept lies on increasing the number of nodes to distribute the workload across multiple VMs. Typically, horizontal scaling is used for applications that have a clustered architecture with a gateway or master node that distributes the load [26]. The benefit with horizontal scaling is that there is no single point of failure which improves performance, reliability and availability. This can in many cases also cause complexity, since user sessions between the nodes needs to be synchronized. Also, to not have the issue with single point of failure, there is a need of at least two gateways or master nodes which also requires a synchronization between them. In this thesis the approach is to coordinate between vertical and horizontal scaling depending on the workload and other metrics.

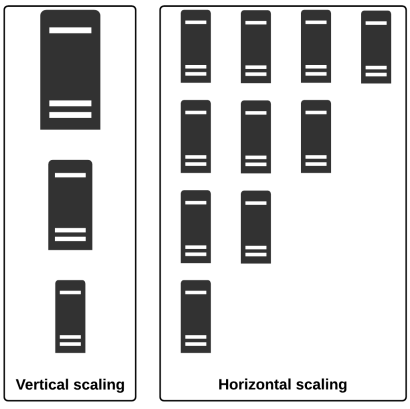

Figure 2.4: Vertical and horizontal scaling

2.4.1 Vertical scaling capabilities of popular hypervisors

Hot-plugging is a concept of appending or removing resources to a running system without having to switch it off [27]. This can be performed on the hypervisor or directly on the operating system. In this thesis the focus will be on the hypervisor, by dynamically adding or removing resources on VMs.

CPU hotplug

CPU hot-plug refers to adding or removing CPU cores to a single or multiple VMs. The support of CPU hot-plug differs between hypervisors. Table 2.1 shows the comparison of vertical capabilities between Xen, KVM, VMware and Hyper-V. Xen has support for both adding and removing CPU cores in run-time. On the other hand, currently KVM and VMware only have support for adding CPU cores, and does not yet have a feature to unplug CPU cores [28]. Microsoft's Hyper-V does not offer adding or removing of CPU cores.

Memory hotplug

Adding or removing memory without having to reboot the system is called *hotplugging*. There has to be support in the hypervisor to be able to perform hot add or remove on virtual machines. With the hypervisor it is possible to reallocate resources between VMs in order to handle load-bursts. However memory is not immediately released like CPU, since applications are not immediately garbage collected [29]. Both KVM and Xen have support for adding or removing memory, while VMware and Hyper-V only have support for adding and not removing memory.

Hot disk add or remove and extend and shrink

The concept of adding or removing disks to a running system is essential in production environments. The two hypervisors Xen and KVM have support for both of the features. VMware has support for removing disk, and partial support for adding, Hyper-V has partial support for the features. Overall there is little support for extending and shrinking disk-size between the hypervisors.

This makes Xen reasonable to use because of capabilities for both adding, removing CPU and memory in run-time. The first point is essential since there will be a need to dynamically reallocate resources in run-time depending on the workload and performance metrics.

Table 2.1: Comparison of vertical scaling capabilities between Xen, KVM, VMware and Microsoft's Hyper-V [30].

	Options	**Xen**	**KVM**	**VMware**	**Hyper-V**
CPU core	Add / Remove	Yes/ Yes	Yes/ No	Yes/ No	No/ No
Memory	Add / Remove	Yes/ Yes	Yes/ Yes	Yes/ No	Yes/ No
Disk	Add / Remove	Yes/ Yes	Yes/ Yes	Partly/ Yes	Partly/ Partly
	Extend / Shrink	Partly/ No	No/ No	Yes/ Partly	Partly/ Partly

Memory ballooning

In difference to *hot memory add or remove*, the concept of memory ballooning lies on dynamically adjusting physical memory address space used by a VM. Instead of allocating static values of physical memory address space to each VM, memory ballooning reduces the chance of performance degradation. Each VM has a *balloon driver* in the VM's kernel, which creates a bridge between the hypervisor and the VM. With for instance Xen, the hypervisor allocates the memory defined in the configuration file, but in most cases the VMs do not need the entire memory at any given time [31]. With the memory ballooning techniques the hypervisor is able to remove the necessary memory pages from the VM when it's unused. The VM will not know that memory has been removed, it will just be empty space, hence "balloon" [32]. If the hypervisor for instance wants to increase memory allocated to a VM, then it would map the memory pages to the VM's space, this gives the balloon driver more access to memory. Further, the balloon driver can release it to the VM's kernel and within a short period, the VM reflects the changes .

In contrast to *hot CPU/memory add or remove*, memory ballooning has some limitations, maximum memory needs to be defined. Memory ballooning is supported by all of the Linux kernels, which makes it an attractive way to perform vertical scaling.

2.5 Control theory

Control theory (CT) is a concept from machine learning, and is used to automate management of different types of information processing systems (IPS) [33]. There are different kinds of IPS systems, such as web server, database and message queuing systems.

Karl Astrom, one of the bigger contributors to control theory, stated that

"magic of feedback" is that one can create a system that works well with components that lack in performance [34]. He mentioned that this is done by adding a new element, a controller that adjusts the behaviour of one or more elements based on the measured outputs of the system [35].

The main idea behind CT is to create a model which defines a reactive or proactive controller that adjust resources based on demand. This makes it possible to perform auto-scaling in relation to increased or reduced workload. There are multiple CT models; feed-forward, open-loop, closed-loop and feedback control loop. In this project, feedback control loop will be used.

As illustrated in figure 2.5, the feedback control loop use a *reference* value which is the desired value of the measured output of the system. The difference between the reference input and measured output value is called *control error*. The *input* variable is the setting of one or more parameters that manipulate the behaviour of the system, which could be for instance to allocate memory, CPU cores or disk space. The system output is the measurable characteristics of a system, such as response time. The feedback or measured output is compared to the reference value. The goal of the controller is to always ensure the measured value is equal as possible to the reference value.

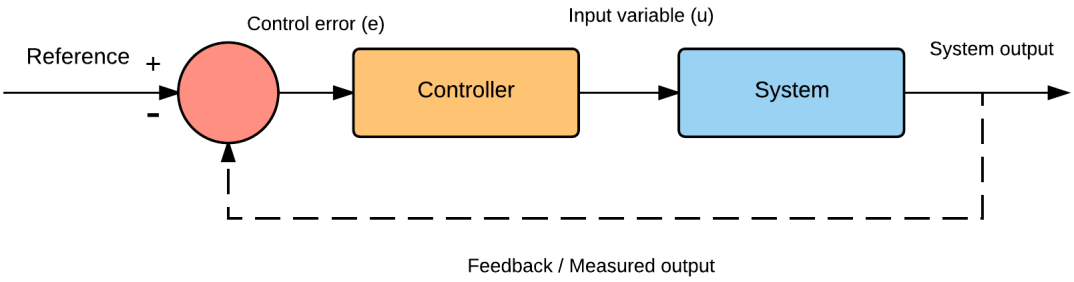

Figure 2.5: Feedback control loop

2.5.1 Libvirt

Libvirt is an open source API, management tool and daemon for multiple virtualization platforms. It currently supports multiple hypervisors such as KVM, Xen, VMware ESX, QEMU and other virtualization platforms [36]. Some of the offered features are;

- VM management

- Network interface management
- Remote machine support
- Virtual NAT and Route based networking

2.6 HAProxy

HAProxy[9] (*High Availability Proxy*) is an Open-source TCP/HTTP load-balancing proxy server, and is widely adopted in order to achieve fault tolerance to guarantee availability and reliability of critical services [37]. HAProxy offers layer 4 (transport layer) and 7 (application layer) load balancing. Several load balancing algorithms are offered, to mention a few commonly used algorithms:

- *RoundRobin* the back-end are served in turns, this is the default algorithm.
- *Leastconn* as the name may tell, selection is based on choosing the server with the least number of connection. This algorithm is often used for longer sessions.
- *Source* algorithm performs balancing based on the hash of the users source IP, this ensures the user is connected to the same back-end server.

Load balancing is often implemented to provide a scalable infrastructure and clustering of for instance web servers is a method to achieve that. Workload is distributed across the pool of servers to maximize performance and optimize resource usage. HAProxy has built-in functionality for health checking of the back-end servers before the traffic is forwarded.

2.7 HandBrakeCLI

HandBrakeCLI[10] is a command-line driven interface to several built-in libraries for performing encoding, decoding and conversion of audio and video streams to multiple of formats. HandBrakeCLI is multi-threaded which makes it possible to perform simultaneously encoding on multiple cores.

[9] http://www.haproxy.org/
[10] https://trac.handbrake.fr/wiki/CLIGuide

2.8 Loader.io

Loader.io[11], also refereed as *Loader* is a cloud-based load testing and scalability service. Loader provides functionality to perform automated testing of web services to measure potentially SLA violations by cloud service providers. Loader also provides a RESTful application programming interface (API), which makes it possible to perform testing of web-applications externally. The tool allows to perform load testing and monitoring of the traffic in real-time.

Three types of tests are offered [38]:

- **Maintain client load** - A constant client count will be maintained throughout the test duration.
- **Clients per second** - Client requests made each second.
- **Client per test** - Clients will be distributed evenly throughout the test duration.

2.9 Httpmon

Httpmon[12] is a web-site monitoring and workload generator tool. Httpmon generates HTTP-requests to a single URL based on an open or closed model. In the open model, the requests are completely random without depending on the response time. In the closed model, each clients wait for a response from the web-server before making a new one. In addition, httpmon also make statistics based on the results from [39]:

- Response time
- 95 and 99-percentile latency
- Requests per second
- Queue length, number of requests vs received reply
- Rate and number of requests - option 1
- Rate and number of requests - option 2

[11] https://loader.io/
[12] http://www.httpmon.com/

2.10 Web-applications

In this thesis, two different interactive benchmark applications will be considered, RUBiS [13] and RUBBoS [14]. These two are popular interactive cloud benchmarking applications and widely used in research experiments [40, 41, 42, 43]. The RUBBoS web-application, is a bulletin board application that models slashdot.org. The RUBBoS's database consist of five tables, containing information about comments, stories and submissions [42]. RUBiS is an online auction site modeled after eBay.com, it is possible to use client workload generators to emulate user browsing and bidding. The architecture consist of a web-server service with PHP and a database in the back-end [44].

2.11 Related research

The concept of *self-adaptive cloud environments* is not new, it covers a broad area of research fields, where there is still ongoing extensive research. Because of the increased use of *cloud computing* [45], cloud service providers are encountering new challenges to ensure SLA- and QoS requirements. There are conducted multiple research experiments to achieve increased efficiency and better resource management.

2.11.1 Performance-based Vertical Memory Elasticity

The research study [40] was performed to explore vertical elasticity features in cloud computing environments. The focus in the study was completely on scaling memory using control theory based on changes in the workload. Control theory with feedback loops is used as decision maker to compare the desired and actual response time (RT) of the application. The application performance is mainly the focus and based on the RT the controller would make changes by either increasing or reducing the memory size of the VM. The results of the experiments shows that they managed to increase memory efficiency by at least 47 %.

2.11.2 Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints

This research work [29] is a continuation of the work mentioned in the previous section. Since applications in most cases are dependent of an arbitrary combination of memory and CPU, a coordination between the resources is essential for efficient resource utilization. The paper

[13] http://rubis.ow2.org/
[14] http://jmob.ow2.org/rubbos.html

describes the novelty of the research by using a fuzzy control approach as a resource coordinator between memory and CPU controller. The application performance is measured in response time as a decision maker. The study shows that without having any coordination between the memory and CPU controller, the VM is in most cases under- or over-provisioned with resources. By using Fuzzy logic and implementing Fuzzy rules which contains; RT, utilization of CPU and memory as a performance vector. Comparing the results of using fuzzy controller and non-fuzzy controller shows that without having any coordination between the controllers most of the times one of the controller over-provisions resources. With a coordination between the controllers, right amount of resources is allocated to meet the desired response time of the application.

2.11.3 Quality of Service Control Mechanisms in Cloud Computing Environments

This study is a PhD dissertation conducted by the same author as in the previous two sections [46].

Cloud providers do not yet offer any performance guarantees, despite of having availability guarantees. The reason for not having any performance guarantees is because of the increased complexity, and how this can be addressed in a cost-effective way from the cloud providers and customers point of view. There is no-linearity between workloads and the required resources, this makes it challenging to meet the desired performance. Controlling the trade-off between QoS and cost is the main focus in the thesis. The scope of the thesis lies on investigating models, algorithms and mechanisms to handle these two perspectives. In more details;

- The first approach looks from the cloud providers point of view to offer a distributed infrastructure placement of virtual machines. In this approach *Bayesian* network model is used to perform decision making.
- In the second approach, the author looks at the trade-off between QoS and cost from the cloud customers point of view. The concept of Fuzzy controller is used to coordinate the resource controllers to meet the performance in a cost-effective way.

The results from the study shows that with the trade-off between QoS and cost for the cloud provider. The proposed approach managed to decrease the energy cost in the infrastructure by up to 69 % in comparison to the first state-of-the-art baseline algorithm, and 45 % in comparison to the second algorithm.

In the second approach; controlling the trade-off between QoS and cost for the cloud customers. Several experiments were conducted with real-world workload traces. They managed to efficiently save at least 47 % memory

usage while keeping the desired performance level. In the experiments, having coordination between resources with the Fuzzy controller, they managed to reduce the memory usage by up-to 60 % in one of the scenarios and up-to 56 % less CPU usage in another one, compared to not having any coordination between the controllers.

2.11.4 Heracles: Improving Resource Efficiency at Scale

This study is conducted by researchers from Stanford University and Google [47]. The main focus in the study lies on increasing resource efficiency by reusing resources of underutilized servers in a production environment. They present a feed-based controller, named *Heracles*, which coordinate resources between best-efforts tasks and latency-critical services (LC). The desired goal is to keep the service level objectives (SLO), since small interference could cause SLO violations for the latency-critical service. The focus is to maintain and guarantee that LC service receives enough amount of shared-resources; memory, CPU and network I/O. Results from the work showed that *Heracles* managed to increase average utilization of 90 % across different scenarios without any SLO violation for LC tasks in a production environment.

2.11.5 Towards Faster Response Time Models for Vertical Elasticity

Resource provisioning is typically coarse-grained, this means that CPU cores are typically leased for periods as long as one hour. Vertical scaling has improved resource efficiency, resources can be provisioned for as least as few seconds. The study [10] present an empirical study where mean of response time is used to measure QoS of popular cloud applications. The interesting points made in the study is that response time is not in a linear relationship with capacity. By presenting a model called *Queue Length Model*, the relationship is presented as:

$$q = \lambda \cdot R$$

As shown, q is is the average queue length, λ is the arrival rate and R is the response time. The second model is called *Inverse model*, where the relationship between an application's mean response time (R) and capacity allocated is represented as:

$$R = \beta / c$$

The parameter β is the model parameter, and as in the *queue* model, earlier measurements of capacity and response time is used to calculate β, c is the capacity and R is the response time. The tool *httpmon* was used as workload generator, with both open and closed system models for emulating user access. The results showed that both of the models described above

managed to predict the needed capacity. With a high desired response time both of the models delivers, while knowing a low response time as target the *invese* modele was more stable than *queue*.

2.11.6 Proactive Memory Scaling of Virtualized Applications

Applications in cloud environments are often subjected to varying workloads, and instead of over-provisioning with resources to accommodate spikes in the workloads. A study performed by researchers from VMware and University of Würzburg [48], developed a solution for proactive scaling of memory on virtualized applications. The researches used statistical forecasting to predict future workloads and scale precisely based on the needed resources. By using real-world traces to create real scenarios, and comparing both a reactive and proactive controller, the researchers managed to show that performance increased with more than 80 % using a proactive controller.

2.11.7 Vertical scaling for prioritized VMs provisioning

The research study is performed by researchers from the University of Dresden in Germany [49]. The aim of the study was to develop a controller to perform elastic provisioning of resources to prioritized VMs and avoid SLA-violations. The paper also evaluates the benefits of performing vertical scaling of prioritized VMs. They use real-world workload traces from WorldCup 98 with the web-application RUBis online auction benchmark. CPU scaling was performed with CPU cap by using *Xen credit-scheduler* to adjust the resources. The results from the paper shows that they managed to improve CPU usage efficiency without having any major SLA violations. The developed controller achieved a higher throughout in comparison to a statistical provisioned VM. In addition, they managed to have a stable low response time for the latency critical application running on the prioritized VM.

Chapter 3

Approach

This chapter provides an overview of the methodology and the steps needed to answer the research questions defined as part of the problem statement:

- *How can we create an autonomic provisioning controller based on elements from control theory to increase server utilization, and at the same time expose available resources to ensure QoS of real-time and non-real-time applications?*
- *How to coordinate vertical and horizontal scaling for better resource allocation?*
- *How to benefit from application level metrics in order to efficiently provision resources to interactive and non-interactive applications?*

The research questions defines several main aspects, which will be used as foundation for the thesis. The research questions are divided into three phases of the study, consisting of creating an autonomic controller to increase server utilization using concepts from control theory. In addition to using concepts as vertical and horizontal scaling to improve resource allocation. The key feature in this study is on QoS of the applications and how application level metrics can be used to better decision making.

The aspects of the approach consist of:

- Design of model
- Implementation of the model
- Experiments with different scenarios
- Expected results

3.1 Objectives

The objectives for this thesis is outlined in the problem statement consisting of mainly three sub-questions that needs to be answered to address the whole problem statement. By using terms and concept described in the background chapter as foundation for the work to design and prototype an autonomic provisioning controller.

The motivation for the approach is to provide experimental scenarios, which can relate to production environments, in addition to increasing server utilization by using control theory. Using two types of applications, namely a interactive and a non-interactive, in order to maximize server utilization. Taking advantage of under-utilized servers to launch batch processing on them is a promising way to increase resource efficiency. This requires a better way to coordinate and fulfil the applications demands in terms of resources. The assumptions made is that by dynamically coordinating resources to applications, without over- or under-provisioning of resources. By doing so, the QoS of the applications will in most cases not violate the SLA-requirements, keeping the desired performance on varying traffic load.

The objectives in this study consist of three main phases, which are essential to adequately answer the problem statement:

1. Design
   (a) Controller models
   (b) Decision model
   (c) Controller metrics
   (d) Expected results
2. Implementation for experiments
   (a) Experimental setup
   (b) Workload patterns
   (c) Experiments
   (d) Autonomic provisioning controller
   (e) Testing and experiments
   (f) Results from initial experiments
3. Measurement and analysis
   (a) Results from main experiments
   (b) Data collection

(c) Data plotting

(d) Analysis and comparison

These three phases will be thoroughly explained and visualized for better understanding of the concepts. The first stage of the preliminary phase consist of creating the architecture and the features. This stage is the foundation for the next phases, the second and the third phase will have more focus on the practical part such as the models, prerequisites and flow charts.

3.2 Challenges with cloud infrastructure management

Efficient management of large-scale infrastructures requires insight in the services and technologies running on the application layer in the data centers. Low resource utilization in data centers is one of the main challenges due to resource stranding and fragmentation [50]. To achieve efficient management of cloud infrastructures, requires solid investment of time in configuration of physical servers and virtual machines. Using concepts as resource elasticity, which can be defined as to dynamically adjust attributes in response to changes in the workloads.

Managing cloud infrastructures is a complex task today because of the rapid increase in usage. To create a production-ready prototype of the controller requires it to be tested in a real environment. The controller will not only monitor one single VM, but several VMs simultaneously. The model of the controller will have several attributes which are collected from the applications running on the VMs and actions are performed reactively. To document the effects of the controller and the increased performance achieved, resource utilization needs to be monitored on each of the VMs. The logs needs to be accurate and precise, and further analyzed in a controlled environment, which is one of the prerequisites for future work.

3.3 Design stage

The design phase is the initial phase for the approach and is also a major part of the thesis study. The steps in the design are divided into several stages which builds on top of each other. The first stage explores the different controller models and the requirements. The second stage explores how concepts from control theory will be applied in the model of the controller, and how the expected behaviour will be in different scenarios. The last stage outlines the technical infrastructure for the experiments.

3.3.1 Controller models

The first step in the design stage consist of evaluating the available controller models for the protoype of the autonomic provisioning controller. The drawbacks and benefits with the models should be outlined, in relation to if their behaviour is proactive or reactive. Two types of models will be looked at: *Capacity- and performance-based* models and the desired characteristics of the controller model includes the following:

- Adaptive: Since real-time applications are latency-critical and have strict SLA-requirements. The controller has to be able to both proactively and reactively act to changes in the environment.
- Scalable: The controller should be able to perform both vertical and horizontal scaling, in terms of resources and VMs. Using techniques to perform cost-effective decisions in real-time without violating the SLA requirements to a greater extent.
- Rapid: The controller should run in a high phase to be able to quickly pick up changes in the environment and monitor resource consumption of the VMs.
- Reliable and robust: The controller has to be able to perform precise and accurate decisions based on changing load dynamics. One of the important characteristic lies on not over- or under-provision resources.

3.3.2 Decision model

The prototype of the controller will be designed as part of the controller model, to provide an overview of all the characteristic and functionalities. The prototype models consist of four sub-models working simultaneously. By using concepts from control theory to define the communication between the models and achieve high performance. There are several parameters which are necessary when performing decision making, and those are collected from the VMs and will be thoroughly outlined in implementation section.

The controller will have the abilities to trigger the following actions:

- Collect information of the state
- Decide to perform vertical scaling
  - Increase or reduce resources
- Decide to perform horizontal scaling
  - Add new VM
- Confirm the state of the environment

The decision model uses control theory as a foundation for the actions and to illustrate the different states of the environment. The first action performs collection of performance and utilization metrics from the VMs, the values are then compared to desired metrics. Based on this the controller either perform vertical or horizontal scaling. The first one is to increase single or multiple resources, and the second consist of adding a new VM to the pool. In the end, the state is checked to confirm the state based on the actions.

By using control theory to model the different states of the environment, actions are compared to the desired state of the environment. Control theory ensures that the state of the environment stays at the desired state without causing the result to diverge from the desired.

3.3.3 Controller metrics

To perform accurate and precise actions, the metrics which the controller relies on are important for the behaviour. The metrics will be collected from several VMs simultaneously, the metrics are important for the controller to trigger the correct actions. The real-time application, web-server is latency-critical and therefore requires the state to be checked more often, in comparison to the non-real-time application, batch job. The metrics listed below are essential for the controller:

- Web server response time
- Video encoding - frames per second (FPS)
- RAM memory usage and allocated
- CPU usage and allocated cores

The response time for the web server can be in short terms be described as the total amount of time it takes to respond to a request for service. On the other hand, the batch processing software heavily relies on CPU power to perform video encoding. The QoS of the application is measured in frames per second, also known as frame frequency.

The resources can be divided into used and the allocated resources. For RAM and CPU, the allocated resources will be monitored in relation to the usage to be able to measure whether or not the VMs are over- or under-provisioned with resources. Throughout the experiments, the controller will use these metrics to perform decision making and this gives the ability perform forensic on the performance and resource logs, which will be explained in more details in the measurements and analysis section.

3.3.4 Expected results

The expected results for the design stage consist of performing a successful analysis of the experimental environment. Several designs of the controller will be proposed and implemented, to analyze the expected behaviour and characteristics. The focus will be on the problem statement, to find an ideal solution which can fit a production environment and give potential value within the research field.

3.4 Implementation stage

For the implementation stage there are several tasks that needs to be completed before the experimentation stage. The implementation stage includes the following tasks:

- Experimental setup
- Tools to build the models
- Application level metrics
- Workload patterns
- Autonomic provisioning controller
- Initial experiments

3.4.1 Experimental setup

After completing designing the controller, architecture of the infrastructure has to be outlined. The physical servers that are going to be used in the thesis needs to be properly configured and maintained in terms of security, reliability and performance.

Since the infrastructure consist of two equal servers, their configuration has to be as similar as possible. The *Xen* hypervissor requires the disk partitioning to be performed during instalment of the operating system. The easiest will be to set the disk with *Logical Volume Manager* (LVM) which gives flexibility in the later stages when creating VMs. Since the *Dom0* is the initial domain and is started by the Xen hypervisor on boot, it will require its own LVM-partition.

The next steps in the process will be to configure the networking, which includes to create a separate subnet for the the VMs, and also a connection between the servers to achieve high performance. The guest VM images

will be manually configured to fit the purpose of the experiments. One of the servers will be hosting the VMs:

- Database
- Webserver
- Batch

The *Database* will be hosting the data storage tier (DS), while the *Webserver* will host the presentation tier and business logic (BL) tier based on the 3-tier pattern [51]. The *Batch* VM will be hosting the batch processing job. The main requirement for the infrastructure as earlier described is security, where the focus lies on isolating the domains from each other. In addition, using Secure Shell (SSH) keys for authenticating and accessing the VMs. Furthermore, achieving reliability and high performance is also beneficial for the experiments.

3.4.2 Tools

To build the autonomic provisioning controller in an stable and reliable environment. Multiple tools will be used to set up the environment and to achieve the desired state. The measures used to find the appropriate tools which can be applied in the project lies on functionality and how easily the tool can be adapted.

Configuring the environment will require manual effort, while during the experiments automation will be in focus. Documentation of configuration files and scripts is one of the objectives for future research, and to have an environment that is reproducible by anyone.

When choosing the tools, open-source has been in focus because of interoperability and flexibility. The following tools have been chosen for the project:

- **Apache** - as web server
- **Git** - as a version control
- **HAProxy** - as an load balancer
- **HandBrakeCLI** - as a batch processing software
- **Libvirt** - as a VM management interface
- **Loader** - as a web traffic simulation and benchmarking tool
- **RUBBoS** - as an benchmark application
- **Python** - as scripting language

3.4.3 Application level metrics

The application level metrics are essential for the project, since the focus lies on the QoS. The web server will be the interactive application, while the HandBrakeCLI will be running as the non-interactive application.

The application level metric for web server is *response time*, that can be defined as; *The elapsed time from sending the first byte of the request to receiving the last byte of the reply*. The main goal is to keep the response time within a desired interval to reduce SLA violations. There are many factors that can cause high response time, which may be everything from slow database queries, slow routing, CPU or memory starvation. All of these have to be considered when trying to find the reason behind a high response time.

The non-interactive application, HandBrakeCLI is a video encoding software and is used for batch-processing because of multi-core processing support. The average frames per second is considered as the application level metric, as illustrated below. In relation to an interactive application, HandBrakeCLI has low SLA requirements in terms of delivering desired FPS at any time. The QoS for the application is to achieve average encoded frames to be within a desired interval from the beginning to the end of the encoding. Based on that, there will be periods where the FPS is below the desired, however there will also be periods where the FPS is above the desired interval, as long as the job is finished within the desired time.

Example output from HandBrakeCLI

```
1 Encoding: task 1 of 1, 50.37 % (14.23 fps, avg 24.26 fps, ETA 00h09m50s)
```

3.4.4 Workload patterns

To perform web traffic simulation and benchmark, the cloud benchmarking tool *Loader* is the most suitable and therefore chosen for the project. There are two main workload patterns that needs to be simulated in the project, *spike*- and *trend*-based traffic.

The difference between these two workload patterns is that traffic spikes is a more extremely rapid and challenging to handle. The term "spike" is commonly used to refer to an unexpected sustained increase in aggregated workload volume [52]. While a so-called *trend* is much more of increasing traffic in a longer time-line, which makes it easier to predict and act towards.

Both *trend*- and *spike*-based traffic will be emulated with *Loader*.

3.4.5 Autonomic provisioning controller

The main purpose of this study consist of designing and developing an autonomic provisioning controller by using concepts from control theory as stated in the problem statement. The foundation of the study relies on increasing server utilization, while achieving a desired QoS of the interactive- and non-interactive- applications.

The controller script will be developed in the programming language *Python* as specified in section 3.4.2. There are some few obstacles that might create issues and needs to be looked into before the implementation of the controller. Using techniques such as *hot-plugging* is essential for the controller, resources needs to be provisioned immediately, the threshold is <1 seconds. Before implementing the controller, some initial experiments will be performed on CPU and memory, these will be thoroughly explained in the experimentation stage.

The controller will perform decisions based on the application level metrics and collect those from the applications running on the VMs. The decisions will be based on whether the metrics are within the desired interval. The controller will also proactively monitor the state of the server to locate lack of resources in an early stage.

Scaling will be performed both vertical and horizontal; the logic will be to perform vertical scaling until there are not any lack of resources on the server. Vertical scaling will be performed until server utilization reaches 80 % of the available resources. Horizontal scaling will be performed right before the server reaches its maximum resources. The second server will be booted up, and the VM will either be live-migrated or a new web server will be created, the load balancer will reroute the traffic to both of the web servers. The solution for performing horizontal scaling will be based on technical difficulties and time constraints.

Since there will be running batch-processing on one VM, and web-server on another. The time-interval for how often the controller will check the application level metrics will also have to be considered. Checking the metrics too often would result in an unstable controller and environment, while performing checks too rarely will not make it possible to detect *spikes* in traffic. To find the ideal time-interval, different scenarios needs to be tested and the behaviour of the controller needs to be analysed, furthermore tuning of the behaviour will be needed based on the results.

3.4.6 Initial experiments

There will be performed three initial experiments that are essential for the project and creates a foundation for the main experiments. The first experiment will be on the CPU performance when performing parallel computing. The second and third experiment lies on how quickly a VM manages to map CPU cores and memory when it is added or removed.

- Initial experiments
  - Parallel computing
  - Memory hotplug
  - CPU hotplug

Parallel computing

In this experiment, the focus lies on measuring how much the performance improves when increasing the number of CPU cores. The results from this experiment is beneficial when implementing the autonomic resource controller.

The experiment will be performed by running a parallel computing CPU-intensive script. This will be done while measuring how long time it takes to perform the prime number calculation with two, three, four etc CPU cores. The expected result from the experiment is to observe a linear increase in the performance when performing parallel computing.

Memory hotplug

The experiment, memory hotplug focuses on adding or removing memory on a VM from the hypervisor without the need to reboot in order to reflect the changes.

The main purpose of this experiment is to observe how long time it takes for a VM to reflect the added or removed memory. The experiments will be performed in two intervals, first with 64 MB segments and then with 1 GB segments. The expected result for this experiment is that it will take a bit longer time for the VM to reflect 1 GB of added or removed memory compared to 64 MB. This is because of the kernel level operations which consist of virtual memory mapping and the page table.

CPU hotplug

This experiment uses the same concept as the *memory hotplug* experiment. The focus lies on adding and removing CPU cores to a VM and measure the amount of time it takes until the cores is available and can be used by the VM.

This experiment will also be performed in two phases, first increasing in a interval with one CPU core at a time, and also removing at the same rate. The second phase will be to do the same but with a interval of three CPU cores.

3.5 Measurements and analysis stage

The last part of the project consist of performing the experiments and gathering performance metrics from the experiments. The results from the experiments are an essential part to illustrate the aspects of the controller. The data will be gathered from multiple sources, this will require accuracy and planning in order to safely store the data. In the end, the data will be used to plot charts for illustration and comparison.

These steps will be performed in the last stage of the project work:

- Performing the main experiments
- Data collection
- Data plotting
- Analyzing the obtained data and charts

3.5.1 Experiments

In the section, the experiments will be explained and their prerequisites will also be listed. The initial experiments are going to be conducted first, which are intended to give proof of concept for the main experiments. Before conducting the main experiments, several experiments will be proposed and each of them will be built on each other based on the results.

The structure for the experimentation stage includes the following experiments:

- Control interval
- Resource conflict - Web server and batch (vertical scaling)
- Resource conflict - Web server and batch (horizontal scaling)

3.5.2 Control interval

Before performing the main experiments, some experiments will be performed to find the most stable control interval for the controller on the real-time application. The main purpose is to have a stable and adaptive controller, that manages to detect different workload patterns at an early as possible stage.

The autonomic provisioning controller will be tested running at these intervals:

- Every 5 seconds
- Every 15 seconds

While for the non-real-time application, the batch job, the average FPS will be calculated every 5 minutes.

3.5.3 Resource conflict - Web server and batch (vertical scaling)

In this experiment both of the applications will be running concurrently, and the workload for the web server will be emulated as *spiky*. This will in the end create a resource conflict when there are no resources left. The latency critical application will then "steal" resources from the batch job to meet desired throughput without SLA violations. While, when the workload reduces, the web server will release the unused resources, which will be available for the batch job. The batch job will then be able to take by the lost encoded frames and reach the desired state.

The main purpose of this experiment is to observe how the autonomic provisioning controller maintains the desired throughput for the latency-critical application without SLA-violations. In the same time the batch job will have reduced resources for some time, but when there is no resource shortage the batch job will able to fulfil the desired FPS throughput.

3.5.4 Resource conflict - Web server and batch (horizontal scaling)

With this experiment the same violations as the experiment above will happen. The difference in this case will be that the workload will be high for a longer period on both of the applications, which will require horizontal scaling.

A new web server will then be spawn up in the second physical server and added to the configuration file of the load balancer - HAProxy. The load

balancer will be configured to use the round-robin algorithm and reroute 50 % of the traffic to the second web server. In the same time, the first web server will release half of its resources which will be available for the batch job, and the second web server will have half of the resources. From there on vertical scaling will be performed until the workload reaches a level where the first web server manages to face the traffic. A new process will start to remove the second web server in the HAProxy configuration and in the end, shut it down.

3.5.5 Data collection

To create evidence that the autonomic controller can increase server utilization and efficiently provision resources, data needs to be collected and stored in order to prove the results. Accuracy and precision when performing the measurements is important when it comes to reproducibility of the measurements.

Since data will be collected from several VMs simultaneously, there has to be possible to determine the relationship between the collected data. This will be performed by synchronizing time with the NTP Linux package which will be implemented in all of the VMs, hence giving identical timestamps. The proposed structure of the web server measurement script is as follows:

Structure of the measurement script

```
1  Timestamp  Available memory  Used memory  Number of vCPU cores  CPU-load
```

The output would look like the following:

Example output of measurement script

```
1  08:00:00  2048  1543  3  180 %
```

The measurements will be saved to a local file with the date and hostname as filename.

3.5.6 Data plotting

After performing the data collection, the measurements will be used to plot charts to visualize the obtained results. This will make it possible to see the relationship between the data-sets and analyze the behaviour of the controller.

Both Microsoft Excel and RStudio with the programming language R will be used to plot the charts. With R, a script will be created to automate the

plotting of the data-sets to charts, while in Excel, manual effort is required to plot the charts.

3.5.7 Analysis

The last task will consist of performing analysis on the charts and plots to compare the results and draw a conclusion. The results from the experiments will be essential to illustrate the expected behaviour of the controller.

When analyzing the results from the experiments, the results will be compared to each other. The main consideration would be if the controller manages to perform decisions in a rapid and adaptive way. In addition, determine that the controller does not under- or over-provision resources and manages to keep the desired QoS without violating SLA.

The following violations will be calculated for each experiment:

- Average amount of violations
- % of violations = SLA > 500 ms
- % of desired = percentage of responses which is between the desired interval

Chapter 4

Result I - Design and models

In this chapter the results from the tasks described in the design stage are outlined. The structure of this chapter is presented in subsections, first containing the two proposed controller models, then a overview of the decisions models. In addition, the controller metrics are reviewed, and in the end an overview of the infrastructure is presented.

4.1 Controller models

After reviewing the desired characteristics described in section 3.3.1, two controller models are proposed. The behaviour of the controller and decisions must be based on the evaluation criteria. Two controller models, *performance-* and *capacity-based* controller will in this section be outlined. Furthermore, the reason for choosing a combination of the two controller models as foundation for the prototype will also be described in detail.

Capacity-based controller is built upon the concept of allocating resources based on the level of utilization. Capacity-based vertical scaling has been widely adopted by cloud providers because of its simplicity. Utilization of resources are used to estimate the required resources in real-time. This does not give any indication on QoS of the applications, and can in most cases lead to over-provisioning of resources. Since applications requires an arbitrary combination of resources to reduce the chances of violating SLA, application level metrics can give a better understanding if the application is suffering.

Performance-based controller emphasis on the QoS rather than on the utilization of resources to perform decision making. The performance is gathered from the application level metrics, such as response time and the metrics gives an indication on the latency of the application. The controller has defined levels of acceptable and non-acceptable values and those are used when performing decision making. There are some few research studies

that uses performance-based controllers, and the results show that the controller manages to increase resource efficiency. As mentioned in section 2.11.1, the research study used a performance-based controller to perform vertical scaling of memory. The results from the study showed atleast 47% less memory usage while keeping an acceptable user experience. The main drawback with a performance controller is the application level metrics does not indicate, which resource or resources are causing the degradation of QoS. Having a performance controller also requires collection of resource utilization to perform decisions in a more efficient way to reduce resource wastage.

Figure 4.1 illustrates the architecture of the performance- and capacity-based controller. The model is built upon concepts from *control theory*. The red-colored lines illustrates the capacity-based controller. The controller is fed with the desired capacity, and then performs collection of utilization metrics, which then are compared to the desired utilization. The black-colored line illustrates the decisions which can be to either add or remove, a single or multiple resources to meet the desired utilization. However, if the utilization meets the desired capacity - nothing is done. For the performance-based controller, the blue-colored dashed lines illustrates the model. The desired performance of the application is fed to the controller, then the performance is measured, and the same decisions as explained above are made. However, choosing a capacity- or performance-based controller does not satisfy the defined criteria, a hybrid version is needed, which consist of a combination of the models. The hybrid version consist of first using the performance-based model to only measure the performance in relation to the desired capacity, and then the capacity-model is used if the performance does not meet the SLA-requirements.

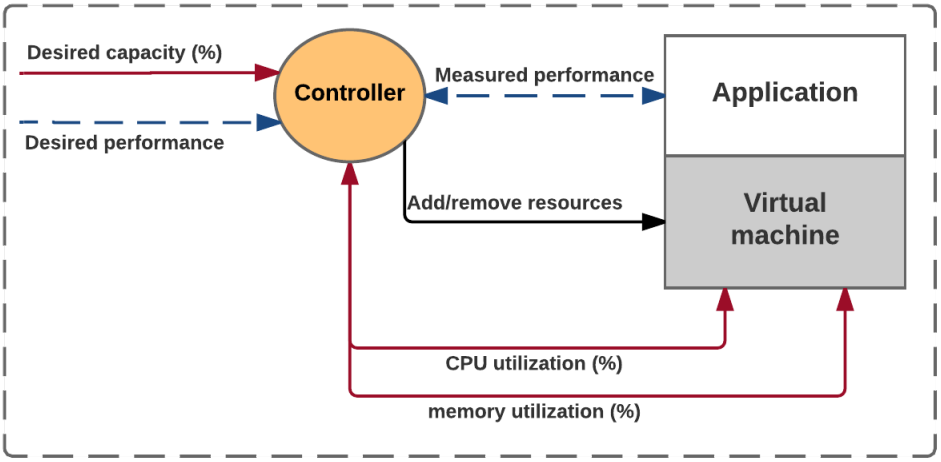

Figure 4.1: Capacity- and performance-based controller

4.2 Decision model

As previously discussed in the design stage, control theory is used as a foundation for the decision making. The control feedback loop in figure 4.2 is based on the feedback control loop described in section 2.5.

The desired QoS can be defined as rt_k, and the measured QoS as rt_i. The control error (e_i) is the difference between these two values in each interval. $Umem_i$ and $Ucpu_i$ are equivalent to utilization of memory and CPU, respectively. While mem_i and cpu_i, is the amount of CPU or memory added. The workload is observed as disturbance, and since the controller has no control over the workload, it adjusts the resources in order to meet the desired QoS.

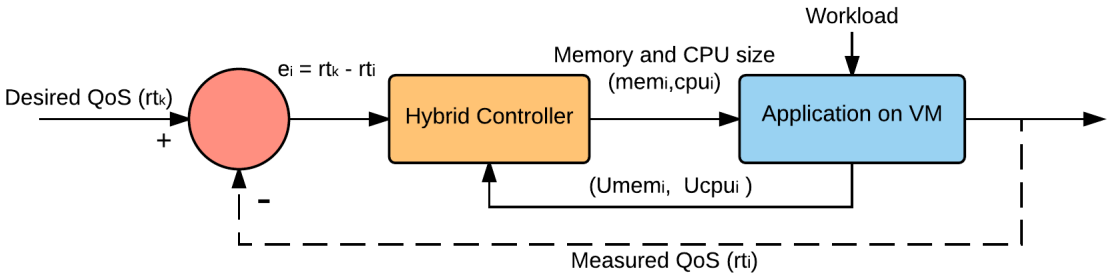

Figure 4.2: The feedback control loop for the hybrid controller

This model is adopted both for the real-time application and non-real-time application, with Response time and frames per second as metrics. As mentioned in the previous section, the hybrid controller utilizes the performance- and capacity-based controller models. The metrics used by the controller will be explained in the section.

4.3 Controller metrics

The controller metrics used in this project can be divided into response time and frames per second. These two SLA-parameters are defined in table 4.1 and 4.2, respectively.

Defining how fast the application should respond is not an easy task, since there are not any specific industry standards. However, based on earlier research on human reaction [53]:

- 0.1 second is the limit for the user to feel that system is reacting instantaneously.

- 1.0 second is the limit for the user to notice the delay.

Based on the research, the SLA policy for the web-server is to keep the average response time to be within the interval of 100 and 500 ms. Each time the response time exceeds 500 ms it is recorded as SLA-violation. When performing the experiments, the violations of SLA will be monitored. If the average response time drops below 100 ms, this means the workload is reducing and therefore the allocated resources needs to be reduced.

The batch-job's SLA policy has lower priority compared to the web-server, the desired average frames per second is defined to be within a interval of 15 and 20. There is a limit of max FPS set to 23 FPS. However if the average frames per second drops below 15 there is not any violation. There will be a need for increased resources to make up for the lost encoded frames. Therefore having an average frames per second which exceeds 20 for some time is not critical. The desired time of the batch is to finish the job approximately within a time-period of 25 minutes.

Table 4.1: SLA: Web server

Average response time	**SLA**
Fast	<100 ms
Medium	100-500 ms
Slow	>500 ms

Table 4.2: SLA: Batch-job

Average frames per second	**SLA**
Fast	>20 FPS
Medium	15-20 FPS
Slow	<15 FPS

The two resource metrics that are also taken into consideration by the controller is utilization of CPU and memory, as illustrated in figure 4.2. To reduce the chances of either over- and under-provisioning, the controller has defined a level containing minimum- and maximum resources, this is illustrated in table 4.3.

The controller always has the state of the VM monitored, containing the usage of CPU and memory. Using those metrics, when performing vertical down-scaling the minimum resources is defined as the memory used plus a buffer of 512 MB. When scaling down there will never be a issue that used memory is removed and causing memory segmentation faults. The VM will always have a buffer to grow into when needed. In addition, if there is no load or the load is manageable with 1 vCPU, then that will be the least possible allocated vCPU.

When performing vertical up-scaling, resources is added when the utilization reaches 80 % of the allocated resources, e.g. if a VM has 5 vCPUs and the CPU usage is above 400 %, a new vCPU is added. With horizontal up-scaling, the same concept is used, however it is based on the total usage of the server. If the total usage exceeds 80 % of available resources, horizontal up-scaling is performed to distribute the load among the servers.

Table 4.3: Utilization of resources

Resources	**Minimum resources**	**Maximum resources**
Memory	UsedMemory + 512 MB	80 % of available resources
CPU	1 vCPU	80 % of available resources

Chapter 5

Result II - Implementation and Experiments

5.1 Experimental setup

The physical equipment where the experiments was conducted consist of two Dell PowerEdge R610 physical machines (PMs) placed at Oslo and Akershus University College. Having access to the physical hardware simplifies the control of the resources and how they are allocated. Both of the servers have equal specifications and are running the operating system Ubuntu 12.04.5 LTS. Based on table 2.1, Xen was the only hypervisor with support for all of the needed features, especially CPU hot-unplugging, and therefore chosen. The PMs have the latest available Xen version 4.1.6.1 configured and installed. Table 5.1 illustrates the specifications of PMs.

Table 5.1: PM specifications

	2xR610
CPU	2xQuad-core Xeon E5530 2.40 GHz
Memory	24 GB (1066 MHz)
Disk	2x146 GB (146 GB in RAID 1)
Network	8xEthernet ports

The main resources which will be used with vertical scaling are CPU and memory, there are in total 16 vCPUs with hyper-threading enabled and 24 GB of memory. These resources will be the limitations when performing vertical scaling.

The PMs are as illustrated in figure 5.1, connected together through the interface *eth1* with a bandwidth capacity of 1 Gbps. This gives

a reliable network connection when performing traffic simulation and benchmarking. The *eth0* is the interface towards internet with a bandwidth capacity of 100 Mbps.

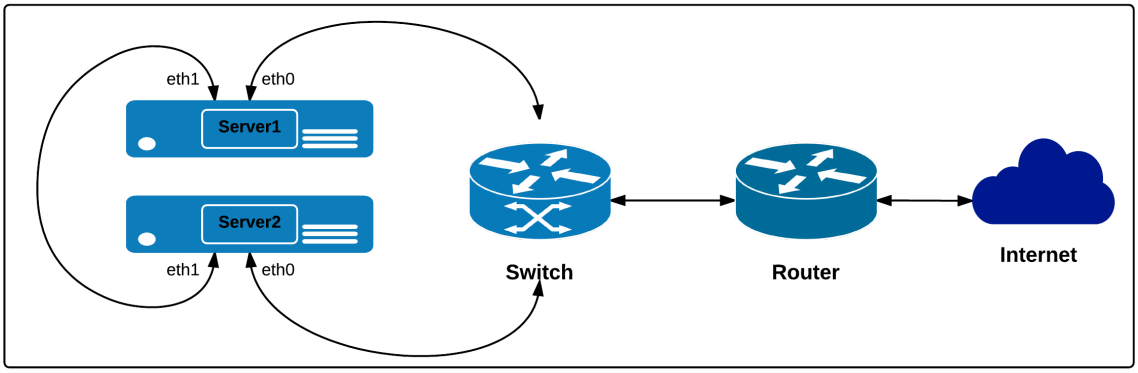

Figure 5.1: Overview of the infrastructure design

5.1.1 Network and virtual machine setup

The VMs participating in the experiments were configured on the private network 10.0.0.0/24. Table 5.2 illustrates the network configuration on the PMs. Server2 is used as the default server where the VMs are stored, while server1 is used as backup server when horizontal scaling is performed.

Table 5.2: Network overview

Physical machine	Interface	IP	Subnet
Server1	eth0	128.39.120.25	255.255.254.0
	eth1	10.0.0.1	255.255.255.0
Server2	eth0	128.39.120.26	255.255.254.0
	eth1	10.0.0.2	255.255.255.0

To ensure network connection between the VMs and internet connection, a bridge named *virbr0* was created, which interface *eth1* is attached to. In addition, two iptables rules are required to ensure that internet traffic from the VMs is permitted. Furthermore, since the VMs has private IP-addresses the traffic needs to be NATed. The following iptables rules are implemented:

Iptables rules

```
1 iptables -I FORWARD -i virbr0 -o eth0 -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT
2 iptables -t nat -I POSTROUTING -o eth0  -j MASQUERADE
```

Virtual machines

As previously described in section 2.3.1, the architecture of Xen consist of having a *Dom0* VM which is the initial domain and has privileged permissions to manage the unprivileged domains.

Each of the VMs requires their own storage space, there are plenty ways to to solve that, either by creating a disk image or by creating a LVM-partition. The most convenient was to create a disk image of 10 GB or 20 GB based on the purpose of the VM. Full hardware virtualization technique was chosen for the VMs to gain optimal performance.

Each VM also requires a configuration file (VM.cfg) as illustrated for the *batch* VM below. The file consist of configuration parameters as; amount of memory, number of vCPUs, IP-address, MAC-address, virtualization technique and so forth. The VMs are managed through console via VNC which simplifies troubleshooting of problems occurring during boot or are network related.

VM configuration file (vm.cfg)

```
1  import os, re
2  arch = os.uname()[4]
3  kernel = "/usr/lib/xen-default/boot/hvmloader"
4  builder='hvm'
5  memory = 1048
6  maxmem = 10240
7  shadow_memory = 8
8  name = "batch"
9  vcpus = 2
10 maxvcpus = 10
11 vif = [ 'bridge=virbr0,mac=02:16:3e:10:11:27,ip=10.0.0.8' ]
12 disk = [ 'file:/home/bilal/domains/batch/batch.img,hda,w']
13 device_model = '/usr/lib/xen-default/bin/qemu-dm'
14 boot="c"
15 vnc=1
16 xen_platform_pci=1
17 vnclisten="0.0.0.0"
18 vncconsole=0
19 vfb = ['type=vnc,vnclisten=0.0.0.0,vncpasswd=password,vncdisplay=1,keymap=no']
20 acpi = 1
21 apic = 1
22 sdl=0
23 stdvga=0
24 serial='pty'
25 usbdevice='tablet'
26 on_poweroff = 'restart'
27 on_reboot   = 'restart'
28 on_crash    = 'destroy'
```

5.1.2 Experimental overview

The experimental setup of the experiments consisted of planning, designing and implementing the setup in this project. The infrastructure, as illustrated in 5.2 can be divided into three; client, control and server side.

Client side is where workload patterns are feed into *Loader*. Which then simulates the traffic by sending HTTP requests and meanwhile measures response time of the sent requests.

Control side is where the traffic arrives and is further distributed to the web-servers, based on if the experiment is vertical or horizontal. The *controller* runs at a specified control interval and collects performance metrics from the real-time and non-real-time applications. Based on the metrics, a decision is made to either increase or decrease resources through the Xen API, however the actions are performed if the utilization is above or less 80 % of available resources, respectively. The utilization of resources are collected either directly from the VMs or from the hypervisor. VM1 is the *Dom0* and provisioned with sufficient resources to avoid being a bottleneck.

Server side is where the applications are running divided on two PMs. Except the database VM, all of the other VMs has elastic resources which are adjusted by the *controller* in run-time. Because of issues with migration of web servers to perform horizontal scaling, the second web server is booted up in the second PM. Both of the RUBBoS web-applications queries the RUBBoS database for each GET request made by Loader.

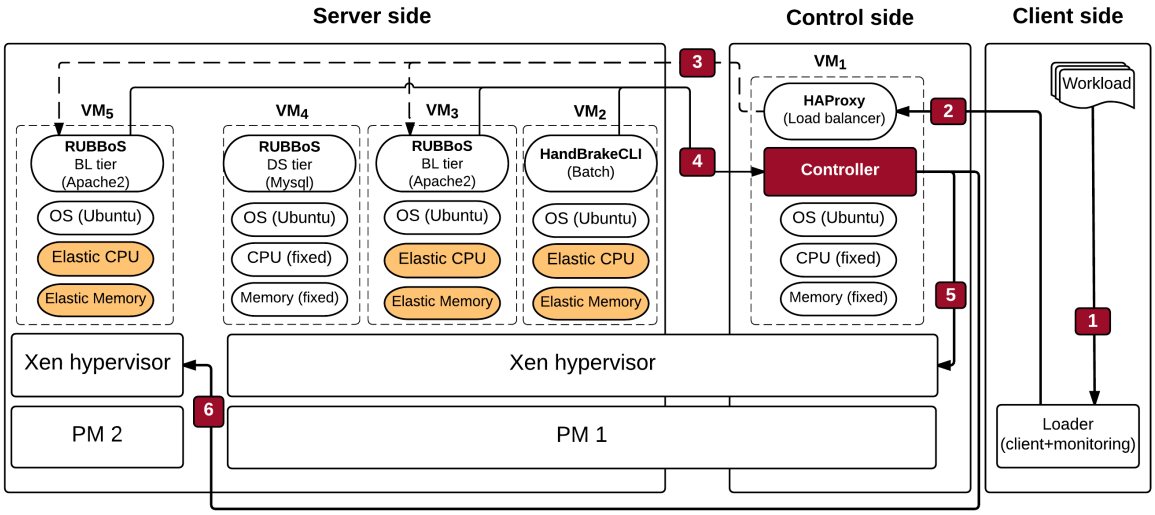

Figure 5.2: Experimental overview

HAProxy

HAProxy was configured to balance the web traffic load between the available web servers. In the first main experiment, with vertical scaling there is only one web server. While in the second experiment with horizontal scaling the traffic is distributed to both of the web servers on the PMs. Since the web traffic is simulated from the internet with the tool *Loader*, *Dom0* runs the load balancers. The algorithm *roundrobin* is used which distributes the traffic equally to both of the web servers. In addition, a statistics front-end is enabled on the default port 1937, this gives real-time overview of the available web servers statistics data. An excerpt from the HAProxy configuration file is shown below, the full configuration file is contained in the appendix section.

HAProxy configuration file (haproxy.cfg)

```
1  frontend http-in
2      bind *:80
3      default_backend web
4
5  listen statistics
6      bind *:1936
7      mode http
8      stats enable
9      stats uri /
10     stats auth user:password
11
12 backend web
13     balance roundrobin
14     mode http
15     server webserver2 10.0.0.11:80 check
16     server webserver1 10.0.0.10:80 check
```

RUBBoS

Setting up the RUBBoS web-application was a fairly involving task, it requires first installing a web server, database server and then initializing the database.

RUBBoS, as earlier described is a bulletin board application, which enables users to browse and submit stories and comments. The application was deployed on a separate VM which runs as a web-server, several packages were needed. *Apache 2.0* with PHP5 was enabled. In addition the VM was configured to have no memory consumption limit for *Apache*. Furthermore, the multi-processing module *Apache MPM prefork* was used, which is thread safe and highly suitable with PHP applications. The *Keep alive timeout* is set to 5 seconds. The parameters for *MaxClients* and *ServerLimit* is set to relatively high values, i.e. 15 000 in the experiments. The values are well above the number of concurrent requests *Apache* has to deal with during the experiments. RUBBoS was configured to display a random webpage which consist of an random item for each request, and each request

requires an *SQL*-query towards the database. The web-page URL was used by the benchmarking tool, *Loader* when performing the experiments.

The database setup consisted of deploying a separate VM where *Mysql* database was installed. To avoid the VM being a bottleneck, it was provisioned with sufficient CPU cores and memory during the experiments. The database table consisted of 1 GB of data and was loaded into the *Mysql* database. The VM had enough allocated memory to avoid disk activity.

HandBrakeCLI

For batch-processing *HandBrakeCLI* was configured on a separate VM. A video file of 3.1 GB was loaded into the VM and a process of converting the file from *.mp4* to *.mkv* was done in the experiments. *HandBrakeCLI* is a CPU intensive tool and is able to perform multi-processing with of all the available CPU cores. The command shown below was used during the experiments. The option *-T* stands for turbo and increases the performance when encoding. Progress of the encoding was followed by looking at the average FPS in the output.

HandBrakeCLI command

```
1 HandBrakeCLI -i input.mp4 -o output.mkv -e x264 -T
```

5.2 Workload patterns

Two types of workload patterns were simulated during the experiments, *trend*- and *spiky*-based traffic.

The *spiky* workload pattern, illustrated in figure 5.3 has two variables which are used as metrics, the number of clients and requests. The simulation of the traffic consist of sudden spikes in the number of requests, being around 20 most of the time and suddenly increasing up to 130 after one, three and and four minutes. There are also some few smaller spikes after the first large spike. The number of clients increases more linearly from zero and up-to in total 120 simultaneous clients. During the experiments, the same spikes are emulated, however when the spiky traffic occurs it varies in time to make it unpredictable. The simulated traffic last for five minutes.

The *trend* workload pattern, illustrated in figure 5.4 is a traffic pattern with more linearly increase in number of clients, from zero and up-to 2900 over ten minutes. The number of requests has a stable increase until six minutes

and then stabilizes around 800 000 requests.

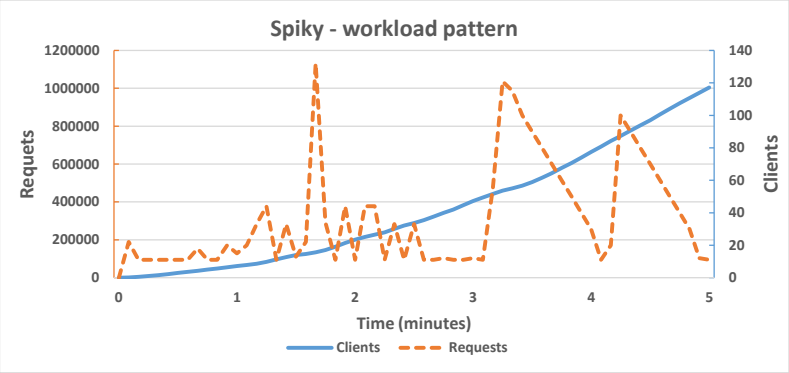

Figure 5.3: Spiky workload pattern

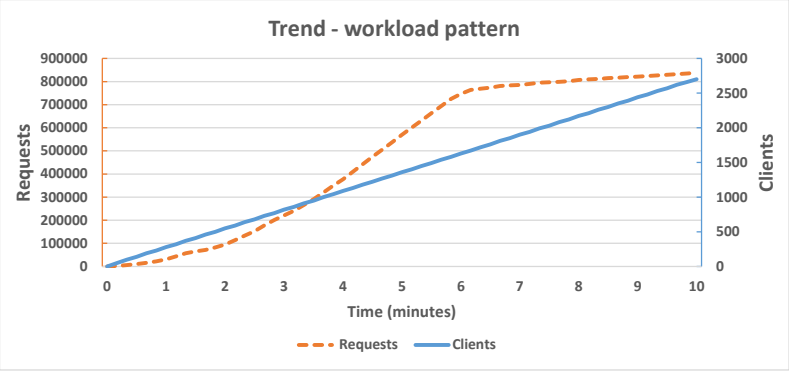

Figure 5.4: Trend workload pattern

These two traffic patterns were simulated during the experiments to analyse the behaviour of the controller with different workload patterns. The simulated traffic arrives from the internet and is distributed by the load balancer on *Dom0* to the running web-servers. Having 1 Gbps network link between the PMs avoided any network congestion while performing the experiments with high traffic load.

5.3 Autonomic provisioning controller

The autonomic provisioning controller solution is developed in Python, which was developed in accordance with the specifications in section 3.4.5. The controller utilizes the hybrid controller model as described in section 4.1. The controller script depends on several modules to work properly,

especially *Libvirt* and some other primary modules which are listed in the main script located in Appendix.

The script contains several functions that are dependant of each other, from measuring the response time to performing actions as to increase resources. The controller runs at an interval of five seconds for the web-application and every five minutes for the batch-job.

There are several parameters that needs to be defined for the controller, as illustrated in table 5.3, minimum-, maximum resources for CPU and memory. The resource parameters are defined for each of the VMs, e.g. the batch-VM is able to allocate in total 10 vCPUs. The minimum memory is defiend as usage plus 512 MB as a buffer, when performing down-scaling. The local url's for the web-servers, desired RT with minimum and maximum interval levels are also defined, the same for the batch-job with FPS.

Table 5.3: Controller parameters

Variable	**Value**
min_vCPU	1
max_vCPU	10
max_memory (MB)	12000
url	http://10.0.0.11/PHP/RandomStory.php
min_RT (seconds)	0.1
max_RT (seconds)	0.5
min_desired_fps	15
max_desired_fps	23
max_fps	25

Because of technical issues with migration of VMs between the PMs, the concept of booting up a new VM on the second PM was adopted. The logic behind performing horizontal scaling lies on achieving desired QoS even if there is a limitation of resources on the first PM, then horizontal scaling is performed on the second PM. For vertical scaling of resources Xen-API is used for *hot-plugging* of memory and vCPUs with the following commands, respectively:

Memory and CPU hot-plugging

```
1 xm mem-set [domain-id] [count in MB]
2
3 xm vcpu-set [domain-id] [count in cores]
```

The decisions in the autonomic provisioning controller script is based on two experiments that has been conducted. The first one is vertical scaling, where there is a conflict of resources on PM and a reallocation is needed to satisfy QoS of the applications. The second is based on horizontal scaling because of resource limitations and each of the experiments covered a

unique use case which is interesting in a scaling scenario.

The decision logic for some of the functionality is illustrated in figure 5.5, starting with measuring the response time and performing decision based on that. Then checking if the maximum resources of the PM is reached, if not vertical scaling is performed based on utilization of the VM. However, if the batch VM is not running with the minimum defined resources, resources are stolen for an amount of time to satisfy QoS-requirements of the real-time application. However for the second experiment, if there are no available resources left, horizontal scaling is performed by booting up a new web-server on the second PM and distributing the traffic between the two web servers. This is done until the simulated traffic comes back to a level where one web server is able to handle the traffic load, then down-scaling is performed.

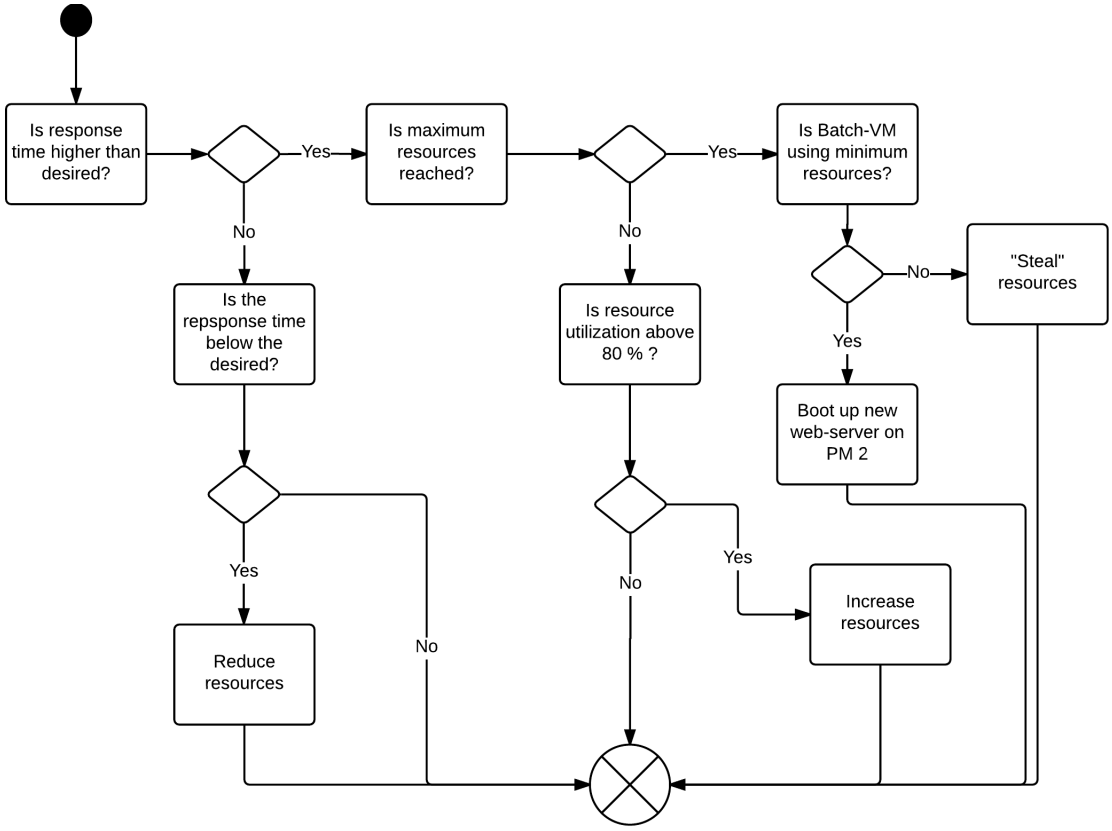

Figure 5.5: An activity diagram showing the decision logic determining whether or not to scale based on QoS-requirements.

The controller has defined control interval for the interactive and non-interactive application. Since the batch-job is less latency critical it is measured every 5 minutes, while the latency-critical application is measured either 5 or 15 seconds based on the initial results. As previously mentioned, Xen API is used for vertical scaling of resources. Resources is added and

removed static by one vCPU at a time based on the QoS of the applications. While memory is added by 624 MB if the utilization is above 80% and if the application level metrics are above the desired. The memory is reduced in smaller amount for each control iteration, by 405 MB until reaching the baseline.

To measure the response time, the *requests* module in Python is implemented as illustrated below, if the web server does not respond to the sent request within the desired interval. An exception triggers, which means the response time is high and resources needs to be added by calling the increase_resources function. While if the response time is below the desired, resources are removed and if the response time is within the desired interval no action is taken by the controller. The controller measures a sample for each control interval, unlike the traffic simulation tool which measures the average response time, however the hit ratio for the controller is close to the average values. Furthermore by measuring the average response time over 5- or 15 seconds is a long enough period to filter measurements noise, but short enough to highlight the transient behaviour of the application.

Decision making function of response time (excerpt)

```
1  def response_time(desired_resp,url,host,server,location):
2      try:
3          time = requests.get(url,timeout=desired_resp).elapsed.
               total_seconds()
4      except requests.exceptions.Timeout as f:
5          print "High response time"
6          increase_resources(host,server,location)
7      else:
8          print "Low reponse time"
9          if time < minresp:
10             reduce_resources(host,server,location)
11         else:
12             print "Ok - Response time"
13             pass
```

5.4 Experiments

5.4.1 Testing and experiments

After a successful implementation of the provisioning controller and configuration of the required tools and applications, the next part of the project consisted of running the experiments. The functionality of the infrastructure was also tested in order to remove bugs in code and configuration files. In addition, each VM was destroyed and re-created for each experiments to be fresh, even rebooting did not help. The cache of the PMs was cleared after each experiment.

Several tests were conducted:

- Deployment of multiple VMs to measure boot time
- Run traffic simulation to measure if there are any bottlenecks in the infrastructure
- Run the autonomic provisioning controller and perform tuning
- Testing the accuracy of the measurements

5.4.2 Initial experiments

For the initial experiments several scripts were developed in order to test the environment. The setup for the three experiments; parallel computing, hot-plugging of CPU and memory will in this section be outlined.

The experiments with parallel computing were performed by using a script in the programming language C. The script has two variables that needs to be defined, maximum prime numbers and the number of CPU cores it is going to run on. The script is executed with the time option to observe how much of the time is spent in *real* and *user* mode, as illustrated below. In this example the system manages to calculate 100 000 prime numbers one time in three seconds.

Output from parallel computing script

```
1 root@bilal2:~# time ./a.out
2 Calculated 9592 primes.
3 This machine calculated all prime numbers under 100000 1 times in 3 seconds
4 real    0m3.894s
5 user    0m3.891s
6 sys     0m0.003s
```

With the memory and CPU hot-plugging experiments, there were developed two Bash scripts. One running on the hypervisor and the second on a VM. As previously mentioned the Xen API was used in order to perform hot-plugging of CPU and memory.

For the CPU part, there were performed two main experiments, adding and removing one and three vCPU cores for each iteration which lasted for one second. The script running on the VM monitored the number available vCPUs every 0.5 seconds.

For the memory part, there were also performed two experiments adding and removing 64 MB and 1 GB for each iteration. Each second memory was added or removed and every half a second the VM measured the amount of available memory from *free* which gets the data from */proc/meminfo*. The */proc* is the virtual file-system and contains the runtime system information

and is the interface towards the kernel data structures.

The initial experiments were conducted 30 times to get as accurate results as possible and to remove systematical errors in the results. During the experiments, processes were turned off and for each experiment the system was rebooted to have a fresh as possible state. In addition, the same experiments were performed on the second PM in order to observe if that showed any significant differences and to verify the acquired results.

5.4.3 Main experiments

The following three main experiments were conducted:

- Control interval
- Resource conflict - Web server and batch (vertical scaling)
- Resource conflict - Web server and batch (horizontal scaling)

The first scenario *control interval* consisted of running the autonomic provisioning controller every 5- and 15 seconds and compare the results. Both of the experiments were conducted with the same testbed, the VMs were destroyed and recreated. The *spiky*-based workload pattern was used with clients increasing from 100 to 1000, with spikes in the number of requests during the experiment lasting for 5 minutes. During the experiments, the allocated memory, CPU cores and usage were monitored on the hypervisor and directly on the web server. The provisioning autonomic controller script was executed right before the the workload was generated in order to observe if the environment was stable and the controller worked as intended.

With the second scenario, resource conflict leading to vertical scaling. Both the web server and batch was initialized for the experiment. The *spiky*-based workload pattern was used, starting from 0 and peaking at 1800 clients during a time-period of 5 minutes. The traffic load was high enough and causing the web server to "steal" resources from the batch for a short amount of time. The autonomic controller measured the QoS of the applications in a interval of 5 seconds and 5 minutes, for the web server and batch. Allocated resources and used were measured and stored in log file further analysis.

The last experiment, resource conflict leading to horizontal scaling was performed with the same settings as in the previous experiment. In addition, horizontal scaling is performed when the maximum number of resources is reached on the PM, in addition horizontal down-scaling is performed in relation to the traffic load. The workload is *trend*-based with

increasing number of concurrent clients from 0 to 2700 during a time-period of 10 minutes.

5.5 Results from initial experiments

The main concept behind performing the initial experiments is on monitoring the impact on the performance, while scaling the resources. The goal is to provide *proof of concept* on how rapid the system reacts to changes in the environment, and if the performance degrades in the long-term. In addition, monitoring how much the performance increases or reduces when increasing or reducing resources, respectively. This gives a better understanding on what we can expect when there is a need to make changes in the environment. The following initial experiments were conducted:

- Parallel computing
- Vertical elasticity (hot-plugging)
  - CPU
  - Memory

5.5.1 Parallel computing

The concept of the first experiment, *Parallel computing* on the multi-core CPU architecture lies on measuring the performance when adding vCPUs. This was done to get an idea about how much the CPU performance increases with Hyper-Threading enabled. The result from this study is essential for the behaviour of the hybrid controller.

This was done by running a script calculating prime numbers up-to 100 000 in parallel on the allocated vCPUs. The two parameters which was considered were *real* and *sys* time. The *real* time is the wall clock time, from the start until finish of the call. While the *sys* time is the CPU time spent in kernel mode during the execution.

The results from the experiment is illustrated in figure 5.6, the graph shows that with one vCPU, the *user-* and *real* time is equal. With two vCPUs the *user* time doubles, which means the two vCPUs manages to perform the prime number calculation in parallel within the same *real* time. This happens until it reaches eight vCPUs, the *user* time increases linearly, by three seconds for each vCPU. Then both the *user-* and *real* time slightly increases for each added vCPU until 16 vCPUs with the 2:1 ratio between vCPUs and physical CPU core.

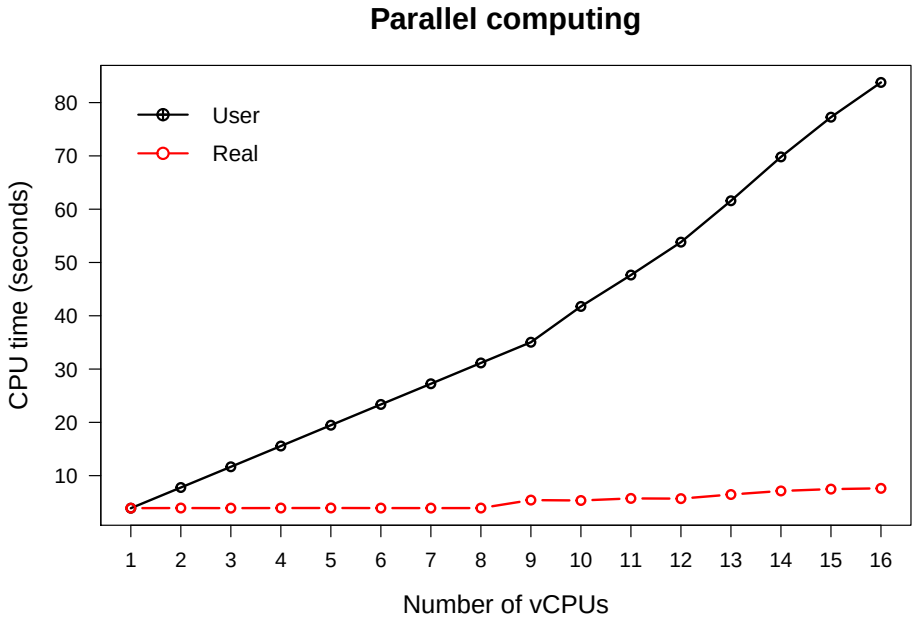

Figure 5.6: Parallel computing from 1 to 16 vCPUs

5.5.2 CPU hot-plugging

In the second part of the initial experiments, the focus lies on measuring the time it takes from allocating vCPUs and until the VM maps the cores. The main purpose of this experiment is to measure the performance between the time vCPUs is added or removed until it is usable by the VM.

The results from the CPU *hot-plugging* are illustrated in figure 5.7 and 5.8, accordingly. In the first experiment, one vCPU was added to a VM in a sequence of one second from two to sixteen vCPUs. Meanwhile the VM measured the number of available vCPUs every half a second. As the figure shows, when appending vCPUs from two to five vCPUs, the cores are immediately mapped on the VM. From six to eight cores, the mapping takes half a second. The time is increasing, and in the last sequence the mapping is one and half second.

In the second part of the experiment, vCPUs were removed with one for each iteration from sixteen to two. As the results shows, for the first six iterations the cores are immediately mapped by the VM. For the next four iterations the reflections took half a second, while for the last part the mapping took one second. Similar to when adding, removal of vCPU cores is in the first part immediately mapped, while in the second half the time increases up-to one and half seconds when adding. The total time for adding is sixteen seconds and for removal, the time is fifteen seconds.

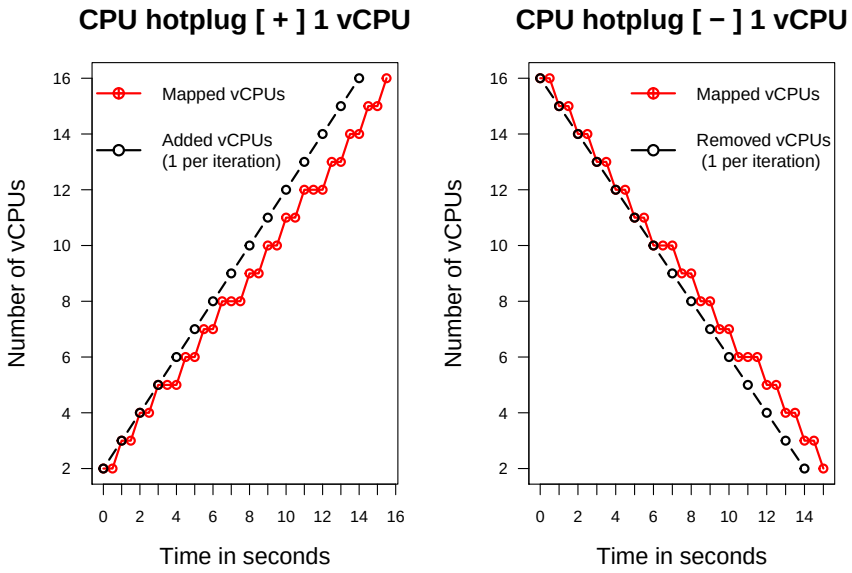

Figure 5.7: CPU hot-plugging when appending and removing vCPUs by one for each iteration

When performing the same experiment, but in this case with three vCPUs for each iteration. The results as illustrated in figure 5.8 starting with four and up-to sixteen vCPUs. As the chart shows, the VM manages to immediately map the added vCPUs from four to sixteen. The VM observes an increase of twelve vCPUs within a four seconds time-period.

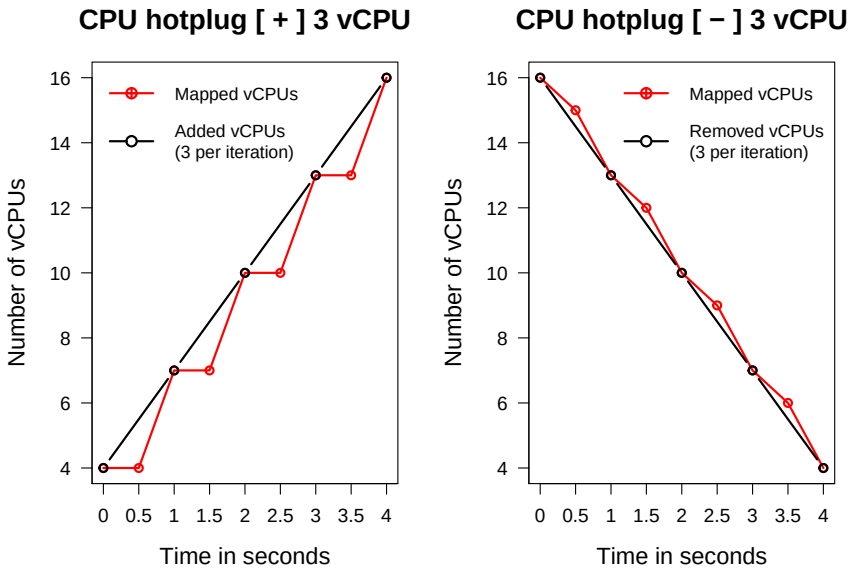

Figure 5.8: CPU hot-plugging when appending and removing vCPUs by three for each iteration

For the second part of the experiment, removal of vCPUs. The VM starts with sixteen cores and for each second three cores are removed. The result shows the VM manages to detect the changes instantly. However, the removal is not instant from sixteen to thirteen, but the value decreases by one for every half a second. The VM observes in this case decrease from sixteen to four vCPUs in time period of four seconds.

5.5.3 Memory hot-plugging

In the last part of the initial experiments, the focus lies on measuring the time it takes from allocating or removing memory and until it is mapped on the VM. The main purpose of this experiment is to measure the performance between the time memory is added or removed until it is usable by the VM. This was first performed with 64 MB segments and then with 1 GB segments, to measure the time until the VM maps the allocated memory address space. The results are illustrated in figure 5.9 and 5.10, accordingly.

When adding 64 MB of memory each second, as the graph illustrates the memory is instantly available for the VM for the first two iterations, beginning from 1088 MB. For the next three iterations there is a delay of half a second before the memory is available for the VM. For the next iterations the delay is one second and gets as high as one and a half seconds in the last iterations.

In the second part memory was removed in 64 MB segments for each iteration starting from 2048 MB to 1024 MB during sixteen seconds. The VM reflects the removed memory instantly for the first five iterations, from there on the delay increases to half a second and then in the end to one and a half second.

For the last experiment, as illustrated in figure 5.10, 1 GB of memory was added starting from 2048 MB and up-to 6144 MB. As the graph shows, the VM manages to map the first iterations immediately, after that the delay is half a second. In addition, the VM maps the values between the segments of 1 GB. In a time-period of four seconds 4 GB of memory is added.

On the other hand, for the removal the VM detects the removed memory rapidly for the first two iterations, for the third iterations it does not manage to detect the correct value, and for the last iteration the delay is half a second. Is this case also, the VM map the values between the segments of 1 GB.

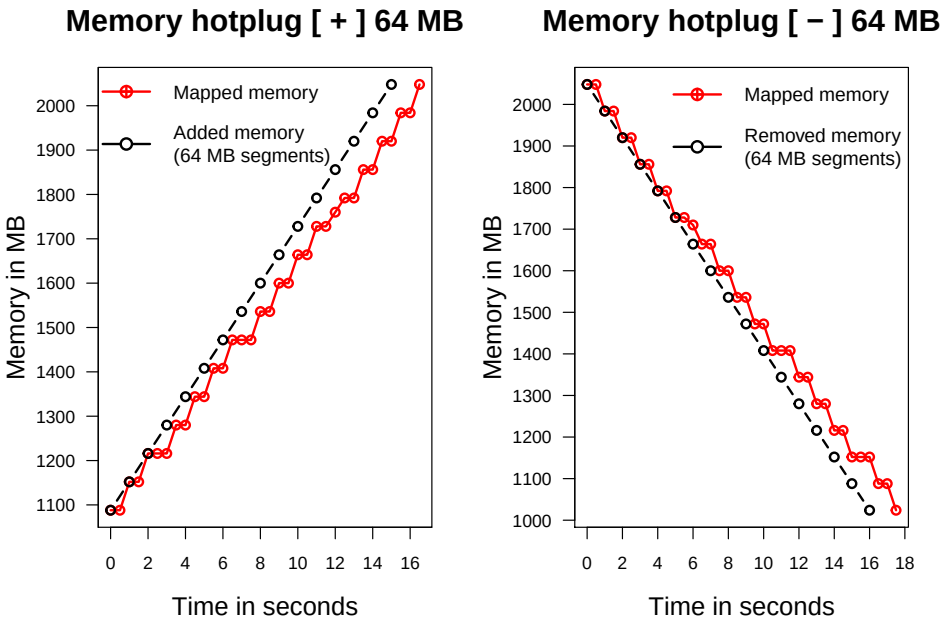

Figure 5.9: Memory hot-plugging of 64 MB for each iteration

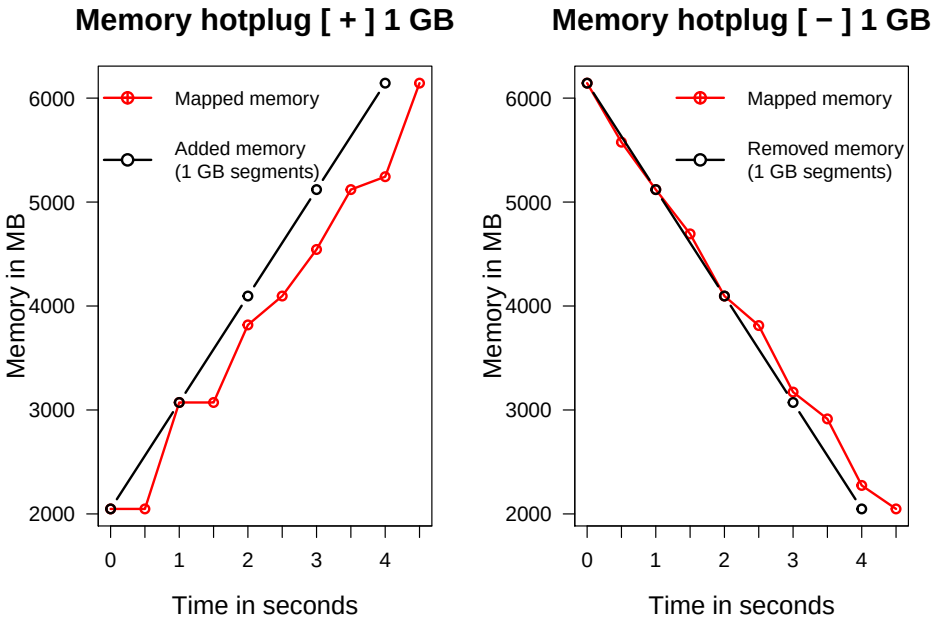

Figure 5.10: Memory hot-plugging of 1 GB for each iteration

Chapter 6

Result III - Measurements and Analysis

This chapter covers the results for all the conducted experiments. The *control interval* experiment are first presented before the main experiments. The results from the experiments are based on each other and will be presented in that order.

After the experimentation process, analysis on the acquired measurements from each VM will be presented in graphs to ease the readability. The data is collected during the experiments and contains utilization metrics. Sometimes other methods will be used if that is deemed necessary.

6.1 Control interval

The control interval experiments were performed in two iterations, with 5 and 15 seconds interval with the controller. The results are illustrated in figure 6.1 and 6.2, with the main metrics, average response time and vCPUs. The min and max response time are visualized with straight dotted lines, the desired response time is within the interval of 100 ms and 500 ms. The experiment lasted for 5 minutes with increasing traffic from 100 to 1000 clients. The *spiky-based* workload pattern was chosen for this experiment, as previously described in section 5.2.

Starting with the 5 seconds interval, there are three main spikes in the response time and this can relate to spikes at the same moment in the workload patterns. The number of vCPUs increases in relation to response time and manages to reduce the time to within the desired interval in all of the three spikes. In the first two spikes there are allocated at maximum three vCPUs and for the last, five are allocated. For the last spike the response time rise rapidly and reaches right below 1600 ms.

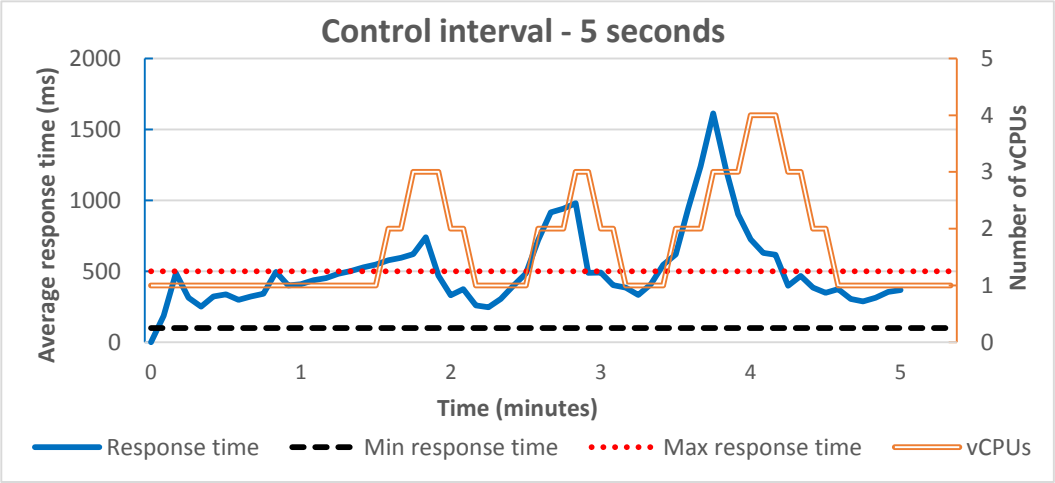

Figure 6.1: Running the controller every 5 seconds with *spiky-based* workload pattern

For the 15 seconds interval, as illustrated in figure 6.2, there are two main spikes in the response time. The first one reaches 800 ms, while the second one reaches right above 1200 ms. For the first spike three vCPUs are allocated and for the second spike four vCPUs are allocated in relation to the response time.

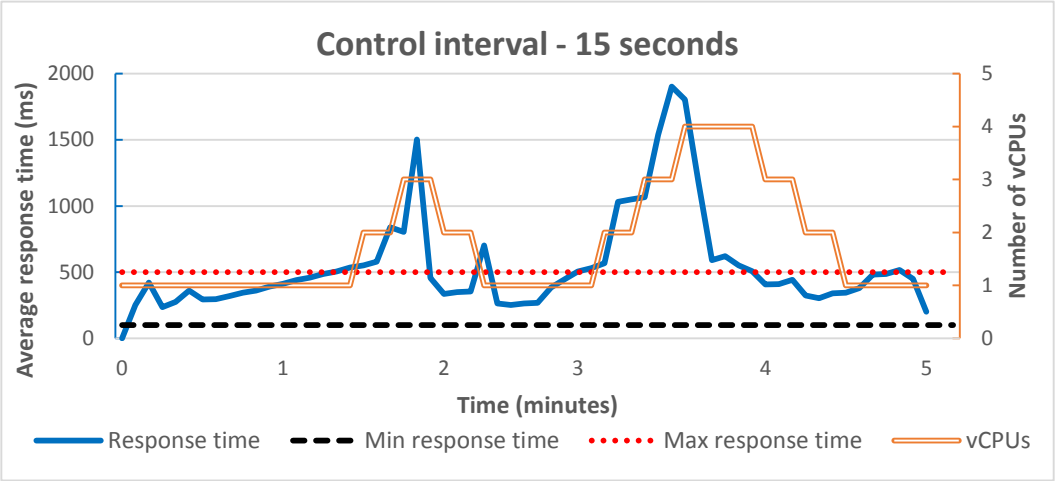

Figure 6.2: Running the controller every 15 seconds with *spiky-based* workload pattern

6.2 Resource conflict - Web server and batch (vertical scaling)

The results from the experiments with resource conflict between the web server and batch on a single PM is illustrated in figure 6.3 and 6.4. The workload pattern for the experiment was *spiky-based* from 0 to 1800 clients during 5 minutes, which means there are several fluctuation in traffic patterns. In addition a control interval of 5 seconds is used based on the results from the *control interval* experiments.

There are two main traffic spikes in the response time, the first one reaching a peak of 900 ms, and the second spike rising up-to 700 ms. The controller allocates resources within a short time after the traffic spikes, and manages to decrease the response time to the desired interval. The number of vCPUs are allocated in relation to the increasing response time and reaches a top of seven vCPUs. Before allocating seven vCPUs, the batch VM gets reduced by one for the web server to manage keep the response time low. After 5 minutes the workload is finish and the web server releases the vCPUs, which then is allocated to the batch VM to take back the lost calculated FPS.

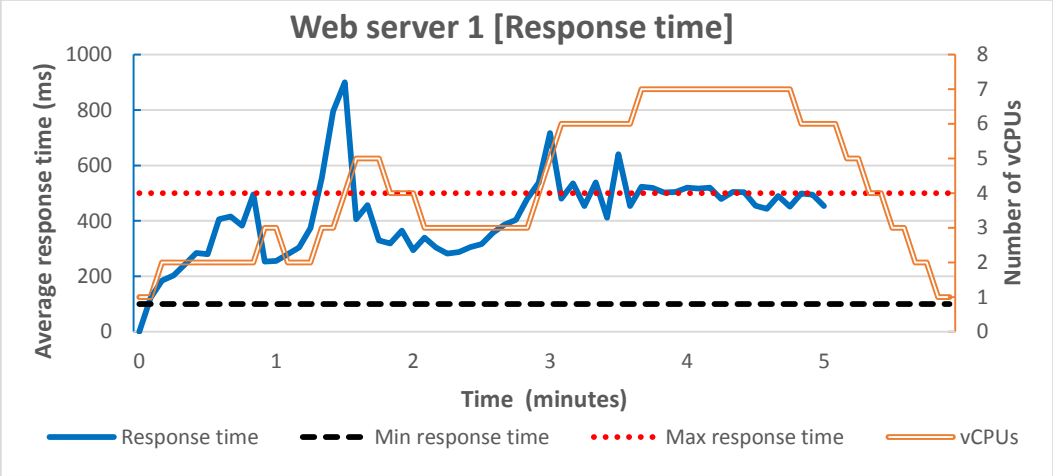

Figure 6.3: Web server 1 - response time in relation to vCPUs

For the batch job, the FPS starts above 40 FPS and drops slowly to the desired interval, during the beginning there are two vCPUs allocated and drops to one which is as explained above allocated to the web server VM. At the tenth minute , the control interval for the batch VM is ran again and the FPS is right at the minimum desired FPS and a new vCPU is allocated. The FPS slowly increases and manages to take back the lost calculated FPS between the fifth and tenth minute. The batch job finishes within the desired time-frame of 25 minutes.

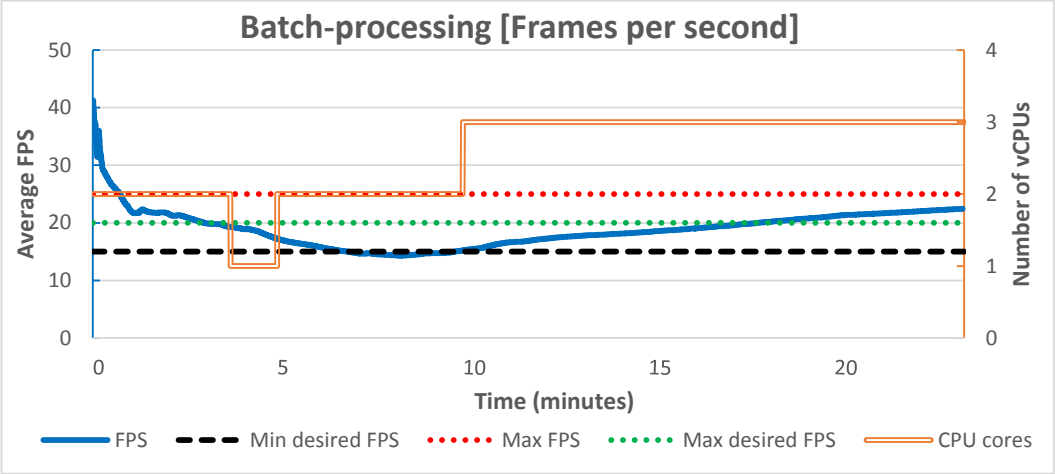

Figure 6.4: Batch-processing - FPS in relation to vCPUs

6.3 Resource conflict - Web server and batch (horizontal scaling)

With the second main experiment, resource conflict between web server and batch leading to horizontal scaling. The *trend-based* workload pattern is simulated in this experiment from 0 to 2700 clients during 10 minutes, which means the traffic increases up to 2700 simultaneous clients by the end of the test.

As illustrated in figure 6.5 and 6.6, the response time jumps to 3000 ms and 4000 ms after three minutes, the number of vCPUs increases in relation to the spikes in response time. Before allocating seven vCPUs, one core is removed from the batch VM, but still the response time is high and a new web server is created on the second PM. The vCPUs then gets reduced to the half, and the second web server get the same number of vCPU. Then the number of vCPUs increases on both of the web servers and manages to decrease the response time to the desired interval.

The batch VM starts right below 45 FPS, but the number of FPS reduces after the web server takes one vCPU. The core is allocated back at the next control interval for the batch job. However the FPS is below the desired at the tenth minute and then another vCPU is allocated. The FPS increases steady and manages to take back the lost calculated FPS. The batch job finishes within the desired time-frame of 25 minutes.

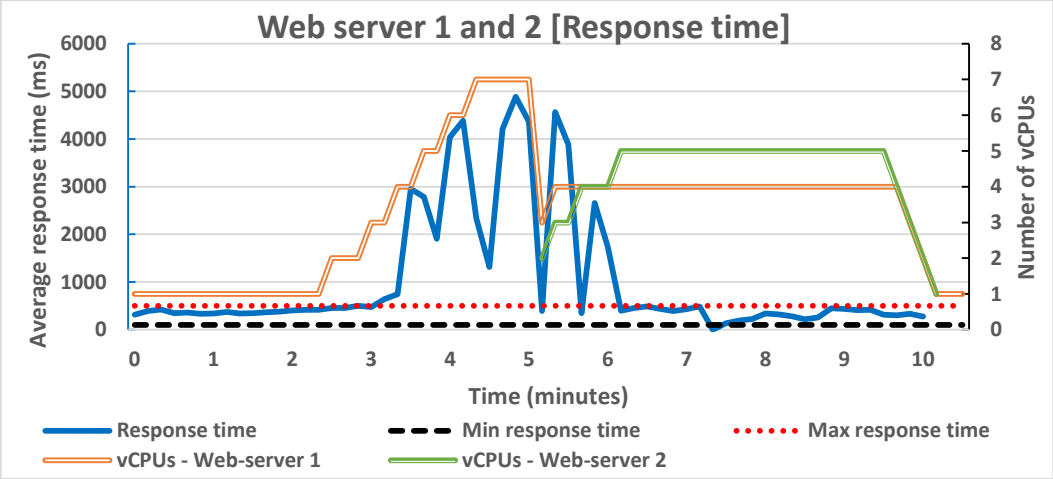

Figure 6.5: Web server 1 and 2 - response time in relation to vCPUs

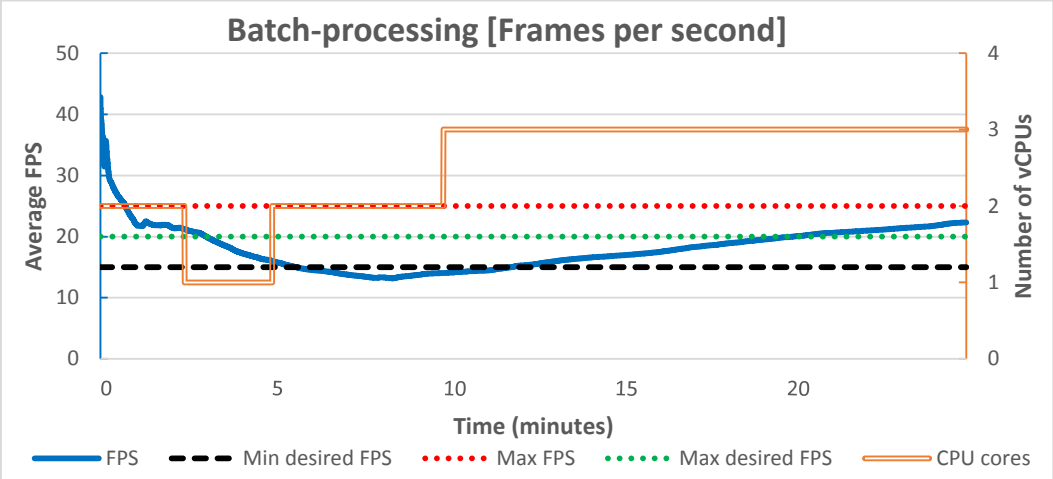

Figure 6.6: Batch-processing - FPS in relation to vCPUs

6.4 Analysis

In this section a thorough analysis will be presented based on the obtained results. A comparison will be performed on the data sets in order to investigate if the behaviour of the autonomic provisioning controller is as expected.

6.4.1 Control interval

The first main experiment, *control interval* in order to find the most suitable control interval for the autonomic provisioning controller. The results shows the 5 seconds controller has an average response time which is 21 ms less than with the 15 seconds control interval.

As shown in table 6.1, with the 5 seconds control interval the web server managed to respond to 17 072 more request with the HTTP response code 200. However with the 5 second control interval the web server had 43 more timeouts in comparison to the 15 seconds control interval.

The average utilization of resources is illustrated in figure 6.7, the 5 seconds control interval had 7.14% higher memory utilization than the 15 seconds control interval during the experiment. The 5 seconds control interval also had 6.42% higher CPU utilization than the 15 seconds control interval. The complete data sent by *Loader* in requests is 2.08 MB larger- and the amount of received data in responses is 42.29 MB larger with the 5 seconds control interval.

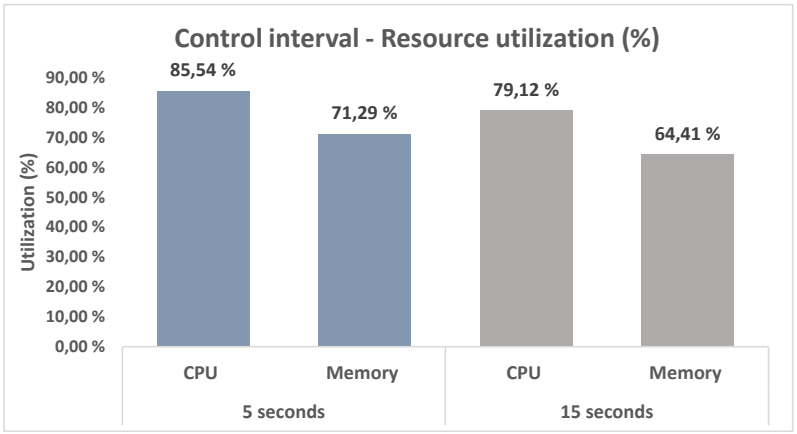

Figure 6.7: Control interval experiment - Average utilization of resources

Table 6.1: Web server metrics with 5 and 15 seconds control interval

Metrics	**Control interval 5 seconds**	**Control interval 15 seconds**
Average response time	442 ms	463 ms
Response code: 200	371 311	354 239
Response code: 400/500	0	0
Timeout	110	67
Bandwidth - Sent	45.11 MB	43.03 MB
Bandwidth - Received	922.43 MB	880.14 MB

The 5 seconds control interval had an average amount of violations of 296.52 ms in terms of response time from the baseline. While 34.64% of the requests violated the SLA-parameters, and had an response time which was above the desired interval. On the other hand, 63.33% of the requests was within the desired interval.

With the 15 seconds control interval the average amount of violations was 368.57 ms in response time from the baseline. The percentage of traffic

that violated the SLA was 37.70%, while 60% of the traffic was within the desired interval.

6.4.2 Resource conflict - Web server and batch (vertical scaling)

In the second experiment based on the results, as shown in table 6.2, the average response time for the web server was 426 ms and 18.96 FPS for the batch-job during the experiment. As shown in table 6.2, the amount of requests which received HTTP response code 200 was 627 911, there were no requests that received the HTTP response code 400 or 500. 136 of the requests received timeout, furthermore the data sent in requests was 72.28 MB and received in responses was 1.52 GB.

Table 6.2: Web server metrics with vertical scaling

Metrics	Web server
Average response time	426 ms
Average FPS	18.96 FPS
Response code: 200	627 911
Response code: 400/500	0
Timeout	136
Bandwidth - Sent	76.28 MB
Bandwidth - Received	1.52 GB

The resource utilization, as illustrated in 6.8, the web server had an 60% average utilization of memory, while the batch had an average memory utilization of 75.61% during the experiment. The web server had an average CPU utilization of 86.32% during the experiments, while the batch on the other hand had an average CPU utilization of 89.34%.

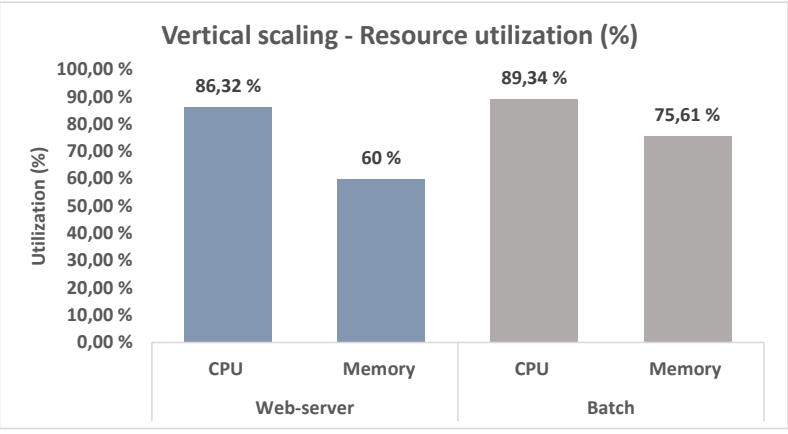

Figure 6.8: Vertical scaling experiment - Average utilization of resources

The average amount of violations was 78.23 ms in response time above the baseline. The overall amount of responses which violated the SLA was 27.87%, and 71.67% of the requests was within the desired interval. The rest of the percentage was below the desired interval.

6.4.3 Resource conflict - Web server and batch (horizontal scaling)

With the horizontal scaling experiment, the results as shown in table 6.3 the average response during the experiment is 494 ms. The average FPS for the batch job is 18.65 within the desired interval of 15- and 20 FPS. There were in total 789 992 requests which received HTTP response code 200, while there were non requests which revived HTTP response code 400/5000. Of the total amount of requests, 13 023 of them received timeout, *Loader* sent 97.83 MB data in requests and received 1.92 GB data in responses.

Table 6.3: Web server metrics with horizontal scaling

Metrics	**Web server**
Average response time	494 ms
Average FPS	18.65 FPS
Response code: 200	789 992
Response code: 400/500	0
Timeout	13 023
Bandwidth - Sent	97.83 MB
Bandwidth - Received	1.92 GB

The average amount of violations was 2463.94 ms in response time above the baseline. The amount of requests which violated the SLA-requirements was 26.23%, and 72.13% of the requests was within the desired interval.

The utilization of resources, as illustrated in figure 6.9. The average utilization of CPU is higher on web-server2 than web-server1, the differentiation is 3.38%. The web-server2 had a higher memory utilization being at 73.99%, while the web-server1 had an average memory utilization of 64.80%. The batch VM had higher utilization of both memory and CPU, 84.43% and 89.67%, respectively.

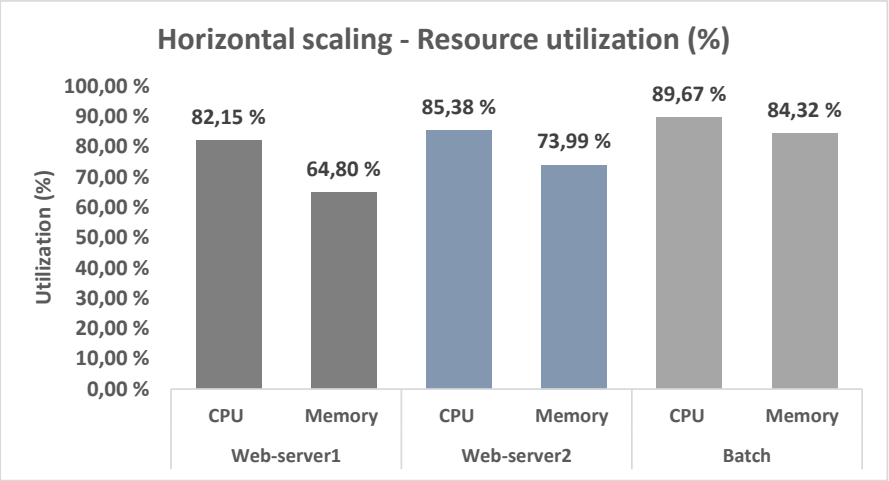

Figure 6.9: Horizontal scaling experiment - Average utilization of resources

Chapter 7

Discussion

This chapters reflects the results from the different stages described in this study, furthermore highlighting both theoretical and practical challenges during the project. In addition, improvements of the developed autonomic provisioning controller are suggested and future work is proposed.

7.1 The problem statement

First of all, the main goal of this project was to design and implement a autonomic provisioning controller, as described in the approach chapter 3. The problem statement has been the main focus during the project period and for convenience the research questions have been listed below:

- *How can we create an autonomic provisioning controller based on elements from control theory to increase server utilization, and at the same time expose available resources to ensure QoS of real-time and non-real applications?*
- *How to coordinate vertical and horizontal scaling for better resource allocation?*
- *How to benefit from application level metrics in order to efficiently provision resources to interactive and non-interactive applications?*

In order to adequately answer the problem statement, several approaches have been considered during the design stage of the project. To address the problem statement, an autonomic provisioning controller has been designed and implemented based on elements from *control theory* with the focus on the QoS of the applications. Several initial experiments has been conducted to have a fully understanding of the environment capabilities.

7.2 Evaluation

Results from the proposed solution in this project have disclosed interesting findings. By taking advantage of hot-plugging capabilities offered from the hypervisor and scaling based on QoS-requirement, the server utilization increased tremendously. The developed autonomic provisioning controller managed to increase the memory- and CPU utilization by 6.88% 6.42% compared to each other in the control interval experiments. The results also showed an improvement in the average response time, and managed to reduce it by 21 ms. The violating of SLA was reduced by 3.06% with the 5 seconds control interval.

Based on these results the 5 seconds control interval was chosen for the vertical and horizontal scaling experiments. With the vertical- and horizontal scaling experiments, the controller worked as intended by performing the correct decisions to achieve the desired QoS for the applications. The controller managed to track the spiky workload pattern in an early stage in order to reduce the SLA violations without over-provisioning to a greater extent. However with the trend-based workload pattern, the controller managed to follow-up with the increasing traffic to the point of reaching the maximum resources. With horizontal scaling it took some time before the new web server spawned, from there on the web servers managed to control the increasing traffic load. The results showed the behaviour of the controller worked as intended, however there are always room for improvements, this will be outlined in the future work section.

7.3 Implementation of the autonomic provisioning controller

The goal of this paper was to improve server utilization by coordinating resource allocation between real-time and non-real-time applications in runtime. Implementation of the autonomic provisioning controller consisted of planning in multiple stages, defining a *hybrid* controller model which was implemented based on logic from *control theory*.

The controller was evaluated with three different scenarios; control interval, resource conflict leading to vertical scaling and resource conflict leading to horizontal scaling. The main goal behind the scenarios was to create situations which could happen in a production environment, and required different actions in order not to violate the SLA-requirements.

The results from the experiments clearly showed the expected improvement with a significant increase in utilization of resources. In addition the

average response time was in all cases within the desired specified interval. Furthermore, the violation varied between the experiments. However due to high load in all of the experiments in a short amount of time was the reason for a somehow high violation in some cases.

The implementation of the controller in the design stage had an early focus on coordinating the memory and CPU sub-controllers with the proposed *hybrid* controller model in order to efficiently provision resources to the interactive and non-interactive applications. The resource utilization during the experiments was in most cases between 70% and 80%. The validity of the results was confirmed during the initial experiments, e.g. the actual achieved performance when allocating vCPU cores with Hyper-Threading enabled. In addition, performing the experiments on both of the PMs and analyzing the collected measurement metrics.

The implementation of the controller also consisted of a lot of time spent in testing and performing optimizing. The decisions made were not always correct, as resources were reduced when the response time was above desired. In order to achieve the desired state of the environment before performing each experiment, several actions needed to be conducted from destroying the VMs and recreating them to cleaning log files.

However the results were not easily achieved as several challenges had to be faced during the project in order to properly configure the environment and get the behaviour of the controller to work as intended.

7.3.1 Challenges during the implementation

During the project period there has been a lot of technical challenges with the implementation. One of the most challenging problems during the project was developing and debugging the code, the amount of if-statements required a lot of logic to be in place in the initial design before the actual development.

During the configuration of the environment the network set-up for the VMs was challenging and different solutions had to be thought out. Since the BIOS version of the PMs did not support Ubuntu 14.04 LTS, the 12.04 version had to be installed. The 12.04 version of Ubuntu only supported Xen version up-to 4.1, while *Open vSwitch* is supported from Xen 4.3 and onwards. Because of this limitation, a bridge between the PMs had to be created with its own subnet.

7.3.2 Other considerations and limitations

Migration limitations

The initial plan of the project was to perform horizontal scaling by live migrating VMs to the second PM. This required a shared-storage to be created where the disk-images had to be placed. A NFS share was created in order to accommodate the VMs on both of the PMs, however during the experimental phase, the VMs started to break down due to errors in memory, when the live-migration was performed. After asking questions in communities and getting replies that the bugfix for the error did not make it into the version of Xen version that was used. This was a huge setback for the project since horizontal scaling was a huge part of it, after further discussion a new plan was made. The solution was then to spawn a web-server on the second PM instead of performing live-migration when there was a limit of resources on the first PM. However, a lot of time was lost in implementing the solution and preparing the environments and this reduced the overall progress of the work. In retrospect, migration of the web server would not been an ideal solution since spawning a new on the second PM is faster to perform, however migration of the batch VM could have been a solution when there is a limit of resources.

Hyper-Threading

The PMs used in the thesis, both had Intel processor with Hyper-Threading, which consisted of 8 physical- and 8 logical CPUs. When performing vertical scaling there was no control whether a virtual or physical CPU was allocated. Even though initial experiment on parallel computing was performed to observe how much the performance increased, and the results showed that it took double the time to execute a task in parallel from 8 to 16 vCPUs in comparison to 1 to 8 vCPUs. If there had been more time in the project, a comparison by performing the experiments on a AMD processor would been highly suggested in order to observe how much the results varied.

Workload patterns

To create more realistic workload patterns and for a longer time-period. Since web-traffic varies during a day with peaks at specific times. It would be ideally to create some similar scenarios in order to reduce the violation rate. The conducted experiments for the web server lasted for 5 and 10 minutes with high load during the experiment period.

7.4 Future work and improvement suggestions

There are always a potential for improvement and this work certainly is not an exception. Several features and functions can be be developed in order to improve and expand the capabilities of the autonomic provisioning controller.

Some main improvements of the autonomic controller consist of more logic into the controller, scaling is performed static, but this has shown good results. However implementing solutions as PID controller as explained below would help in terms of allocating resources based on the QoS metrics. To illustrate, e.g, if the response time increases by 200 ms every second, a proportional based logic can be implemented to predict and allocate the needed resources for the next seconds. This would help to remove some of the spikes in response time as observed in the results.

In addition with horizontal scaling, instead of using *round-robin*-based distribution of traffic with the load-balancing, a solution for *weight*-based distribution based on capacity of the web servers. This solution would be more ideally, proportional-based distribution would reduce the fluctuation in the response time. Traffic will be distributed equally based on amount of resources and this would give a better QoS of the applications. Another solution would be to distribute the traffic load based on each of the web servers local response time. If the response time is low more traffic can be distributed to the server, while if the response time is high less traffic is distributed.

Machine learning could be implemented in order to perform *peak-detection*, the approach in this thesis focuses on reactive actions, however in order to detect peaks in the traffic and to achieve proactive decisions to limit the SLA violations, prediction has to be implemented with machine learning.

Bin packing can be used in order to improve consolidation when performing horizontal down-scaling. However this requires a method to know when the system is stable before the consolidation is performed. When the applications are satisfying the QoS requirements, the consolidating can be performed from two to one web server.

PID is an acronym for the mathematical terms proportional–integral–derivative, and the PID controller works as *control theory* with a control loop feedback mechanism. Implementing PID controller in order to adjust resources, the controller provides formal guarantees on the properties of the controller. However the problem with PID controller is that the same actions are performed if the measured value is above or below the action point. In our case the more important is to at any cost reduce SLA violations, in addi-

tion PID requires a lot of testing and the values has to be tuned in order to achieve the desired results.

Reinforcement learning (RL) is a technique which consist of trial-and-errors methods [54]. Using RL, different actions are performed during numerous system states and learns from the consequences of each action. In this project scaling is performed based on performance and capacity, however since applications needs an arbitrary combination of resources in order to achieve the desired SLA-requirements. RL can be adopted to dynamic environments to support distributed controllers in order to suit complex, and large-scale systems. RL offers potential benefits in autonomic computing and this is highly needed in the context of a web server environment.

7.5 Potential impact of the thesis

This thesis started with the desire to increase server utilization in data centres by using techniques such as, *hot-plugging* to perform efficient decision-making. Vertical elasticity is recognized as a key enabler for efficient resource utilization of cloud infrastructure due to fine-grained resource provisioning, fractions of a CPU can be leased for as short as a few seconds. However, at the time of writing there is still limited *hot-plugging* support offered by several hypervisor vendors, as illustrated in table 2.1.

In this thesis, an autonomic provisioning controller has been designed, implemented and tested in several scenarios. The controller has been evaluated in terms of improvements and future work. The controller is SLA-driven by facilitating QoS of heterogeneous applications using a hybrid controller model. Both vertical- and horizontal scaling has been evaluated. While some recent work has focused on vertical scaling (section 2.11) to improve QoS of applications, however the proposed solution is scalable and takes horizontal scaling of the environment into consideration.

It is worth mentioning that we currently are working on a research article in order to disseminate the results of the current work.

Chapter 8

Conclusion

The aim of this thesis was to investigate how server utilization in data centres could be improved by coordinating vertical and horizontal scaling. The main purpose was to find out how resources could be managed more efficiently.

The problem statement was addressed by designing and implementing an autonomic provisioning controller based on elements from control theory. The hybrid controller model was chosen as a foundation for the controller after evaluating several models. The experiments were conducted on two physical machines running Xen Hypervisor with support for *hot-plugging* of resources to tackle load bursts of heterogeneous applications. Several experiments were conducted, consisting of both vertical and horizontal scaling to achieve the desired QoS of the real-time and non-real-time applications. The application level metrics were used in decision-making in order to efficiently provision resources to the applications.

The analysis of the results showed that the proposed solution was able to achieve the desired average response time for the web application across all of the experiments. In addition, the batch job managed to finish the workload within the desired time. However, due to high traffic load during the experiments, violation of SLA was somehow high. The average utilization of CPU across all experiments was 83.70% for the web application and 89.51% for the batch, while average memory utilization was 67.52% and 74.78% for the web server and batch, respectively.

Further testing with more diverse workload patterns and several improvements have been suggested for the future work , such as adopting reinforcement learning into the controller to achieve prediction-based detection of traffic flows.

Bibliography

[1] Minqi Zhou et al. "Security and Privacy in Cloud Computing: A Survey." In: *Semantics Knowledge and Grid (SKG), 2010 Sixth International Conference on*. Nov. 2010, pp. 105–112. DOI: 10.1109/SKG.2010.19.

[2] Louis Columbus. *Roundup Of Small & Medium Business Cloud Computing Forecasts And Market Estimates, 2015*. May 2015. URL: http://www.forbes.com/sites/louiscolumbus/2015/05/04/roundup-of-small-medium-business-cloud-computing-forecasts-and-market-estimates-2015/#4ae123a81646.

[3] Kaplan M. James, Forrest William and Kindler Noah. *Revolutionizing data center efficiency*. July 2008. URL: http://www.mckinsey.com/clientservice/bto/pointofview/pdf/revolutionizing_data_center_efficiency.pdf.

[4] A. Vasan et al. "Worth their watts? - an empirical study of datacenter servers." In: *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. Jan. 2010, pp. 1–10. DOI: 10.1109/HPCA.2010.5463056.

[5] Charles Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis." In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 7:1–7:13. ISBN: 9781450317610. DOI: 10.1145/2391229.2391236. URL: http://doi.acm.org/10.1145/2391229.2391236.

[6] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 1st. Morgan and Claypool Publishers, 2009, pp. 54–57. ISBN: 159829556X, 9781598295566.

[7] Marcus Carvalho et al. "Long-term SLOs for Reclaimed Cloud Computing Resources." In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: ACM, 2014, 20:1–20:13. ISBN: 9781450332521. DOI: 10.1145/2670979.2670999. URL: http://doi.acm.org/10.1145/2670979.2670999.

[8] Whitney Josh and Delforge Pierre. *Data Center Efficiency Assessment*. Aug. 2014. URL: http://www.nrdc.org/energy/files/data-center-efficiency-assessment-IP.pdf.

[9] David Lo et al. “Towards Energy Proportionality for Large-scale Latency-critical Workloads.” In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA ’14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 301–312. ISBN: 9781479943944. URL: http://dl.acm.org/citation.cfm?id=2665671.2665718.

[10] E.B. Lakew et al. “Towards Faster Response Time Models for Vertical Elasticity.” In: *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. Dec. 2014, pp. 560–565. DOI: 10.1109/UCC.2014.86.

[11] Germán Moltó et al. “Elastic Memory Management of Virtualized Infrastructures for Applications with Dynamic Memory Requirements.” In: *Procedia Computer Science* 18 (2013). 2013 International Conference on Computational Science, pp. 159–168. ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2013.05.179. URL: http://www.sciencedirect.com/science/article/pii/S1877050913003220.

[12] F.B. Shaikh and S. Haider. “Security threats in cloud computing.” In: *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*. Dec. 2011, pp. 214–219.

[13] P. Hofmann and D. Woods. “Cloud Computing: The Limits of Public Clouds for Business Applications.” In: *Internet Computing, IEEE* 14.6 (Nov. 2010), pp. 90–93. ISSN: 1089-7801. DOI: 10.1109/MIC.2010.136.

[14] Oracle. *Brief History of Virtualization*. URL: https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html.

[15] diegocalleja. *Linux 2 6 20*. Sept. 2007. URL: http://kernelnewbies.org/Linux_2_6_20.

[16] S.G. Soriga and M. Barbulescu. “A comparison of the performance and scalability of Xen and KVM hypervisors.” In: *Networking in Education and Research, 2013 RoEduNet International Conference 12th Edition*. Sept. 2013, pp. 1–6. DOI: 10.1109/RoEduNet.2013.6714189.

[17] Sisu Xi et al. “RT-Xen: towards real-time hypervisor scheduling in xen.” In: *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*. IEEE. 2011, pp. 39–48.

[18] Emmanuel Ackaouy. *[Xen-devel] New CPU scheduler w/ SMP load balancer*. May 2006. URL: http://old-list-archives.xenproject.org/archives/html/xen-devel/2006-06/msg00935.html.

[19] Wiki Xen. *Credit Scheduler*. [Online; accessed 3-February-2016]. 2016. URL: http://wiki.xen.org/wiki/Credit_Scheduler.

[20] Diwaker Gupta, Rob Gardner, and Ludmila Cherkasova. “Xenmon: Qos monitoring and performance profiling tool.” In: *Hewlett-Packard Labs, Tech. Rep. HPL-2005-187* (2005).

[21] Diego Ongaro, Alan L. Cox, and Scott Rixner. "Scheduling I/O in Virtual Machine Monitors." In: *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '08. Seattle, WA, USA: ACM, 2008, pp. 1–10. ISBN: 9781595937964. DOI: 10.1145/1346256.1346258. URL: http://doi.acm.org/10.1145/1346256.1346258.

[22] M. Tim Jones. *Inside the Linux 2.6 Completely Fair Scheduler*. Dec. 2009. URL: http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/.

[23] C.S. Wong et al. "Fairness and interactive performance of O(1) and CFS Linux kernel schedulers." In: *Information Technology, 2008. ITSim 2008. International Symposium on*. Vol. 4. Aug. 2008, pp. 1–8. DOI: 10.1109/ITSIM.2008.4631872.

[24] Ingo Molnar. *[patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]*. [Online; accessed 7-February-2016]. Apr. 2007. URL: http://lwn.net/Articles/230501/.

[25] Sangar Jitendra. *Scheduling : O(1) and Completely Fair Scheduler (CFS)*. [Online; accessed 8-February-2016]. Mar. 2014. URL: http://algorithmsandme.in/2014/03/scheduling-o1-and-completely-fair-scheduler-cfs/.

[26] S. Dutta et al. "SmartScale: Automatic Application Scaling in Enterprise Clouds." In: *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. June 2012, pp. 221–228. DOI: 10.1109/CLOUD.2012.12.

[27] Narayanan Gopalakrishnan. *Hot-Plugging CPU's to conserve power*. [Online; accessed 10-February-2016]. Jan. 2013. URL: http://linuxforthenew.blogspot.no/2013/01/hot-plugging-cpus-to-conserve-power.html.

[28] Zhu Guihua. *[libvirt] [RFC PATCH v2 00/12] qemu: add support to hot-plug/unplug cpu device*. [Online; accessed 10-February-2016]. Feb. 2015. URL: https://www.redhat.com/archives/libvir-list/2015-February/msg00084.html.

[29] S. Farokhi et al. "Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints." In: *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*. Sept. 2015, pp. 69–80. DOI: 10.1109/ICCAC.2015.20.

[30] Marian Turowski and Alexander Lenk. "Service-Oriented Computing - ICSOC 2014 Workshops: WESOA; SeMaPS, RMSOC, KASA, ISC, FOR-MOVES, CCSA and Satellite Events, Paris, France, November 3-6, 2014, Revised Selected Papers." In: ed. by Farouk Toumani et al. Cham: Springer International Publishing, 2015. Chap. Vertical Scaling Capability of OpenStack, pp. 351–362. ISBN: 9783319228853. DOI: 10.1007/978-3-319-22885-3_30. URL: http://dx.doi.org/10.1007/978-3-319-22885-3_30.

[31] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. "Working Set-based Physical Memory Ballooning." In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 95–99. ISBN: 9781931971027. URL: https://www.usenix.org/conference/icac13/technical-sessions/presentation/chiang.

[32] George Dunlap. *Ballooning, rebooting, and the feature you've never heard of*. [Online; accessed 16-February-2016]. Feb. 2014. URL: https://blog.xenproject.org/2014/02/14/ballooning-rebooting-and-the-feature-youve-never-heard-of/.

[33] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments." In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592. ISSN: 1572-9184. DOI: 10.1007/s10723-014-9314-7. URL: http://dx.doi.org/10.1007/s10723-014-9314-7.

[34] Karl Johan Åström. "Challenges in control education." In: *Proceedings of the 7th IFAC Symposium on Advances in Control Education (ACE)', Madrid, Spain*. 2006.

[35] Tarek Abdelzaher et al. "Performance Modeling and Engineering." In: ed. by Zhen Liu and Cathy H. Xia. Boston, MA: Springer US, 2008. Chap. Introduction to Control Theory And Its Application to Computing Systems, pp. 185–215. ISBN: 9780387793610. DOI: 10.1007/978-0-387-79361-0_7. URL: http://dx.doi.org/10.1007/978-0-387-79361-0_7.

[36] Wiki Archlinux. *libvirt*. [Online; accessed 12-February-2016]. n.d. URL: https://wiki.archlinux.org/index.php/Libvirt.

[37] Anju Bala and Inderveer Chana. "Fault tolerance-challenges, techniques and implementation in cloud computing." In: *IJCSI International Journal of Computer Science Issues* 9.1 (2012), pp. 1694–0814.

[38] Lorin. *Understanding the different test types*. July 2014. URL: https://loader.io/blog/2014/07/16/understanding-different-test-types/.

[39] Cristian Klein. *Closed and open HTTP traffic generator*. [Online; accessed 12-February-2016]. (n.d.) URL: https://github.com/cloud-control/httpmon.

[40] S. Farokhi et al. "Performance-Based Vertical Memory Elasticity." In: *Autonomic Computing (ICAC), 2015 IEEE International Conference on*. July 2015, pp. 151–152. DOI: 10.1109/ICAC.2015.51.

[41] Tao Chen and Rami Bahsoon. "Self-adaptive and Sensitivity-aware QoS Modeling for the Cloud." In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '13. San Francisco, California: IEEE Press, 2013, pp. 43–52. ISBN: 9781467344012. URL: http://dl.acm.org/citation.cfm?id=2663546.2663556.

[42] S. Sivasubramanian. "Scalable hosting of web applications." Doctoral dissertation. VU University Amsterdam, Netherlands, 2007. URL: http://hdl.handle.net/1871/10753.

[43] C. Stewart, M. Leventi, and Kai Shen. "Empirical examination of a collaborative web application." In: *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. Sept. 2008, pp. 90–96. DOI: 10.1109/IISWC.2008.4636094.

[44] Timothy Wood et al. "Profiling and Modeling Resource Usage of Virtualized Applications." In: *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Middleware '08. Leuven, Belgium: Springer-Verlag New York, Inc., 2008, pp. 366–387. ISBN: 3540898557. URL: http://dl.acm.org/citation.cfm?id=1496950.1496973.

[45] Rajkumar Buyya et al. "Cloud computing and emerging {IT} platforms: Vision, hype, and reality for delivering computing as the 5th utility." In: *Future Generation Computer Systems* 25.6 (2009), pp. 599–616. ISSN: 0167-739X. DOI: http://dx.doi.org/10.1016/j.future.2008.12.001. URL: http://www.sciencedirect.com/science/article/pii/S0167739X08001957.

[46] Soodeh Farokhi. "Quality of service control mechanism in cloud computing environments." Doctoral dissertation. Vienna University of Technology, Dec. 2015.

[47] David Lo et al. "Heracles: Improving Resource Efficiency at Scale." In: *SIGARCH Comput. Archit. News* 43.3 (June 2015), pp. 450–462. ISSN: 0163-5964. DOI: 10.1145/2872887.2749475. URL: http://doi.acm.org/10.1145/2872887.2749475.

[48] S. Spinner et al. "Proactive Memory Scaling of Virtualized Applications." In: *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. June 2015, pp. 277–284. DOI: 10.1109/CLOUD.2015.45.

[49] L. Yazdanov and C. Fetzer. "Vertical Scaling for Prioritized VMs Provisioning." In: *Cloud and Green Computing (CGC), 2012 Second International Conference on*. Nov. 2012, pp. 118–125. DOI: 10.1109/CGC.2012.108.

[50] Albert Greenberg et al. "The Cost of a Cloud: Research Problems in Data Center Networks." In: *SIGCOMM Comput. Commun. Rev.* 39.1 (Dec. 2008), pp. 68–73. ISSN: 0146-4833. DOI: 10.1145/1496091.1496103. URL: http://doi.acm.org/10.1145/1496091.1496103.

[51] Nikolay Grozev and Rajkumar Buyya. "Multi-Cloud Provisioning and Load Distribution for Three-Tier Applications." In: *ACM Trans. Auton. Adapt. Syst.* 9.3 (Oct. 2014), 13:1–13:21. ISSN: 1556-4665. DOI: 10.1145/2662112. URL: http://doi.acm.org/10.1145/2662112.

[52] Peter Bodik et al. "Characterizing, Modeling, and Generating Workload Spikes for Stateful Services." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 241–252. ISBN: 9781450300360. DOI: 10.1145/1807128.1807166. URL: http://doi.acm.org/10.1145/1807128.1807166.

[53] Clay James. *Forget application response time "standards" – it's all about the human reaction*. May 2014. URL: http://me.riverbed.com/blogs/human-reaction-drives-application-response-time-standards.html.

[54] G. Tesauro et al. "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation." In: *2006 IEEE International Conference on Autonomic Computing*. June 2006, pp. 65–73. DOI: 10.1109/ICAC.2006.1662383.

Appendices

Appendix A

Autonomic provisioning controller

Autonomic provisioning controller script (controller.py)

```
1
2  #!/usr/bin/python
3  from __future__ import division
4  import libvirt
5  import sys
6  import xml.etree.ElementTree as ET
7  import paramiko
8  import time
9  import requests
10 from time import sleep
11 import subprocess
12 import logging
13 import math
14 from datetime import datetime
15 import os
16
17 # URLs
18 url1 = "http://10.0.0.11/PHP/RandomStory.php"
19 url2 = "http://10.0.0.10/PHP/RandomStory.php"
20
21 # Resources
22 min_vcpu = 1
23 max_memory = 12000
24 max_vcpu = 10
25
26 # Xen - Libvirt
27 local = "xen:///"
28 remote = "xen+ssh://10.0.0.1/"
29
30 # Servers
31 server2 = '127.0.0.1'
32 server1 = '10.0.0.1'
33 batch = 'batch'
```

```
34 webserver1 = 'webserver1'
35 webserver2 = 'webserver2'
36
37 # Batch metrics
38 max_frame = 23
39 min_frame = 15
40
41 # Web servers metrics
42 desired_response_time = 0.5
43 minresp = 0.1
44
45 def runbash(command):
46     bashcommand = subprocess.call(command,shell=True)
47
48 def ssh(hostname,command):
49     sshcon = paramiko.SSHClient()
50     logging.basicConfig()
51     sshcon.load_system_host_keys()
52     sshcon.set_missing_host_key_policy(paramiko.AutoAddPolicy())
53     sshcon.connect(hostname, username='root')
54     stdin, stdout, stderr = sshcon.exec_command(command)
55     return stdout.read()
56
57 def retrieve_vcpu_count(host,location):
58     conn = libvirt.openReadOnly(location)
59     if conn == None:
60         print 'Failed to open connection to the hypervisor'
61         sys.exit(1)
62
63     try:
64         hostname = conn.lookupByName(host)
65         info = hostname.info()
66         vcpu = int(info[3])
67         cpu_usage_max = vcpu*100
68         # vcpu's and maximum cpu usage e.g 100 if 1 vcpu
69         return vcpu,cpu_usage_max
70
71     except:
72         print 'Failed to find the main domain'
73         sys.exit(1)
74
75 def retrieve_cpu_usage(server,host):
76     cpu = ssh(server,"xentop -b -d 1 -i 2 | awk '/%s/ {print $4}'"
           %(host))
77     cpu_usage = float(cpu.split()[1])
78     return cpu_usage
79
80
81 # Batch commands
82 HandBrakeCLI = "screen -d -m /home/user/run.sh"
83 convert = "tr '\r' '\n' < /home/user/batch.log > /home/user/batch-
       final.log"
84 read = "cat /home/user/batch-final.log | cut -f 11 -d ' ' | tail -1"
85
```

```
86  def relocate_resources():
87      cpus = int(retrieve_vcpu_count(webserver2, local)[0])
88      if cpus >= 3:
89          value = int(cpus/2)
90          ssh(server1, "xm vcpu-set %s %s" % (webserver1, value))
91          time.sleep(20)
92          ssh(server2, "xm vcpu-set %s %s" % (webserver2, value))
93      else:
94          print "3 or less CPUs "
95          pass
96
97  def start_batch(hostname):
98          client = paramiko.SSHClient()
99          client.load_system_host_keys()
100         client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
101         client.connect(hostname=hostname)
102         client.exec_command(HandBrakeCLI)
103         print "Starting batch"
104         start_log(hostname)
105         time.sleep(70)
106
107 def retrieve_frame(hostname):
108         client = paramiko.SSHClient()
109         client.load_system_host_keys()
110         client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
111         client.connect(hostname=hostname, port=22)
112         client.exec_command(convert)
113         stdin, stdout, stderr = client.exec_command(read)
114         frame = stdout.read().strip()
115         if frame == "" or frame == None:
116                 pass
117         else:
118                 return float(frame)
119
120 def check_frame(hostname,server,location):
121         if retrieve_frame(hostname) > max_frame:
122                 print "Reduce resources - batch"
123                 reduce_resources(hostname,server,location)
124         elif retrieve_frame(hostname) < min_frame:
125                 print "Increase resources - batch"
126                 increase_resources(hostname,server,location)
127         else:
128                 print "Desired average frames/s"
129                 pass
130
131 start = "xm create /home/user/webserver/webserver1.cfg"
132 stop = "xm shutdown %s" %(webserver1)
133 webserver1_IP='10.0.0.10'
134 def addTo_loadbalancer():
135     f = open('/etc/haproxy/haproxy.cfg', 'a')
136     f.write('      server ' + str(webserver1) + ' ' + str(
        webserver1_IP) + ':' + '80 check')
137     f.write('\n')
138     f.close()
```

```
139     runbash('service haproxy restart')
140
141 def start_web():
142     client = paramiko.SSHClient()
143     client.load_system_host_keys()
144     client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
145     client.connect(hostname=server1, port=22)
146     client.exec_command(start)
147     response = 1
148     while response != 0:
149         response = os.system("ping -c 1 -w0 webserver1 > /dev/null
                2>&1")
150         print 'Attempting to connect to webserver1'
151     print "Connected"
152     start_logg_web1()
153     addTo_loadbalancer()
154     relocate_resources()
155
156 def removeFrom_loadbalancer():
157     lines = file('/etc/haproxy/haproxy.cfg', 'r').readlines()
158     del lines[-1]
159     file('/etc/haproxy/haproxy.cfg', 'w').writelines(lines)
160     runbash('service haproxy restart')
161
162 def stop_web():
163     client = paramiko.SSHClient()
164     client.load_system_host_keys()
165     client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
166     client.connect(hostname=server1, port=22)
167     removeFrom_loadbalancer()
168     time.sleep(10)
169     client.exec_command(stop)
170
171 # Memory allocation
172 total_memory = "free -m | grep Mem: | awk {'print $2'}"
173 used_memory = " free -m | grep Mem: | awk '{print $3}'"
174
175 def retrieve_memory(host):
176     memory_usage_max = int(ssh(host,total_memory))
177     memory_usage = int(ssh(host,used_memory))
178     min_memory = memory_usage+512
179     return memory_usage_max,memory_usage,min_memory
180
181 # Response time decision making
182 def response_time(desired_resp,url,host,server,location):
183     try:
184         time = requests.get(url,timeout=desired_resp).elapsed.
                total_seconds()
185     except requests.exceptions.Timeout as f:
186         print "|----------------------------------------|"
187         print "High response time"
188         increase_resources(host,server,location)
189     else:
190         print "|----------------------------------------|"
```

```
191         print "Low reponse time"
192         if time < minresp:
193             reduce_resources(host,server,location)
194         else:
195             print "Ok - Response time"
196             pass
197 
198 def reduce_check_cpu(cpu, max_cpu):
199         if cpu < max_cpu * 1:
200                 return False
201         else:
202                 return True
203 
204 # Check CPU usage
205 def check_cpu(cpu,max_cpu):
206     if cpu > max_cpu*0.8:
207                 return True
208     else:
209         return False
210 
211 # Check memory usage
212 def check_memory(memory,max_memory):
213     if memory > max_memory*0.8:
214         return True
215     else:
216         return False
217 
218 def check_total_CPU_usage():
219      if retrieve_vcpu_count(batch,local)[0] > 2:
220         print "Removing CPU cores"
221         ssh(server2,"xm vcpu-set %s %s" % (batch, retrieve_vcpu_count
                (batch,local)[0] - 1))
222         ssh(server2,"xm vcpu-set %s %s" % (webserver2,
                retrieve_vcpu_count(webserver2,local)[0] + 1))
223      else:
224         print "Adding new web-server to pool"
225         start_web()
226 
227 def increase_resources(hostname,server,location):
228     if check_cpu(retrieve_cpu_usage(server,hostname),
            retrieve_vcpu_count(hostname,location)[1]) == True and
            check_memory(retrieve_memory(hostname)[1],retrieve_memory(
            hostname)[0]) == True:
229 
230                 print "Increasing CPU cores and memory"
231 
232         if retrieve_vcpu_count(hostname,location)[0] < max_vcpu and
                retrieve_memory(hostname)[1] < max_memory:
233             sum = int(retrieve_vcpu_count(webserver2, local)[0]) +
                    int(retrieve_vcpu_count(batch, local)[0])
234             if sum >= 8:
235                 print "Trying to remove from batch"
236                 check_total_CPU_usage()
237             else:
```

```
238                 return ssh(server,"xm vcpu-set %s %s && xm mem-set %s
                        %s" %(hostname,retrieve_vcpu_count(hostname,
                        location)[0]+2,hostname,retrieve_memory(hostname)
                        [0]+1024))
239         else:
240             print "Error: Maximum CPU cores and memory reached -
                    Reducing from batch"
241             check_total_CPU_usage()
242 
243     elif check_cpu(retrieve_cpu_usage(server,hostname),
            retrieve_vcpu_count(hostname,location)[1]) == True:
244 
245                 print "Increasing CPU cores"
246 
247         if retrieve_vcpu_count(hostname,location)[0] < max_vcpu:
248             sum = int(retrieve_vcpu_count(webserver2, local)[0]) +
                        int(retrieve_vcpu_count(batch, local)[0])
249             print sum
250             if sum >= 8:
251                 print "Removing CPU cores"
252                 check_total_CPU_usage()
253             else:
254                 return ssh(server,"xm vcpu-set %s %s" %(hostname,
                            retrieve_vcpu_count(hostname,location)[0]+2))
255 
256         else:
257             print "Maximum CPU cores reached - Reducing from batch"
258             check_total_CPU_usage()
259 
260     elif check_memory(retrieve_memory(hostname)[1],retrieve_memory(
            hostname)[0]) == True:
261         print "Increasing memory"
262         if retrieve_memory(hostname)[1] < max_memory:
263             return ssh(server,"xm mem-set %s %s" %(hostname,
                        retrieve_memory(hostname)[0]+1024))
264         else:
265             print "Error: Maximum memory reached"
266     else:
267                 print "No lack of resources causing high response time"
268 
269 def reduce_resources(hostname,server,location):
270     if reduce_check_cpu(retrieve_cpu_usage(server,hostname),
            retrieve_vcpu_count(hostname,location)[1]) == False and
            check_memory(retrieve_memory(hostname)[1],retrieve_memory(
            hostname)[0]) == False:
271         print "Reducing resources"
272 
273         if retrieve_vcpu_count(hostname,location)[0] > min_vcpu and
                retrieve_memory(hostname)[0] > retrieve_memory(hostname)
                [2]:
274             print "Reducing CPU and memory"
275             return ssh(server,"xm vcpu-set %s %s && xm mem-set %s %s"
                         %(hostname,retrieve_vcpu_count(hostname,location)
                        [0]-1,hostname,retrieve_memory(hostname)[0]))
```

```
276
277         elif retrieve_vcpu_count(hostname,location)[0] > min_vcpu:
278             print "Reducing CPU cores"
279             return ssh(server,"xm vcpu-set %s %s" %(hostname,
                    retrieve_vcpu_count(hostname, location)[0]-1))
280
281         elif retrieve_memory(hostname)[0] > retrieve_memory(hostname)
                [2]:
282             print "Reducing memory"
283             return ssh(server,"xm mem-set %s %s" %(hostname,
                    retrieve_memory(hostname)[0]))
284         else:
285             print "Minimum memory and CPU reached"
286             list=ssh(server1, "xm list")
287             if "webserver1" in list:
288                 print "Removing Webserver1"
289                 stop_web()
290             else:
291                 pass
292
293     elif reduce_check_cpu(retrieve_cpu_usage(server,hostname),
            retrieve_vcpu_count(hostname,location)[1]) == False:
294         print "Reduce resources"
295
296         if retrieve_vcpu_count(hostname,location)[0] > min_vcpu:
297              print "Reducing CPU cores"
298              return ssh(server,"xm vcpu-set %s %s" %(hostname,
                     retrieve_vcpu_count(hostname,location)[0]-1))
299         else:
300             print "Minimum CPU cores used"
301
302     elif check_memory(retrieve_memory(hostname)[1],retrieve_memory(
            hostname)[0]) == False:
303
304         if retrieve_memory(hostname)[0] > retrieve_memory(hostname)
                [2]:
305             print "Reducing memory"
306             ssh(server,"xm mem-set %s %s" %(hostname,retrieve_memory(
                    hostname)[1]))
307         else:
308                 print "Minimum memory is reached"
309     else:
310         print "Minimum CPU and memory reached"
311
312
313 start_batch(batch)
314
315 while True:
316     print "|------- Batch -----------|"
317     check_frame(batch,server2,local)
318     i=1
319     while i <= 60:
320         print "|------- Webserver2 -----------|"
```

```
321         response_time(desired_response_time, url1, webserver2,
                server2, local)
322         list=ssh(server1, "xm list")
323         if "webserver1" in list:
324                         horizontal('2')
325             print "|------- Webserver1 -----------|"
326             response_time(desired_response_time, url2, webserver1,
                    server1, remote)
327         else:
328              horizontal('1')
329              pass
330         time.sleep(5)
331         i+=1
```

Appendix B

Parallel computing

The script is retrieved from Stackoverflow[1].

Parallel computing script (parallel.c)

```
1
2  #include <stdio.h>
3  #include <time.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <errno.h>
7
8  #define NUM_OF_CORES 8
9  #define MAX_PRIME 100000
10
11 void do_primes()
12 {
13     unsigned long i, num, primes = 0;
14     for (num = 1; num <= MAX_PRIME; ++num) {
15         for (i = 2; (i <= num) && (num % i != 0); ++i);
16         if (i == num)
17             ++primes;
18     }
19     printf("Calculated %d primes.\n", primes);
20 }
21
22 int main(int argc, char ** argv)
23 {
24     time_t start, end;
25     time_t run_time;
26     unsigned long i;
27     pid_t pids[NUM_OF_CORES];
28
29     /* start of test */
30     start = time(NULL);
31     for (i = 0; i < NUM_OF_CORES; ++i) {
32         if (!(pids[i] = fork())) {
```

[1] stackoverflow.com/questions/9244481/how-to-get-100-cpu-usage-from-a-c-program

```
33              do_primes();
34              exit(0);
35          }
36          if (pids[i] < 0) {
37              perror("Fork");
38              exit(1);
39          }
40      }
41      for (i = 0; i < NUM_OF_CORES; ++i) {
42          waitpid(pids[i], NULL, 0);
43      }
44      end = time(NULL);
45      run_time = (end - start);
46      printf("This machine calculated all prime numbers under %d %d
            times "
47             "in %d seconds\n", MAX_PRIME, NUM_OF_CORES, run_time);
48      return 0;
49  }
```

Appendix C

HAProxy configuration file

This configuration requires: HAProxy version 1.6.5 2016/05/10

HAProxy configuration file (haproxy.cfg)

```
1  global
2          log /dev/log  local0
3          log /dev/log  local1 notice
4          chroot /var/lib/haproxy
5          stats socket /run/haproxy/admin.sock mode 660 level admin
6
7          stats timeout 30s
8          user haproxy
9          group haproxy
10         daemon
11
12         # Default SSL material locations
13         ca-base /etc/ssl/certs
14         crt-base /etc/ssl/private
15
16         # Default ciphers to use on SSL-enabled listening sockets.
17         # For more information, see ciphers(1SSL). This list is from:
18         #  https://hynek.me/articles/hardening-your-web-servers-ssl-
               ciphers/
19         ssl-default-bind-ciphers ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH
               +AES256:ECDH+AES128:DH+AES:ECDH+3DES:DH+3DES:RSA+AESGCM:
               RSA+AES:RSA+3DES:!aNULL:!MD5:!DSS
20         ssl-default-bind-options no-sslv3
21
22 defaults
23         log    global
24         mode   http
25         option httplog
26         option dontlognull
27         timeout connect 5000
28         timeout client 50000
29         timeout server 50000
30         errorfile 400 /etc/haproxy/errors/400.http
31         errorfile 403 /etc/haproxy/errors/403.http
```

```
32          errorfile 408 /etc/haproxy/errors/408.http
33          errorfile 500 /etc/haproxy/errors/500.http
34          errorfile 502 /etc/haproxy/errors/502.http
35          errorfile 503 /etc/haproxy/errors/503.http
36          errorfile 504 /etc/haproxy/errors/504.http
37
38
39  frontend http-in
40          bind *:80
41          default_backend web
42
43  backend web
44          balance roundrobin
45          mode http
46          server webserver2 10.0.0.11:80 check
47                  server webserver1 10.0.0.10:80 check
48  listen statistics
49          bind *:1936
50          mode http
51          stats enable
52          stats uri /
53          stats auth user:password
```

Appendix D

Xen - VM creation file

Xen - VM creation file (vm.cfg)

```
1  import os, re
2  arch = os.uname()[4]
3  kernel = "/usr/lib/xen-default/boot/hvmloader"
4  builder='hvm'
5  memory = 2449
6  maxmem = 20480
7  shadow_memory = 8
8  name = "batch"
9  vcpus = 1
10 maxvcpus = 10
11 vif = [ 'bridge=virbr0,mac=02:16:3e:10:11:27,ip=10.0.0.8' ]
12 disk = [ 'file:/home/bilal/domains/batch/batch.img,hda,w']
13 device_model = '/usr/lib/xen-default/bin/qemu-dm'
14 boot="c"
15 #boot="d"
16 vnc=1
17 xen_platform_pci=1
18 vnclisten="0.0.0.0"
19 vncconsole=0
20 vfb = ['type=vnc,vnclisten=0.0.0.0,vncpasswd=test123,vncdisplay=1,
       keymap=no']
21 acpi = 1
22 apic = 1
23 sdl=0
24 stdvga=0
25 serial='pty'
26 usbdevice='tablet'
27 on_poweroff = 'restart'
28 on_reboot   = 'restart'
29 on_crash    = 'restart'
```