SERVICE LEVEL AGREEMENT DRIVEN ADAPTIVE RESOURCE MANAGEMENT FOR WEB APPLICATIONS ON HETEROGENOUS COMPUTE CLOUDS

by

Waheed Iqbal

A PFM report submitted in partial fulfillment of the requirements for the degree of Master in Information Technology

Director: Dr. David Carrera
Co-director: Dr. Matthew N. Dailey

Previous Degree: Bachelor of Software Engineering
Bahria University, Pakistan

Technical University of Catalunya, Barcelona School of Informatics, Spain
Asian Institute of Technology, School of Engineering and Technology, Thailand
September 2009

Acknowledgments

In the first place I am thankful to Almighty Allah, the most beneficent and merciful who enabled me to complete this research. Then, I would like to record my gratitude to Dr. David Carrera and Dr. Matthew N. Dailey for their supervision, support, and guidance throughout this research. I gratefully acknowledge AIT and UPC to select me for double degree program. I express my love and gratitude to my beloved family, for their endless love and prayers, throughout the duration of my studies. I am also thankful to all my friends for providing help and encouragement to successfully complete this research.

Abstract

Cloud computing is an emerging topic in the field of parallel and distributed computing. Many IT giants such as IBM, Sun, Amazon, Google, and Microsoft are promoting and offering various storage and compute clouds. Clouds provide services such as high performance computing, storage, and application hosting. Cloud providers are expected to ensure Quality of Service (QoS) through a Service Level Agreement (SLA) between the provider and the consumer. In this research, I develop a heterogeneous testbed compute cloud and investigate adaptive management of resources for Web applications to satisfy a SLA that enforces specific response time requirements. I develop a system on top of EUCALYTPUS framework that actively monitors the response time of the compute resources assign to a Web application and dynamically allocates the resources required by the application to satisfy the specific response time requirements.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Cloud computing is an emerging topic in the field of parallel and distributed computing. Many IT giants such as IBM, Sun, Amazon, Google, and Microsoft are promoting and offering various storage and compute clouds. Clouds provide services such as high performance computing, storage, and application hosting. Cloud providers are expected to ensure Quality of Service (QoS) through a service level agreement (SLA) between the provider and the consumer. Cloud providers need to establish a robust infrastructure that can grow dynamically, with easy maintenance and update.

A cloud is a combination of physically and virtually connected resources. Virtualization allows us to instantiate virtual machines dynamically on physical machines and allocate them resources as needed; therefore, virtualization is one of the key technologies behind the cloud computing infrastructure. There are several benefits that we expect from virtualization, such as high availability, ease of deployment, migration, maintenance, and low power consumption that help us to establish a robust infrastructure for cloud computing.

EUCALYPUTS is an open source framework for developing clouds. Its primary developers are at the University of California, Santa Barbara. The aim of EUCALYPTUS is to enable academics to perform research in the field of cloud computing. In this research study, I use the EUCALYPTUS framework to establish a cloud and host a simple Web application on a Web farm of virtual machine instances.

1.2 Problem Statement

In Cloud computing, provider and consumer sign a service level agreement (SLA) that defines quality of service requirements. Cloud providers such as Amazon and Google allow consumers to the select resources they need for their cloud-based applications. Usually consumers reserve resources without any optimal prediction or estimation, leading either under or overutilization of reserved resources. Web application owners, in order to ensure the usability of their applications, need to maintain a minimum response time for their end users. Web application owners should be able to host their applications on a cloud by specifying a desire Quality of Service (QoS) in terms of response time. Cloud providers should ensure those QoS requirements using a minimal amount of computational resources. When violations of the SLA are predicted or detected, cloud providers should horizontally scale up the hosted Web application to ensure that the specific response time requirement is satisfied. Currently, neither the commercial cloud providers nor the existing open source frameworks support maximum response time guarantees.

1.3 Objectives

The main goal of this research study is to satisfy a cloud consumer that specifies Quality of Service (QoS) requirements in terms of response time by monitoring the average response time of the application and adaptively scaling up the Web application when necessary. Towards reaching this goal, the following are my specific objectives:

1. Understand EUCALYPTUS and establish a heterogeneous compute cloud testbed.
2. Write a simple Web based application and develop a workload generator for it.
3. Host the Web application on the cloud and establish a Web farm (load balancer) for the application.
4. Detect SLA violations by monitoring request response times.
5. Develop software components to horizontally scale up the Web application dynamically.

1.4 Limitations and Scope

This research investigates adaptive resource management for a simple Web application that contains one Java servlet. The testbed cloud is only accessible on the CSIM LAN at AIT.

1.5 Thesis Outline

This document is organized as follows:

Chapter 2 reviews the relevant literature and technology, including cloud computing, virtualization, platforms for virtualization, EUCALYPTUS, and dynamic resource allocation.

Chapter 3 describes the methodology I use for this research, including a system overview, the system design, and the system architecture.

Chapter 4 describes the experiments I performed and the results I obtained.

Chapter 5 contains conclusions and recommendations.

Chapter 2

Literature and Technology Review

This chapter presents the literature and technology relevant to cloud computing, virtualization, and dynamic resource provisioning.

2.1 Cloud Computing

Cloud computing is an emerging technology in the field of parallel and distributed computing. In Geelan (2009), many experts define cloud computing with their points of view. Buyya, Yeo, and Venugopal (2008) provide a very concise and clear definition of cloud computing:

"A Cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers" (p. 05).

Storage and compute clouds are two major types of clouds that aim to provide services without exposing the underlying infrastructure to customers. Instead of investing and managing personal hardware, users rent virtual machines or services from cloud providers (Christine, Jérôme, Yvon, & Pierre, 2009). Following is a brief overview of the different clouds available for consumers today. Costs are up to date as of March 2009.

2.1.1 Amazon Elastic Computing "EC2"

Amazon Elastic Computing also known as EC2 is a commercial Web service that enables users to rent hardware resources and execute their applications on those resources. EC2 provides a Web service interface to customers for requesting virtual machines (VMs) as server instances. Consumers are allowed to demand more server instances or reduce the number of acquired server instances; therefore, this feature enables consumers to scale their applications up or down according to the changes in their computing needs. Server instances are available in three different sizes: small, medium and large. Each option has different memory, processing power, and bandwidth (Wikipedia, 2009a). EC2 offers many features such as Elastic IP Addresses, Multiple Locations, and Amazon Elastic Block Store (Amazon.com, Inc, 2009). Table 2.1 and 2.2 show the cost (per CPU hour of used compute time) of different EC2 server instances. One EC2 compute unit is equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor in CPU capacity. EC2 also charges for data transfer and Elastic IP addresses.

Table 2.1: Amazon EC2: cost of standard instances in the USA availability zone. Costs are given per CPU hour of used compute time. One EC2 compute unit is equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor in CPU capacity.

Type	Memory	Compute Unit	Storage	Platform	Linux/Unix	Windows
Small (default)	1.7 GB	1 EC2	160 GB	32 bit	$0.10	$0.13
Large	7.5 GB	4 EC2	560 GB	64 bit	$0.40	$0.50
Extra Large	15 GB	8 EC2	1690 GB	64 bit	$0.80	$1.00

Table 2.2: Amazon EC2: cost of high CPU instances in the USA availability zone. Costs are given per CPU hour of used compute time. One EC2 compute unit is equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor in CPU capacity.

Type	Memory	Compute Unit	Storage	Platform	Linux/Unix	Windows
Medium	1.7 GB	5 EC2	350 GB	32 bit	$0.20	$0.30
Large	7 GB	20 EC2	1690 GB	64 bit	$0.80	$1.20

2.1.2 Sun Grid

Sun Grid is a commercial service by Sun Microsystems that provides Solaris 10 X86 machines for consumers to execute their applications. Consumers are allowed to upload their applications or select applications from catalogue that contains many open source applications. Users need to buy tokens (it is a prepaid services) for executing their applications. Sun provides a portal for uploading applications and data. It also provides various levels of security for user accounts and their applications (Sun Microsystems, 2008). There are few important features in SunGrid:

- Applications should be able to execute on Solaris 10.
- Only users from 24 countries are able to use this facility. Users from Thailand and Pakistan are not allowed but users from Spain are allowed to access this service.
- Only MPICH v1.2.6 is available for parallel environments.
- GCC and Studio 10 compilers are available.
- Sun charges $1/CPU hour.

2.1.3 Google App Engine

Google App Engine allows consumers to run their Web applications on Google infrastructure (Google Inc., 2008). It only supports the Python run time environment. Google App Engine

has both free and paid accounts. Free accounts allow persistent storage up to 1 GB and other resources such as bandwidth and CPU required to support 5 million page views per month (Wikipedia, 2009b).

2.1.4 GoGrid

GoGrid is a commercial cloud that offers many services similar to Amazon EC2. GoGrid provides a Web-based GUI for the management of Virtual Machine instances. It provides many ready-to-use VM instances but does not allow building custom VM images. Strong Windows operating system support exists in GoGrid (Wayner, 2008). An API is also available for developers to manage the cloud programmatically.

2.1.5 AppNexus

AppNexus is another commercial cloud that offers services similar to Amazon EC2. It provides command line tools for managing VM instances. AppNexus allows rebuilding of virtual machine images from other sources like Amazon EC2 (Wayner, 2008).

2.1.6 IBM Blue Cloud

In November 2007, IBM introduced its cloud computing project named Blue Cloud (Sims, 2007). It aims to provide a cloud for testing and prototyping Web 2.0 applications within their enterprise environment. The idea is to help users working on Web 2.0 technologies (mashups, social networks, mobile commerce) by providing them infrastructure to test and prototype their applications.

2.1.7 Resource and Services Virtualization without Barriers (RESERVOIR)

RESERVOIR is a European Union FP7 funded project that aims to develop an infrastructure for Cloud Computing by using virtualization, grid computing, and business service management techniques. This project will help users to deploy and use services or applications requiring huge computation power and resources while taking care of QoS and security. The high-level objective of this project is to increase the competitiveness of the European ICT industry by introducing this infrastructure (RESERVOIR Team, 2008).

2.1.8 Project Caroline

Project Caroline is another project from Sun Microsystems. Project Caroline is a hosting platform for the development and delivery of Internet-based services. Project Caroline is

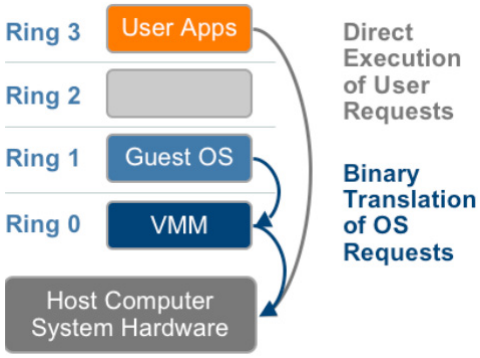

Figure 2.1: Full virtualization using binary translation on the X86 architecture. The VMM performs binary translation for OS requests while user requests execute directly. Reprinted from VMWare (2007).

hosted at `https://www.projectcaroline.net/` and available for consumers to use. Services developed using Java, Ruby, Python and Perl can easily be hosted on Project Caroline. Developers are allowed to manage deployed services programmatically. Services can be scaled horizontally by acquiring more resources dynamically.

2.2 Virtualization

Virtualization allows multiple operating systems to execution simultaneously on a physical machine. It is achieved through a virtual machine monitor (VMM). VMMs are also known as hypervisors. VMMs are responsible for keeping track of all activities performed by virtual machines. VMMs are categorized into Type-1 and Type-2 VMMs. Type-1 VMMs run directly on hardware without the need for a host OS, while Type-2 VMMs run on top of a host OS. Server virtualization can be achieved using Type-1 or Type-2 VMMs. Virtualization achieved through a Type-1 VMM is known as paravirtualization, and virtualization achieved through a Type-2 VMM is known as full virtualization.

On the X86 architecture, four privilege levels are available and known as Ring 0, 1, 2, and 3. Privilege levels are used to manage hardware resources to the applications and OS. Ring 0 is the most privilege level and Ring 3 is the least privilege level. Usually, OS instructions execute in Ring 0 and user application instructions execute in Ring 3. Virtualization needs to place VMM in the most privilege level (Ring 0) for managing the share resources for multiple operating systems.

2.2.1 Full Virtualization

Full virtualization is achieved using direct execution of user application code and binary translation of OS requests. The hypervisor traps and translates all OS instructions and caches the results for future use. User-level instructions run directly without any impact on their execution speed. Figure 2.1 shows how full virtualization is organized on the x86 architecture.

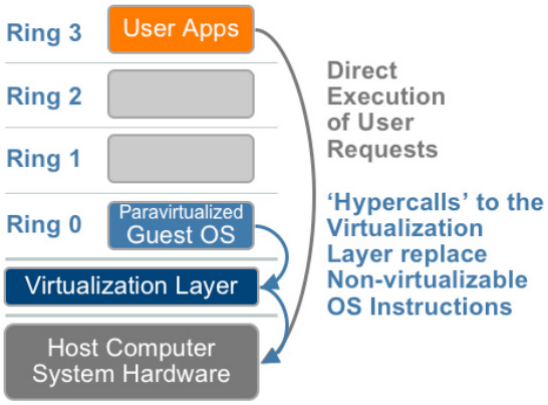

Figure 2.2: Paravirtualization on X86 architecture. Paravirtualization modifies the guest OS kernel and replaces non-virtualized instructions with custom hyper calls. Paravirtualization has low overhead because it does not perform binary translation for all OS calls. Reprinted from VMWare (2007).

2.2.2 Paravirtualization

Paravirtualization is achieved through the modification of a guest OS. It is only supported for open source operating systems. Paravirtualization is better than full virtualization in performance because it does not need to binary translate all OS calls. Since paravirtualization does not support unmodified operating systems such as Windows XP, its compatibility is poor. Paravirutlization is not recommended for production environments because it modifies the OS kernel; this can challenge the integrity and security of the host operating system. Xen is pioneer in the paravirtualization and popular among academics. Figure 2.2 shows how the paravirtualization is organized on the X86 architecture.

2.2.3 Hardware Assisted Virtualization

Hardware vendors realize the importance of virtualization. Intel and AMD offer enchantments to hardware for virtualization. Intel VT-x and AMD-V currently offer hardware assistance. VT-x and AMD-V introduce a new execution mode that allows the VMM to run in a new root mode below ring 0. Figure 2.3 shows the hardware-assisted approach to virtualization. As shown in Figure 2.3, the VMM executes in a root mode that enables the VMM to execute at a new privilege level, eliminating the need to trap sensitive instructions or modify the guest OS. The guest state is stored in Virtual Machine Control Structures (VT-x) or Virtual Machine Control Blocks (AMD-V).

2.3 Virtualization Platforms

In this section I review commercial and free/open source virtualization platforms.

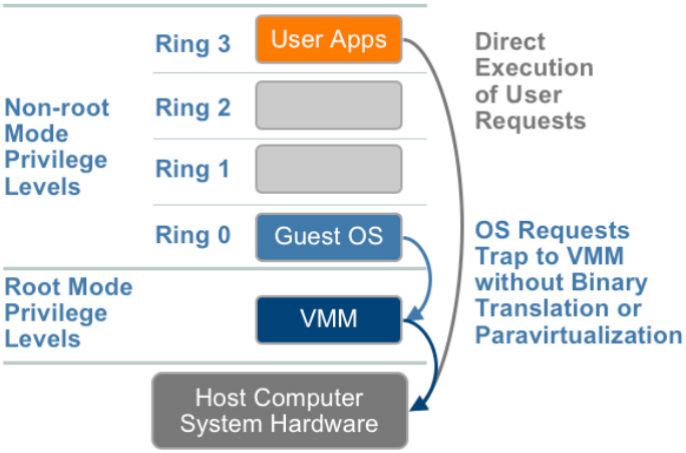

Figure 2.3: Hardware Assisted Virtualization approach for the X86 architecture. Hardware assistance in modern X86 CPUs introduces a new CPU privilege mode that allows VMMs to execute below ring 0. Sensitive OS calls automatically trap without using paravirtualization or binary translation, so virtualization overhead is decreased. Reprinted from VMWare (2007).

2.3.1 Xen

Xen is open source virtualization software that provides paravirtualization. It allows several guest operating systems to execute on one physical machine simultaneously. The first guest OS is known as Domain-0 in Xen terminology. Domain-0 automatically boots whenever Xen software boots. Users need to login on Domain-0 to execute other guest OSes (Paul et al., 2003). Xen provides paravirtualization and does not support the Window operating system.

2.3.2 VMWare

VMWare is commercial virtualization software that provides full virtualization. VMWare has many flavors such as VMWare Workstation, VMWare Server, and VMWare ESX that provide different levels of flexibility and functionality. VMWare is highly portable as it is independent of the underlying physical hardware, making it possible to create one instance of a guest OS using VMWare and copy it to many physical systems (Nieh & Leonard, 2000).

2.3.3 KVM

The Kernel-based Virtual Machine (KVM) is a free virtualization platform licensed under GPL version 2, provides full virtualization. KVM contains a Type-2 hypervisor and supports Linux-based host operating systems. It allows Windows or Linux as the guest OS. KVM works on the X86 architecture and supports hardware virtualization technologies such as Intel VT-x and AMD-D (Avi, Yaniv, Dor, Uri, & Anthony, 2007). The Ubuntu community provides KVM support.

2.3.4 Virtual Box

Virtual Box is a software package that provides paravirtualization. It was initially developed by a German company, Innotek, but now it is under the control of Sun Microsystems as part of the Sun xVM virtualization platforms. It supports Linux, Mac, Solaris, and Windows platforms as the host OS (Jon, 2008).

2.3.5 Hyper-V or Viridian and Microsoft Virtual PC

Hyper-V is a virtualization platform developed by Microsoft specifically for Windows 2008 Server. It allows creating Linux or any flavor of Windows guests. Prior to Hyper-V, Microsoft introduced "Microsoft Virtual PC," which is a freely available package initially developed by Connectix and later acquired by Microsoft. It only supports Windows for the host and guest operating systems (Wikipedia, 2009c).

2.4 Virtual Machine Management APIs and Tools

This section describes the available tools and APIs for virtual machine management.

2.4.1 Enomalism

Enomalism is an open source, Web based management tool for Xen virtual machines. It aims to help organizations with the development of scalable and efficient infrastructure for hosting applications within and outside their premises. The Enomalism feature list includes monitoring, resource management, live migration, SSH, and graphical login to virtual machines.

2.4.2 Xen Management User Interface (xm)

xm is a command line tool for the management of Xen guest domains. Executing xm commands requires administrative privileges on the system. This tool can be used to create, pause, shutdown, reboot, restore, list, and migrate Xen domains.

2.4.3 XenAPI

XenAPI is a management interface for remote virtual machines deployed on the Xen platform. This API aims to provide easy management of virtualized environments based on Xen. It provides a set of Remote Procedure Calls (RPC) with an invocation format based on XML-RPC. XML-RPC based bindings are available for the Python, Java, and C languages.

2.4.4 Libvirt

Libvirt is an open source toolkit for interacting with virtualization platforms. It is written in the C language. Libvirt supports the Xen, QEMU, KVM, LXC, OpenVZ, and User Mode Linux virtualization platforms. *Libvirt* bindings for C, Python, OCaml, and Ruby are available.

2.4.5 Virt-manager

Virt-manager is an open source GUI tool for the management of virtual machines. It was developed by Red Hat and its GUI is constructed using Glade and GTK+. It was originally designed for Xen-based virtual machines, but it also supports KVM and Qemu virtualization platforms because it is developed on top of *libvirt*. It provides a dashboard view of the currently running domains and their statistics. *Virt-manager* contains an embedded VNC client viewer that allows connection to guest domains and display of their graphical consoles. It also includes virt-install, virt-clone, and virt-image command-line tools.

2.4.6 Convirt

ConVirt is an open source solution for the management of virtualized platforms. It provides a graphical tool that aims to provide easy management of Xen and KVM based virtual machines. ConVirt provides most of the command line functionality that we can get from *xm*, but through a nicer user interface. Convirt allows users to start, pause, reboot, shutdown, kill, snapshot, resume, and access the console.

2.5 Open Source Cloud Frameworks

This section describes the available open source frameworks for developing private clouds.

2.5.1 EUCALYPTUS

EUCALYPTUS is an open source cloud computing framework developed by the University of California, Santa Barbara as an alternative to Amazon EC2. The aim of EUCALYPTUS is to enable academics to perform research in the field of cloud computing. It is composed of several components that interact with each other with well-defined interfaces. EUCALYPTUS therefore allows the open source community to replace or modify any component according to their requirements. Key functionalities of EUCALYPTUS are VM instance scheduling, construction of virtual infrastructure, administrative interfaces, user management of clouds, and definition and execution of service-level agreements. EUCALYPTUS allows the use of Amazon EC2 interfaces for interacting with the cloud (Nurmi et al., 2008). The three main

components of EUCALYPTUS are the Node Controller, the Cluster Controller and the Cloud Controller. Each is described below.

2.5.1.1 Node Controller

The Node Controller (NC) is responsible for controling the execution, inspection, cleanup, and termination of virtual machine instances on a physical machine. A EUCALYPUTS cloud may contain many NCs, but one physical machine only requires one NC. A single NC is therefore able to control all virtual machines executing on a single physical machine. The NC interface is exposed via a WSDL document that explains the data structure and control operations that the NC supports. The NC control operations are runInstance, terminateInstance, describeInstance, startNetwork, and describeResource.

2.5.1.2 Cluster Controller

A collection of NCs form Cluster Controller (CC) that typically executes on a head node or server that is able to access both the public and the private network. The CC is responsible for managing a collection of associated NCs. The CC interface is also exposed via a WSDL document and contains the same operations, but all operations are plural instead of singular (runInstances, describeInstances, terminateInstances, describeResources). Whenever a CC receives a runInstances request, it invokes the describeResource of all NCs and chooses the NC with enough resources to host another virtual machine. It is possible for one EUCALYPTUS installation to contain more than one CC.

2.5.1.3 Cloud Controller

Each EUCALYPTUS-based cloud must include one Cloud Controller (CLC) that is the entry point for users. The CLC is responsible for making global decisions that include user authentication, processing of user or administrative requests, high level decisions for VM instance scheduling, and processing of SLAs. Figure 2.4 shows the hierarchal interaction of NCs, CCs, and CLC in a cloud with client requests.

2.5.2 Nimbus

Nimbus (Globus Alliance, 2008) is an open source toolkit developed by the Globus Alliance community. It allows users to expose a cluster of physical resources into a cloud. Nimbus allows development of a virtual private cluster known as a virtual work space in which users are allowed to instantiate multiple virtual machines and configure them according to their needs. Workspace-control is an important component that installs on all physical nodes and us used for virtual machine management. Nimbus partially supports the EC2 client interface and contains custom tools for client interaction. Nimbus is able to utilize Xen and KVM virtualization platforms.

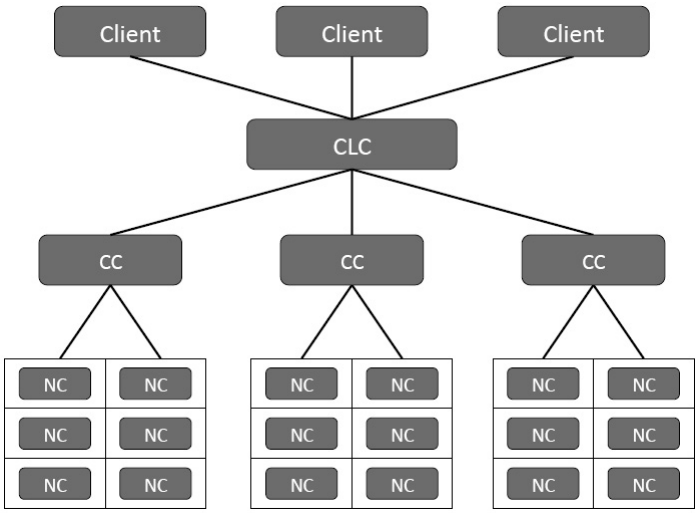

Figure 2.4: Hierarchal interaction of EUCALYPTUS components(Node Controller (NC), Cluster Controller (CC), and Cloud Controller (CLC)) with client requests. Each box at the bottom that contains NC represents one physical node. Reprinted from Nurmi et al. (2008).

2.5.3 OpenNebula

OpenNebula (OpenNebula.org, 2008) is an open source engine that enables dynamic deployment of virtual machines on a set of physical machines. It supports Xen and KVM virtualization platforms. The RESERVIOR project, described in section 2.1.7, enhanced OpenNebula. It is possible to integrate an OpenNebula-based cloud with other remote clouds due to its modular design. OpenNebula cloud contains a repository on the cluster frontend to store virtual machines. The Virtual Network Manager and the Transfer Manager are two important components of OpenNebula. The Virtual network Manager allows users to create virtual networks by mapping physical networks into multiple virtual networks. The Transfer Manager fetches virtual machines from the repository and transfers them to physical machines. OpenNebula exposes its core functionality via XML-RPC.

2.6 Dynamic Resource Provisioning

This section provides brief overview of recent research in dynamic resource allocation.

Kalyvianaki, Charalambous, and Hand (2008) present a feedback controller scheme using Kalman filters for multi-tier applications hosted on virtual machines. They use this scheme for dynamic allocation of CPU resources to virtual machines. The figure 2.5 shows a prototype of their design. Their experiments use the RUBIS Web application deployed on the Xen virtualization platform. Three physical machines are used to host three virtual machines. Tocmcat, JBoss and MySQL are hosted on separate virtual machines. They developed a module called "manager," that obtains CPU usage information for certain intervals and submit that information to another module called "Controller." The controller module is responsible for computing the CPU allocation for the next time interval and adjust the CPU resources for specific VMs. In the future, authors hope to demonstrate that their model

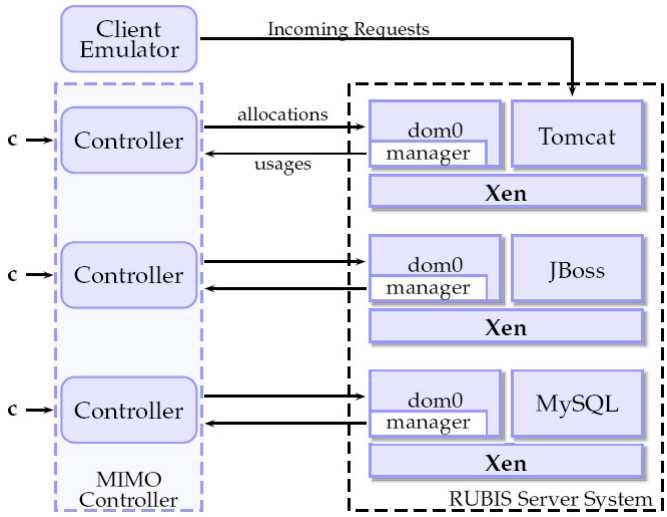

Figure 2.5: Prototype dynamic CPU resource allocation system using a Kalman filter based controller. Reprinted from Kalyvianaki et al. (2008).

performs well even when many virtual machines share a single CPU.

Nowadays, e-commerce applications are gaining in popularity. E-commerce Web applications contain sensitive resources and require access over HTTPS. Cryptographic techniques are used in e-commerce application to ensure confidentiality and authentication. These techniques are CPU-intensive and may lead to overload condition when many concurrent clients connect to the system. Guitart, Carrera, Beltran, Torres, and Ayguadé (2008) use admission control and dynamic resource provisioning to develop an overload control strategy for secure Web applications hosted on SMP (Symmetric MultiProcessing) platforms. They implemented a global resource manager for the Linux hosting platform that is responsible for allocating resources to the application servers running on it and ensures the desire QoS in terms of performance stability for extreme overloads. Server machines are able to manage themselves on changes of workload and adapt the load.

Piyush et al. (2007) demonstrate two software systems, Shirako (David et al., 2006) and NIMO (Piyush, Shivnath, & Jeff, 2006). Shirako is a Java toolkit for dynamic resource allocation. It allocates virtual resources to guest applications from available resources. NIMO creates an application performance model using active learning techniques. The authors use NIMO to estimate the minimum resources required for a guest application, and they use Shirako to allocate those resources. NIMO builds performance models by capturing resource requirements, characteristics of the data and the workload of the guest application.

Data centers offer services for storage and application hosting. Most of the applications are real time and require SLAs. The real challenge is to deal with unpredictable workload and Web application failure. Oftentimes, virtual appliances are preconfigured and ready to run virtual machines for specific applications. XiaoYing et al. (2007) develop a Virtual Application Environment (VAE) using virtual appliances as a mechanism for resource allocation. The idea is to deploy multiple virtual appliances on separate physical machines for specific applications and use an on-demand router that makes scheduling decisions on the basis of workload. Whenever more resources are required, they deploy another virtual appliance. The key issue with this approach is wastage of resources, because they allocate the same amount of resources to the new virtual appliance as the other virtual appliances are using.

Emerson, Paddy, and Simon (2008) address this problem, proposing a fine-grained technique for on demand resource provisioning in data centers. They designed a *Global Controller* that decides the CPU resource assignment and places the application on an appropriate machine.

Chapter 3

Methodology

3.1 System Overview

This chapter describes the methodology I used to achieve the specific objectives of this research. I developed a private network using four workstations to establish a heterogeneous cloud on the CS&IM LAN. I used the EUCALYPTUS framework to establish a cloud with one Cloud Controller (CLC), one Cluster Controller (CC), and three Node Controllers (NCs). I installed the CLC and CC on a front-end node that is on the CSIM LAN and the clouds private network. I installed the NCs on three separate machines connected to the private network. I used EUCALYPTUS to instantiate virtual machines on the node controllers.

Ngnix is a HTTP and mail proxy server capable of load balancing. I used Ngnix as a load balancer because it offers detailed logging and allows reloading the configuration file without termination of existing client sessions. I installed the Ngnix as a load balancer on the front-end node and the Tomcat application server on the virtual machine. I stored the virtual machine image in EUCALYPTUS, which cached it on all three NCs. My Web application contains one Java servlet that performs a matrix calculation. The Tomcat server hosts the Web application while Ngnix balances the load. I used Httperf to generate a HTTP workload for the Web application. I developed a software component named *VLBManager* to monitor the average response time of requests by real-time monitoring of Ngnix logs. I developed another software component named *VLBCoordinator* to invoke the virtual machines on the EUCALYPTUS-based cloud. Both components interact with each other using XML-RPC. The *VLBManager* detects the violation of the average response time for each virtual machine in the Web farm due to the heterogeneous testbed cloud. Whenever *VLBManager* detects a violation of the average response time, it signals the *VLBCooridnator* to invoke another virtual machine and add it to the Web farm.

3.2 System Design

The system design is divided into five sections: network design, EUCALYPTUS installation, software components, sample Web application, and workload generation, described below.

3.2.1 Network Design

The private network consists of four physical machines and one 100 Mbps Ethernet switch. Ubuntu 8.04 (hardy) runs the front-end node, and Debian 5.0 (lenny) runs installed on all other nodes. Figure 3.1 shows the basic network design. The front-end node is equipped with two Network Interface Cards (NICs). One NIC is used to connect the front-end with CSIM LAN, and another connects with the EUCALYPTUS private network. SMTP and

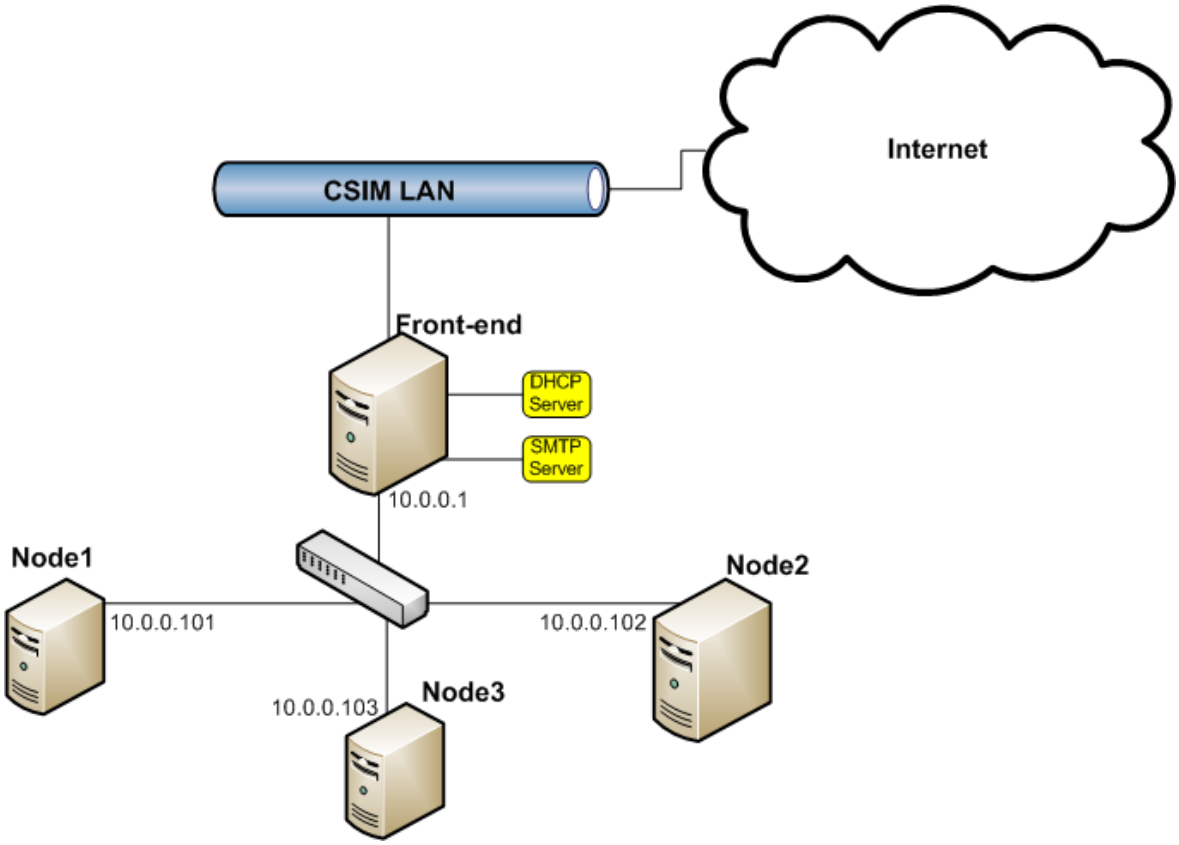

Figure 3.1: Basic network design. This network consists on four physical machines and one 100 Mbps ethernet switch.

DHCP servers are installed on the front-end node for the private network as required by EUCALYPTUS installation. Node1, Node2, and Node3 are members of the private network and are able to access the Internet using IP Masquerading (Ranch, 1999). For heterogeneity, I used machines with different configurations. Table 3.1 shows the hardware configuration of machines.

3.2.2 EUCALYPTUS Installation

EUCALYPTUS is installed on top of the network described in Section 3.2.1. Figure 3.2 shows the deployment of EUCALYPTUS components on the private network. The Cloud Controller (CLC), Cluster Controller (CC), and Ngnix are installed on Front-end. Three Node Controllers (NCs) are installed on Node1, Node2, and Node3. The EUCALYPTUS Web portal is accessible on the CSIM LAN.

3.2.3 Software Components

I developed two software components, namely *VLBCoordinator* and *VLBManager*, using Java. The components are explained in Section 3.2.3.1 and 3.2.3.2, respectively.

Table 3.1: Hardware configuration of physical machines.

Node Name	Type	CPU Frequency	RAM
Front-end	Intel Pentium	2.80 GHz	2 GB
Node1	Intel Pentium	2.66 GHz	1.5 GB
Node2	Intel Celeron	2.4 GHz	2 GB
Node3	Intel Core 2 Duo	1.6 GHz	1 GB

3.2.3.1 VLBCoordinator

VLBCoordinator is used to interact with the EUCALYPTUS cloud. Typica (Google Code, 2008) is a simple API written in Java to access a variety of Amazon Web services such as SQS, EC2, and SimpleDB. Currently, Typica is not able to interact with Eucalyptus-based clouds. I hacked Typica to interact with EUCALYPUTS. VLBCoordinator uses Typica to interact with EUCALYPTUS. The core functions of VLBCoordinator are *instantiateVirtualMachine* and *getVMIP*, which are accessible through XML-RPC. Figure 3.3 shows a sequence diagram of a use case in which a user wants to instantiate a virtual machine by providing its virtual machine id. Figure 3.4 shows a sequence diagram of a use case in which the user needs to obtain the IP address of a virtual machine by providing its instance id.

3.2.3.2 VLBManager

VLBManager is used to monitor the logs of the load balancer and detects violations of the response time requirements. It reads the real time logs of the load balancer and calculates the average response time after every 60 seconds for each virtual machine in a Web farm. Whenever it detects that the average response time of any virtual machine is exceeding the required response time, it invokes the *instantiateVirtualMachine* method of *VLBCoordinator* with the required parameters and obtainer a new instance id. After obtaining the instance id, *VLBManager* waits for few seconds and tries to obtain instance id by using XML-RPC call to *VLBCoordinator*. After receiving the IP address of the newly instantiated virtual machine, it updates the configuration file of Ngnix and sends a signal to Ngnix requesting it to reload the configuration file. Figure 3.5 shows a sequence diagram of the main use case, in which the system reads the Ngnix logs and analyzes them to obtain the average response time of each virtual machine. Figure 3.6 shows a sequence diagram of the use case in which the system detects that the average response time is greater than the maximum allowed response time and adds another virtual machine to the Web farm to distribute the load.

3.2.4 Sample Web Application

The CPU is normally the bottleneck in the generation of dynamic contents (Challenger et al., 2004). Therefore I consider a simple Web application consisting of one Java servlet that performs a CPU-intensive job. The servlet accepts two parameters, *baseline* (baseline

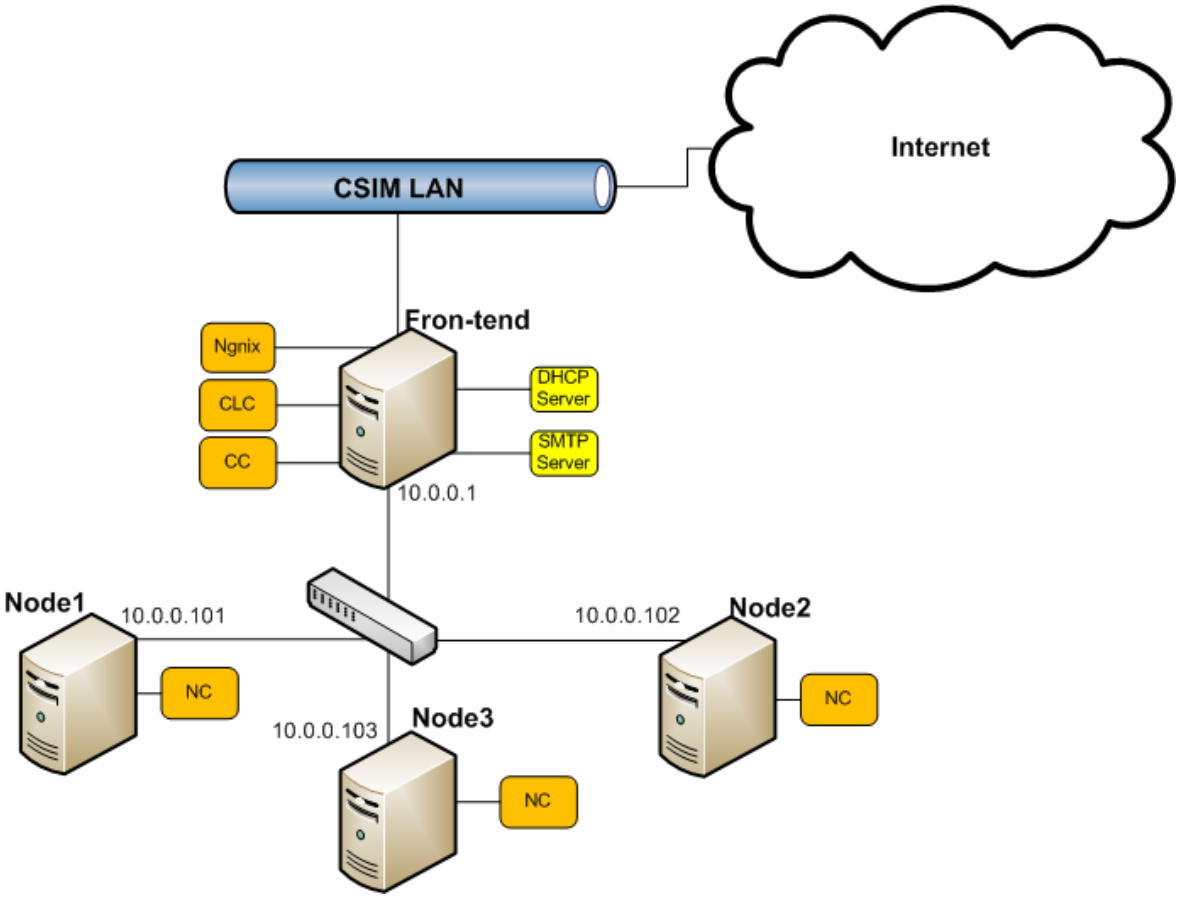

Figure 3.2: EUCALYPTUS installation on top of the basic network from Figure 3.1.

time) and *iters* (iterations) and performs a matrix calculation up to the baseline time. The complete code of the servlet is given in Appendix A.

3.2.5 Workload Generation

I developed a shell script to generate synthetic workload for the sample Web application. The shell script contains the following httperf command:

```
httperf --hog  --timeout=100 --server=10.0.0.1 --port=80
--wsesslog=10000000,0,input.txt --rate $1 -v
```

The parameter *–rate* is used to define the number of sessions created per second. Usually, Web applications users make requests interactively and take some time during requests, known as user think time. The parameter *–wsesslog* defines the total number of sessions to generate, user think time, and input file name. The input file allows specification of a sequence of requests, the request method, and user think time per burst. The shell script allows control of the duration of the workload. My input.txt file contains ten requests of four bursts and each burst contains some think time in seconds. This script allows generation of workload

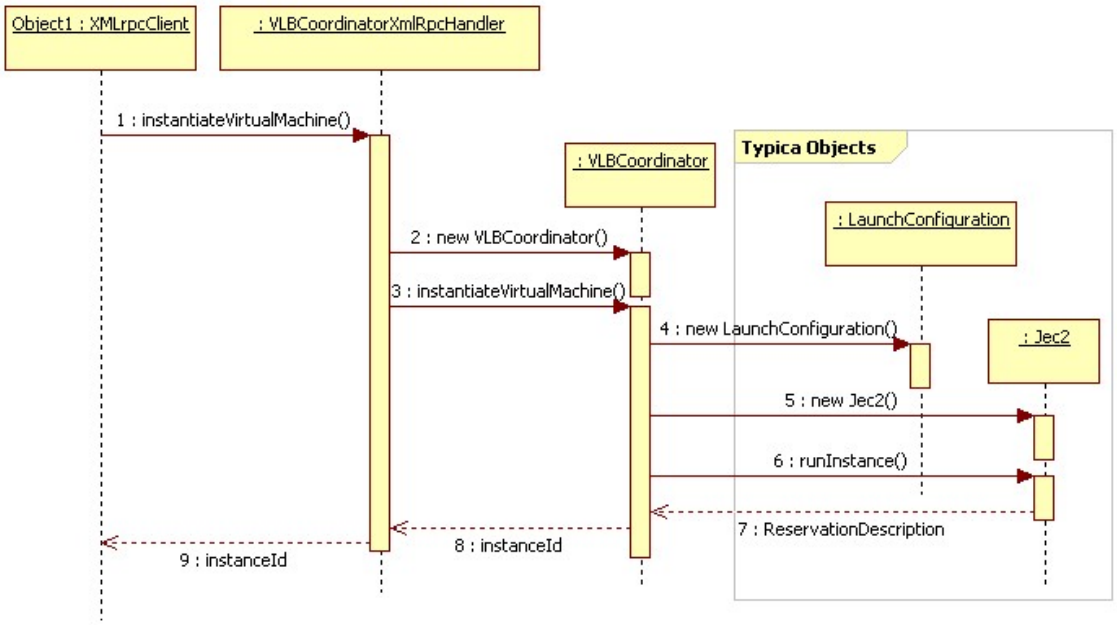

Figure 3.3: VLBCoordinator sequence diagram for instantiating a virtual machine on a EUCALYPTUS cloud. User makes an XML-RPC call to the instantiateVirtualMachine method, providing a virtual machine id. VLBCoordinatorXMLRpcHandler performs the necessary steps and returns the new instance id to the client.

for a specific duration with a required number of sessions per second. Each session contains 10 requests defined in input.txt file. The contents of my input.txt file are:

```
/app1/cars?baseline=37&iters=40 think=7.428
        /app1/cars?baseline=73&iters=40
        /app1/cars?baseline=73&iters=40
/app1/cars?baseline=64&iters=40 think=1.444
        /app1/cars?baseline=73&iters=40
/app1/cars?iters=40&baseline=124 think=8.194
        /app1/cars?baseline=73&iters=40
        /app1/cars?baseline=73&iters=40
/app1/cars?iters=40&baseline=124 think=2.194
        /app1/cars?baseline=73&iters=40
```

3.2.6 System Architecture

Figure 3.7 shows the system architecture. For simplicity, I installed the *VLBCoordinator* and *VLBManager* on the front-end node. VM1, VM2 and VM3 are copies of the same virtual machine, containing the sample Web application. They are cached on the physical machines using EUCALYPTUS. The NCs are configured to host only one virtual machine each. Initially, only one virtual machine (VM1) is alive and part of the Web farm. The workload generator generates traffic for the Web application through the Ngnix load balancer.

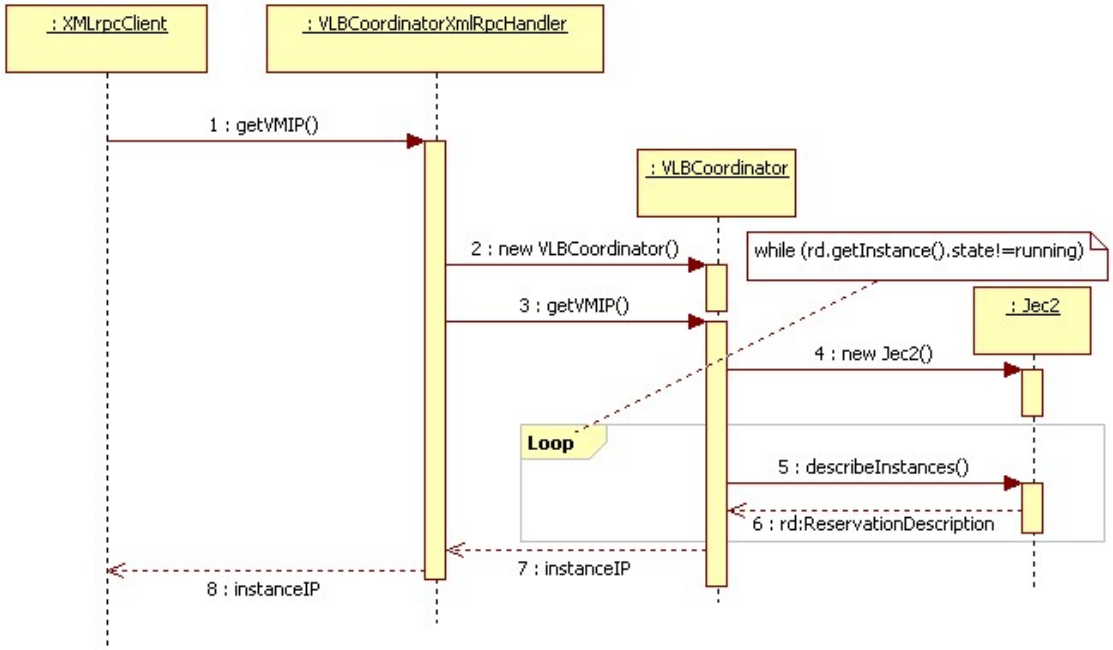

Figure 3.4: VLBCoordinator sequence diagram for obtaining the IP address of a virtual machine. User makes an XML-RPC call to the getVMIP method, providing a virtual machine instance id. VLBCoordinatorXMLRpcHandler performs the necessary steps and returns the IP address to the user.

VLBManager continuously monitors the Ngnix logs and detects violations of the response time requirements for each virtual machine in the Web farm. Whenever *VLBManager* detects a violation, it signals *VLBCoordinator* to deploy another virtual machine and add the newly instantiated machine to the Web farm.

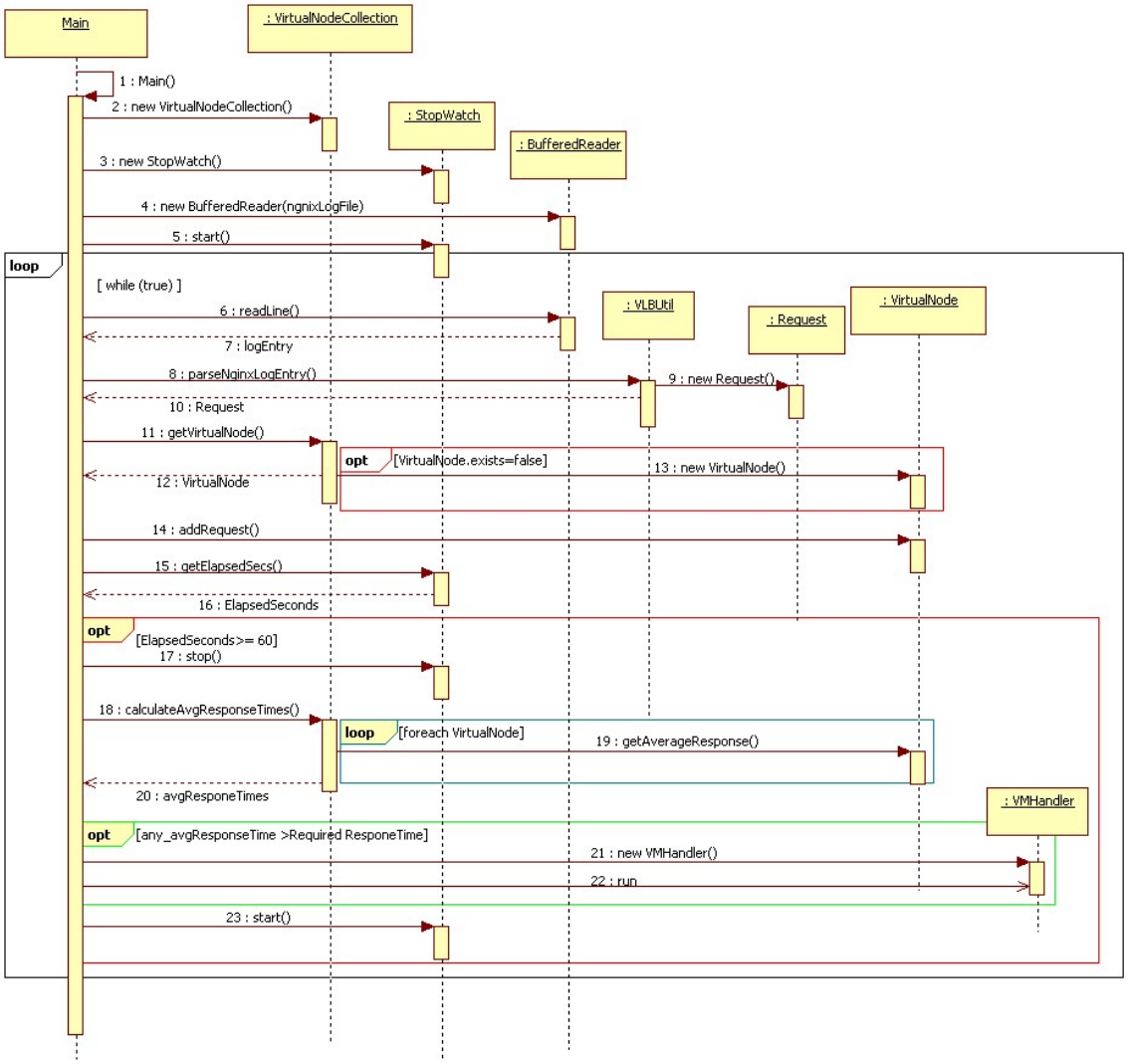

Figure 3.5: VLBManager sequence diagram for the main use case in which the system reads the Ngnix logs and analyzes them to obtain the average response time of each virtual machine. When VLBManager detects a violation of the response time requirements, it makes an asynchronous call to VMHandler.

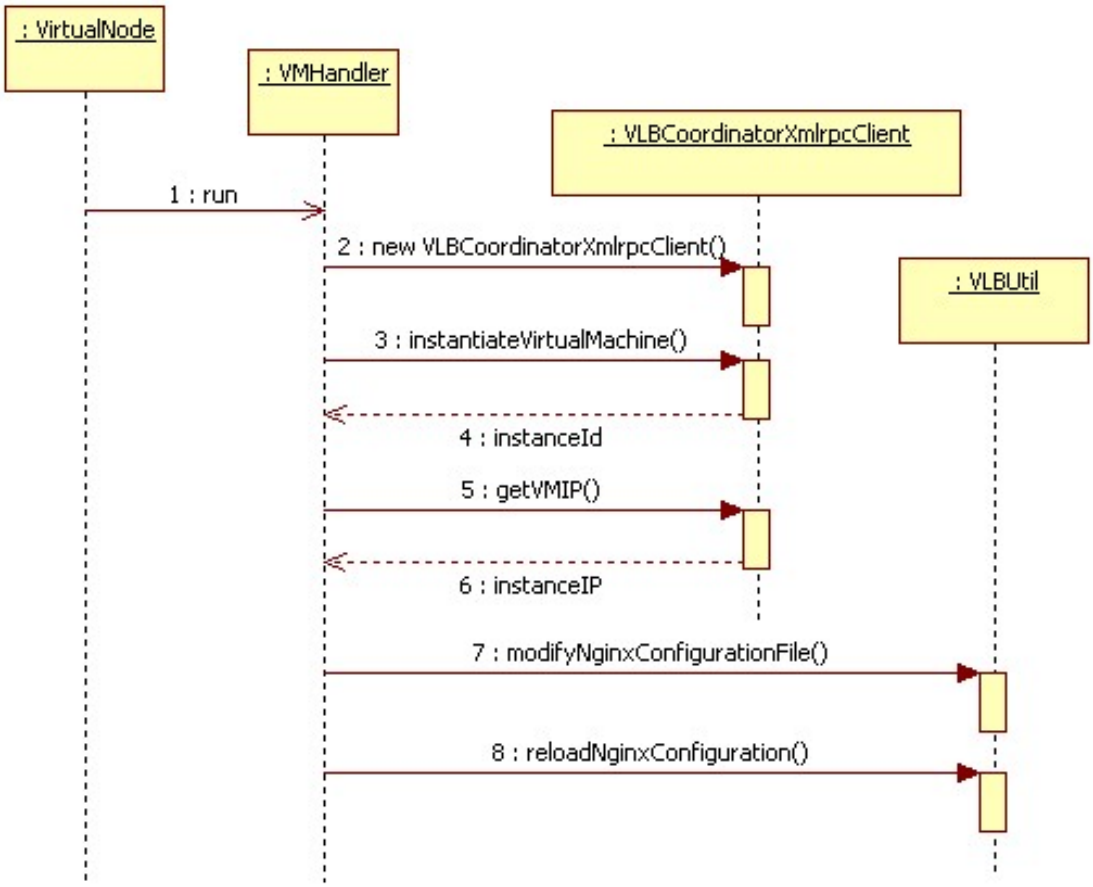

Figure 3.6: VLBManager sequence diagram for adaptive instantiation of a virtual machine and adding it to the Web farm.

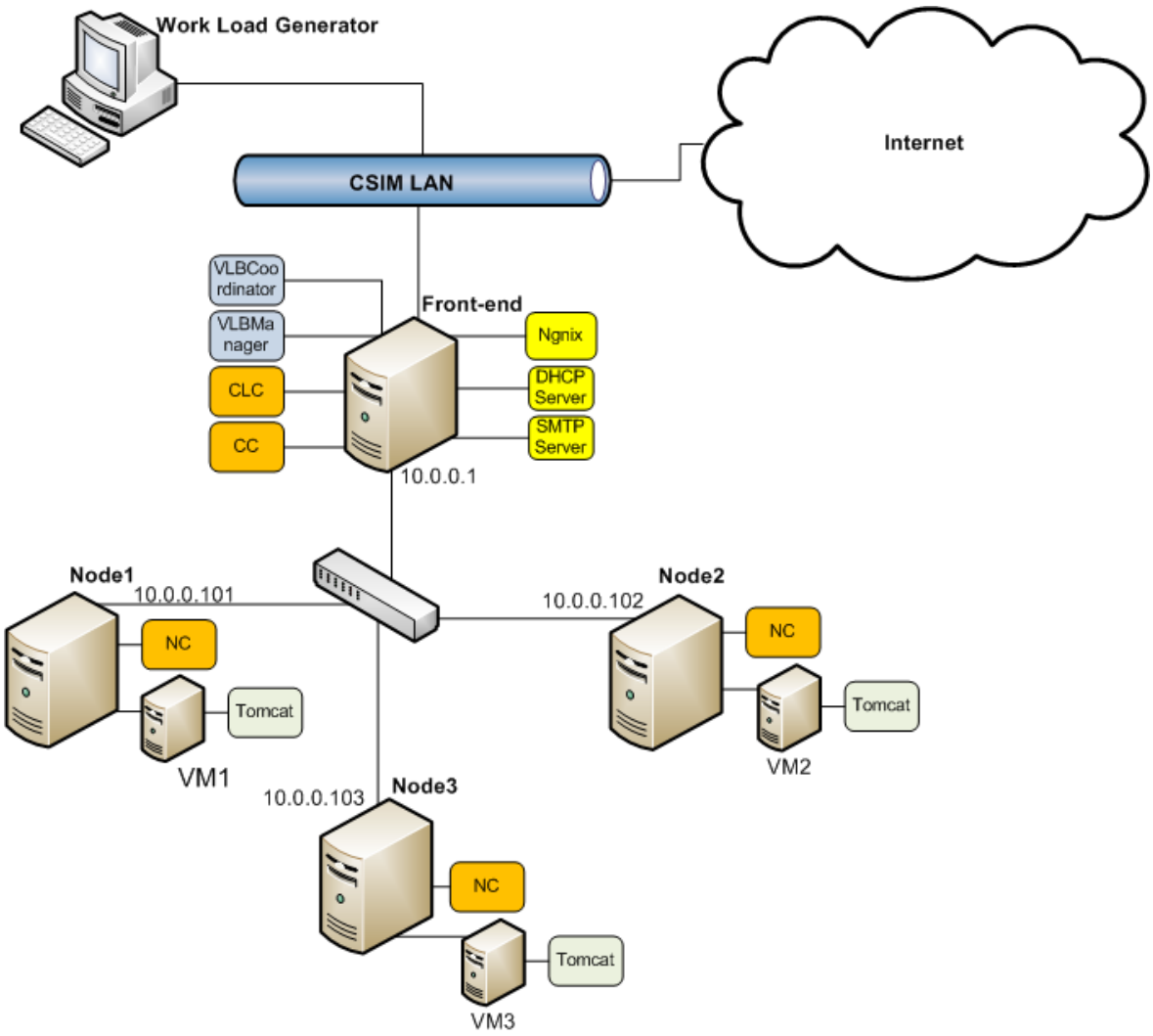

Figure 3.7: System architecture diagram. VM1, VM2 and VM3 are copies of the same virtual machine hosting the same Web application. Initially, only VM1 is alive, while VM2 and VM3 are invoked adaptively to satisfy response time requirements.

Chapter 4

Experiments and Results

4.1 Overview

This chapter describes the experiments I performed to evaluate my prototype system. I performed two experiments to obtain average response times for sample Web application. Experiment 1 investigates the system's behavior with static allocation of the resources to the sample Web application. Experiment 2 investigates the system's behavior with adaptive allocation of resources to the sample Web application to satisfy specific response time requirements. The same workload is generated for both experiments.

4.2 Experiment 1: Static resource allocation to Web application

In this experiment, I established an experimental setup in which only one virtual machine (VM1) hosts the Web application. Figure 4.1 shows the experimental setup established for this experiment. The Ngnix based Web farm consists of only one virtual machine (VM1). VLBManager is only used to obtain the average response time by actively monitoring the Ngnix logs.

Figure 4.2 shows the synthetic workload generated during this experiment. We linearly increase the load level from load level 1 through load level 12. The duration of each load level is four minutes. Each load level represents the number of sessions per second, and each session invokes 10 requests to the sample Web application, as previously described in Section 3.2.5.

4.2.1 Results

This section describes the results I obtained in Experiment 1. Figure 4.3 shows the number of requests served by our system. After load level 6, we do not observe any growth in the number of served requests because our Web server reaches the saturation point. Although the load level increases with time, our system is unable to serve all requests, and it either rejects or queues the remaining requests.

Figure 4.4 shows the average response time obtained during each load level. The duration of each load level is 4 minutes. It is clear that from load level 1 to load level 5, we observe a nearly constant response time, but after load level 5, we see rapid growth in the average response time. This is the saturation point for our Web server. When the Web server reaches the saturation point, requests spend more time in the queue. As the Web server gets increasingly loaded, it starts rejecting incoming requests. From load level 5 to load level 10, some requests spending time in the queue and relatively few requests get rejected. After

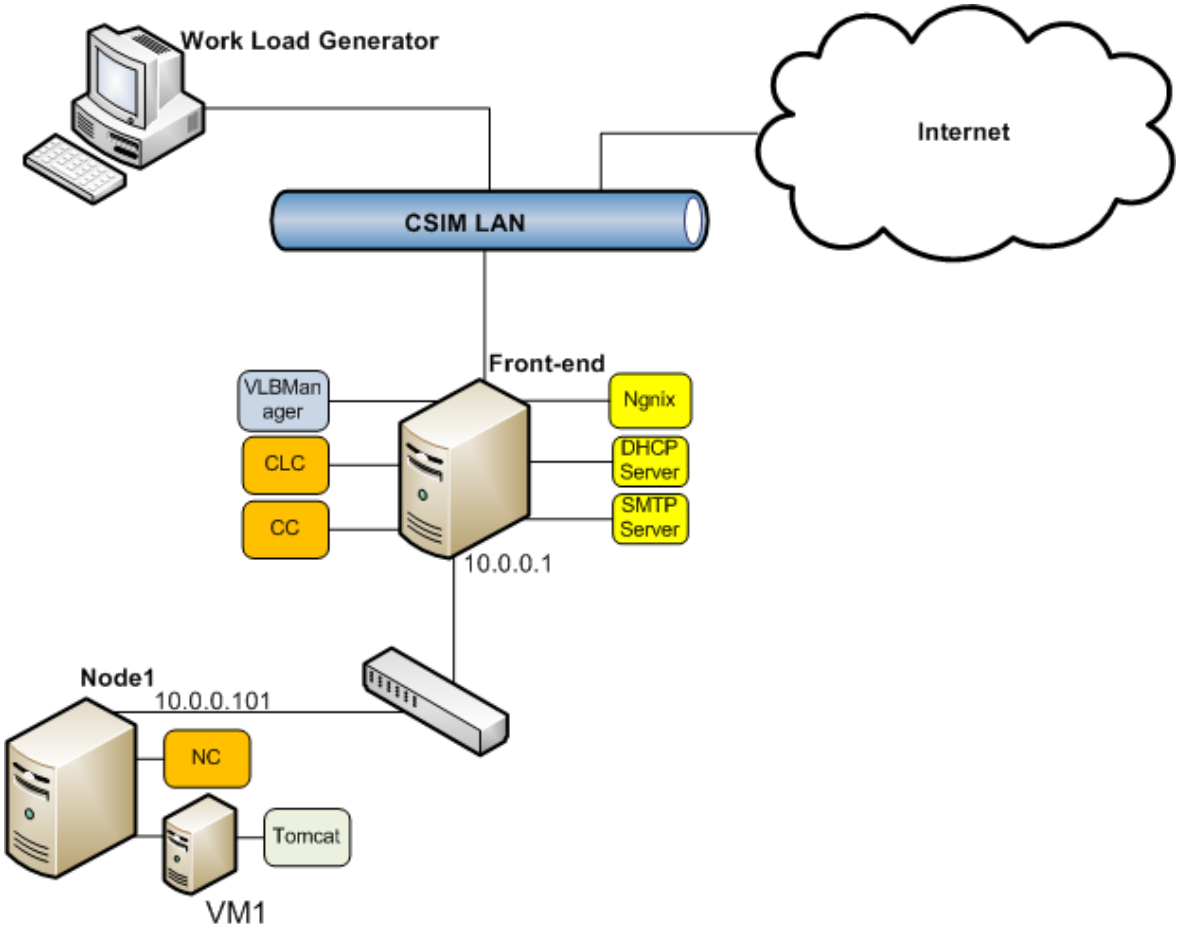

Figure 4.1: Experimental setup for Experiment 1. This setup is used to investigate the system's behavior with statical allocation of resources to the sample Web application. The Web application is deployed on one virtual machine (VM1). The Ngnix Web farm contains only one virtual machine. VLBManager is only used to obtain the average response time by actively monitoring the Ngnix logs.

load level 10, the Web server queue is also saturated and the system rejects more requests. Therefore we do not observe further growth in the average response time. Figure 4.5 shows the CPU utilization of VM1 during Experiment 1. Each load level duration is 4 minutes (240 seconds), so each tick on the X-axis represents an increment in the load level. After load level 5, the CPU is almost fully utilized by the Tomcat application server.

4.2.2 Discussion

Whenever the arrival rate exceeds the limit of the Web server's processing capability then requests spend more time in the queue, leading to higher response times. When the Web server queue is also full or saturated, it starts to reject incoming requests, and every instance serves only a fixed number of requests. The increase in response time thus tapers off after some time. We cannot sign a SLA guaranteeing a specific response time with an undefined load level for a Web application using static resource allocation.

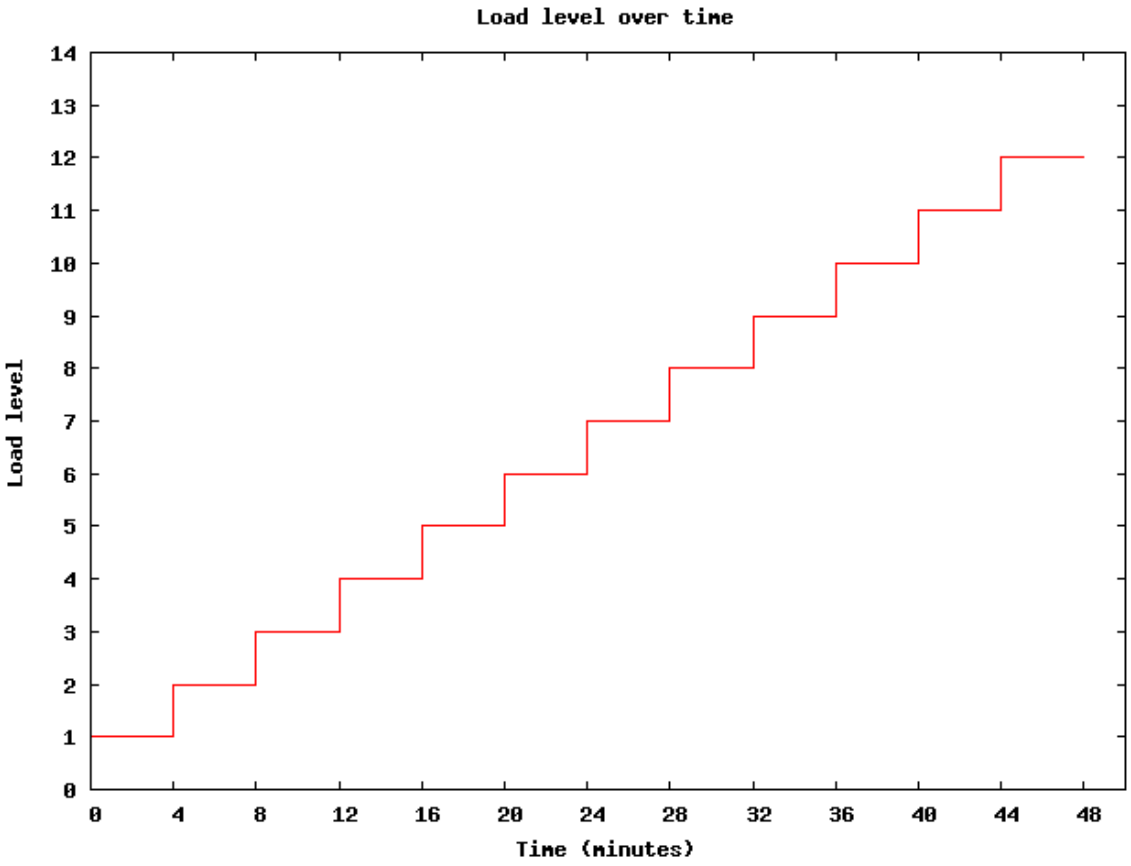

Figure 4.2: Work load generation for both experiments. We linearly increase the load level from load level 1 through load level 12. The duration of each load level is four minutes. Each load level represents the number of sessions per second and each session makes 10 requests to the sample Web application.

4.3 Experiment 2: Adaptive resource allocation to Web application

In this experiment, I used our proposed system to prevent response time increases and rejection of requests by the Web server. Figure 4.6 shows the experimental setup I established for this experiment. Initially, the Web farm is deployed with one virtual machine (VM1), while VM2 and VM3 are cached to the physical system using EUCALYPTUS. In this experiment, I try to satisfy a Service Level Agreement (SLA) that enforces a two-second maximum average response time requirement for the sample Web application on an undefined load level. I use VLBManager to monitor the Ngnix logs and detect the violations of the SLA. I use VLB-Coordinator to adaptively invoke additional virtual machines as required to satisfy the SLA. The same workload from Experiments 1 is used in this experiment, as explained in Figure 4.2.

4.3.1 Results

This sections describes the results of Experiment 1. Figure 4.7 shows the number of requests served by the system. We observe linear growth in the number of requests served by the

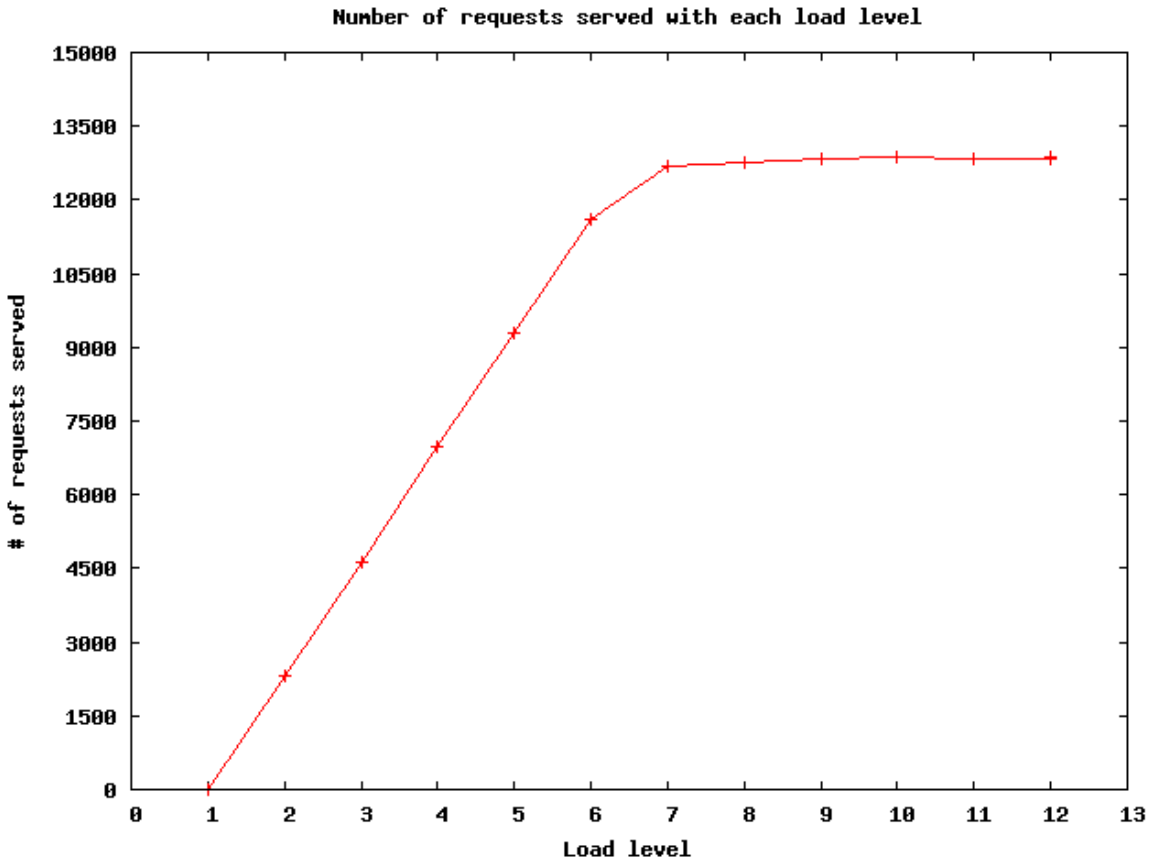

Figure 4.3: Number of requests served by system during Experiment 1. After load level 6, we do not observe any growth in the number of served requests, because Web server reaches its saturation point. Although load level continues to increase with time, the system is is unable to serve all requests, and it either rejects or queues the remaining requests.

system with each load level. Figure 4.8 shows the average response time of each virtual machines as it is adaptively added to the Web farm. Whenever the system detects a violation of the response time requirment from any virtual machine, it dynamically invokes another virtual machine and adds it to the Web farm. We observe continued violation of the required response time due to the latency of virtual machine boot-up. Figure 4.9 shows the CPU usage of each virtual machine. The duration of each load level is four minutes or 240 seconds, so each tick on the X-axis represents an increment in the load level. After load level 6 (1440 seconds), VM2 is adaptively added to the Web farm to satisfy the response time requirement. After load level 10 (2400 seconds), VM3 is adaptively added to the Web farm. Different load levels for different VMs reflect the use of round robin balancing and differing processor speeds for the physical nodes.

4.3.2 Discussion

Adaptively management of resources on compute clouds for Web application helps us offer SLAs that enforce specific response time requirements. To avoid the continued violation of

the SLA during VM boot-up, it would be better to predict violation of response time rather than to wait until the requirement is violated.

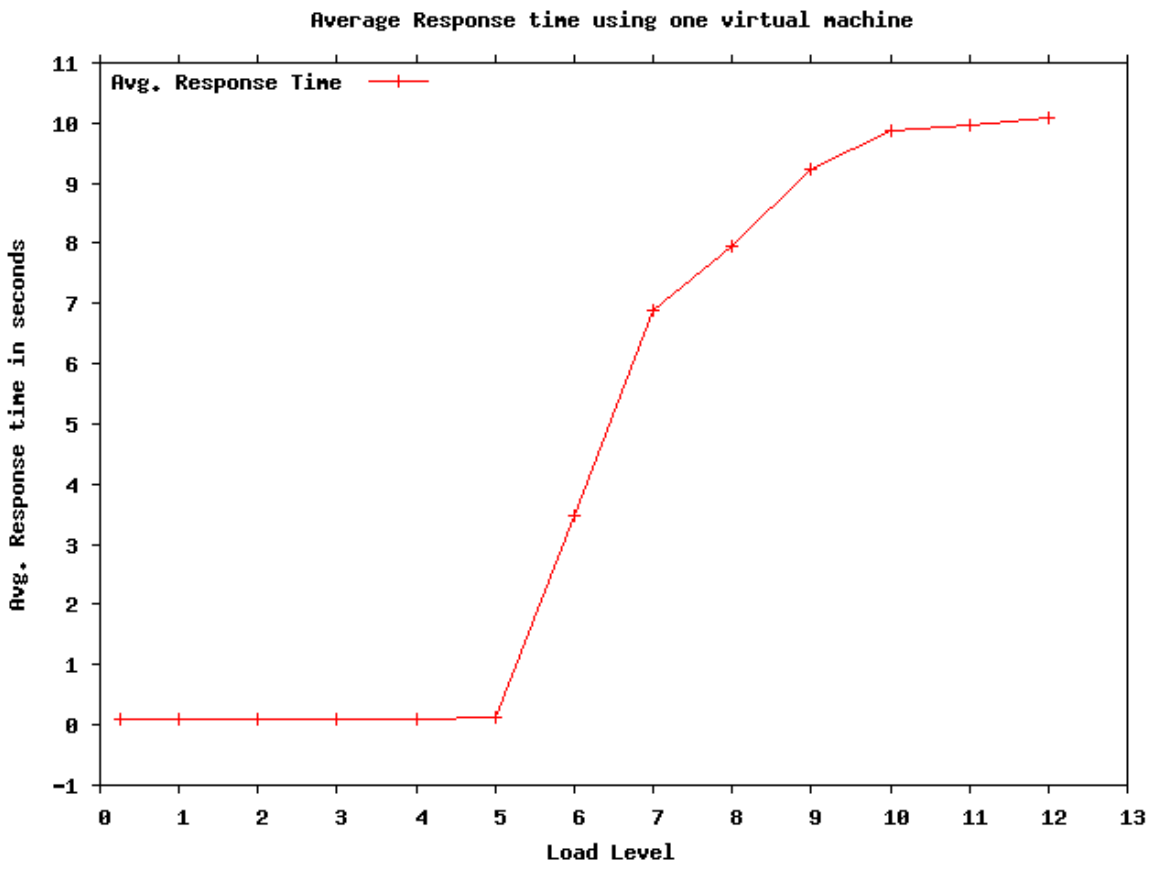

Figure 4.4: Average response time for each load level during Experiment 1. The duration of each load level is 4 minutes. From load level 1 to load level 5, we observe a nearly constant response time. After load level 5, we see rapid growth in the average response time. This is, when from Figure 4.3, the saturation point for Web server is reached. When the Web server reaches the saturation point, requests spend more time in the queue, and the system rejects some incoming requests. From load level 6 to load level 10, some requests spend time in the queue and few requests get rejected. After load level 10, the Web server queue is also saturated, and the system rejects more requests.

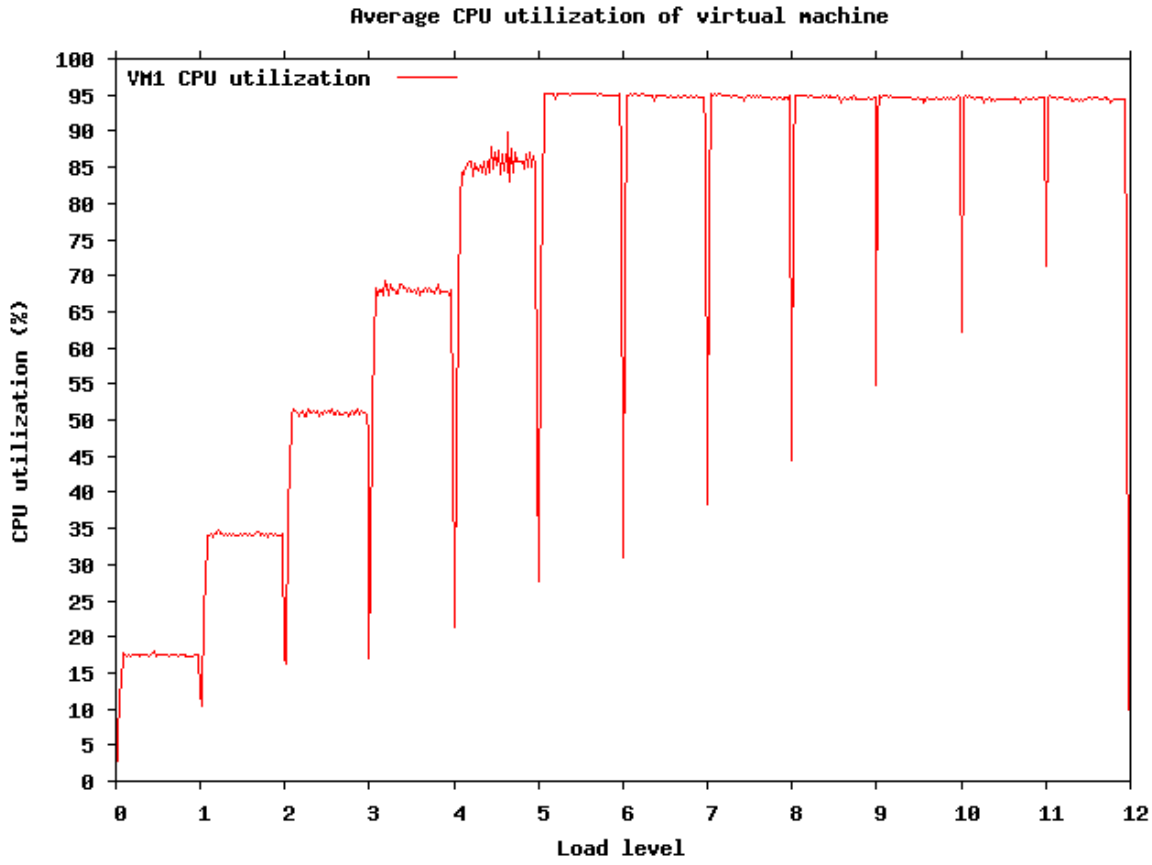

Figure 4.5: CPU utilization of VM1 during Experiment 1. Each load level duration is 4 minutes (240 seconds), so each tick on the X-axis represents an increment in the load level.

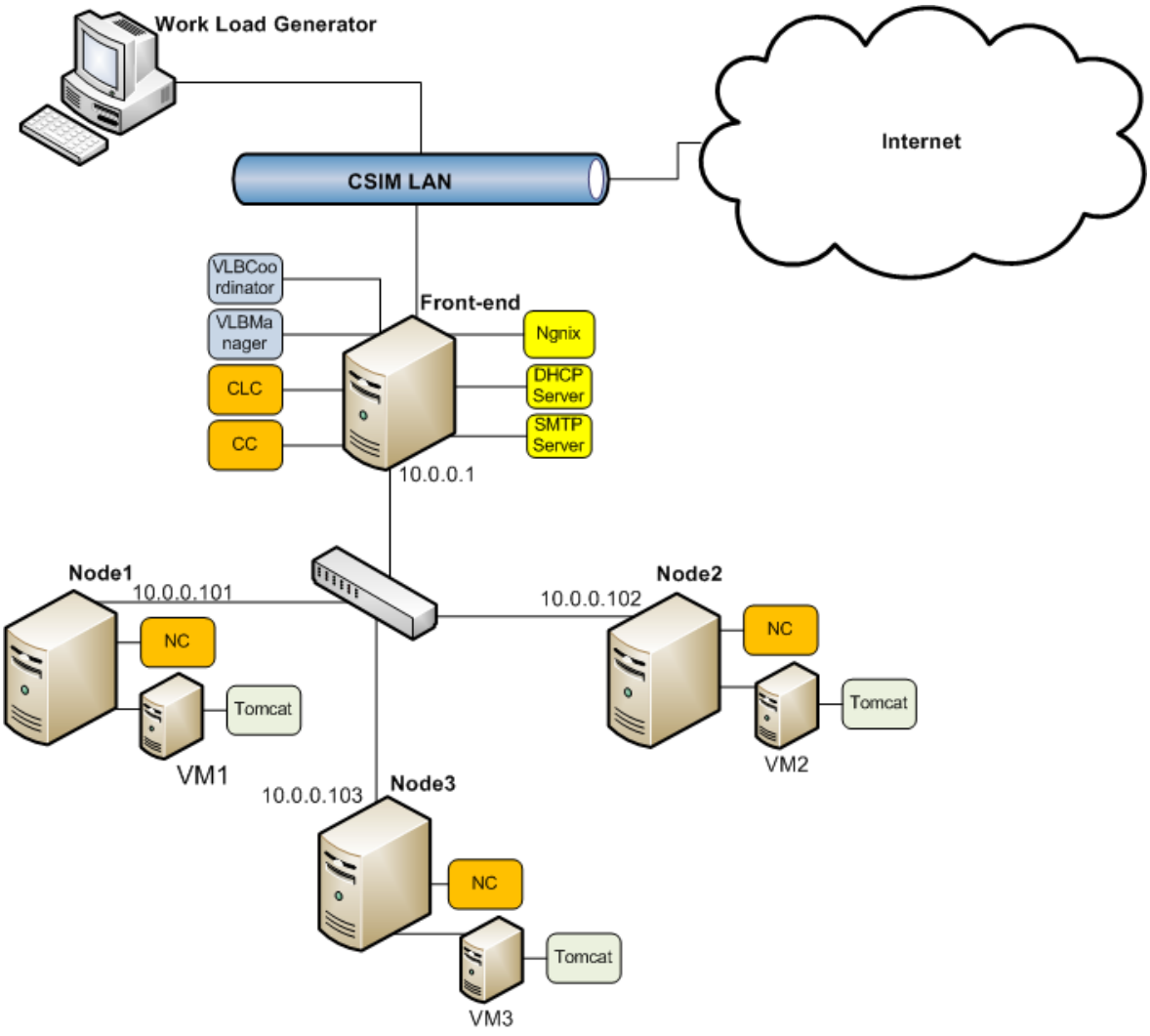

Figure 4.6: Experimental setup for Experiment 2 (dynamic resource allocation). Initially, the Web farm is deployed with one virtual machine (VM1), while VM2 and VM3 are cached to the physical system using EUCALYPTUS. VLBManager monitors the Ngnix logs and detects violations of the SLA. VLBCoordinator adaptively invokes additional virtual machines as required to satisfy the SLA.

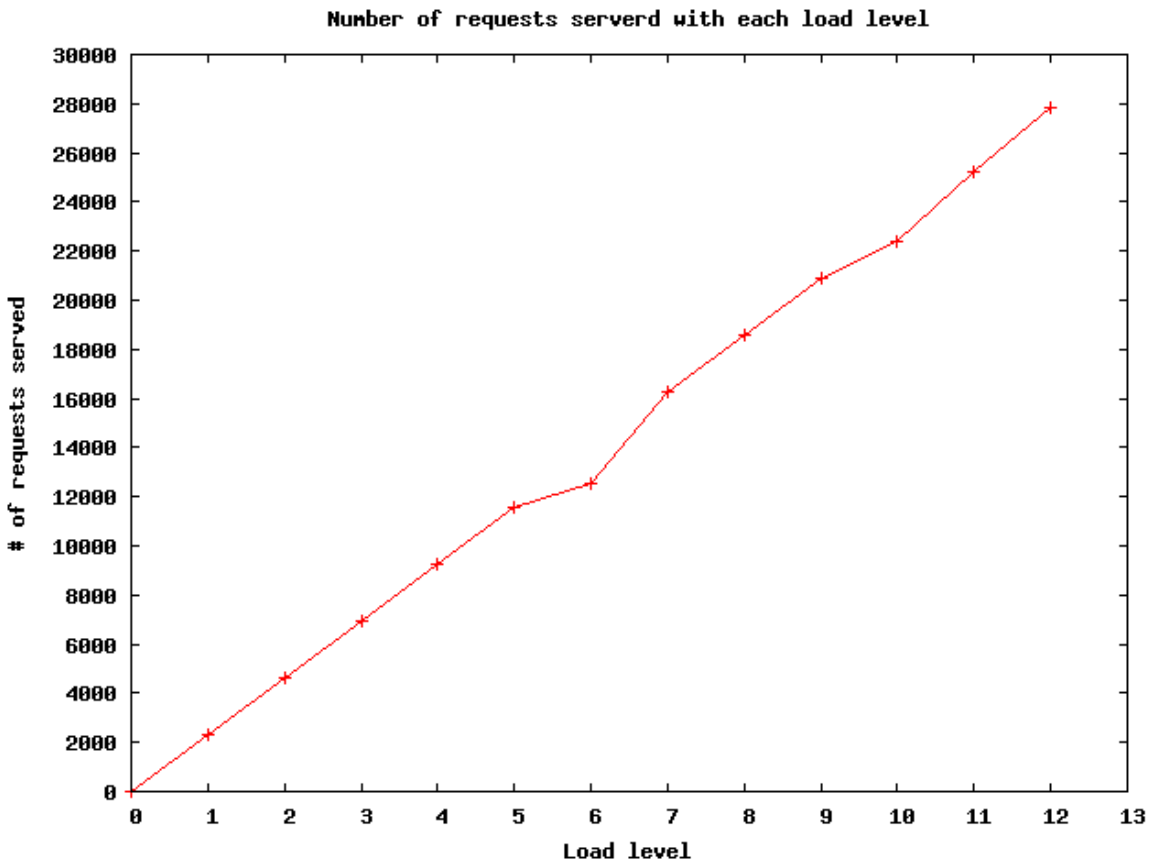

Figure 4.7: Number of requests served by system during Experiment 2. We observe linear growth in the number of requests served by system with each load level.

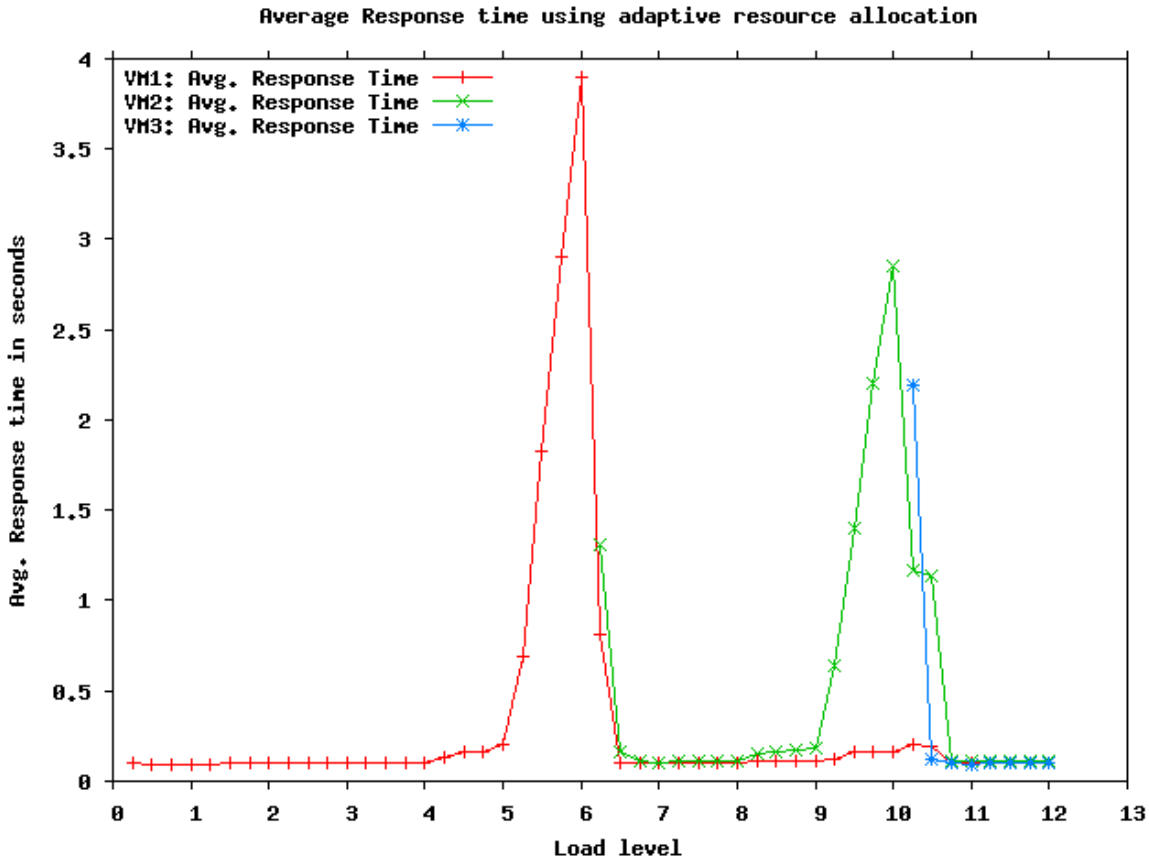

Figure 4.8: Average response time for each virtual machine during Experiment 2. Whenever the system detects a violation of the response time requirement from any virtual node, it dynamically invokes another virtual machine and adds it to the Web farm.

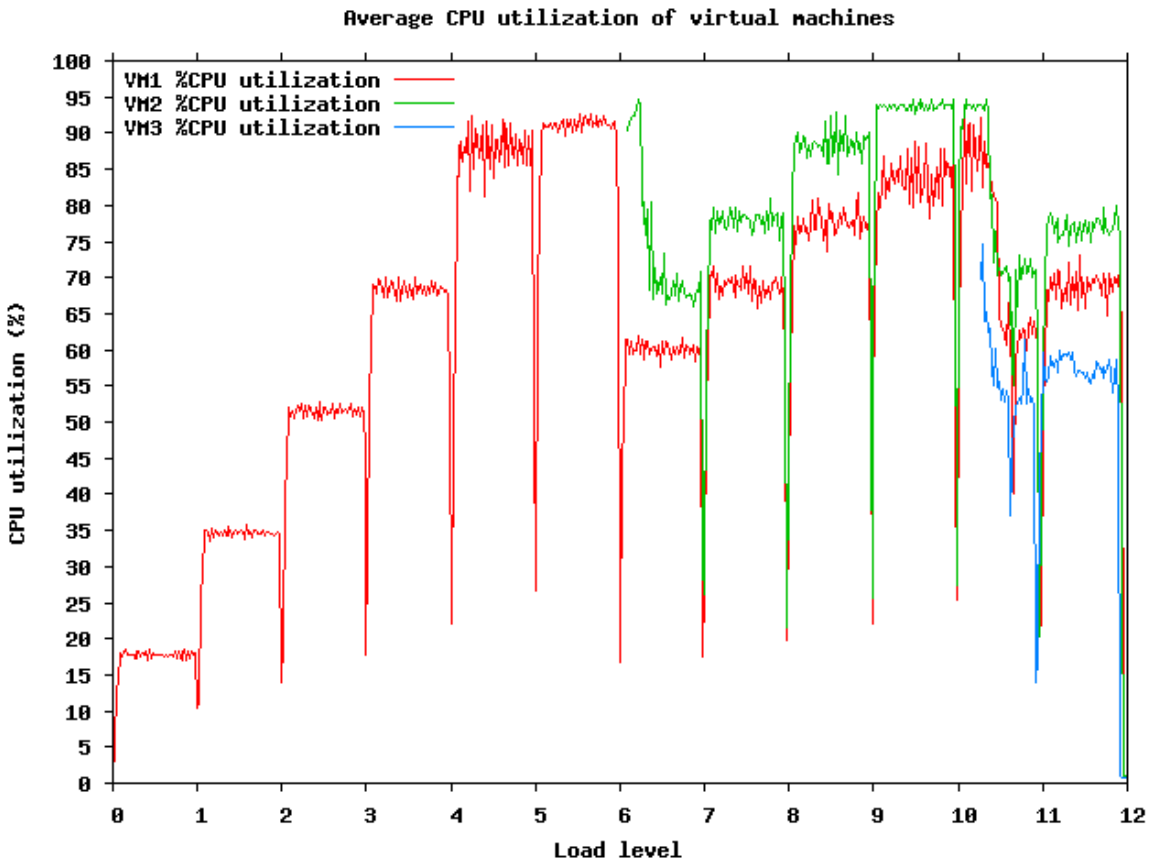

Figure 4.9: CPU utilization of virtual machines during Experiment 2. The duration of each load level is four minutes or 240 seconds, so each tick on the X-axis represents an increment in load level. After load level 6 (1440 seconds), VM2 is adaptively added to the Web farm to satisfy the response time requirement. After load level 10 (2400 seconds), VM3 is adaptively added to the Web farm. Different load levels for different VMs reflect the use of round robin balancing and differing processor speeds for the physical nodes.

Chapter 5

Conclusion and Recommendations

5.1 Overview

Cloud computing is an emerging topic in the field of parallel and distributed computing and requires a huge contribution from the research community. This research study explored cloud computing and identified the advantage of adaptive resource management in heterogeneous clouds for Web applications. I developed a system on top of A EUCALYPTUS cloud that adaptively grows to satisfy specific response time requirements for Web applications.

5.2 Contribution

This research identified a new kind of Service Level Agreement (SLA) for heterogeneous compute clouds in which the consumer wants specific response time for his or her Web application. I developed a system on top of EUCALYPTUS that actively monitors the response time of a Web application hosted on multiple virtual machines and adaptively manages the Web application's resources. Two experiments demonstrated the advantages of the proposed system. VLBManager and VLBCoordinator might be useful for the cloud computing community.

5.3 Recommendations

The CPU is typically the bottleneck in dynamic content generation and I observed that whenever response time starts to grow, the CPU usage exceeds above 90%. I recommend investigating system behavior by adaptively invoking other virtual machines by actively monitoring this behavior. To help overcome the virtual machine boot-up latency problem it would be useful to predict VM response times in advance then invoke additional virtual machines before the system exceeds the response time limit. It should not be necessary to check each VM response time if load balancing is fair instead of round robin. We would only need to check aggregated response time.

I also recommend moving VLBManager and Ngnix to a virtual machine and naming this VM as the head node. Whenever a user needs to deploy a Web application on cloud, the provider needs to deploy two virtual machines, one for the Web application and one for the head node to actively monitor the response time of the Web application hosted on the virtual machine. The head node can detect violations of response time requirements and adaptively deploy more virtual machines to satisfy the response time requirements. Figure 5.1 shows the recommended architecture of the system.

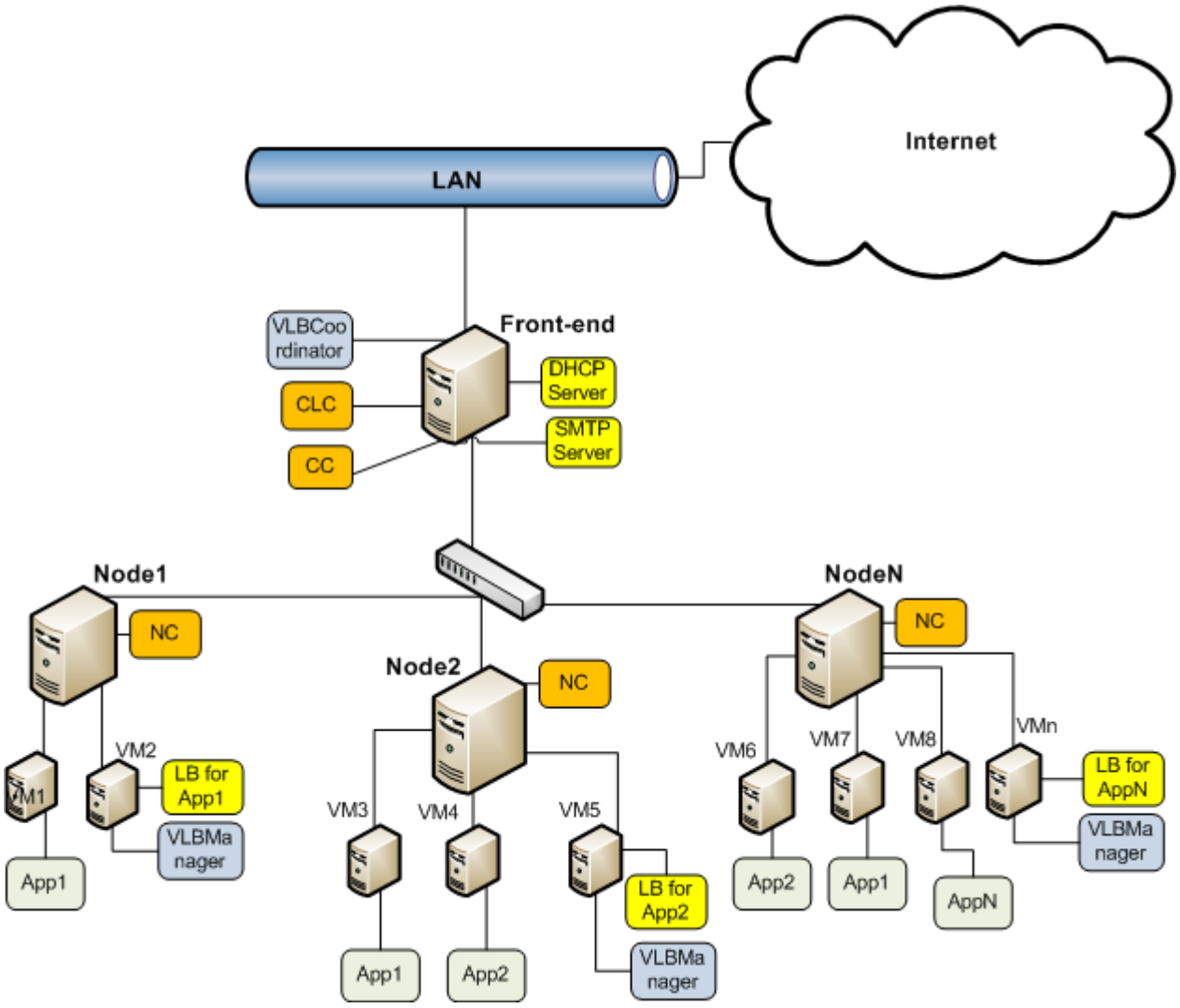

Figure 5.1: Recommended system architecture for multiple Web applications. Each Web application consists of at least two virtual machines. One virtual machine is the load balancer (head-node) and uses VLBManager to adaptively manage resources for the target Web application. The second VM contains the Web application.

References

Amazon.com, Inc. (2009). *Amazon Elastic Compute Cloud (EC2).* (Available at `http://aws.amazon.com/ec2/` [Online; accessed 1-March-2009])

Avi, K., Yaniv, K., Dor, L., Uri, L., & Anthony, L. (2007). KVM: The Linux Virtual Machine Monitor. In *Ottawa Linux Symposium* (pp. 225–230).

Buyya, R., Yeo, C. S., & Venugopal, S. (2008). Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications* (pp. 5–13). Washington, DC, USA: IEEE Computer Society.

Challenger, J. R., Dantzig, P., Iyengar, A., Member, S., Squillante, M. S., & Zhang, L. (2004). Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Transactions on Networking*, *12*, 233–246.

Christine, M., Jérôme, G., Yvon, J., & Pierre, R. (2009). *Clouds, a new playground for the xtreemos grid operating system* (Tech. Rep. No. RR-6824). INRIA.

David, I., Jeffrey, C., Laura, G., Aydan, Y., David, B., & G., Y. K. (2006). Sharing networked resources with brokered leases. In *ATEC '06: Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (pp. 18–18). Berkeley, CA, USA: USENIX Association.

Emerson, L., Paddy, N., & Simon, D. (2008). A fine-grained model for adaptive on-demand provisioning of CPU shares in data centers. In *IWSOS '08: Proceedings of the 3rd International Workshop on Self-Organizing Systems* (pp. 97–108). Berlin, Heidelberg: Springer-Verlag.

Geelan, J. (2009). *Twenty-one experts define cloud computing.* (Available at `http://cloudcomputing.sys-con.com/node/612375/`)

Globus Alliance. (2008). *Nimbus.* (Available at `http://http://workspace.globus.org//` [Online; accessed 06-March-2009])

Google Code. (2008). *Typica: A Java client library for a variety of Amazon Web Services.* (Available at `http://http://workspace.globus.org//` [Online; accessed 06-March-2009])

Google Inc. (2008). *What Is Google App Engine?* (Available at `http://code.google.com/appengine/docs/whatisgoogleappengine.html` [Online; accessed 1-March-2009])

Guitart, J., Carrera, D., Beltran, V., Torres, J., & Ayguadé, E. (2008). Dynamic CPU provisioning for self-managed secure Web applications in SMP hosting platforms. *Computer Network*, *52*(7), 1390–1409.

Jon, W. (2008). VirtualBox: Bits and bytes masquerading as machines. *Linux Journal*, *2008*(166), 1.

Kalyvianaki, E., Charalambous, T., & Hand, S. (2008). Applying Kalman filters to dynamic resource provisioning of virtualized server applications. In *FeBid '08: Proceedings of the 3rd International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks.* Edinburgh, Scotland.

Nieh, J., & Leonard, O. C. (2000). Examining VMWare. *Dr. Dobb's Journal.*

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., et al. (2008).

Eucalyptus: A technical report on an elastic utility computing archietcture linking your programs to useful systems (Tech. Rep. No. 2008-10). UCSB Computer Science.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., et al. (2008). The EUCALYPTUS Open-source Cloud-computing System. In *CCA'08: Proceedings of the Cloud Computing and Its Applications Workshop.* Chicago, IL, USA.

OpenNebula.org. (2008). *The engine for data center virtualization and cloud solutions.* (Available at `http://www.opennebula.org/` [Online; accessed 06-March-2009])

Paul, B., Boris, D., Keir, F., Steven, H., Tim, H., Alex, H., et al. (2003). Xen and the art of virtualization. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (pp. 164–177). New York, NY, USA: ACM.

Piyush, S., Azbayar, D., Pradeep, G., David, I., Laura, G., Aydan, Y., et al. (2007). Automated and on-demand provisioning of virtual machines for database applications. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (pp. 1079–1081). New York, NY, USA: ACM.

Piyush, S., Shivnath, B., & Jeff, C. (2006). Active and accelerated learning of cost models for optimizing scientific applications. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases* (pp. 535–546). VLDB Endowment.

Ranch, D. A. (1999). *Setting up ip masquerade.* (Availalbe at `http://www.linux-mag.com/id/282` [Online; accessed 2-March-2009])

RESERVOIR Team. (2008). Resources and Services Virtualization without Barriers (RESERVOIR) Whitepaper. *23rd Open Grid Forum (OGF32).*

Sims, K. (2007). *IBM introduces ready-to-use cloud computing.* (Available at `http://www-03.ibm.com/press/us/en/pressrelease/22613.wss` [Online; accessed 1-March-2009])

Sun Microsystems. (2008). *Sun Grid utility computing.* (Available at `http://www.sun.com/service/sungrid/index.jsp` [Online; accessed 20-October-2008])

VMWare. (2007). *Understanding full virtualization, paravirtualization, and hardware assist.* Whitepaper.

Wayner, P. (2008). *Cloud versus cloud: A guided tour of Amazon, Google, AppNexus, and GoGrid.* (Available at `http://www.infoworld.com/infoworld/article/08/07/21/30TC-cloud-reviews_1.html` [Online; accessed 1-March-2009])

Wikipedia. (2009a). *Amazon Elastic Compute Cloud.* (Available at `http://en.wikipedia.org/w/index.php?title=Amazon_Elastic_Compute_Cloud&oldid=272762076` [Online; accessed 1-March-2009])

Wikipedia. (2009b). *Google App Engine.* (Available at `http://en.wikipedia.org/w/index.php?title=Google_App_Engine&oldid=273555538` [Online; accessed 3-March-2009])

Wikipedia. (2009c). *Hyper-V.* (Availalbe at `http://en.wikipedia.org/w/index.php?title=Hyper-V&oldid=271655821` [Online; accessed 1-March-2009])

XiaoYing, W., DongJun, L., Gang, W., Xing, F., Meng, Y., Ying, C., et al. (2007). Appliance-based autonomic provisioning framework for virtualized outsourcing data center. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing* (p. 29). Washington, DC, USA: IEEE Computer Society.

Appendix A

Sample Web Application

Web application servlet code.

```
import java.util.Enumeration;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class cars extends HttpServlet {


  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {

long in_time = System.currentTimeMillis();

response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";

long baseline_time = 0;
long iters = 0;

Enumeration e = request.getParameterNames();
while (e.hasMoreElements()) {
String name = (String)e.nextElement();
String value = request.getParameter(name);

if(name.equalsIgnoreCase("baseline"))
baseline_time = Long.parseLong(value);
if(name.equalsIgnoreCase("iters"))
iters = Long.parseLong(value);
}
long spent_time = 0;
while(spent_time < baseline_time) {


spent_time += loop((int)iters);
try{
Thread.currentThread().sleep(20);
spent_time +=20;
```

```
} catch (Exception x) {};
}

long out_time = System.currentTimeMillis();
out.println(docType +
                "<HTML>\n" +
                "<BODY>\n" +
"Baseline time: "+(baseline_time)+"\n" +
"iters: "+iters+"\n" +
                "Spent time: "+(out_time - in_time)+"\n" +
                "</BODY></HTML>");
  }

  private long loop(int limit) {

long array[][] = new long[limit][limit];
for(int i = 0; i < limit; i++)
for(int j = 0; j < limit; j++)
for(int k = 0; k < limit; k++)
array[i][j] = (array[i][j] * array[j][k] + array[k][j]) / (array[k][j]+1);

return 10;

  }
}
```