# THUNDER: A PRIVATE CLOUD ARCHITECTURE DESIGNED FOR HIGH USABILITY

by

GABRIEL JACOB LOEWEN

SUSAN VRBSKY, COMMITTEE CHAIR
MONICA ANDERSON
JOHN LUSTH
ASHRAF SAAD
JINGYUAN ZHANG

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2015

ABSTRACT

Cloud computing is a technological strategy for saving time, money, and resources within an organization. Underfunded and understaffed organizations benefit the most from a cloud architecture because it can help to alleviate a cost burden allowing funds to be used more effectively. Therefore, we believe that non-profit organizations, such as schools, libraries, non-profit medical facilities, and others have the most to gain from cloud computing. Cloud computing has played a major role in shaping large for-profit businesses like Google, Amazon, and Microsoft. Research has suggested that cultural barriers make it difficult for professionals in non-profits to adopt cloud computing technology.

One key challenge faced by organizations for which a cloud architecture would be beneficial is the deployment and management process. In order for private cloud computing to become a viable solution for struggling organizations, much work needs to be done to simplify and improve the deployment process. We describe a new cloud architecture called THUNDER, which is a recursive backronym meaning "THUNDER Helps Underfunded Nonprofits Distribute Electronic Resources."

THUNDER introduces strategies which are meant to help struggling organizations to decrease costs. Virtual machine load balancing attempts to distribute the load across multiple nodes in order to maximize potential performance of virtual machines. Virtual machine consolidation attempts to utilize as few computational resources as possible in order to maximize the potential cost savings of maintaining idle nodes.

We present an evaluation of THUNDER using metrics designed to compare its ease of use with that of other architectures. We also present an empirical evaluation of THUNDER by which users deploy and use the architecture, and then participate in a survey. THUNDER utilizes a new load balancing algorithm, which we have called "RAIN", meaning "Rating Assisted Instantiation Negotiation". RAIN attempts to optimize virtual machine instance placement by choosing compute nodes that have the best potential based on real-time metrics aggregation and a rating algorithm. Results show that the THUNDER is preferable for inexperienced users when compared to that of OpenStack and Eucalyptus. Additionally, experimental results show that RAIN is more efficient at placing virtual machines than the more typical approach of round-robin.

## DEDICATION

To Christopher, thank you for putting up with me during this process. To my parents Arbra and Charles, my brothers Soluman and Elijah, my nieces Amalie and Evelyn, and my aunt Mary. Thank you all for the love and support, especially during the last four years which I have dedicated to my graduate studies. To the memory of my uncle Rick and cousin Peter, I miss you both very much.

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

— Alan Turing, *Father of Computer Science*

"If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility."

— John McCarthy, *Father of Cloud Computing*

LIST OF ABBREVIATIONS AND SYMBOLS

| | |
|---|---|
| API | Application Programming Interface |
| ARPANET | Advanced Research Projects Agency Network |
| AWS | Amazon Web Services |
| $C_{cloud}(n)$ | Cost of THUNDER with Respect to $n$ Clients |
| $C_{comp}$ | Cost of the THUNDER Compute Node |
| $C_{image}$ | Cost of the THUNDER Image Repository |
| $C_{head}$ | Cost of the THUNDER Head Node |
| $C_{store}$ | Cost of the THUNDER Storage Node |
| $C_{client}$ | Cost of a Thin Client Device |
| $C_{disp}$ | Cost of a Display |
| $C_{kbm}$ | Cost of a Keyboard and Mouse |
| $C_{traditional}(n)$ | Cost of a Traditional Network of Workstations with Respect to $n$ Clients |
| $C_{ws}$ | Cost of One Desktop Workstation |
| CPU | Central Processing Unit |
| DHCP | Dynamic Host Control Protocol |
| EBS | Elastic Block Storage Controller |
| EC2 | Elastic Compute Cloud |
| Eucalyptus | Elastic Utility Computing Architecture Linking Your Programs to Useful Systems |
| GB | Gigabyte |

| | |
|---|---|
| Gbps | Gigabits per second |
| GUI | Graphical User Interface |
| HDD | Hard Disk Drive |
| HTML | Hyper-text Markup Language |
| IaaS | Infrastructure as a Service |
| ID | Identity |
| IP | Internet Protocol |
| IT | Information Technology |
| KVM | Kernel-based Virtual Machine |
| LAN | Local Area Network |
| Lab | Laboratory |
| MB/s | Megabytes per second |
| Mbps | Megabits per second |
| MySQL | My Structured Query Language |
| NAS | Network Attached Storage |
| NAT | Network Address Translation |
| NC | Node Controller |
| NFS | Network File System |
| NIST | National Institute of Standards and Technology |
| PaaS | Platform as a Service |
| PC | Personal Computer |
| PHP | PHP: Hypertext Preprocessor |

| | |
|---|---|
| QOE | Quality of Experience |
| QOS | Quality of Service |
| RAM | Random Access Memory |
| $Rasp_{CPU}$ | CPU Utilization of a Raspberry Pi Thin Client Device |
| $Rasp_{Throughput}$ | Network Throughput of a Raspberry Pi Thin Client Device |
| RDP | Remote Desktop Protocol |
| RSA | Rivest, Shamir, Adleman (encryption algorithm) |
| S3 | Simple Storage Service |
| SaaS | Software as a Service |
| SDK | Software Development Kit |
| SOAP | Simple Object Access Protocol |
| SSD | Solid State Disk |
| SSH | Secure Shell |
| SSL | Secure Sockets Layer |
| TFTP | Trivial File Transfer Protocol |
| USD | United States Dollar |
| VM | Virtual Machine |
| W | Watt |
| WS3 | Walrus Storage Controller |
| XML | Extensible Markup Language |

ACKNOWLEDGMENTS

First and foremost I would like to acknowledge my research adviser, Dr. Susan Vrbsky. I am very thankful for her involvement in my academic and professional training. She has helped to guide me in my research and has given me a lot of insight into the nature of life in academia. As her student I have been given the opportunity to conduct original research in cloud computing, as well as the privilege to travel to conferences all around the country to present my work. Dr. Vrbsky also showed me a level of generosity and compassion that is far beyond my expectations. Whether it is the simple gesture of buying me lunch, or letting me borrow her van to run an errand, Dr. Vrbsky's generosity has never wavered. I very much hope to remain in contact with Dr. Vrbsky now that I am *hopefully* moving on to new and exciting things. Thank you for everything that you have done for me over the past several years.

To Dr. Nicholas Kraft, who was my first contact when arriving at the University of Alabama. Thank you for making sure that I found a place to live after I arrived in Tuscaloosa. Additionally, I would like to thank you for keeping in touch with me periodically over the years to check on my progress.

To Dr. John Lusth, my teaching adviser. I would like to show my gratitude to you for supporting me as I have strived to become a better teacher. I have acquired a lot of skills while teaching the Introduction to Programming course under your guidance. I am happy to be able to take these skills with me into any future teaching endeavors.

To my other committee members, I appreciate your cooperation and support during my time as a Ph.D. student.

To Kathy DeGraw and Jamie Thompson, thank you for everything that you put up with on a daily basis. You both do so much to support the computer science department, but rarely get acknowledged. Keep on doing what you are doing!

I am also privileged to be associated with the current and former cloud and cluster computing group members at UA: Michael Galloway, Jeffrey Robinson, Ashfakul Islam, and Kazi Zunnurhain. I really enjoyed working with this group and will always remember my time in this lab.

To Dr. Ferosh Jacob, thank you for your discussions about computer science and software engineering topics, and approaches we should make in relaying these topics to students in the classes we teach at UA.

Finally, I would like to thank the Department of Computer Science at UA for the financial support. Thank you for giving me the opportunity to pursue my Ph.D. in the area of cloud computing and for making it possible for me to take my professional and academic career to places that seemed impossible only a few years ago.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

# Chapter 1

# INTRODUCTION

Many view cloud computing as nothing more than a buzzword, giving rise to much speculation as to the precise definition of cloud computing. In fact, there is no formal definition of cloud computing, as it is typically defined by its usage within a particular product or service. There has been some effort in formalizing a definition for cloud computing [Grandison, Maximilien, Thorpe, and Alba, 2010], however there has not been widespread adoption of any formal definition. We will define cloud computing as a set of service-oriented architectures, which allow users to access a number of resources in a way that is elastic, scalable, on-demand, and cost efficient. Elasticity and scalability are closely related, but often misunderstood concepts. Elasticity is the ability of the physical resources of the architecture to scale transparently by means of adding or removing hardware. Scalability refers to the ability of virtualized resources to expand as resource demands increase and contract as resource demands decrease. Hardware virtualization is also a hallmark of cloud computing. Virtualization is the process by which the physical hardware may be shared by several concurrent operating systems running on top of a hypervisor, or virtual machine management layer. Within the general-purpose cloud computing stack there are three architectures, each with a specific set of intended purposes and target audiences.

Infrastructure-as-a-Service, or IaaS, is the lowest level cloud architecture. It is designed for provisioning of hardware resources, such as virtual machines, and storage devices. We typically see IaaS architectures used in organizations which may not be able to afford to purchase ade-

quate physical hardware and must rely on public third party vendors to provide access to services. However, we also see IaaS architectures used in private clouds which are local to an organization. Private IaaS clouds are typically deployed by organizations that require more flexibility in how data is managed, such as the case of the health information management industry. Built upon the IaaS architecture is Platform-as-a-Service, or PaaS. Usage of a PaaS architecture really varies depending on the intentions of the user or organization. However, typically a PaaS architecture is used by a developer or a development firm with the intent of utilizing cloud services within an application. At the highest level of the cloud computing stack is Software-as-a-Service, or SaaS. In recent years we have seen a large influx of new software services being deployed by public cloud vendors. Such software services include Google Gmail, Salesforce POS, and Dropbox.

Given that cloud computing is defined by the three aforementioned architectures available in the typical cloud stack, there are general purpose, open source, cloud implementations that users may deploy in order to construct a private cloud. Eucalyptus is an open source cloud implementation that is compatible with the Amazon Elastic Compute Cloud (EC2) [Amazon Web Services, inc., 2013]. Additionally, architectures which are not strictly based on EC2 are available, such as OpenStack [OpenStack Foundation, 2013a], CloudStack [Apache Foundation, 2013], and OpenNebula [OpenNebula, 2013]. However, these solutions are designed to be general-purpose, which allows for a very broad set of uses, and configurations. Interfacing with a general-purpose cloud implementation typically relies heavily on command line tools, as well as having to edit text-based configuration files. This methodology is sufficient for general use; however, it becomes complicated for users with limited technical experience. Limitations in technical experience is one of the many reasons why general-purpose cloud computing is not always suitable for underfunded or understaffed organizations. This is especially true for the public education industry, where avail-

ability of funding has a direct impact on the ability of the school or district to hire or train qualified personnel.

## 1.1 Vertical Architectures

Vertical cloud architectures are relatively new within the scope of the cloud computing research community. A vertical cloud architecture is defined by a specific market, each with their own specifications and organizational requirements. At the moment there are very few vertical cloud architectures, with the majority of them focused on the field of medical informatics and health management. The authors of [Delgado, 2011] discuss healthcare IT, and whether or not the healthcare industry is ready to endorse a cloud-based architecture for managing patient records. Although there is progress towards a widespread e-heath cloud, there is far less widespread usage of vertical cloud architectures for other markets, such as education.

Education, much like healthcare, is a fundamental component of modern society. One possible avenue leading towards a more robust and complete education system is the use of technologies in the classroom, such as mobile devices and thin clients. These technologies are highly capable of utilizing remote resources in the cloud, but there are challenges with moving towards cloud services in education, which differ greatly from the challenges encountered in moving health records to the cloud. Where health professionals are apprehensive with regard to the security of an e-health cloud, educators seem to be apprehensive with regard to the applicability of cloud computing. Additionally, it is common for educators to adapt slowly to changes in educational technology due to a number of factors, including fear, lack of monetary support, and lack of professional development. There is much research within the education research community on how technology may be used to improve the quality of education provided to young people across the world [Alabbadi, 2011] [Banister, 2010] [Ercan, 2010]. Unfortunately, there has not been significant research

into how cloud computing may be nicely integrated into a learning environment in a way that is easy to use, unobtrusive, and natural for all stakeholders. In addition to the education market, an easy to use, easy to deploy, and easy to manage private cloud architecture is highly desirable in other non-profit organizations. This is due to the fact that many organizations share the challenges seen in public educational institutions with regard to budget cuts, and the lack of qualified staff.

## 1.2 Motivation

When deciding to use a cloud provider, organizations have very few choices. An organization may choose to hire a public cloud vendor, such as Amazon to maintain their data and services. However, many organizations are skeptical of public cloud vendors due to issues regarding security and privacy. Therefore, it may be advantageous for some organizations to deploy a local private cloud and manage it themselves. In order to deploy a general purpose private cloud, such as Eucalyptus or OpenStack, the following steps are typically required:

1. Procure the necessary equipment, including computers, and networking hardware.
2. Create private network between cloud servers.
3. Install a host operating system onto each cloud server.
4. For each cloud server, install the appropriate cloud service.
5. For each cloud server, generate an SSH RSA key.
6. For each cloud server, share the RSA key with every other cloud server.
7. For each cloud server, start the appropriate cloud service.

8. Establish a virtual private overlay network on each cloud server using a virtual network bridge adapter.

9. For each non-cloud controller service, register that server with the cloud controller.

These steps can cause significant burden upon the individuals deploying the cloud architecture. This burden can not only reduce the effectiveness of the staff at a particular organization, but it can also produce an excessive expense if the organization has to hire specially trained employees to maintain and manage the private cloud. The motivation towards reducing the complexity of deploying, maintaining, and managing a private cloud architecture is towards reducing excess costs within an organization. Additionally, we would like to give organizations the tools to potentially deploy and manage a private cloud without needing to hire additional IT staff.

## 1.3 Eucalyptus Methodology

The methodology for management of resources in Eucalyptus is predominantly reliant upon establishing a control structure between nodes, such that one cluster is managed by one second-tier controller, which is managed by a centralized cloud controller. In the case of Eucalyptus, there are five controller types: cloud controller, cluster controller (CC), block-based storage controller (EBS), bucket-based storage controller (S3), and node controller (NC). The cloud controller is responsible for managing attributes of the cloud, such as the registration of controllers, access control management, as well as the facilitation of user interaction through the use of the command-line and in some cases web-based interfacing. The cluster controller is responsible for managing a cluster of node controllers which entails transmission of control messages for instantiation of virtual machine images and other necessities required for compute nodes. Block-based storage controllers provide an abstract interface for the creation of storage blocks, which are dynamically

allocated virtual storage devices that can be utilized as persistent storage. Bucket-based storage controllers are not allocated as block-level devices, but instead are treated as containers by which files, namely virtual machine images may be stored. Node controllers are responsible for hosting virtual machine instances and for facilitating remote access via RDP [Surhone, Timpledon, and Marseken, 2010], SSH [Barrett, Silverman, and Byrnes, 2005], VNC [Richardson, 2010], and other remote access protocols.

## 1.4 OpenStack Methodology

Similar to the methodology used by Eucalyptus, OpenStack maintains a control structure based on the elements present in the Amazon EC2 cloud. However, there is a difference in terminology between Eucalyptus and OpenStack. OpenStack consists of many "components," whereas Eucalyptus consist of many "controllers." The usage of these terms is consistent between the architectures and their meanings are taken to be the same. OpenStack maintains five components: compute component (Nova), object-level storage (Swift), block-level storage (Cinder), networking component (Quantum), and dashboard (Horizon). There are many parallels between the components of OpenStack and the controllers of Eucalyptus. The Nova component of OpenStack is similar to the node controller of Eucalyptus. Similarly we see parallels between Swift in OpenStack with the bucket-based controller in Eucalyptus, and Cinder in Openstack with the block-based storage of Eucalyptus. There seems to be a discrepancy in implementation between the highest level component in each architecture. OpenStack maintains a different component for interfacing and network management, while Eucalyptus maintains a single cloud controller, combining these functionalities. Additionally, OpenStack does not maintain a higher-level control structure for managing compute components, which is a deviation from the cluster controller mechanism present in Eucalyptus.

## 1.5 THUNDER

Based on the needs of non profit organizations, such as public educational institutions, we propose a new cloud architecture, called THUNDER. THUNDER is a recursive backronym, which stands for "THUNDER Helps Underfunded Nonprofits Distribute Electronic Resources." The goal of this architecture is to decrease the complexity of deploying, using, and maintaining a private cloud compared to current general purpose cloud architectures, such as Eucalyptus and OpenStack [Eucalyptus Systems, 2013b; OpenStack Foundation, 2013a]. Specifically, THUNDER represents a completely new architecture, which simplifies the process by which users may interface with cloud resources, such as virtual machines and persistent storage devices. The methodology by which THUNDER reduces system complexity is twofold. Firstly, interfacing with resources is wholly web-based, which removes the necessity to interface with resources via command line environment. Secondly, deployment of THUNDER utilizes an automated solution, such that configuration of the cloud resources during system deployment requires no user intervention. Although THUNDER is not a general-purpose cloud architecture, it is designed to replace the computing infrastructure of an organization by providing access to resources in the cloud using cheap, low power, thin client devices. Therefore, THUNDER is not suitable for the typical uses of a PaaS or SaaS cloud, which could be developed on top of a general purpose cloud solution. For this reason, we consider THUNDER as a vertical cloud architecture, which is intended to make low cost computing accessible to users that do not have any experience in deploying the more complicated general purpose private cloud architectures.

THUNDER is composed of several components that borrow names of various thunder gods from Greek, Vedic, and Norse mythology. The cloud controller, which is named *Zeus* is responsible

for managing and relaying communication between all other components, and hosting a number of services. Zeus maintains a web and database service for launching and managing virtual machine instances and other aspects of the cloud. Additionally, a DHCP server, TFTP server, and PXE server are maintained by Zeus for local networking support. The storage controller, which is named *Indra* is responsible for storing virtual machine images and maintaining storage space for registered users using the network storage service, Samba [Samba team, 2013]. The compute controller, which is named *Thor* is responsible for hosting virtual machine instances using a local hypervisor, such as KVM. Each controller is managed by Zeus using a middleware service called ThunderRPC.

ThunderRPC is a middleware service that utilizes an event driven model for sending and receiving data between nodes. Communication between nodes is mitigated by Zeus, such that requests from the user are sent to Zeus, which then relays the message to one or more controller nodes. Messages are in the form of JSON [ECMA International, 2013] objects containing remote procedure names and corresponding arguments. However, some special messages are handled by Zeus directly for administrative purposes. ThunderRPC also serves as a means for aggregating data regarding controller nodes, and for implementing common cloud specific tasks, such as virtual machine load balancing and deployment of new nodes.

By combining the three controller types, coupled with the ThunderRPC middleware service, THUNDER provides all main attributes that define a cloud architecture. These attributes are scalability, elasticity, cost efficiency, and virtualization. We address scalability with the ability to define resource requirements for virtual machine instances, thereby scaling a particular virtual machine up or down based on demand. We address elasticity with the ability of THUNDER to deploy new controller nodes using a turn key solution, thereby allowing hardware to be seamlessly

added or removed from the cloud. We support cost efficiency through virtualization technologies that allow many virtual clients to be hosted on a single server instead of multiple desktop solutions, thereby reducing the overall energy load seen in non-virtualized computing environments. Finally, virtualization is the underlying factor by which THUNDER is able to fully utilize the computational demands of many users.

## 1.6 Challenges

The design and implementation of a new cloud architecture comes with a myriad of challenges. The THUNDER architecture is being designed to decrease complexity and overhead for underfunded organizations. However, with such broad goals it is difficult to determine the correct set of methodologies in order to come to an appropriate conclusion. Developing a automated deployment solution is difficult because there are a number of questions that must be answered before such a solution may be proposed.

1. Is it possible for nodes to be discovered automatically on the local network?
2. How can communication between nodes be established without user intervention?
3. Is there a good solution to distributing the required software to each node automatically?
4. Is there an alternative to RSA key sharing with regard to authentication and security?
5. How does THUNDER differ from general purpose architectures?
6. Can we produce some qualitative or quantitative metrics to demonstrate that THUNDER is better for users of non-technical backgrounds?
7. Are there any other constraints to deployment that have not been uncovered?

These deployment challenges and our solutions are are addressed in this work. Unfortunately, these challenges are not addressed in most general purpose cloud solutions. For example, a multi-node Eucalyptus deployment depends upon an RSA key sharing mechanism, which means that the system administrator must generate RSA keys on every node in the cloud. After the RSA key generation process is complete, the generated keys must be shared with every node in the cloud. Additionally, automatic discovery of nodes in Eucalyptus is possible, but not easily accessible or usable by average users. We would like to address these questions in order to develop an architecture that may easily be deployed by average users, and those with little to no experience in cloud computing.

In this dissertation we introduce solutions for the challenge questions introduced above. The middleware solution used in THUNDER utilizes non-persistent socket connections with a pre-shared key challenge/response authentication mechanism, which allows node-node communication without RSA keysharing. Additionally, we introduce automatic node discovery using a multicast based fault tolerance mechanism, which is maintained by Zeus. Distribution and automated deployment of THUNDER is facilitated by the PXE boot architecture, combined with the ThunderRPC management layer. Additionally, we show qualitative metrics regarding the benefit of using THUNDER over other architectures. These metrics were gathered by empirical analysis through a series of anonymous surveys.

## 1.7 Outline

The following chapters are devoted to discussing the work in the design and development of THUNDER. Chapter 2 discusses the background work and literature review. Following the literature review is a discussion of the THUNDER architecture in Chapter 3. Our message-oriented remote procedure call middleware is presented in Chapter 4. Chapter 5 presents an empirical study

on the deployment process for THUNDER and an analysis of the results of the study. Chapter 6 presents the "RAIN" virtual machine instance placement and load balancing strategy. Chapter 7 discusses future work and goals for the THUNDER architecture. Chapter 8 concludes this dissertation with an overview of the results and final remarks.

# Chapter 2

# BACKGROUND AND LITERATURE REVIEW

The modern notion of "Cloud Computing" is one which has its roots firmly placed in the 1960's. During this era, the idea that computing resources could be shared over a vast network of devices was first envisioned. John McCarthy, to whom many have attributed the title "Father of Cloud Computing," was a pioneer in the early development of grid computing. One of the earliest declarations for computing as a public utility came from John McCarthy during his speech at the MIT Centennial celebration of 1961. McCarthy suggested that the computers of the future could simply "plug in" to a global network in order to access computing services as a utility, much like the way in which electricity and other public utilities are provided to homes. McCarthy envisioned a system that greatly resembled the typical cloud computing architecture that we have today. In fact, it can be argued that cloud computing is the product of the evolution of three system architectures: grid computing, cluster computing, and mainframe computing.

Around the same time that idea of grid computing was being envisioned, the predominant mode of accessing compute resources was in the form of mainframe computing and dumb terminals. The mainframe architecture is designed in such a way that computing resources reside on a centralized system, but are accessible via remote terminals. By connecting many terminals to single mainframe hundreds of users can execute jobs using a time-sharing mechanism. Although the notion of computing as a utility was not a goal of mainframe computing, both grid and mainframe architectures provide a means to access compute resources.

Cluster computing represents a system architecture comprised of loosely coupled computing hardware connected by a shared network bus. The goal of cluster computing is that the resources provided by many computers may be accessed in such a way that the end user perceives only one cohesive system. Historically, cluster computing was used to execute highly parallel jobs on expensive and often times massive clusters. In modern times cluster computing presents an opportunity to provide high performance using loosely coupled hardware that individually may be mediocre.

Throughout the past fifty years these three architectures have evolved, and many conceptual details have been funneled into what we now know as "Cloud Computing." Grid computing presents a vast network of computational resource sharing, and introduces the notion of computing as a utility. Mainframe computing presents computational resources as a centralized and highly accessible infrastructure. Lastly, cluster computing presents the coupling of several computational resources as a means to construct the illusion of a single cohesive system.

In Sections 2.1 to 2.3 we present an architectural overview of the three main architectures that contributed to the eventual design of the modern cloud. Section 2.4 presents an architectural overview of general purpose cloud computing. Finally, in Sections 2.5 and 2.6 we present a literature review of current cloud technologies that help to support cloud services in the public sector.

## 2.1 Grid Computing

Grid computing represents a computing ideology that was initially conceived of in the 1960s, but was not fully realized until the late 1990s and early 2000s. The idea behind grid computing was that computing resources could be distributed in a very loosely coupled and geographically distributed network. The purpose of this vast network, or "grid" of computing devices was to serve as a means to construct an open and available "utility," such that users could simply connect

when and where they choose. The realization and full potential of grid computing came about by work done by Ian Foster and Carl Kesselman [Foster, Kesselman, and Tuecke, 2001; Foster and Kesselman, 2000].

Although, the advent of grid computing spurred the development of new and novel distributed applications, such as Folding@Home[Stanford University, 2011], many issues still remained. The fact that grid computing relies upon the coupling of computing hardware, which in many cases are set apart by hundreds of miles, latency and performance issues are abundant. In keeping true to the words spoken by John McCarthy, utility computing must be able to provide computing resources reliably, in much the same way that electricity and water are provided to homes. It would be unacceptable for water service to suffer from a trickling delay, and the same should be true of a computing utility. For this reason, the goal of grid computing to transcend traditional computing methodologies and become a true public utility has often been viewed as a failure.

## 2.2 Mainframe Computing

Mainframe computing represents an architecture that has not been popular in the last three decades, due to the advent of cheaper and faster hardware which made personal computing much more feasible. The ideology behind mainframe computing is that time critical and bulk processing jobs could be offloaded to a centralized computing device for which the results of the computation could be retrieved at a later time. Batch processing became popular in the 1960's as a means to accept large quantities of data and programs, which were typically provided as paper cards or magnetic tape. Although mainframe computing has been mostly phased out as newer and more sophisticated architectures have been introduced since the late 1990's and early 2000's, it is evi-

dent that some traces of mainframe computing and batch processing can be seen in modern cloud implementations.

## 2.3 Cluster Computing

Cluster computing refers to the general concept of combining multiple computing devices to form larger and more sophisticated logical systems. This technique has allowed cluster computing to form the basis for many supercomputing architectures. The originating idea behind cluster computing was introduced by J.C.R. Licklider [Licklider and Taylor, 1968], who discussed an interconnected computing system such that computing jobs could be submitted and distributed amongst many systems. Licklider proposed his idea to the Intergalactic Computer Network group, which helped in the development of ARPANET in 1969, the precursor to the modern internet. Figure 2.1 illustrates the modern concept of cluster computing with a four node star network topology.

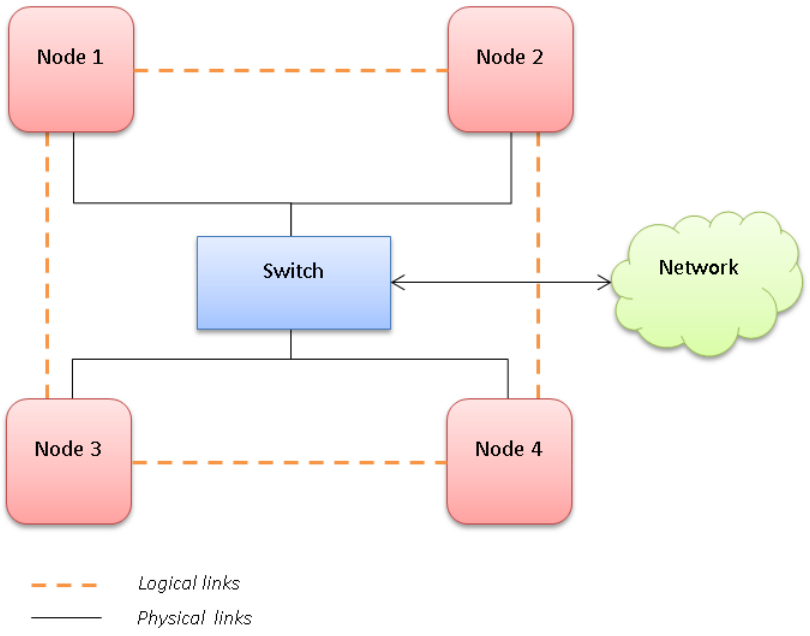



Figure 2.1: Cluster of four computers using the star network topology.

## 2.4 General-Purpose Cloud Computing

According to current literature [Canonical Group Ltd., 2010; Edmonds, Johnston, Metsch, and Mazzaferro, 2010; Liu, Tong, Mao, Bohn, Messina, Badger, and Leaf, 2011; Toby Velte, 2009; Van, Tran, and Menaud, July; Weiss, 2007; Zeng, Zhao, Ou, and Song, 2009; Zhang and Zhou, July] cloud computing is a set of distributed computing architectures which are designed to provide resources to users in a way that is scalable, cost-efficient, and on-demand. This architecture is composed of three general layers. Infrastructure-as-a-Service provides virtual machines and storage devices to users. Virtual machines are provided to users by utilizing a hypervisor, which is essentially a virtual machine manager. Persistent storage is provided at the IaaS layer in a number of different ways, and each architecture may provide storage differently. For example, storage may be provided by means of a local SAN device or by a simple Samba [Samba team, 2013] share. The middle layer of the general purpose cloud stack is the Platform-as-a-Service (PaaS) layer. PaaS typically provides an operating system, an API, and development tools, which allow users to develop software that utilizes cloud resources. The highest layer in the general purpose cloud stack is the Software-as-a-Service (SaaS) layer. SaaS provides complete software solutions to users in the form of web services and standalone applications. Figure 2.2 presents the logical structure of the general purpose cloud stack.

## 2.5 Research in Cloud Computing for Education

Many researchers have suggested that cloud computing is purely a business model, which is focused on increasing profit by reducing costs. However, cloud-computing research in education paints a different picture that is centered on the efficient use of technology in the classroom intended to increase student production and motivation. Researchers in education and educators

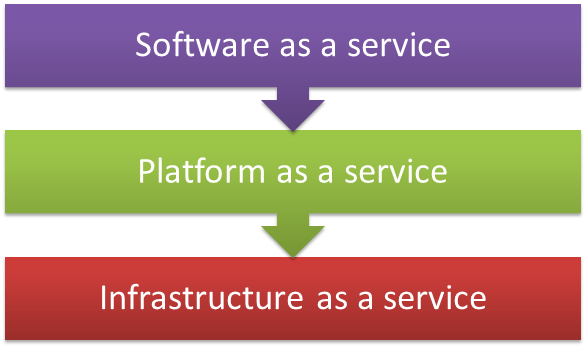



Figure 2.2: Layered architecture of general-purpose cloud computing model.

are focused on providing young students with the tools necessary to be productive members of society [Stein, Ware, Laboy, and Schaffer, 2013]. Nevertheless, there is a need to prioritize efforts based on the requirements of each educational institution, because the needs of one school may not correlate with the needs of another. The idea that each school may have different needs and requirements is interesting from a cloud computing perspective, which is identified by scalability of resources, and reusability and adaptability by different institutions and markets. Cloud computing is also identified as a low cost alternative to general purpose computing by providing on-demand access to resources. The cloud can scale up resource availability based on need, and scale down to save energy. This notion of scalability and on-demand is beneficial to educational institutions, especially schools struggling with reductions in funding. The past decade has seen in some cases a dramatic decrease in state, local, and federal funding for public K-12 educational institutions. This reduction in funding indicates that a paradigm shift in how technology is utilized in the classroom may be necessary in order to maintain the quality of our public education system.

### 2.5.1 Viability of Cloud Computing for Education

The authors of [Sultan, 2010] discuss the viability of cloud computing for education as well as provide some valuable insight from a business perspective. Cloud computing is described

as an immense opportunity, such that an educational establishment may take advantage of the cloud to pull itself out of a funding crisis. Maintaining an IT infrastructure comes with many advantages to an organization. However, these advantages are paired with additional economic burdens associated with maintenance and ongoing costs, such as power and cooling. The authors attempt to explain how cloud computing can benefit educational institutions by describing the cloud paradigm at a high level while showing parallels between cloud computing and current popularly used service-oriented solutions, such as web services.

Based on a study done by the United Kingdom's National Computing Center (NCC) [The National Computing Group, 2015], the use of hosted computing solutions, such as cloud and web-based solutions can drastically reduce the total cost of ownership of technology in the classroom. Additionally, it is revealed by a survey [CBR Online, 2013] of small and medium sized enterprises, as well as educational institutions, that the driving force behind adoption of cloud services is cost saving, and 54% of the respondents of the survey indicated that they expect to adopt cloud services by 2010. Although the survey did not specifically target educational institutions, but captured a broad base consisting of schools as well as businesses, it shows a trend of interest in cloud computing by organizations in the UK. Considering that this survey was executed in 2009 and that new data has not been released since this time, the utilization of cloud services in more recent times by the respondents is unclear based solely on this study. One of the current educational institutions taking advantage of cloud services is Washington State University School of Electrical Engineering and Computer Science (EECS). Washington State University has seen budget cuts, which affect the ability of EECS to purchase and maintain computer equipment. This has led EECS to utilize vSphere 4 as a private IaaS architecture to alleviate the funding cuts.

Many parallels can be made between cloud services and web services. However, not all


Figure 2.3: Intended users for each layer of the cloud stack.

web services are hosted in the cloud, and not all cloud services are web-accessible via a typical web browser. Additionally, each layer of the cloud stack is intended for different types of users. Figure 2.3 illustrates the intended audience for each layer of the cloud stack. One of the concerns noted by the authors of [Sultan, 2010] is that if a cloud is implemented for education then there must be specific uses for each layer of the cloud stack. For example, from Figure 2.3 it is clear that both IaaS and SaaS clouds benefit both students and instructors. However, PaaS clouds seem to only benefit software developers. This is in contradiction to [Doelitzscher, Sulistio, Reich, Kuijs, and Wolf, 2011] which supports the notion of PaaS for students in STEM learning. The authors of [Doelitzscher et al., 2011] illustrate the usage of PaaS by students taking computing courses, where a PaaS architecture allows students to take advantage of an API and services, such as MySQL in the cloud. The authors also discuss green computing in the cloud by citing statistics that show that computing and information systems have contributed 2% or more to the total global carbon emissions in 2010. It is likely that this trend will continue to increase as utilization of computational resources is always increasing worldwide, both by individual consumers as well as organizations. One of the benefits of cloud computing described by the authors is a potential to decrease power consumption and provide a more environmentally friendly infrastructure when compared to traditional paradigms.

Although, it seems like cloud services are ready for educational institutions, there are concerns over privacy and security in the cloud. This concern is definitely legitimate for public cloud vendors, which may host many organizations' data in offsite data centers. However, private cloud architectures are not generally of much concern when it comes to privacy and confidentiality because they are maintained solely by the organization using the architecture. The conclusion of the authors is that more research is needed to determine if privacy concerns can be adequately solved, because privacy seems to be the leading concern among educators and other staff. The authors of [Stein et al., 2013] provide some additional support to cloud computing for education by highlighting the benefits that cloud computing presents to educational institutions using a case study from two rural North Carolina schools.

Even though the authors of [Sultan, 2010] are skeptical of cloud computing in education because of privacy concerns, they are hopeful that developments in cloud computing will improve upon data privacy strategies in the cloud. Privacy is an issue that many cloud researchers are facing and there are solutions that allow for greater privacy in the cloud by utilizing methods of securing data, such as by the use of encryption. However, privacy is not an issue that is specific to education. It is a necessity in many cases to keep certain records secure and private, such as medical records, which are handled by medical facilities as well as educational institutions.

### 2.5.2 Virtual Computer Lab

Researchers at North Carolina State University (NCSU) have developed a cloud architecture which is designed to provide young students with tools that help to engage them in the field of mathematics [Stein et al., 2013]. This cloud architecture, known as "Virtual Computing Lab," or "VCL" has been provided as a public service to rural North Carolina ninth and tenth grade algebra and geometry classes. The virtual computer lab was developed because of concerns over

computer maintenance costs at schools in rural North Carolina as well as a growing concern that students attending these schools were not achieving adequate scores in standardized testing. The idea to introduce cloud computing into the schools was spurred by the belief that an on-demand computational resource, such as the cloud could help to alleviate the pedagogical concerns as well as provide a more cost-efficient computational solution.

The authors of [Stein et al., 2013] have begun to gather data regarding the effectiveness of the VCL for certain math courses offered in two North Carolina schools via an NSF funded case study titled "Scaling Up STEM Learning with the VCL." During this study, algebra and geometry courses at four rural North Carolina schools were selected. Selection of these schools were made by determining which schools are comprised of the most underprivileged and minority students. These students were given access to the VCL in order to help understand what sort of impact the introduction of cloud computing would have. Additionally, the four schools chosen for this study were selected for having a low pass rate in geometry and algebra, as measured by the average final exam scores.

The goal of this study was to broaden the education of STEM related topics using the VCL in these schools, with the desired end result being an increase in the pass rate of the two courses selected. Two applications were selected to be used in the course curriculums: *Geometer's Sketchpad 5*, and *Fathom 2*. Additionally, instructors were given a six-month period of training in these applications. Although the pass rate statistics regarding these courses are not available yet, the authors describe a set of key challenges that were encountered during the study, including: diversity of software, software licensing, security, network availability, life expectancy of hardware, and affordability, as well as technical barriers.

Software availability is a prime concern when it comes to provisioning educational tools

for academic use. The specific needs of the classroom in many cases require pre-selected standardized software packages. When deploying software to a cloud it is not always possible to provide certain applications as cloud services. For this reason, it is common to bundle software with virtual machine images, which are launched on an IaaS cloud. Additionally, software packages may have some conflicts with one another, which can create an issue with the logistics of the system [Stein et al., 2013].

Another software concern is related to software-specific licensing and how it affects the cloud. Many software packages require licensing fees to be paid per user of the system, or a volume license which may or may not impose a maximum number of users allowed access to the software. Therefore, depending on the specific requirements of the school and course, software-licensing fees must be paid for accordingly. For example, when *Geometers Sketchpad* was deployed to the VCL, the authors made sure that the software licensing fees were paid for in accordance with the software publishers' license agreement. The necessity for licensing does affect the cost effectiveness of using a cloud in this setting; however it is no different than licensing software for traditional workstations [Stein et al., 2013].

In addition to technical barriers, the authors of [Stein et al., 2013] discuss cultural barriers as well as the issue of stubbornness when it comes to adaptation of new technologies in the classroom. Not only is technology slow to be adapted in education, but the interest in new technology is sometimes absent. Some educators are more comfortable using outdated equipment, or are forced to do so by superiors due to both fear and ignorance of technological innovations that might be greatly appreciated by students. One of the ways to mitigate this barrier is to educate our educators. For those educators that are wary about adaptation of new technologies in the classroom, there should be training and materials available for them to learn about how these technologies,

notably cloud computing, can benefit the learning experience for students and improve the jobs and lives of educators in the process.

Although the authors expressed design concerns, such as adequately securing private data and software licensing, several benefits were uncovered in the process of developing the VCL architecture. For example, the VCL allows students and faculty to access educational content and material from their homes, without having to be on campus. Additionally, the software available on the VCL is not restricted to on-campus use, which allows users to utilize the software in the cloud at home. Before introduction of the cloud in rural North Carolina schools these features were unavailable, which meant that if a student needed material from the school then the student would have to wait until they could access a computer on campus. This flexibility and high availability makes the VCL desirable for both students and instructors, and helps to establish a more open and easily accessible learning environment.

There has been some criticism of the VCL when it comes to the efficiency of the way in which the VCL manages resources. Data gathered from the VCL between the years 2004 and 2007 were used by the authors of [Tian, 2008] to create a more efficient VCL model. The authors of [Tian, 2008] have proposed to use the Erlang delay model to increase efficiency when users request on-demand resources. This model, although not yet tested on the VCL to verify the authors' claims, is expected to improve the performance of the system by effectively managing requests for resource allocations, as opposed to the first come first serve model currently in effect. The necessity of this improvement is due to the fact that since its induction in 2004, the VCL has experienced a large increase in utilization that spans across several state lines, in addition to its primary use in North Carolina.

Based on the literature it is clear that the VCL is highly successful in high schools, con-

sidering that schools are using it all across the United States. In actuality, the authors of [Sultan, 2010] present the only evidence from the literature of a cloud computing architecture currently being used in middle schools. The utilization of VCL makes it important in the context of this survey, and demonstrates that cloud computing can be successful in educational institutions. However, the case studies presented in [Sultan, 2010] only show a subset of uses for the VCL, namely mathematical software for two rural North Carolina schools. It is unclear if the applicability of the VCL extends further into other subjects and if the architecture is capable of providing other features, such as IaaS and PaaS services.

### 2.5.3 CloudIA Architecture

The authors of [Doelitzscher et al., 2011] have created a private cloud architecture named CloudIA, which supports e-Learning services at every layer of the cloud stack. At the IaaS layer, the CloudIA architecture supports an automated virtual machine image generator, which utilizes a web interface for creating custom virtual machine images with predefined software packages installed. At the PaaS layer, the CloudIA architecture supports computer science students with a robust API for writing software that utilizes cloud services. At the SaaS layer, the CloudIA architecture supports collaborative software for students to utilize for projects and discussion. The CloudIA architecture is composed of several layers, each responsible for a specific attribute of the cloud. The descriptions of each layer are:

- *User interface layer*: Provides access points to users and administrators of the content management system in accessing CloudIA.

- *Business layer*: Regulates resource provisioning and demand through the use of price and

Service-Level Agreements (SLA). The business layer also enables users to reserve virtual machines in advance and manage their virtual machine configurations.

- *System layer*: This layer is responsible for daily operation of the content management system, such as job submission, management of user accounts and monitoring Quality of Service (QoS).

- *Resource interface layer*: This layer deals with the physical hardware. It provides interfaces and plugins to virtualization services, database and distributed systems, as well as other technologies, such as Xen, Amazon EC2, Amazon S3, and node monitoring services, such as Nagios.

- *Monitoring and management component*: To ensure the reliability of each layer in the cloud, a monitoring and management component is needed. This component allows the system administrator to monitor activities at each layer.

- *Security component*: Ensures that each layer is hardened with security components that protect transactions and data from unauthorized access or misuse.

The CloudIA architecture utilizes the aforementioned layers to support educational services. Additionally, the authors describe their experience with using third party educational software. Specifically, authentication concerns with software components prompted the use of a single sign-on solution called Shibboleth, which acts as an intermediary authentication service between the third party software components and the user. Figure 2.4 illustrates the Shibboleth authentication framework within the CloudIA architecture.

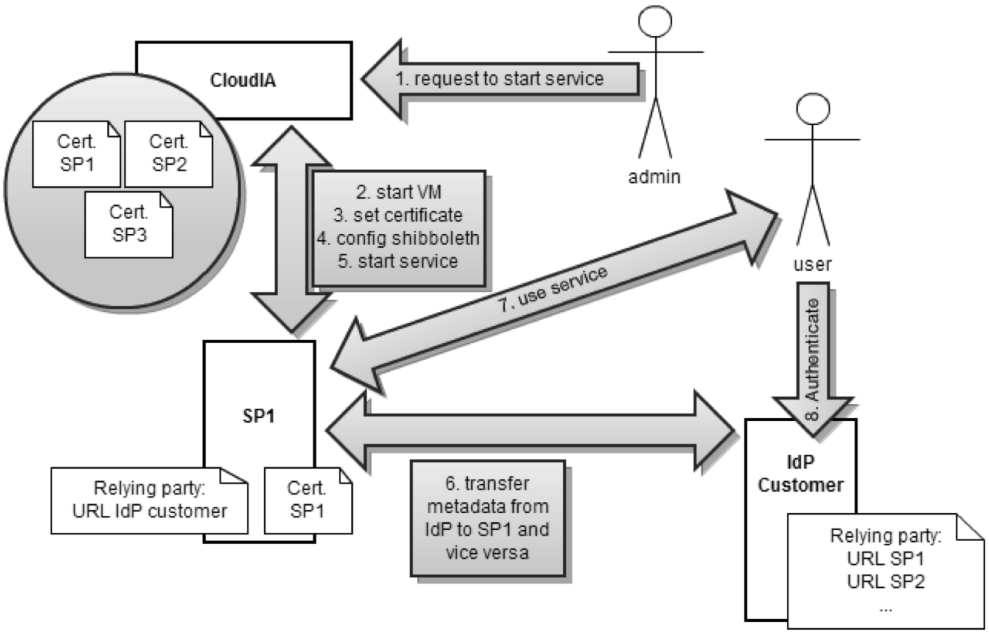



Figure 2.4: Single sign on architecture with Shibboleth.

Since CloudIA offers IaaS, PaaS, and SaaS services, the authors made sure that each layer of their cloud stack contributed to particular attributes of education. At the IaaS layer an automated virtual machine generator constructs custom virtual machines using a base operating system and a selection of predefined packages. By utilizing the cloud to construct virtual machine image,s each student can guarantee that the software a student needs will always be available when they need it, and any superfluous software will not be a distraction to the educational development of a particular student.

CloudIA implements a platform for software development in the cloud, which is called the servlets container platform. The servlets container platform allows CloudIA to offer students a robust API and development tools and resources, such as Apache and MySQL server software. Students that are taking a computer science or related course can utilize the servlets container

platform to take advantage of these resources. Figure 2.5 shows how students may interact with the servlets container platform.

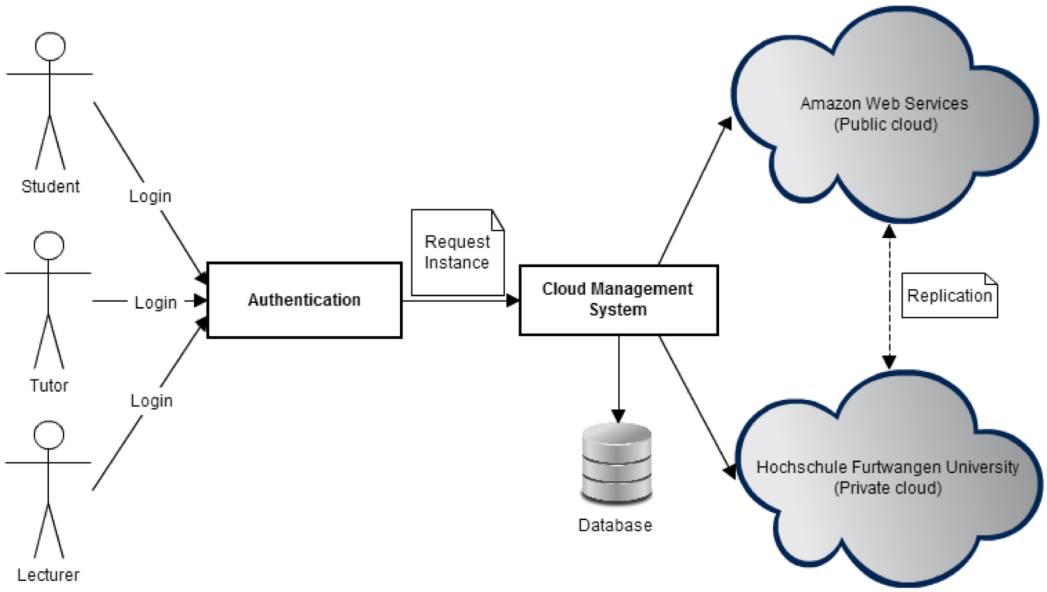



Figure 2.5: Servlet container platform.

In terms of the third party software discussed in [Doelitzscher et al., 2011], the authors focus heavily on CollabSoft, which is a collaboration tool for students and instructors. CollabSoft enables students to participate in discussions, group projects, and testing. Additionally, it allows instructors to post messages and monitor student progress using an easy to use interface, which is reminiscent of BlackBoard. The CloudIA architecture is composed of several freely available third party tools; however, the authors fail to explain how these tools interact with one another to form a cohesive system. The authors present CloudIA as an architecture that has been introduced to university level education at Hochschule Furtwangen University, but do not present any information about the viability of CloudIA at other education levels, such as high schools and middle schools. Although, education at all levels has similar goals, future work in education clouds should address cloud computing in secondary educational institutions.

### 2.5.4 Other Cloud Architectures and Platforms for Education

The authors of [Sultan, 2010] describe the benefits of cloud computing for education. The main point that the authors make is that cloud computing provides a flexible and cost effective way to utilize hardware for improving the way information is presented to students. Additionally, the authors describe details about the ability of cloud computing to shift the traditional expenses from a distributed IT infrastructure model, to a more pay-as-you-go model, where services are paid for based on specific needs.

Authors of [Cappos, Beschastnikh, Krishnamurthy, and Anderson, 2009] discuss "Seattle," which is a cloud application framework and architecture that enables users to interact with the cloud using a robust API. By using this platform students can execute experiments for learning about cloud computing, networking, and other STEM topics. The authors also describe a complimentary programming language, built upon Python, which gives students easy access to the Seattle platform. The authors of [Masud and Huang, 2011] discuss a new model for SaaS, which they have named ESaaS. ESaaS is defined as a Software-as-a-Service cloud architecture with a focus on providing educational resources. The authors discuss the need for a managed digital library, and a global repository for educational content, which is easily accessible through a web interface. The proposed architecture is meant to integrate into existing secondary and post-secondary institutions as a supplementary resource to their existing programs.

The authors of [Baev, Weaver, and Weaver, 2011] describe their experiences in implementing a low cost desktop virtualization solution at Georgia Southwestern State University. This solution only offers access to Windows instances via an RDP protocol and takes advantage of commercial tools for provisioning of resources. The authors describe the implementation but give

little detail about the cost savings obtained by using virtualization services over a traditional non-virtualized computer lab.

The authors of [Behrend, Wiebe, London, and Johnson, 2011] discuss the implementation of cloud architectures for use in community colleges and the authors' experience with deploying cloud architectures for educational purposes. Specifically the authors discuss the acceptance of new technology, namely cloud computing in higher education and, based on surveys, predict that cloud computing will emerge as an important technology in higher education.

Although, these works in cloud computing for education are valuable in describing various experiences in education oriented cloud solutions, many of them focus on very specific uses. For example, the authors of [Cappos et al., 2009] discuss cloud computing in the context of software development with the Seattle PaaS. Although, Seattle seems like a good platform for students in computing, it is not robust enough to be considered a fully featured cloud computing architecture for education, which should also provide virtualization services. Similarly, the authors of [Masud and Huang, 2011] discuss e-Learning software in the cloud, but do not address other aspects of cloud computing for education, such as IaaS and PaaS architectures. The authors of [Baev et al., 2011] and [Behrend et al., 2011] provide an overview of experiences in cloud computing deployments for education, but fail in describing the details of the cloud architecture being deployed. Additionally, considering that one of the biggest motivating factors in moving towards cloud computing is energy conservation, the literature lacks data on energy utilization. The authors of [Behrend et al., 2011], [Cappos et al., 2009], and [Masud and Huang, 2011] do not address energy concerns, while the authors of [Baev et al., 2011] and [Sultan, 2010] discuss energy conservation in little detail.

## 2.6 Useful Technological Strategies for Cloud Deployment

In addition to education-specific technologies, general purpose technologies are also pertinent in this discussion. Although, these technologies are not designed specifically with educational institutions in mind, they are useful in many applications, both general purpose and vertical.

### 2.6.1 Virtualization with KVM

At the heart of every IaaS cloud architecture is a hypervisor, which is responsible for instantiating virtual machine instances and providing an interface between the physical hardware and the virtual machine instance. One of the most popular hypervisors is the Kernel Virtual Machine (KVM), which is primarily used for virtualization of Linux virtual machines. KVM provides a shared kernel interface, which means that the base operating system and the virtualized guest operating system share a common Linux kernel. This kernel sharing mechanism allows the virtual machine image to be scaled down and smaller in size, which decreases the required network bandwidth for virtual machine instantiation. Authors of [Redhat, Inc., 2013]] and [Goto, 2011] discuss the KVM architecture in depth and provide valuable information in its usage. The information presented in [Redhat, Inc., 2013]] is related to educational clouds because design considerations for an educational cloud must take into account virtualization methodology and how these services are implemented.

KVM allows multiple virtual machine instances to run in a sandboxed environment on the same server. Sandboxing requires security privileges at multiple levels in order to ensure that multiple sandboxed environments cannot interact with each other. The operating system has 4 levels of security privileges, which are separated using the notion of rings, where ring 0 represents the most privileged layer and ring 3 represents the least privileged layer. Due to the fact that KVM

must have access to physical hardware, it is located in ring 0. Figure 2.6 illustrates the privilege layers of the host operating system running KVM. Rings 1 and 2 are prevented from executing privileged instructions, but are traditionally not used in modern operating systems.

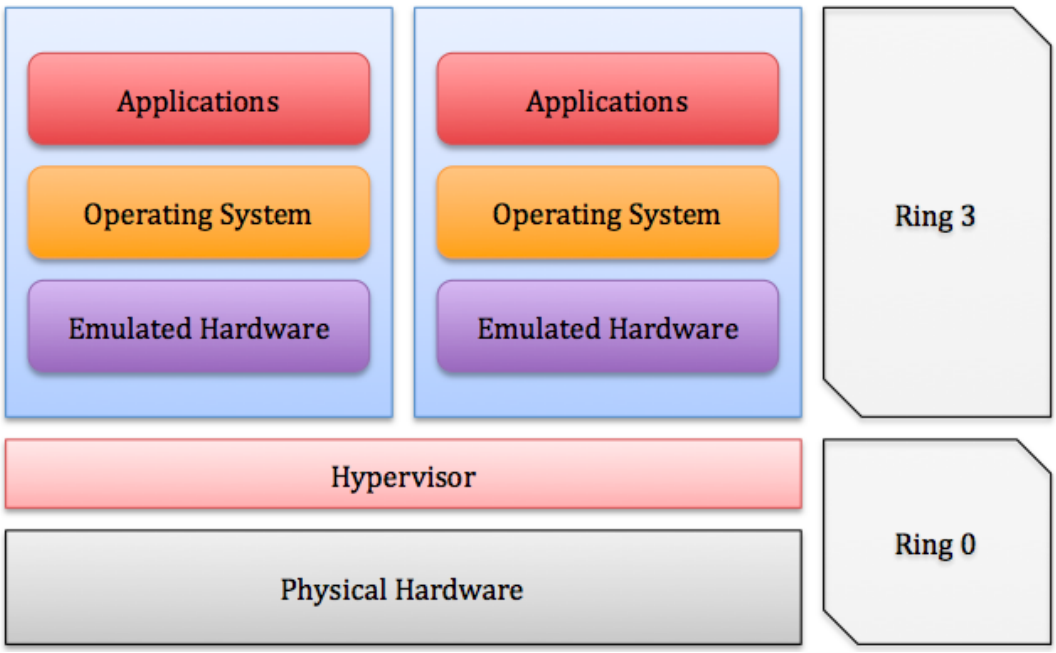



Figure 2.6: KVM hypervisor rings of privilege.

KVM allows two different types of virtualization, full virtualization and paravirtualization. In full virtualization the guest operating system is not aware that it is running in a virtualized environment, and does not need to be modified to support virtualization. Guest operating systems are able to perform privileged instructions in the same way that a non-virtualized operating system would. Paravirtualization is another approach to virtualization in which the guest operating system is modified, by which all privileged instructions are replaced with calls to the hypervisor.

Computer processor manufacturers, namely Intel and AMD have managed to support hardware-assisted virtualization in almost all of their products. Hardware assisted virtualization is important because without hardware assistance the amount of overhead associated with virtualization becomes a burden to the user. Hardware assisted virtualization effectively reduces the overhead

produced by the hypervisor by removing the necessity of the hypervisor to translate privileged instructions between the hypervisor and the host operating system. Additionally, virtual memory access is implemented in the hardware via nested page tables, which allow a guest operating system to rapidly access memory without requiring the hypervisor to translate memory addresses.

KVM is implemented as a kernel module that may be unloaded if unneeded. However, KVM has been included in the Linux kernel as a standard module since Linux 2.6.20, released in 2007. One interesting feature supported by KVM is live migration of virtual machine instances. This feature allows a virtual machine instance to be moved to a different server without a loss in service for the user. This is a desirable feature because if a cloud server fails for any reason, the virtual machine instance may be replicated across the network and resumed without the client even knowing that a failure has occurred.

When considering a cloud architecture for education there are few choices available for the hypervisor. KVM is one of the possible choices that could be used. The benefits of using KVM are highlighted by its open nature. Thus, KVM is free to use and free to modify, if the need arises. This makes it attractive for an education cloud because it presents no additional costs to the institution for licensing, and it performs the task of virtualization very well, considering that it is used in almost all open source cloud implementations, such as Eucalyptus and OpenStack. Additionally, KVM is included by default with the Linux kernel, which makes it very attractive, as installation is not necessary. Other hypervisors are available and provide similar functionality to KVM, namely being the Xen hypervisor and VSphere ESXi. Xen is also provided as an open source solution, and should be considered in addition to KVM. VSphere ESXi is a commercial product, and although it is available for free for personal use, it is not available for free for commercial use and requires the payment of licensing fees.

### 2.6.2 Energy Efficiency in Thin Client Solutions

Project such as the Linux Terminal Server Project (LTSP)čiteltsp present an effort towards energy efficiency of shared computing environments that predates that of the modern cloud. LTSP is a relatively popular low power thin client solution for accessing free and open source Linux environments running on a cluster of servers. LTSP does not offer the more modern attributes of hardware virtualization, scalability and elasticity that come with cloud computing. Alternatively, LTSP provides each user with a private GNU/Linux session by which the same physical hardware is shared. However, more recently there is an effort towards energy efficiency with thin client solutions paired with an IaaS cloud. The authors of [Vereecken, Deboosere, Simoens, Vermeulen, Colle, Develder, Pickavet, Dhoedt, and Demeester, 2009] analyze energy savings opportunities in the thin-client computing paradigm.

Utilization of thin client devices for general purpose computing is not a new concept. In the 1960's this model was used in the mainframe-computing paradigm, where thin clients would dial into a central data and computational "mainframe" to perform computational and data-driven tasks. The mainframe model is not so different from the cloud-computing model. However, the cloud-computing model extends mainframe computing to a new level with improvements in scalability, elasticity, and overall power reduction by taking advantage of technologies, such as voltage and frequency scaling.

The current approach to decreasing the energy footprint of a computing infrastructure is based on downscaling of performance and even completely powering devices down based on overall system utilization [Galloway, Loewen, and Vrbsky, 2014; Vrbsky, Galloway, Carr, Nori, and Grubic, 2013]. Additionally, energy aware algorithms have shown potential in saving energy by

directing tasks to execute on devices where the least amount of energy will be consumed. Thin client solutions are also utilized in organizations mainly as a means to decrease hardware acquisition costs and improve manageability of the clients. However, the authors of [Vereecken et al., 2009] take a different approach by analyzing the implications of thin client solutions with respect to power utilization and efficiency.

Power utilization of a thin client device compared to a traditional desktop is necessary to determine the benefits in using thin clients over traditional computers. The authors of [Vereecken et al., 2009] have made such a comparison by introducing an analytical model. The model proposed by the authors compares power consumption of a typical set of desktop computers with that of the same number of thin client devices along with the power consumption of the networking and servers.

For each power comparison study performed by the authors of [Vereecken et al., 2009] Equation (2.1) is used to calculate the power utilization of the system, regardless of whether it is a desktop, thin client, or cloud server.

$$P = P_0 + \alpha_{CPU} \times \lambda_{CPU} \tag{2.1}$$

Where:

$P_0$ : Power utilization of components (networking, etc.)
$\alpha$ : CPU wattage
$\lambda$ : CPU utilization (0% to 100%)

$$C = \#Cores \times CINT2006 \tag{2.2}$$

Where:

$C$ : Computational load
$CINT2006$ : Single threaded benchmark result

$$\lambda^{s}{}_{CPU} = N \times \left[ \lambda^{d}{}_{CPU} \times \frac{C^{d}}{C^{s}} + \varepsilon \right] \tag{2.3}$$

Where:

$\lambda^{s}{}_{CPU}$ : CPU utilization of the cloud server
$\lambda^{d}{}_{CPU}$ : CUP utilization of the desktop
$C^{d}$ : Computational load of the desktop
$C^{s}$ : Computational load of the cloud server
$N$ : Number of clients connected to the cloud
$\varepsilon$ : Additional load attributed to protocols (VNC, RDP, etc.)

$$P^{thin} = P^{c}{}_{0} + P^{n} + \frac{P^{s}{}_{0}}{N} + \alpha^{s}{}_{CPU} \times \left[ \lambda^{d}{}_{CPU} \times \frac{C^{d}}{C^{s}} + \varepsilon \right] \tag{2.4}$$

Where:

$P^{thin}$ : Power utilization of the thin client solution
$P^{c}{}_{0}$ : Base power utilization of the thin client device
$P^{n}$ : Power utilization of the networking devices
$P^{s}{}_{0}$ : Power utilization of the cloud servers
$\alpha^{s}{}_{CPU}$ : CPU wattage of the cloud server

By applying Equation (2.1) the authors were able to perform a power comparison of traditional desktop computers and thin client solutions during peak load. However, additional information is necessary in order to achieve usable results. Namely, the computational capacity of the desktop workstation should be compared with the computational capacity of the cloud server. This

study utilized the SPEC CINT 2006 benchmarking tool. Equation (2.2) shows the computational load measurement.

Another consideration must be made with regards to CPU utilization of the cloud servers because each server hosts multiple users, whereas one desktop computer only hosts one user. For the purposes of the power comparison it is important that for each operation performed on the desktop computer, the same operation should be performed on the cloud server. Therefore, there exists a dependency between the CPU utilization of the cloud server and the CPU utilization of the desktop computer in this study. Due to this dependency the formula for CPU utilization of the cloud server is shown in Equation (2.3).

Based on the Equations (2.1) to (2.3), the total power consumption for a thin client solution, including the thin client device, networking components, and cloud servers can be expressed using Equation (2.4). The formula presented in Equation (2.4) is used to calculate the power consumed by the thin client solution, which will be compared to the power consumption of a traditional desktop using Equation (2.1).

The results of this study show that power savings by using a thin client solution can reach up to 350% over that of a traditional desktop solution. Utilizing low power states for unused or idle devices additionally increases the cost savings. Low power states may be applied to both the thin client systems as well as the cloud server, effectively powering down unused equipment.

The authors of [Baliga, Ayre, Hinton, and Tucker, Jan] discuss green computing and how energy conservation in computing is gaining momentum, especially with the current energy crisis. The authors discuss cloud computing as an architecture that has a lot of potential for being environmentally friendly. The focus of the paper is in describing power conservation models used in cloud computing, which are similar to those discussed in [Vereecken et al., 2009].

Being capable of comparing the power utilization of a cloud-based solution using thin clients and a traditional non-virtualized solution is important when considering cloud computing for education. One of the largest motivating factors discussed in the literature is energy conservation and cost savings. The authors of [Vereecken et al., 2009] derive the formulas for such a comparison, and also show that power utilization of a thin client cloud-based solution can be up to 350% over a traditional non-virtualized computer lab. Although, this is a theoretical measurement using the derived power consumption formulas, it illustrates that budget cuts can be alleviated in the long term by utilizing thin clients and a cloud architecture. Considering that the authors have taken a theoretical approach to comparing power conservation in thin client solutions, the work could have been strengthened by showing a practical power measurement and comparing results with the theoretical figures.

### 2.6.3 Design Considerations for Cloud Architectures

Authors of [Cardellini and Iannucci, 2012] discuss design considerations for a low power and modular cloud solution. In this study the LTSP architecture is reviewed and compared to the cloud architecture designed by the authors of [Cardellini and Iannucci, 2012]. For performance reasons the authors have chosen to segment the backend network into several subnets, where each service category utilizes a different subnet in order to separate the network traffic according to service, namely being storage, virtualization, and software. The storage architecture is designed using Distributed Redundant Block Devices (DRBD) [Distributed Redundant Block Devices, 2013], which provide highly redundant storage across a network. DRBD allows data to be replicated using a primary/secondary device scheme, where the primary device maintains the original data and the secondary device maintains the replicas. Additionally, Logical Volume Management (LVM) [Sourceware, 2013] is used for managing the storage volumes without having to deal with the

physical storage devices directly. LVM provides an abstraction from the physical storage devices such that volumes are created and modified through the LVM service rather than partition tables. Figure 2.7 illustrates the storage architecture designed in [Cardellini and Iannucci, 2012].

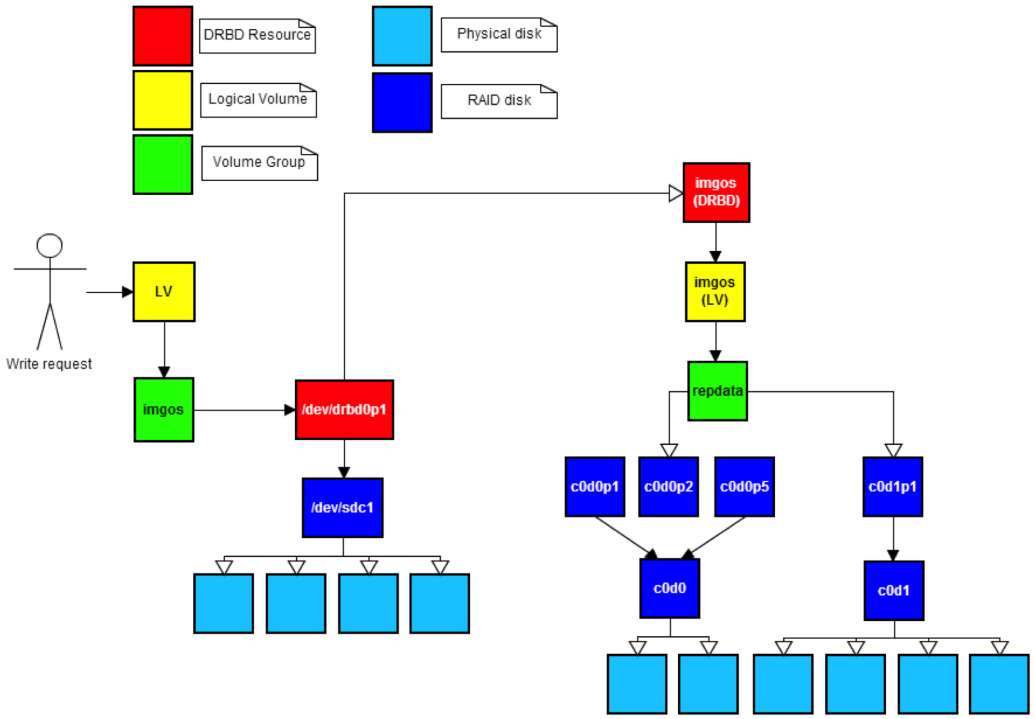



Figure 2.7: Cloud storage architectural design.

The LTSP architecture provides services, which are very similar to an IaaS cloud architecture with a few notable limitations. Firstly, LTSP only offers Linux environments, which differs from an IaaS cloud in that the cloud can host Linux, Windows, and in some cases Apple OSX virtual machine instances are supported. Additionally, LTSP does not utilize virtualization technology, rather it provides several minimal Linux and X windows environments on the same host computer. Interfacing with an LTSP instance also differs from an IaaS cloud in that an LTSP terminal will boot directly from the host machine using PXE or NetBoot [Syslinux, 2013], which is a remote booting protocol. Based on observations of current IaaS cloud implementations, a

client connected to an IaaS cloud will typically rely upon the Remote Desktop Protocol (RDP) for accessing Windows instances, or the Virtual Network Computing (VNC) protocol for Linux instances.

Because the cloud designed in this study is an IaaS cloud, selection of an appropriate management platform for virtualization services is pinnacle in the design. The authors chose to use OpenQRM [openQRM, 2013] in addition to KVM and Xen hypervisors [Goto, 2011] for managing virtualization and the storage architecture. This design consideration was made due to the fact that OpenQRM is customizable and compatible with another cloud management architecture called OpenNebula [OpenNebula, 2013].

The design proposed in [Cardellini and Iannucci, 2012] utilizes open source tools to create a modular and scalable IaaS cloud without resorting to all-in-one solutions like Eucalyptus [Eucalyptus Systems, 2013b] or OpenStack [OpenStack Foundation, 2013a]. Additionally, the authors provide insight into the differences between their design and a typical LTSP cluster. Namely, LTSP relies on accessing multiple X windows environments from the same Linux computer, which differs from a cloud architecture, which is designed to instantiate multiple virtual machines on the same computer. Additionally, LTSP is a developed project, which utilizes both open source and custom software, whereas the authors of [Cardellini and Iannucci, 2012] utilize third party solutions only. Although, the design proposed is not specifically targeted towards educational institutions, it serves as a valuable resource when considering a cloud architecture for education. LTSP, however, has been used successfully by educational institutions, and the insight gained in comparing LTSP with a cloud architectural design is interesting because cloud computing is an emerging technology in the classroom while LTSP has a longer history of use in education.

Whenever a new architecture is being designed, the architect must make some design con-

siderations. The authors of [Cardellini and Iannucci, 2012] outline their design considerations for a cloud architecture, including the usage of OpenQRM for managing virtualization and storage, DRBD for storage redundancy, LVM for managing storage volumes, and both KVM and Xen hypervisors. Although, these elements of the design are chosen for specific reasons, the authors do not adequately describe alternative approaches and reasons behind their decisions. However, the design considerations do provide insight into what a cloud architecture should be composed, which can help designers of an education cloud in the case of confusion or misunderstanding of the reasoning of a cloud architectural design.

## 2.7 Cloud Middleware Solutions

The authors of [An, Pradhan, Caglar, and Gokhale, 2012] present their solution, SQRT-C, which is a light weight and scalable resource monitoring and dissemination solution using the publisher/subscribe model. The SQRT-C solution is designed specifically as a means to improve quality of service in real time systems. The approach considers three major design implementations as a top proiority: Accessing physical resource usage in a virtualized environment, managing data distribution service (DDS) entities, and shielding cloud users from complex DDS QoS configurations. SQRT-C can be deployed seamlessly in other cloud platforms such as Eucalyptus and OpenStack since it relies on the libvirt library for information on resource managment in the IaaS cloud.

In [Sun, Wang, Zhou, Huang, and Liu, 2010], the authors begin to build a cloud computing for service oriented softwre development. Their SOARWare tool is used for software development, but users still need to install various tool on their local machine for development. The authors this need by moving their tools to a PaaS architecture. MyCloud and App Engine provide the platform and interface for developers to begin work quickly on service oriented architecture projects.

In [Lee, Tang, Chen, and Chu, 2012], the authors proposed a middleware for enterprise cloud computing architectures whcih can automatically manage the resource allocation of services from services, platforms, and infrastructures. The middleware API set used in their cloud is built around servicing specific entities using the cloud resources. For end users, the API toolkit provides interaction for requesting services. These requests are submitted through a web interface. Internal interface APIs communicate between physical and virtual cloud resources to construct interfaces for users and determine resource allocation. A service directory API is provided for users based on that users permissions. A monitoring API is used to monitor and calculate the use of cloud system resources.

The authors of [Apostol, Baluta, Gorgoi, and Cristea, 2011] proposed a resource manager that handles users requests for virtual machines in a cloud environment. Their architecture deploys a resource manager and a policy enforcer module. First the resource manager decides if the user has the rights to request a certain virtual machine. If the decision is made to deploy the virtual machine, the policy enforcer module communicates with the cloud front-end and executes an RPC procedure for creating the virtual machine.

The authors of [Khalidi, 2011] describe how cloud platforms should provide a rich set of services on-demand that help the user complete their job quickly. Also mentioned is the cloud's responsibility of hiding low-level technical issues such as hardware configuration, network management, and maintenance of guest and host operating systems. The cloud should also reduce costs by using dynamic provisioning of resources, consuming less power to complete jobs (within the job constraints), and by keeping human interaction to cloud maintenance to a minimum.

Developing cloud APIs are discussed in [Cooper, 2010]. The author mentions three goals of a good cloud API: Consistency, Performance, and Dependencies. Consistency implies the guar-

antees that the cloud API can provide. Performance is relatively considered in forms of decreasing latency while performing actions. Cloud dependencies are other processes that must be handled other than spawning virtual machines and querying cloud resource and user states.

## 2.8 Cloud Deployment Strategies

This section will briefly cover the deployment process of the major open source IaaS cloud architectures Eucalyptus and OpenStack [OpenStack Foundation, 2013a]. The deployment process of these cloud architectures will be confined to a single machine. Most of the open source cloud architectures provide this deployment opportunity not as a production option, but as a way to learn the specific cloud services, dependencies, interface, and networking options.

### 2.8.1 Eucalyptus

Eucalyptus Provides users with a Faststart [Eucalyptus Systems, 2014], a process for building an IaaS cloud architecture using one or two computers running GNU/Linux. Deploying Eucalyptus using only one machine uses their *Cloud-in-a-Box* method. Once the user begins to install Eucalyptus, they will be asked for network information, such as the local network interface, IP address, Netmask, Default Gateway, DNS servers, etc. After the initial network information is gathered, the user is asked to enter a Eucalyptus root password. The user then is prompted for the common Eucalyptus cloud configuration options. Since this uses the *Cloud-in-a-Box* method, most of the configuration options are set as default, requiring no change from the user. The only thing the user must enter is the range of IP addresses that are available to be assigned to virtual machine instances launched by Eucalyptus. Next, the user will be prompted to enter persistent storage information. Since Eucalyptus will assume it is the primary application, it will attempt to use all available disk space. The user has the option to change this if needed. Eucalyptus will begin to install and reboot the computer. Once the system is rebooted, the user can access (some)

cloud information by using the default Eucalyptus web interface. It should be noted that the default web interface for Eucalyptus does not give the user the ability to launch or terminate virtual machines. The most common installation of Eucalyptus generally requires at least five computers. Those are used for Cloud Controller, Cluster Controller, VM Image Controller, Persistent Storage Controller, and Compute Node Controller. The administration of this type of Eucalyptus architecture requires a substantial amount of time and effort using the Linux terminal, as most of the architectural refinements are not accessible with the Eucalyptus web interface.

### 2.8.2 OpenStack

Users can install a minimal OpenStack cloud by using DevStack [Openstack Foundation, 2014]. DevStack provides a one-node OpenStack cloud deployment and is meant to allow new users to get accustomed to the OpenStack architecture. Before deployment of DevStack the user has to install a GNU/Linux based operating system. Next, the DevStack installer needs a username with sudo privileges for installing the OpenStack packages. The user can then download the latest version of DevStack. The user has to configure *stack.sh* and create *local.conf*. The *local.conf* file has to be modified to adhere to the settings used by the local network. The OpenStack, MySQL, RabbitMQ, and service administrative accounts and passwords must also be configured in this file. Once *local.conf* is configured, the user can run *stack.sh* to install OpenStack. Once the installer finishes, the user can access OpenStack's web interface (Horizon). This interface gives the user the ability to launch and terminate virtual machine instances.

## 2.9 Load Balancing

Load balancing in cloud computing architectures is defined as scheduling tasks (or virtual machines) for deployment on cloud servers based on the current state of the cloud architecture. The authors of [Lanjewar, Surwade, Patil, and Ghumatkar, 2014], [Ajit and Vidya, 2013], [Mahajan and

Dahiya, 2014] give the goals of cloud load balancing as a way to improve overall performance, mitigate hardware and software failures, maintain system stability, and for accommodation of future cloud software and hardware modifications.

Task scheduling on cloud systems is defined in [Tsai, Huang, Chiang, Chiang, and Yang, 2014] as a workflow problem, which can be classified into two levels: service-level and task-level [Daniel and Lovesum, 2011]. The task scheduling problem for cloud computing is different from grid computing and other cluster architectures since users can deploy virtual machines for interfacing with resources. [Tsai et al., 2014] presents a Hyper-Heuristic Scheduling Algorithm, which has better performance handling workloads than rule-based deterministic algorithms.

A scalable, stochastic model-driven approach to resource management in cloud computing, where failures are handled by virtual machine migration, is covered in [Ghosh, Longo, Frattini, Russo, and Trivedi, 2014]. Cloud servers in this model are grouped into three categories: running, powered on but not ready, and powered off. Three approaches are considered for constructing a stochastic modeling approach for resource allocation: monolithic modeling of cloud resources [Ciardo, Blakemore, Chimento, Muppala, and Trivedi, 1993], interacting sub-models, and simulation.

The authors of [Zhang, Zhani, Boutaba, and Hellerstein, 2013] present a cloud resource monitoring tool called *Harmony* that is capable of performing dynamic capacity provisioning [Zhang, Zhani, Zhang, Zhu, Boutaba, and Hellerstein, 2012] in heterogeneous cloud architectures. This monitoring tool adjusts the resource provisioning capabilities on a varying number of machines in the cloud architecture to create a balance between energy savings and workload scheduling delays.

A guest-aware scheduler is introduced by [Kim, Kim, Jeon, Seo, and Lee, 2008]. This

scheduler takes an intrusive approach since guest virtual machines expose their information to the hypervisor. The guest-aware scheduler reduces scheduling latency and I/O response time of tasks running on virtual machines in the Xen hypervisor. These advantages come with drawbacks, such as not working with closed-source operating systems like Microsoft Windows and potentially introducing security risks by communicating information from the host operating system to the guest operating system.

An energy-aware scheduler, *EARH*, is defined in [Zhu, Yang, Chen, Wang, Yin, and Liu, 2014]. Large scale data centers consume a large amount of energy, generally at high costs to the cloud vendor [Koomey, 2007], [IBM, 2019]. The energy-aware scheduler is needed to negate the fact that failure rates of computing devices double for every 10°C[Feng, 2003]. *EARH* employs a rolling-horizon optimization scheme which allows tasks to be queued based on the current system state, which could reduce energy consumption. The scheduler then creates a task-oriented energy consumption model that produces an optimized scheduling algorithm that reduces the energy required to complete tasks while maintaining task deadline constraints.

The authors of [Cao, Li, and Stojmenovic, 2014] also introduce a load balancing method that is optimized for reducing power consumption and increasing cloud architecture performance. This load balancer considers power constrained or performance constrained optimizations.

[Hu, Gu, Sun, and Zhao, 2010] introduces a load balancing algorithm using genetic algorithms that takes into account historical data and the current state of the cloud resources to achieve an optimal task load. This load balancing algorithm tries to avoid or reduce the number of dynamic virtual machine migrations [Sotomayor, Keahey, and Foster, 2007]. The authors of [Zhang, Xiao, Tao, Tian, Wang, and Liu, 2013] use a similar approach for creating a scalable load balanc-

ing method that accounts for different types of virtual machines and physical nodes in the cloud architecture.

## 2.10 Summary

There is a great deal of research in the literature that focuses on general purpose cloud architectures; however, research in vertical cloud architectures is limited. One of the few substantial studies on vertical clouds appears in [Stein et al., 2013] where the authors discuss the results of a study on a vertical cloud designed for middle and high school students. Within the scope of general purpose cloud architectures the literature contains much research relating to power and energy conservation, load balancing and consolidation of virtual machines, and large scale deployment strategies.

Based on the literature, we are interested in using the observations from the studies on vertical cloud computing as well as proposed solutions regarding general purpose cloud computing to design a new architecture. By reflecting on the results in the literature our goal is to produce a vertical cloud architecture that helps to break the cultural and technical barriers found in [Stein et al., 2013]. In addition to breaking these barriers, we want to provide robust cost savings and usability features as described by general purpose cloud computing research.

# Chapter 3

# ARCHITECTURE DESIGN

The THUNDER project aims to provide on-demand compute resources to both organizations and individuals, especially underfunded non-profit organizations and individuals with little technical experience. Our primary focus is to provide organizations with the means to facilitate the computing needs of their users in a cost-effective manner as well as to help support the user's needs with dynamic and scalable cloud services. The THUNDER architecture is composed of a special purpose private cloud stack, which can be leveraged using traditional thick-client devices, as well as low power thin-client devices. The THUNDER stack differs from the general purpose private cloud model in a number of ways. General purpose cloud stacks, such as Eucalyptus [Eucalyptus Systems, 2013b] and OpenStack [OpenStack Foundation, 2013a], are focused on providing users with many different options as to how the cloud may be configured. These general purpose solutions are suitable for some organizations because the architecture is flexible enough to be useful for diverse markets. However, underfunded non-profit organizations do not typically have the resources to deploy general purpose cloud solutions and many of these organizations simply need to satisfy basic computing needs of their constituents. Therefore, a special-purpose, or vertical cloud architecture is desirable because it circumvents the typical cloud deployment process by making assumptions about the use of the architecture, such as computers residing in a shared office setting or a computer lab at a public secondary school, as described in Chapter 2. One example in which this architecture might be particularly useful is in education. Research in cloud computing for

education as described in Chapter 2 has shown that educational services in high school settings are successful in motivating students to learn and possibly achieve greater success in the classroom [Stein et al., 2013].

The THUNDER cloud stack utilizes a number of compute nodes, in addition to persistent storage nodes with a redundant backup, as well as a custom DHCP, MySQL, and system administration server. One example scenario for which THUNDER could be beneficial is a computer lab. This lab scenario consists of embedded thin client systems with keyboards, mice, and monitors, as well as a connection to a private gigabit network. A custom web-based interface allows users to log in, select a desired virtual machine from a list of pre-defined images, and then launch the virtual machine. For example, users that are interested in programming might be interested in a GNU/Linux based virtual machine with development tools. However, users that require Microsoft Office, or other software designed for Microsoft Windows might need a Windows based virtual machine. Therefore, regardless of their specific requirements, THUNDER will be able to provide all necessary software components to each user independently. Additionally, software services are supported by means of custom private virtual machine server instances, which can be used to host private web servers, application servers, and other services. Figure 3.1 illustrates the THUNDER network topology, and Figure 3.2 illustrates a typical general purpose Eucalyptus cloud topology.

## 3.1 Network Topology

One of the common advancements in wired Ethernet technology is the use of switches [Metz, 1998]. Ethernet switches allow for adjacent nodes connected to the switch to communicate simultaneously without causing network collisions. Almost all network interface cards support full duplex operation which further allows nodes to send and receive data over the network at the same time. Ethernet is a Link-Layer protocol [Metz, 1998], which determines how physical devices on

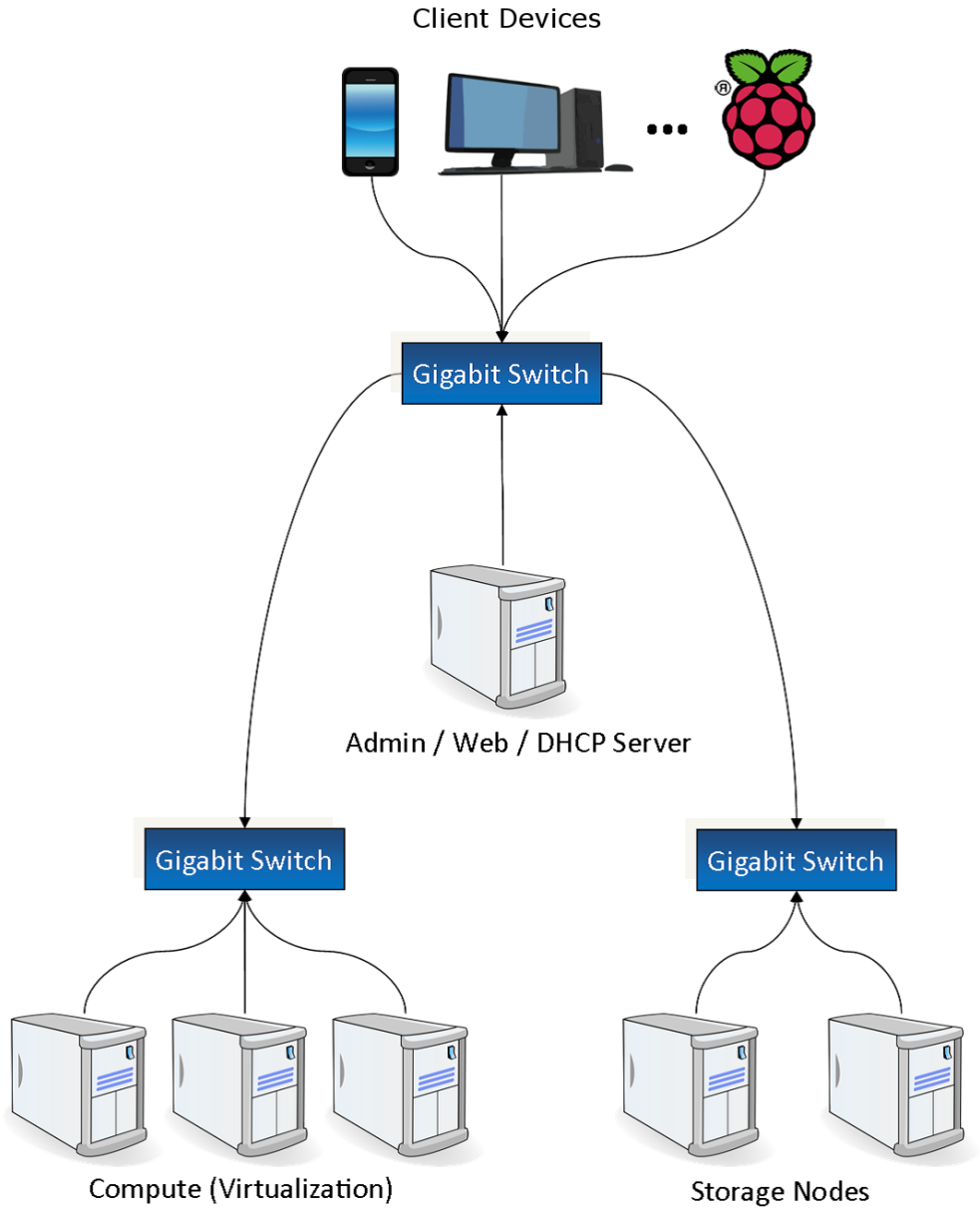



Figure 3.1: Network topology of THUNDER architecture.

the network communicate. THUNDER is implemented with the assumption that network switches are used as a means to separate clusters of nodes.

The THUNDER network topology is designed for use with any device that has sufficient access to a web browser and internet access, since the THUNDER interface is entirely web-based. However, local access to THUNDER resources provides the best network performance because

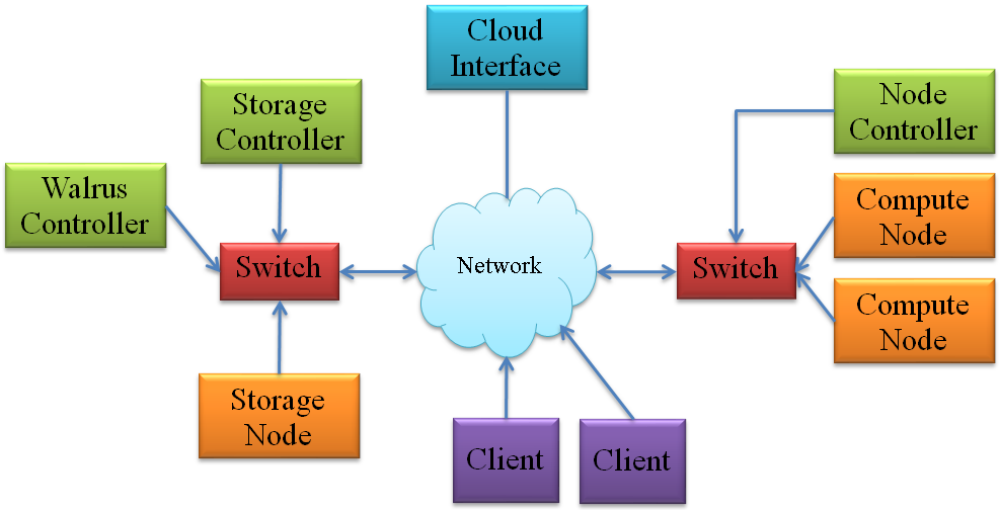



Figure 3.2: Typical network topology for a Eucalyptus cloud deployment.

| Typical Lab | | | THUNDER | | |
|---|---|---|---|---|---|
| # | Hardware | Watts | # | Hardware | Watts |
| 20 | PC Desktops | 6,000 | 20 | Thin Client | 60 |
| 20 | Display | 2,000 | 20 | Display | 2,000 |
| | | | 3 | Compute Node | 1,200 |
| | | | 2 | Storage Node | 500 |
| | | | 1 | Admin/Web | 250 |
| Total | | 8,000 | Total | | 4,010 |
| Monthly Bill: $152 | | | Monthly Bill: $77 | | |

Table 3.1: Power cost comparison between THUNDER with low power thin clients and a typical lab.

local users have direct access the private LAN. THUNDER is designed to use the common star gigabit Ethernet topology, which separates compute and storage resources, administrative resources, and thin clients. The maximum sustainable throughput of gigabit Ethernet is 125MB/s (1024b/8). By isolating resources we can ensure that local network latency is minimized, however this has little effect on network latency between THUNDER and remote clients.

The THUNDER network topology resembles a typical cloud topology, where the compute cluster is connected to a single shared LAN switch, and support nodes share a separate LAN

switch. Additionally, the topology shows the client devices and how they interface with the rest of the system. The results found in Table 3.1 shows a power cost comparison between THUNDER and a typical twenty PC lab. These results demonstrate a possible savings of 50% with THUNDER when compared to a traditional computer lab.

### 3.1.1 Compute and Store Resources

THUNDER compute and storage resources consume a considerable amount of network bandwidth. The compute nodes are responsible for hosting virtual machines that are accessed by the users. These compute nodes mount the user's persistent data as the virtual machine is booting. Each compute node communicates with the THUNDER cloud resources and the client devices using a 1Gbps network interface. The Raspberry Pi, which is one potential thin client device, is only equipped with a 10/100Mbps network adapter; however utilizing a 1Gbps network switch ensures that network bandwidth won't be saturated. Userspace storage nodes are connected to the same switch as the compute nodes, which allows for tighter coupling of storage nodes and compute nodes, decreasing the potential delay for persistent file access. Backup storage nodes are connected to a separate network switch, and are used to backup the cloud system, in case of a system failure.

### 3.1.2 Administrative Resources

The THUNDER administrative resources include a MySQL database, web interface, and networking services. These services are hosted on a single physical machine, with a backup machine isolated to the same network switch. There will be no need for a high amount of resources in the head node since the numbers of compute and storage nodes determine the amount of clients that can be connected to THUNDER. The THUNDER cloud can only be accessed by secure login, leaving the head node handling only the requests from the users as they launch virtual machines. This ensures that the virtual machines cannot be reserved by anyone except for authenticated users.

### 3.1.3 Thin Client Resources

Thin clients are responsible for displaying Microsoft Windows or Linux virtual machines to the users. Even though many thin client devices only maintain a 10/100Mbps interface, the maximum transfer rate from the thin client to THUNDER resources is not only determined by the network link, but by the system, memory bus and the capability of the CPU to handle network traffic.

In most cases the network interface to THUNDER is sufficient for utilizing VNC [Richardson, 2010], which is a protocol for connecting to a computer remotely. The VNC protocol automatically adjusts to the amount of available bandwidth by adjusting the color depth and applying compression to accommodate for lower bandwidths. This bandwidth is determined based on what the user is viewing. Multimedia viewing and editing will require more resources that composing text documents. This is due to the fact that multimedia is more difficult for VNC to compress before sending it to the client device. Based on the data given from VNC [Richardson, 2010], 33Kbps is sufficient for interfacing with virtual desktop environments with color compression, reduced frame rate, and no multimedia data involved. The client will need to sustain 128Kbps of bandwidth to display desktop environments with color compression and small amounts of multimedia. At 1Mbps, the client should accommodate virtual desktops with little color compression and little to no reduction in frame rate. A 100Mbps connection will be sufficient in making most tasks running in the virtual environment no different than if it were hosted on the local machine.

An example configuration involves Raspberry Pi devices. The Raspberry Pi has a 700 MHz ARM processor, which may be overclocked to 1 GHz, and 512MB RAM. The network interface is integrated into the USB 2.0 hub. Given the computation resources and power draw of the Raspberry

Pi (4 Watts max) virtual machine display performance is reduced when displaying multimedia. Network bandwidth between THUNDER and the Raspberry Pi is measured using IPERF [Iperf team, 2013], and demonstrates the potential network latency issues with this particular device. When the CPU is at idle, the maximum throughput from the Raspberry Pi to the THUNDER resources is 94.5Mbps. This throughput can be sustained +/- 5% for as long as needed. During the sustained maximum throughput network tests, the CPU utilization averages around 55%. With the CPU utilization at 30%, the average network throughput from the Raspberry Pi to THUNDER is 71Mbps. When the CPU utilization reaches 100%, the maximum sustainable throughput from the Raspberry Pi to THUNDER is 51Mbps. In all cases, the throughput from THUNDER to the Raspberry Pi is 92Mbps. The low network throughput from the Raspberry Pi is a result of the low power, single core CPU and integrated USB/Ethernet interface. As a result, we experience choppy video playback while connected to a THUNDER virtual machine, and further investigation into alternative thin client devices is underway. Equation (3.1) presents an extrapolation for the network throughput of Raspberry Pi's using a polynomial equation, relating network throughput to CPU load.

$$Rasp_{Throughput} = 0.005 \times Rasp_{CPU}^2 - 0.9326 \times Rasp_{CPU} + 94.5Mbps \pm 5\% \quad (3.1)$$

Where:

$Rasp_{Throughput}$ : Network Throughput of a Raspberry Pi Thin Client Device
$Rasp_{CPU}$ : CPU Utilization of a Raspberry Pi Thin Client Device

Although, it is possible to utilize the Raspberry Pi device as a thin client, many more restrictions must be made with regard to the host operating system. It is unreasonable to expect

a device such as the Raspberry Pi to offer adequate performance unless the host operating system reduces potential overhead. Overhead in this case consists of the windowing manager and any excessive background tasks typically found in general purpose consumer desktop environments.

### 3.1.4 Network Provisioning for Low Latency

Using a top down approach, the maximum amount of bandwidth needed for a twenty node THUNDER lab can be described. If we assume that each thin client requires at least a sustained 1Mbps network throughput, we would need to accommodate for sustaining 20Mbps within the network switch used by the thin clients. Since this switch is isolated to communicating with the resources of THUNDER, this throughput needs to be sustainable on the uplink port. This should be relatively easy, given the costs of gigabit switches on the market today. The specification that needs close attention is the total bandwidth of the switch backplane. Making the assumption that this bandwidth is the number of ports multiplied by the switch speed is not always true. In our case, the bandwidth needed, 20Mbps, is much lower than the maximum throughput of a 24 port gigabit switch.

The THUNDER storage node resources are also isolated to the same gigabit network switch as the compute node resources. When the user logs into THUNDER and requests a virtual machine, their persistent storage is mounted inside the virtual machine for them to use. The data created by the users have to be accessed while they are using a virtual machine.

## 3.2 Inter-Node Communication

The THUNDER architecture depends on a private cloud middleware service, which allows each node to communicate with the head node. This methodology is very similar to the client-server model [Berson, 1992], where the server waits to receive requests from clients. We utilize simple socket programming for inter-node communication using the middleware API presented in

Chapter 4. The head node is responsible for performing a number of tasks, including node registration and updating system metadata. As these tasks are event driven, we implement a socket listener on the head node, which waits for requests on a pre-defined port and then responds accordingly. A service is instantiated on each of the support nodes, which sends and receives requests to and from the head node. We utilize inter-node communication for the following tasks:

- Registering nodes
- Launching virtual machines
- Synchronization of metadata
- Removing and/or deleting nodes from the system

Tasks such as these are executed in a way that is not visible to the user, and users do not typically have explicit access to terminal commands, unless in the case of system administration. Additionally, it is important to note that each client has a table in a database located on the head node. When information about a client changes, for example after a successful virtual machine instantiation, the table for the client will be updated with information about the VM instance.

When the the middleware service receives a request, it will respond accordingly. For example, in the case of launching a virtual machine, when the head node responds to a VM request it will perform the following actions:

- Determine if the VM name is valid
- Select a compute node using a VM placement and load balancing algorithm (see Section 3.6)
- Send a request to the selected compute node

- Wait for response from the compute node, being either “Success," or “Failure"
- On success, the head node will read the IP address of the VM from the clients table in the database, and relay the IP address to the web interface
- On failure, the head node will send a failure response to the web interface

During this server-side process the client will be waiting on a successful instantiation of a virtual machine image. If the database contains successful VM instance information, then the client will launch a remote desktop client and connect to the VM instance IP address. If the client determines that the VM instantiation has failed, it will notify the user and send an email to the administrator for a service request. Figure 3.3 shows the sequence diagram for this process.

## 3.3 Authentication

Other private cloud architectures, such at Eucalyptus [Eucalyptus Systems, 2013b] and OpenStack [OpenStack Foundation, 2013a], utilize the secure shell (SSH) protocol for communication between nodes. This is done mainly for security purposes, as SSH encrypts packets before transmission. However, in order to implement a system based on SSH for inter-node communication it is necessary for the system to utilize some kind of key-sharing process whenever a new node is registered with the cloud. Additionally, passwordless, or key-based SSH connections are necessary for private clouds, which utilize the SSH protocol. THUNDER does not use SSH for inter-node communication. Instead, we rely upon socket connections for communication between nodes. This is done for simplicity reasons, because SSH introduces additional system overhead, and given that this is an architecture which is focused on decreasing complexity and overhead, the use of SSH could be counter intuitive. The downside of using non-encrypted socket connections is that there are no default security precautions provided with socket programming [Stevens, Fenner,

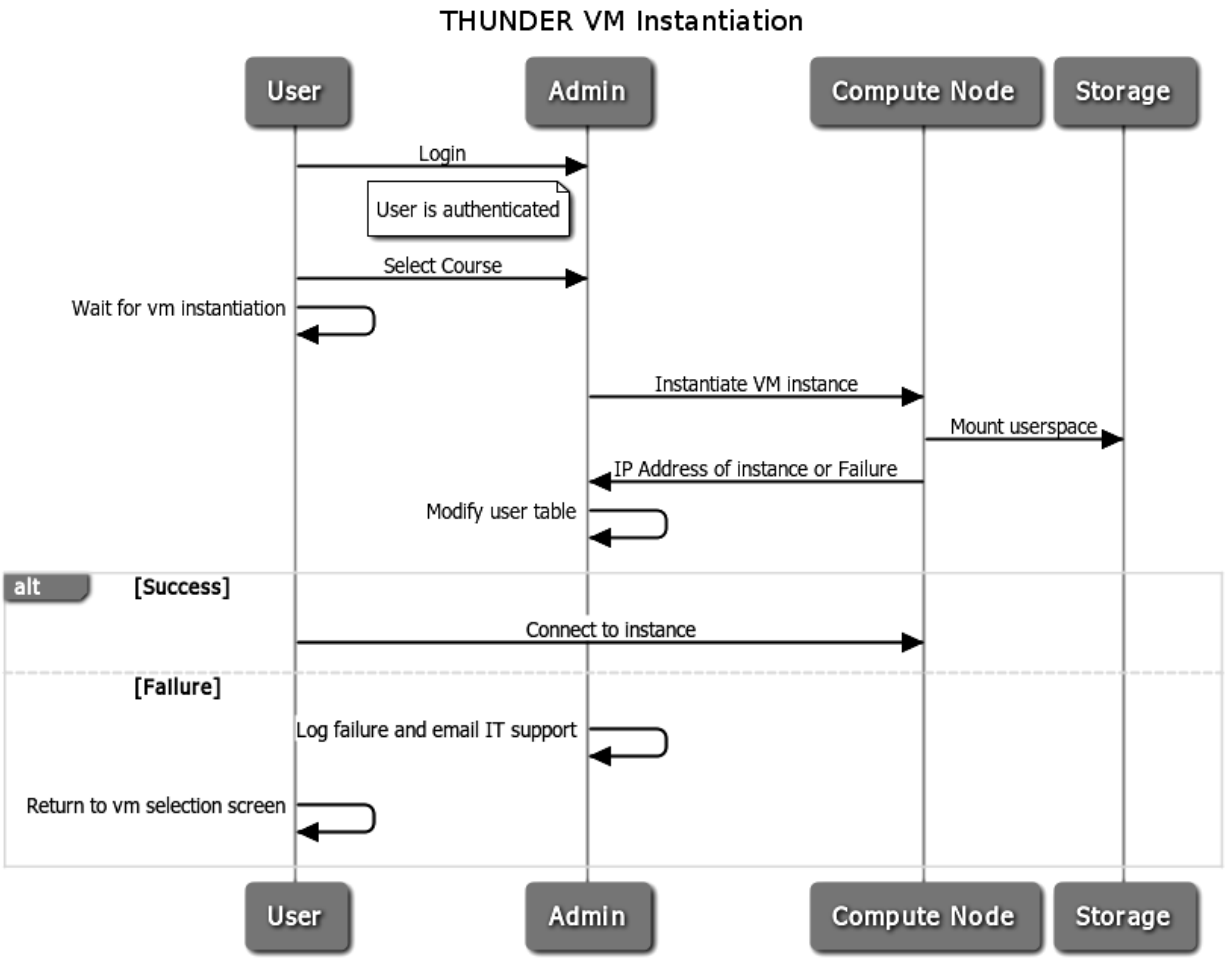



Figure 3.3: Sequence diagram for instantiation of virtual machines.

and Rudoff, 2004]. Therefore, the THUNDER architecture provides authentication using a simplified challenge-response mechanism. This approach ensures that the communication between nodes is authentic. However, it does not ensure that communication is encrypted. Although due to the nature of the communication, which is comprised mainly of commands for resource management, such as instantiation of virtual machines, encryption of messages is unnecessary. The security requirements of THUNDER may change in such a way that management traffic needs to be obfuscated, in which case encryption of network traffic may be desirable.

## 3.4 Multicast Based Node Registration for Fault Tolerant Cloud Servers

One of the shortcomings of general purpose cloud architectures, such as Eucalyptus and OpenStack, is that these architectures do not provide support for redundant head nodes. We have accomplished redundant cooperative head nodes via multicast based re-authentication. Essentially, we allow for $k$-node fault tolerance such that the cloud can continue to operate normally if up to $k$ head nodes fail. At periodic intervals each cloud server sends out a "heartbeat" request to the active head node. When the head node fails to respond to a heartbeat request from a particular cloud server, the server will send out a re-authentication request via the multicast group *224.0.0.2*, which corresponds to every router located on the local network. Additionally, replica nodes will also be sending out heartbeat requests. However, when a replica node does not receive a response to a heartbeat request, it will execute a cooperative failover operation, such that one of the replica nodes will be selected to become the head node for the cloud. As soon as a replica executes the failover operation, it will respond by broadcasting over the multicast group. Cloud servers will receive the message from the new head node and attempt a re-authentication. Furthermore, during a failover operation the new head node will dynamically set it's IP address to that of the previous head node in order to preserve the local web address used by clients.

## 3.5 Supporting Software Services

The preceding sections discussed our implementation of the software system necessary for supporting IaaS cloud services, namely hardware virtualization and persistent storage. Building upon virtualization of hardware, we are able to provide software services as custom virtual machine instances. The approach that THUNDER takes is instantiation of server virtual machine images, which deploy web services. By implementing services in this fashion, no modifications

are required to the infrastructure of the cloud, and administrators may easily start services by allocating hardware resources, and stop services by deallocating resources. This approach differs from typical SaaS architectures in that no additional configuration is necessary outside of what is required for regular instantiation of virtual machines.

## 3.6 Load Balancing of Virtual Machine Instances

Quality of service is an important factor when dealing with services such as virtualization, because one goal of virtualization is to ensure that the performance of virtual machines is comparable to that of non-virtualized systems. We can help to ensure quality of service, while not sacrificing cost effectiveness by utilizing a schedule in which virtual machine instances are either consolidated to one node or dispersed over many nodes. We define consolidation as the process by which virtual machines are migrated to a small subset of nodes. Consolidation of virtual machines decreases costs associated with power, because unused compute nodes can be put into a low power state. A fully balanced cloud, in which virtual machines are dispersed over many compute nodes ensures maximum virtual machine performance. We have devised a load balancing schedule, shown in Figure 3.4, in which a change from zero to one indicates that virtual machines will be distributed across all compute nodes and a change from one to zero indicates that virtual machines will be consolidated. Our schedule ensures high performance during peak hours, and cost saving during off-peak hours.

## 3.7 Accessibility of Virtual Machine Instances

Because of the special nature of the THUNDER architecture we believe that virtual machine instances should be made as accessible to the user as possible. It is true that other architectures fully support remote access to virtual machine instances via secure shell and remote desktop, if available on the instance. However, these access methods are not necessarily obvious to inex-

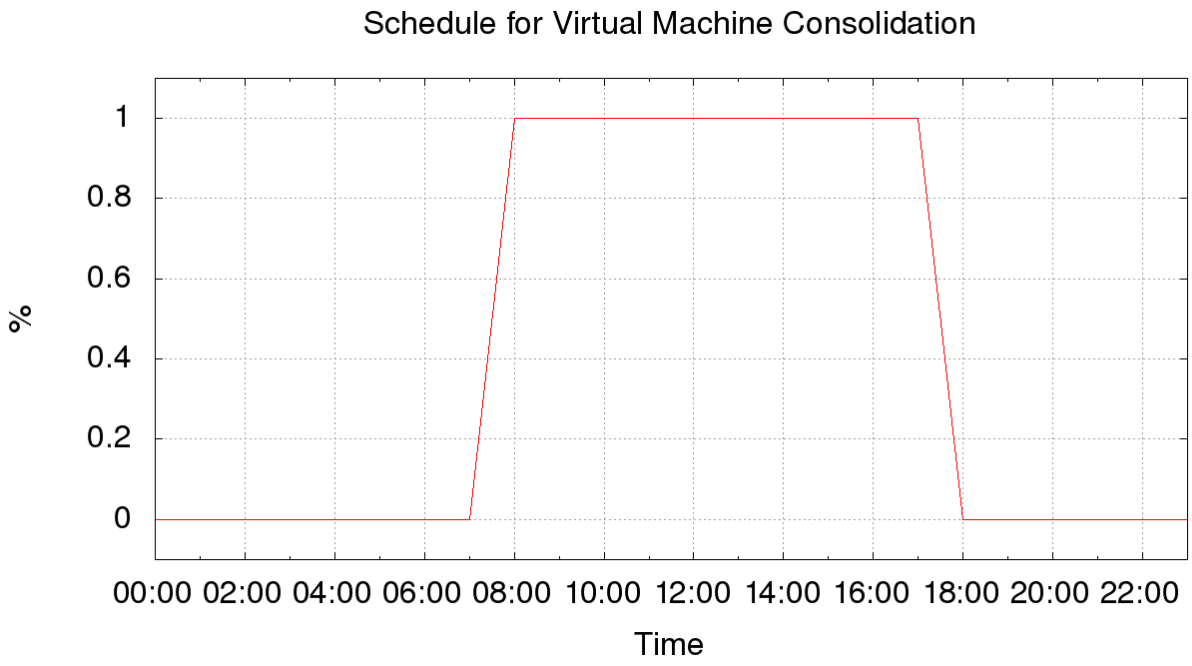



Figure 3.4: Virtual machine consolidation/distribution schedule over a typical work day.

perienced users. The approach made by THUNDER is such that virtual machine instances are accessible both via secure shell and remote desktop using an HTML5 based web client. This client connects to the virtual machine instance over the private IP address assigned during instantiation and displays the remote session in the web browser in such a way that the user does not need to use any tools other than the browser itself. In order for this process to work the virtual machine images provided by THUNDER must include a web server or web application server, such as Apache Tomcat. Additional specifications are outlined in Section 3.8.

## 3.8 Virtual Machine Image Specifications

THUNDER requires virtual machine images to be bundled using a scheme that encapsulates one or more images along with a specification written in YAML. The YAML descriptor and an LZMA compressed archive containing one or more serialized filesystem images are packaged

into an uncompressed tarball. Naming of the files is unimportant, however the YAML descriptor must end in a *.yml* extension to be recognized properly. The YAML descriptor contains data that is required by THUNDER for properly importing an image and storing it on one or more storage nodes. Additionally, multiple images can be defined within the same package, but only one is required. Figure 3.5 shows an example of a valid YAML descriptor with associated fields.

```
# Example image descriptor
---
image :
    - name : ubuntuxfce
      title :  Ubuntu XFCE
      description : Ubuntu GNU/Linux with the XFCE desktop.
      archive : xubuntu.tar.xz
      type : qcow2
      baseimage : base.img
      configimage : config.img
      overlayimage : disk.img

    - name : fedoragnome
      title : Fedora GNOME
      description : Fedora GNU/Linux with the GNOME desktop.
      archive : fedora.tar.xz
      type : qcow
      baseimage : base.img
      configimage : config.img
      overlayimage : fedora.img
```

Figure 3.5: Example YAML descriptor for two bundled THUNDER virtual machine images.

The LZMA compressed tarball contains between two and three serialized filesystems corresponding to a base image, a configuration image containing a NoCloud data store specified by cloud-init [Canonical, Ltd., 2015] and an optional overlay image. If no overlay image is used then the associated field in the YAML descriptor may be omitted. A valid THUNDER virtual machine image is created by first building a base disk image using a tool such as *qemu* or *kvm*. The base image is an unmodified base installation of a GNU/Linux distribution that is typically

formatted using qcow or qcow2, which are serialized filesystems optimized for copy-on-write. Copy-on-write delays physical disk allocation until absolutely necessary. In addition to the base disk image, THUNDER images support an optional overlay disk image. An overlay image is also a qcow or qcow2 serialized filesystem that builds upon the base image with additional packages, files, and configuration details. At the very minimum, the filesystem contained in either the base image or the overlay image must have support for some means of remote accessibility such as SSH or remote desktop services. Additionally, one service must be propagated over port 80 using a proxy or application service, such as Apache Tomcat and a remote connection service such as Guacamole [Glyptodon LLC., 2015]. This web service allows THUNDER images to be accessed easily using a web browser. Figure 3.6 shows an example structure of a THUNDER virtual machine image package.

```
images.tar
----- desc.yml
----- xubuntu.tar.xz
---------- base.img
---------- config.img
---------- disk.img
----- fedora.tar.xz
---------- base.img
---------- config.img
---------- disk.img
```

Figure 3.6: Example structure of a THUNDER virtual machine image.

## 3.9 Startup Cost Comparison

In continuation of the power cost comparison shown in Table 3.1 it is also important to understand the startup costs of a private cloud versus a traditional workstation setup. In this comparison we have devised the formula shown in Equation (3.2) with each individual cost shown in Table 3.2 to calculate startup costs for THUNDER with respect to the number of clients. Addi-

tionally, Equation (3.3) calculates the cost of a traditional set of workstations with respect to the number of clients. We have plotted the startup costs for THUNDER alongside that of a traditional set of workstations using the formulas from Equations (3.2) and (3.3) in Figure 3.7. Percent savings of THUNDER over a traditional set of workstations has been calculated in Equation (3.4) as well as plotted in Figure 3.8, which seems to plateau at around 50% as shown in Figure 3.9.

$$C_{cloud}(n) = C_{comp} + C_{image} + C_{head} + C_{store} + \lfloor \frac{n}{8} \rfloor \times C_{comp} + \lfloor \frac{n}{50} \rfloor \times C_{store} + (C_{client} + C_{disp} + C_{kbm}) \times n \tag{3.2}$$

Where:

$C_{cloud}(n)$ : Cost of THUNDER with Respect to n Clients
$C_{comp}$ : Cost of the THUNDER Compute Node
$C_{image}$ : Cost of the THUNDER Image Repository
$C_{head}$ : Cost of the THUNDER Head Node
$C_{store}$ : Cost of the THUNDER Storage Node
$C_{client}$ : Cost of a Thin Client Device
$C_{disp}$ : Cost of a Display
$C_{kbm}$ : Cost of a Keyboard and Mouse

$$C_{traditional}(n) = n \times C_{ws} \tag{3.3}$$

Where:

$C_{traditional}(n)$ : Cost of a Traditional Network of Workstations with Respect to n Clients
$C_{ws}$ : Cost of One Desktop Workstation

As shown in Figures 3.7 to 3.9 and eqs. (3.2) to (3.4), an organization with at least twelve employees can decrease startup costs by utilizing the THUNDER architecture with thin client devices. An organization with less than twelve employees could initially spend less money on a more traditional private network of workstations. However, if such an organization anticipates any

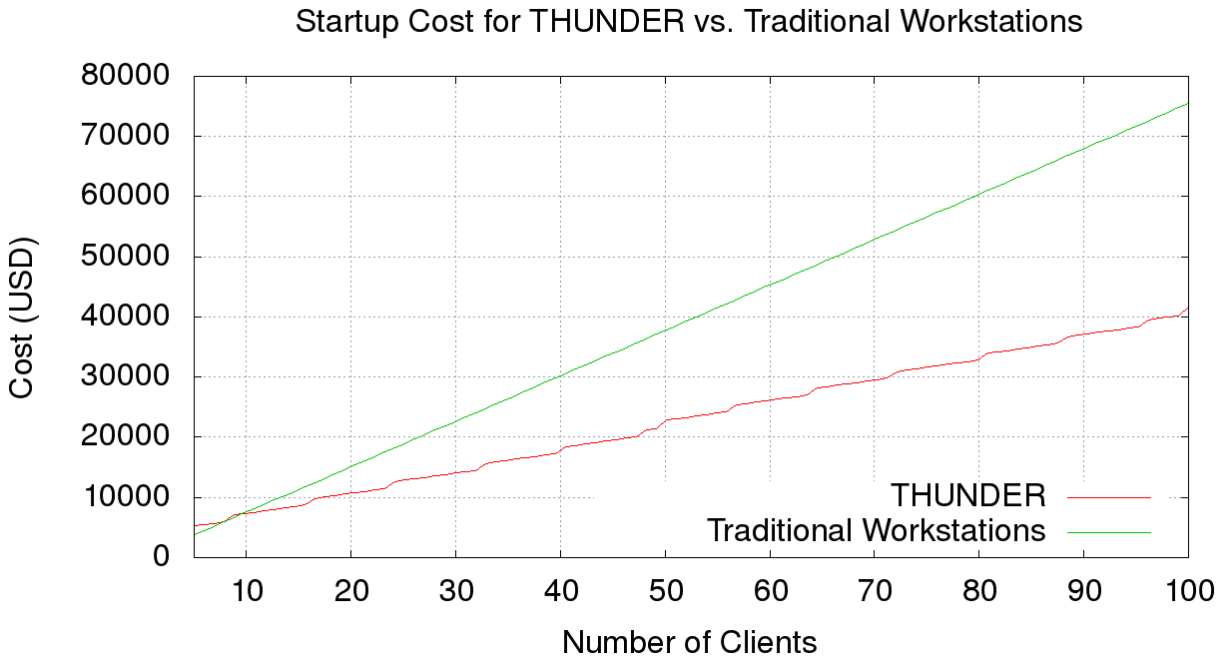



Figure 3.7: Startup cost comparison of THUNDER versus a traditional set of workstations with respect to N clients.

$$S(n) = \frac{C_{traditional}(n) - C_{cloud}(n)}{C_{traditional(n)}} \times 100\% \tag{3.4}$$

Where:

$S(n)$ : Percent savings of THUNDER over traditional workstations

sort of future expansion, then the traditional workstation setup would become less cost efficient when compared to that of the THUNDER architecture.

The THUNDER architecture design describes a new model for simplifying private cloud deployments. We believe that the THUNDER model can be successfully used to decrease costs in struggling nonprofit organizations. Additionally, an important goal of the THUNDER architecture is for THUNDER to be easy to deploy and easy to manage, which we believe is beneficial

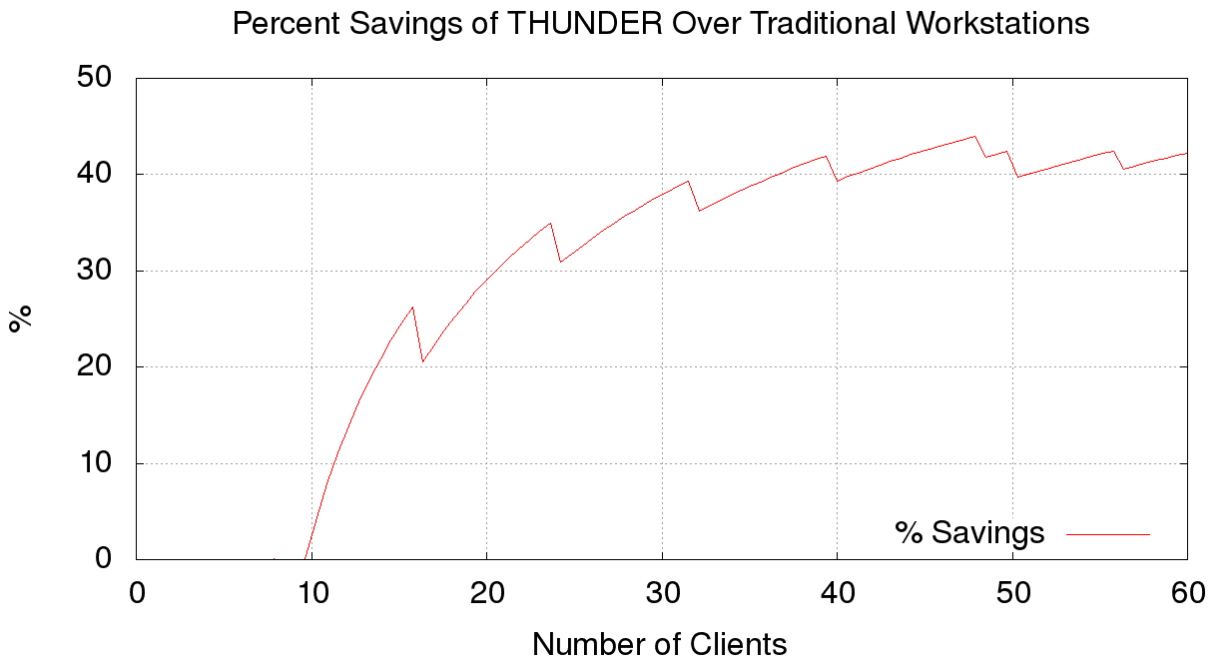



Figure 3.8: Percent savings of THUNDER over a traditional set of workstations.

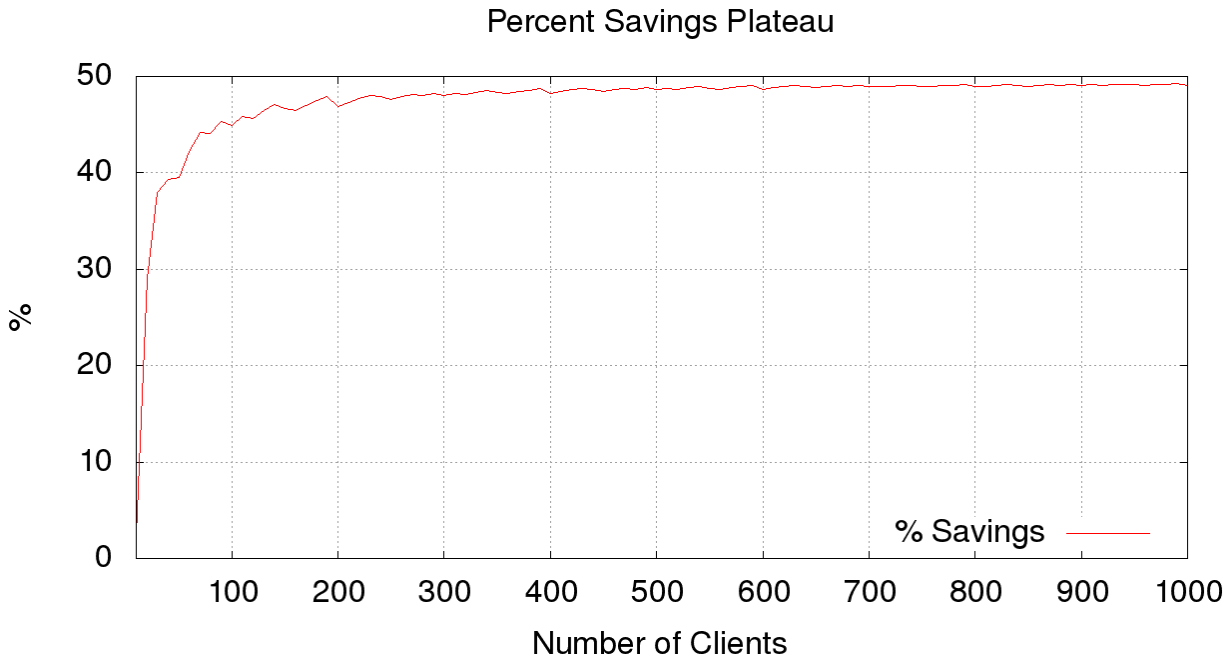



Figure 3.9: Percent savings graph showing plateau of 50%.

| Item | Cost |
|---|---|
| THUNDER Compute Node | $900.00 |
| THUNDER Image Repo | $1156.00 |
| THUNDER Head Node | $800.00 |
| THUNDER Storage Node | $1156.00 |
| Thin Client Device | $120.00 |
| Display | $100.00 |
| Keyboard and Mouse | $25.00 |
| Dell Workstation | $755.00 |

Table 3.2: Initial costs for each component of THUNDER compared to a single workstation.

for organizations that do not employ an IT staff. In the next chapter we introduce the THUNDER middleware implementation, which forms the infrastructure necessary to facilitate inter-node communication of the THUNDER architecture.

## 3.10 Improvements to Usability Over Other Architectures

By integrating support for browser based virtual machine accessibility features we believe that THUNDER is a better solution for non-profits and inexperienced users. Although, both OpenStack and Eucalyptus can be modified to support similar services, neither of these architectures provide the out-of-the-box experience that is provided by THUNDER with respect to accessibility of virtual machines. Additionally, fault tolerance of cloud servers using a multicasting approach allows for cloud servers to be able to be automatically recovered from intermittent connectivity issues or power failure without user intervention. Additionally, THUNDER provides a load consolidation schedule Section 3.6 and a heuristic approach for selecting the location of virtual machine instances. As will be shown in Chapter 6, this approach outperforms the more common round-robin approach, and further aids the usability of the system due to less frequent instantiation failures.

# Chapter 4

# MIDDLEWARE FOR REMOTE METHOD INVOCATION

Management of resources is a key challenge in the development of a cloud architecture. Moreover, there is a necessity for minimizing the complexity and overhead in management solutions in addition to facilitating attributes of cloud computing, such as scalability and elasticity. Another desirable quality of a cloud management solution is modularity. We define modularity as the ability to easily add or remove components on-the-fly without the necessity to reconfigure any services or systems. The field of cloud management exists within several overlapping domains, which include service management, system deployment, access control management, and others.

When examining the current state of the art in cloud management, there are few options. As mentioned earlier, we are confined to free and open source (FOSS) cloud implementations, such as Eucalyptus [Eucalyptus Systems, 2013b], and Openstack [OpenStack Foundation, 2013a]. Cloud management solutions used in closed-source, and often more popular cloud architectures, such as Amazon EC2, are often times out of reach from an academic and research perspective due to their cost and closed nature. However, there has been an effort to make private IaaS cloud architectures compatible with Amazon EC2 by implementing a compatible API and command line tools, such as euca2ools [Eucalyptus Systems, 2013a] and Nova [OpenStack Foundation, 2013b], respectively. The compatibility of API's makes it easy to form a basis of comparison between different architectures. However this compatibility may also serve as a downfall because if one API suffers from a bug, it may also be present in other API's.

In this chapter we introduce the middleware layer of the THUNDER cloud stack. We utilize a threaded, message-oriented, event-driven remote procedure call service to form the backbone of all THUNDER nodes. In this methodology we employ a publisher and subscriber model by which one or more subscribers connect to one publisher responsible for receiving and relaying messages. Messages in this context signify remote procedure names and arguments, and are passed over the local network between nodes. Nodes receiving such messages look up the method name in a lookup table, and if the name has been found it will then invoke the appropriate procedure. The return value of the invocation is then sent back to the subscriber. Similarly, each service is responsible for registering one or more events, which are simply pushed into a dictionary of one-to-one mappings between procedure names and the procedure definition.

## 4.1 Architecture

The middleware layer of THUNDER is composed of several tightly coupled services. At the lowest level we utilize a socket server which accepts external connections using a thread pool. As connections are established the THUNDER middleware will enqueue the incoming message, which will reside in the queue until a thread becomes available to handle the request. Once the request has been handled the thread will return to the thread pool and continue to wait. The incoming message is always a JSON object, which is parsed and interpreted by the middleware and the appropriate response is generated and sent back to the sender. This process is illustrated in Figure 4.1.

The handler examines the message to determine whether or not there is an enclosing WebSocket frame. If the message is enclosed in a WebSocket frame then the frame is parsed and the data is decoded using the ”Sec-WebSocket-Key” field found in the HTTP header. Messages enclosed in a WebSocket frame indicate that the message is coming from a web browser, and there-

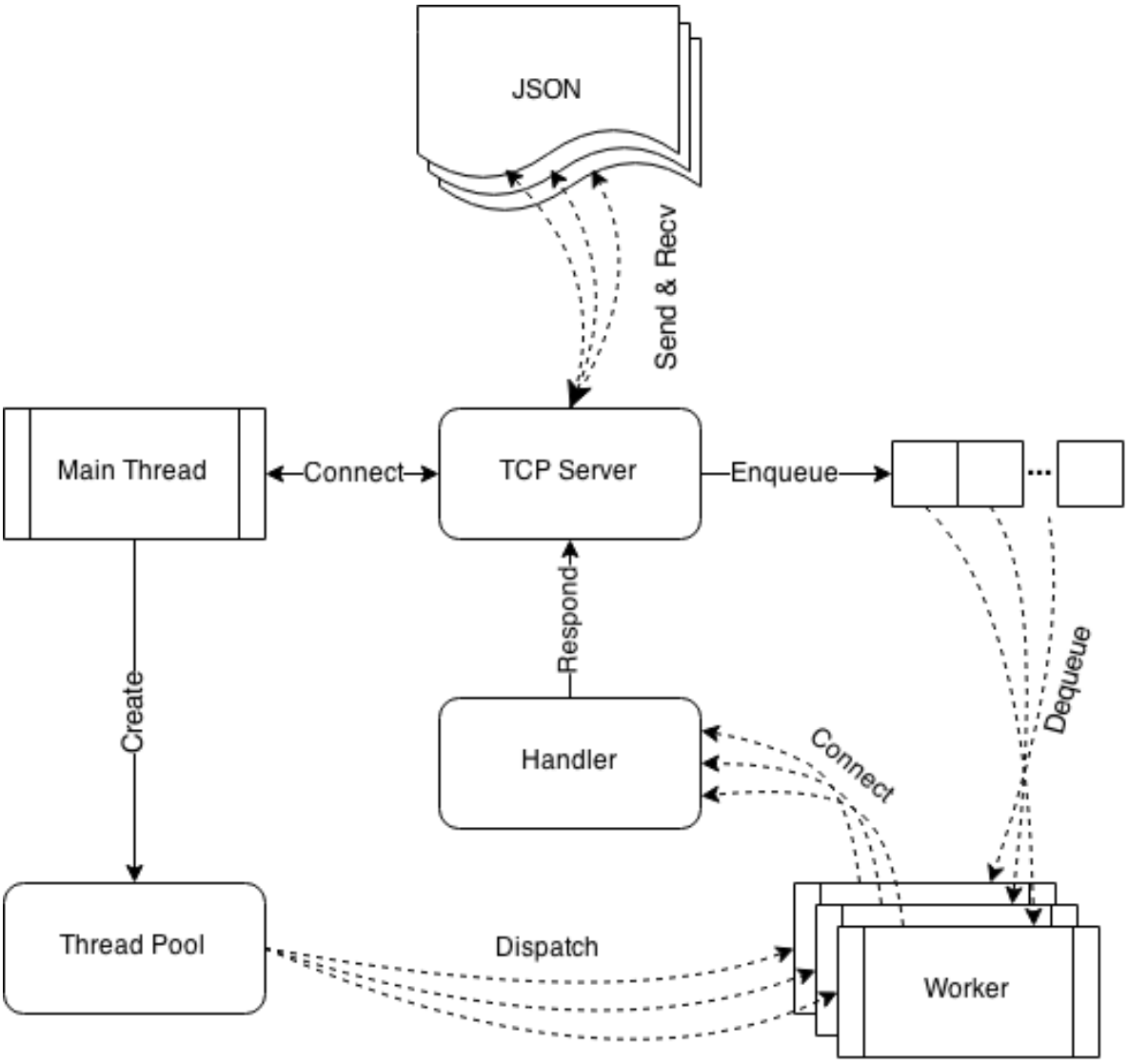



Figure 4.1: Diagram of connection request handler.

fore restrictions are put into place to prevent the RPC call embedded in the message from calling any insecure method. We only allow messages that are decoded from WebSocket frames to execute RPC calls that are directly related to the web interface for administrator and user accounts, such as requesting a virtual machine and polling nodes for utilization statistics. Messages that are

not enclosed by a WebSocket frame are those that occur due to inter-node communication. These messages are for internal administrative purposes, such as node authentication and load balancing.

## 4.2 Communication Scheme

In contrast to the methodologies used by Eucalyptus, OpenStack, and presumably Amazon, our cloud middleware API addresses resource management in a simplified and more direct manner. The hierarchy of controllers used in Eucalyptus introduces extra complexity that we have deemed unnecessary. For this reason, our solution utilizes a simple publisher and subscriber model [Birman and Joseph, 1987] by which compute, storage, and image repository nodes may construct a closed network. The publisher and subscriber model operates in conjunction with event driven programming, which allows events to be triggered via remote procedure calls over the private network to groups of nodes subscribed to the head node. Figure 4.2 shows the logical topology and lines of communication constructed using this model.

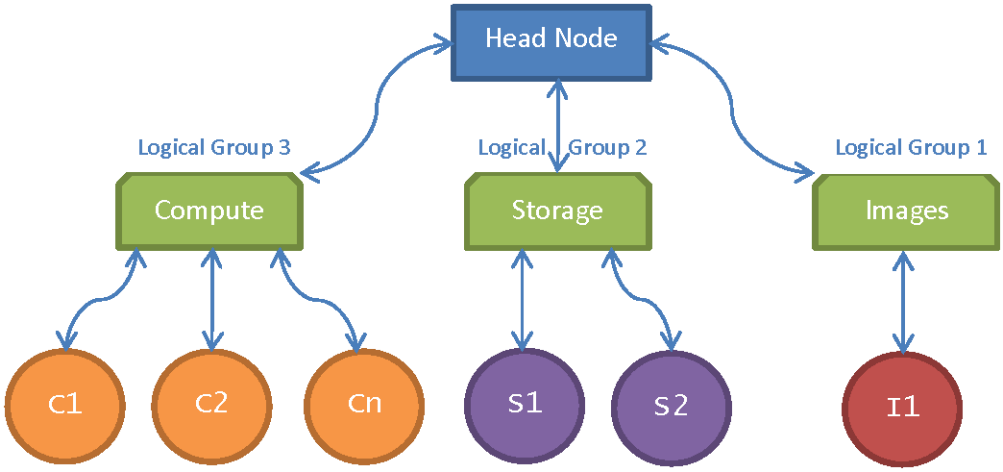



Figure 4.2: Logical topology showing group-wise membership in publisher/subscriber model.

In constructing the communication in this manner we are able to broadcast messages to logical groups in order to gather metadata about the nodes subscribed to that group. Message

passing is useful for retrieving the status of nodes, including virtual machine utilization, CPU and memory utilization, and other details pertaining to each logical group. Additionally, we are able to transmit messages to individual nodes in order to facilitate virtual machine instantiation, storage allocation, image transfer, and other functions that pertain to individual nodes.

### 4.2.1 Registration of Nodes

Communication between nodes utilizes non-persistent socket connections, such that the head node maintains a static pre-determined port for receiving messages, while other nodes may use any available port on the system. Thus, each node in the cloud, excluding the head node automatically selects an available port at boot time. Initial communication between nodes is done during boot time to establish a connection to the head node. We utilize a methodology for automatically finding and connecting to the head node via a multicast authentication request (Section 3.4). Once a communication link is established between a node and the head node, the node will request membership within a specific logical group, after which communication between the head node and that logical group will contain the node in question. The use of multicast messaging for authentication introduces challenges with regard to head node replication. If the head node malfunctions and a replica is needed to take its place, then every system in the cloud would need to select the same replica, which may not be guaranteed unless only one replica is allowed to respond to multicast requests. Therefore, we ensure that only one replica is allowed to take control of the cloud, while other replicas must return to an idle state.

The registration methodology used in our middleware solution differs from the methodology used by Eucalyptus and OpenStack. For example, Eucalyptus relies upon command line tools to perform RSA keysharing and for establishing membership with a particular controller. We do not perform key sharing, and instead rely upon a pre-shared secret key and plaintext passphrase,

such that each message is prepended with the passphrase. This approach is commonly known as challenge-response authentication. Upon transmission of the message the passphrase is encrypted with the pre-shared secret key and sent back to the originating node. We validate the message by comparing the cyphertext produced by the receiver and the cyphertext produced by the sender. Thus, we do not rely upon manual sharing of RSA keys beforehand, and instead we eliminate the need for RSA keys altogether and utilize a more dynamic approach for validation of communication during the registration process.

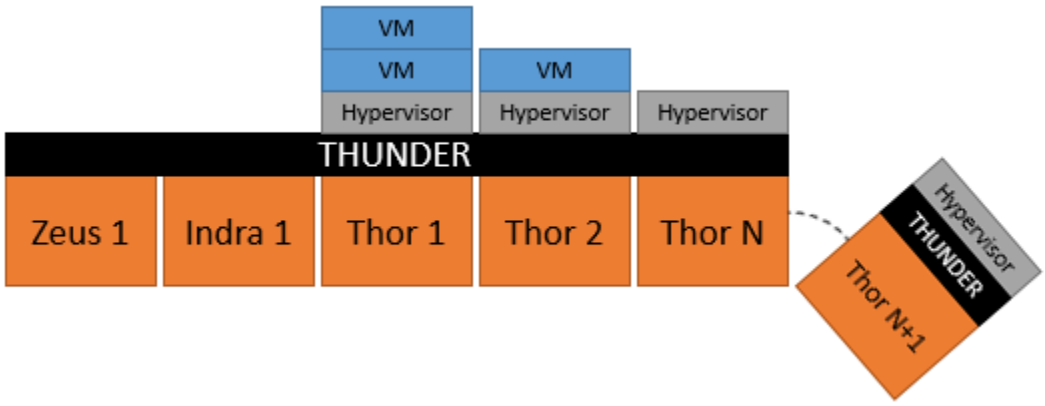



Figure 4.3: New node registration emphasizing independent operation of node Thor N+1.

As new nodes attach to a network for which a sibling node is a THUNDER Zeus controller, a number of events begin to take place. Firstly, we assume the new node has the THUNDER middleware process in a running state and is transmitting requests over the local multicast group for a connection to a Zeus controller. Before the Zeus controller responds to the request sent out by the new node we assume that this node is independent and not operating within a logical cluster of nodes. Figure 4.3 illustrates a new node, labeled "Thor N+1" operating independently before joining a logical cluster. Once the Zeus node responds to the request, an authentication request is sent by the new node and identification of each node is exchanged. The Zeus controller then

stores the identification of the new node as a tuple of IP address and port number for which the THUNDER process is accepting TCP connections. The new node stores the identification of the Zeus controller in the same way. The entire communication process for registration is facilitated over multicast; however, all communication between nodes after this process has completed is facilitated by TCP connections.

## 4.3 Middleware API

Our methodology for constructing a middleware API for cloud resource management centers around the notion of simplicity. In order to facilitate a simple middleware solution, our API is designed to provide a powerful interface for cloud management while not introducing excessive code overhead. This API is utilized within our private IaaS cloud architecture as a means for communication, management of resources via RPC, and interaction with our cloud interface. Figures 4.4 and 4.5 illustrate the manner in which the API is accessed. Although the code examples presented here are incomplete, they illustrate the simplicity of creating events to be triggered by the system for management of resources.

In Figure 4.4 we present sample code for the creation of a head management node, which is responsible for issuing commands to the rest of the cloud. The head node establishes a special event, which is triggered by the cloud interface for gathering data and interacting with the resources. In Figure 4.5 we present sample code for the creation of a compute node with events written for instantiation of virtual machine images and for retrieving the status of the node. Although, we do not present code for the implementation of storage or image repository nodes, the implementations are similar to that of the compute node. In addition, the code examples presented in this paper show only a subset of the functionality contained within the production code.

The API presented in Figures 4.4 and 4.5 provides a powerful interface for implementing

```
from thunder import *
import sys

# Invoke a function and return the result.
def invoke(*params):
   tag = params[0]
   data = params[1]

   if (len(data) > 1):
      if (data[0] == "CONTROL"):
         if (data[1] == "GET_CLIENT_LIST"):
            return server.getClientList()
         elif (data[1] == "GET_CLUSTER_LIST"):
            return server.getClusterList()
      ...
   # Invalid number of params.  Return error code.
   else:
      return None

# Instantiate Thunder and register the invoke event
server = ThunderRPC(role = 'PUBLISHER', group = 'Cloud')
server.registerEvent('INVOKE', invoke)
```

Figure 4.4: Example head node service.

```
from thunder import *

def instantiate(*params):
   # Instantiate a virtual machine
   ...

def destroy(*params):
   # Destroy a virtual machine instance
   ...

client = ThunderRPC()
client.registerEvent("INSTANTIATE", instantiate)
client.registerEvent("DESTROY", destroy)
client.findPublisher()
```

Figure 4.5: Example compute node service.

private cloud architectures because it can be extended easily to support many different distributed systems that require a managed approach. By means of event triggering over a private network we are able to instantiate virtual machine images, mount storage volumes, retrieve node status data, transfer virtual machine images, monitor activity, and more. The implementation of the system is completely dependent upon the developer's needs and may even be used in distributed systems, which may or may not be implemented as a cloud architecture. Remote procedure calls allow a program to call a procedure from another program, and then return the result of the remote procedure

to the caller. ThunderRPC performs this task by utilizing JSON as a descriptor language, which is a common approach to implementing RPC. A descriptor language is a standard domain specific language, which allows different pieces of software to access remote procedures in a standardized way.

## 4.4 Interfacing

In Section 4.3 we introduced our middleware API for managing cloud resources. However, another important component in our design is a reasonable way to interface with the middleware solution. Although, the middleware API solution is completely independent from the interface, we have chosen to use a message passing approach. In this approach our web interface, which is written in PHP, HTML5 and Javascript, connects to the head node in order to trigger the "INVOKE" event (see fig. 4.4). By interfacing with the head node we are able to pass messages to groups or individual nodes in order to manage the resources of that node and receive responses. The ability to interface in this manner allows our interface to remain decoupled from the logical implementation, while allowing for flexibility in the interface and user experience.

In order to allow for greater flexibility and ease of use we have designed the interface to utilize jQuery [The jQuery Foundation, 2015] and Javascript for dynamic content. When requesting data about cloud resources, our approach is to construct JSON objects on the fly, which are in turn processed by the web interface dynamically in order to remove the necessity for refreshing the web page. Figure 4.6 shows an example PHP script for interfacing with the resources in the manner described in this section.

The PHP interface presented in Figure 4.6 illustrates the methodology behind how we may capture and display data about the nodes, as well as provide a means for user interaction in resource allocation and management. Although we do not present the full source code here, additional

```
class Communication {
  var $sock;

  /* Insert socket connect/send/receive code here */

  function getStatus($group) {
    $this->connect();
    $this->send('INVOKE GROUP ' . $group . ' STATUS');
    $response = $this->receive();
    $this->close();
    if ($response == '') {return array();}
    return explode('\;',$response);
  }

  function getClusterList() {
    $this->connect();
    $this->send('INVOKE CONTROL GET_CLUSTER_LIST');
    $response = $this->receive();
    $this->close();
    if ($response == '') {return array();}
    return explode('\;',$response);
  }
}
```

Figure 4.6: Example communication interface in PHP.

functions were written and appear in Appendix P. One important aspect of this system is that at all levels the mode of communication remains consistent. Every message sent is implemented via non-persistent socket connections. This allows for greater data consistency without modifying the semantics of messages between the different systems.

### 4.4.1 Websocket Interface

The power behind ThunderRPC is that it has built in support for the WebSocket [Fette and Melnikov, 2011] protocol. The WebSocket protocol allows direct socket connections to be made between a client web browser and a web server. WebSockets have become the new standard for dynamic web content and they simplify the flow of interactive content, where previously asynchronous Javascript and XML (AJAX) was the industry standard. THUNDER manages a WebSocket server, which intercepts incoming connections in order to identify valid HTTP header information. Within the header of the incoming data stream is a key, which must be used to establish the connection by the accepting web server. When a valid HTTP header is found in the

incoming data stream, the THUNDER WebSocket server attempts to perform a handshake, in which a magic number is appended to the base-64 encoded key and then hashed via SHA-1. If the client accepts the handshake, then the WebSocket connection is established and messages may be transmitted between the client and server in full duplex mode.

By utilizing the WebSocket protocol, THUNDER maintains loose coupling between client and server in a way that is secure, and which provides a means to leverage THUNDER events via a web browser. This methodology differs from other open source general-purpose cloud architectures. Neither Eucalyptus nor OpenStack uses the WebSocket protocol for interfacing with the cloud. Instead, OpenStack and Eucalyptus both use a WSGI (Web Server Gateway Interface) implementation for interfacing over HTTP. Figure 4.7 shows the model by which THUNDER communicates using the WebSocket protocol, and Figure 4.8 shows the Javascript event handler used for responding to messages from the THUNDER WebSocket server to the client.

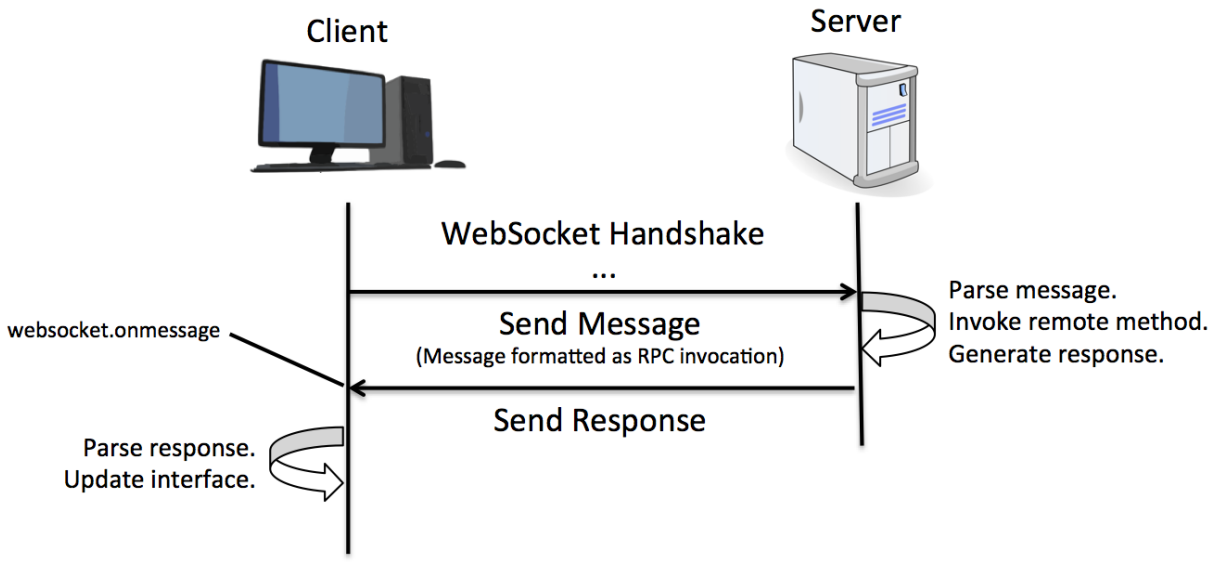



Figure 4.7: Client/Server Communication via WebSockets.

```
...
// Insert any helper functions here.
...

window.onload+=start();

function start()
   {
   // Update the list of groups/clusters in the DOM
   event('GROUPNAMES');
   }

function event(msg)
   {
   // Open a WebSocket connection to the server on port 6667
   var ws = new WebSocket("ws://" + location.host + ":6667");
   // When a connection is opened, send the message.
   ws.onopen = function() { ws.send(msg); };
   // Get the response back from the server.
   ws.onmessage = function(evt) { processEvent(msg,JSON.parse(evt.data)); };
   }

// process the response and update the DOM
function processEvent(msg,response) {
   // we need to parse the response and add links to the clusters div
   if (msg['cmd'] == 'GROUPNAMES')
      {
      // If the original message is 'GROUPNAMES', parse the
      // response as a list of groups/clusters
      ...
      // Update the UI
      }
   else if (msg['cmd]== 'NODESINGROUP')
      {
      // If the original message is 'NODESINGROUP', parse the
      // response as a list of nodes in a particular group/cluster
      ...
      // Update the UI
      }
   else if (msg['cmd'] == "EXECNODE")
      {
      // We've executed an event on a node, lets deal with the response.
      ...
      // Update the UI
      }
   else if (msg['cmd'] == ...)
      {
      ...
      }
   }
```

Figure 4.8: Event Handler for Responding to Messages from THUNDER Over WebSocket Protocol.

## 4.5 Built-In Events for Administrative Tasks

Some tasks used in managing THUNDER are defined as built-in events that are not able to be customized at run time by the service itself. These events are tightly coupled to the data man-

aged by ThunderRPC, such as the list of active cloud servers, the list of service-defined events, and details specific to service authentication and deployment. Most importantly are the events which manage the deployment of additional cloud servers. Upon initialization of the controller node, ThunderRPC manages the services for deployment of the base operating system and software for additional cloud servers. This is accomplished with a web accessible repository containing the base operating system used on each cloud server, and a preseed configuration which is customized automatically at run time. Customizations made to the preseed configuration allow for the deployment process to know the correct IP address of the controller node, and which service type is actively being deployed. Additionally, these customizations are made per MAC address of each target system, which allows for deployment to multiple systems possible. The process by which this is accomplished is as follows:

- One or more MAC addresses along with the server role are selected by the user and transmitted to the controller node.
- Upon receiving notification, the controller node writes custom configurations for each node.
- The controller node begins to poll the DHCP server for IP renewal notifications.
- The cloud server(s) must be manually restarted by the user.
- The controller node detects an IP renewal on boot of the cloud server(s). Configuration for this server is reverted to local boot to prevent redeployment.
- The cloud server receives configuration details using the NetBoot [Alexander and Droms, 1993, 1997] protocol, which points to the appropriate kernel, ramdisk image, and preseed options made available by the controller node.

- The cloud server is automatically provisioned and reboots. The appropriate THUNDER service starts automatically.

## 4.6 Improvements to Usability Over Other Architectures

THUNDER provides increased usability over Eucalyptus and OpenStack. The hallmark of communication between nodes in Eucalpytus is the use of SSH connections between head cloud controllers and the service controller nodes. By utilizing SSH connections cloud controllers may invoke processes on service controller nodes remotely, but not before every the service controller node and the cloud controller have exchanged RSA keys. The RSA keysharing process is not automated, and must be executed manually by the user. Additionally, neither OpenStack nor Eucalyptus supports automated deployment of new cloud servers. There are third party solutions, such as Puppet [Puppet Labs, 2015] which may be used in conjunction with OpenStack or Eucalyptus in order to support an automated multi-node deployment, but these tools are not officially supported by either architecture. THUNDER provides a level of automation which is built in to the middleware layer to support deployment, node registration and metadata aggregation that is not provided natively by other architectures.

# Chapter 5

# EMPIRICAL STUDY ON THE EASE OF DEPLOYMENT

Public clouds are becoming more popular with Amazon, Google, and Microsoft leading the way in providing public cloud services. However, struggling organizations may not have the funds to pay the costs associated with subscribing to a public cloud. Local and private cloud architectures have not been widely adopted by organizations, except for those with an appropriate budget and technical resources. One of the reasons for this has been attributed to the difficulties associated with deploying local clouds. Additionally, it can be argued that the usability of local clouds is diminished from that of public clouds. One aspect of local private clouds that may be of benefit to an organization is the fact that existing computing hardware can be re-purposed as cloud servers. A recent study [Stein et al., 2013] on the pedagogical applicability of cloud computing in public schools illuminates several challenges which make local clouds less than appealing. According to the study, many of the participants were seemingly unwilling to use the new technology, and many participants did not understand the benefits associated with cloud computing. In order to determine if THUNDER helps to alleviate some challenges shown in previous studies we have performed a series of empirical studies. These studies targeted both university students, and professionals either working for non-profit organizations or in fields related to non-profit organizations. The studies presented in this chapter were approved by the University of Alabama Internal Review Board [Loewen and Vrbsky, 2014] and IRB documentation is located in Chapters B to F. In each study the participants were asked the following set of questions:

**Quantitative Questions**

1. How much time was spent?

2. How difficult was the deployment?

3. How clear were the instructions?

**Qualitative Questions**

1. Which instructions, if any, were unclear?

2. What could be done, if anything, to simplify the deployment process?

## 5.1 Survey of University Students

We performed an empirical study by which two groups composed of both undergraduate and graduate students in computer science and computer engineering were asked to deploy both a general purpose cloud architecture and THUNDER. These groups will be referred to in the remainder of this section as *group one* and *group two*. Group one consisted of twenty-four undergraduate students which were asked to compare the deployment process of Eucalyptus to that of THUNDER. The optional survey given to students after the deployment process completed was given anonymously. Group two consisted of fourteen undergraduate students which were asked to compare the deployment process of OpenStack to that of THUNDER. Similarly to group one, the surveys given to group two were both optional and anonymous. Qualitative results from the combined thirty-eight participant study show that the deployment process of THUNDER is easier and less time consuming than that of both Eucalyptus and OpenStack. Results of this study are presented in Sections 5.1.1 and 5.1.2

### 5.1.1 Eucalyptus Deployment

Results from the Eucalyptus deployment study are shown in Figures 5.1 to 5.3. In these results we see that Eucalyptus is not very favorable when compared to THUNDER.

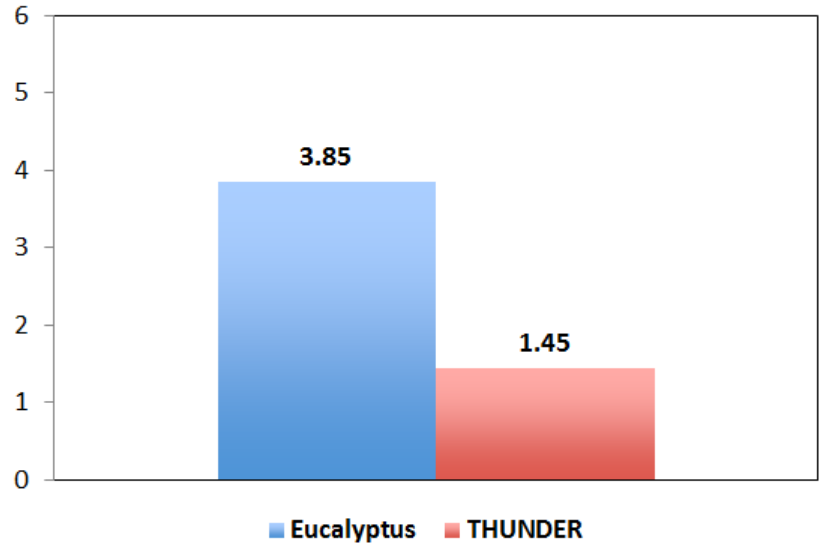



Figure 5.1: Quantitative Results: Time spent to deploy Eucalyptus and THUNDER.

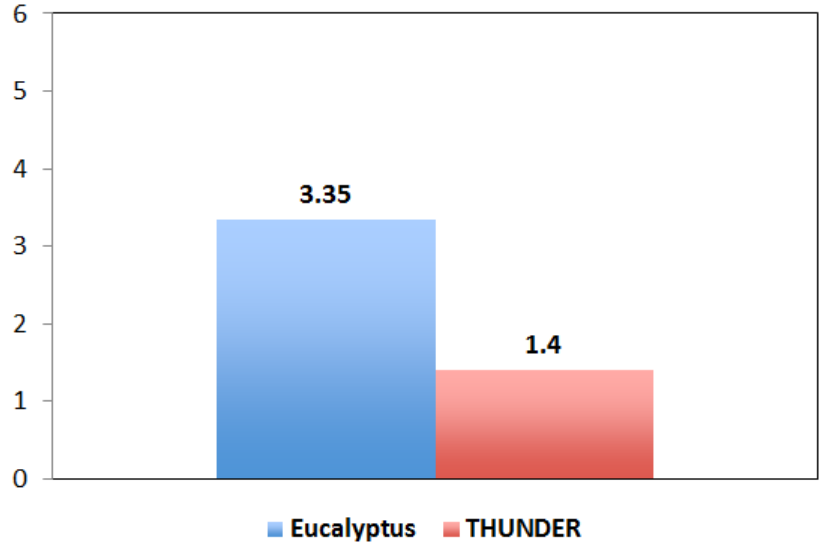



Figure 5.2: Quantitative Results: Perceived difficulty of Eucalyptus and THUNDER deployment.

Twenty-four participants from the University of Alabama in group one were given the

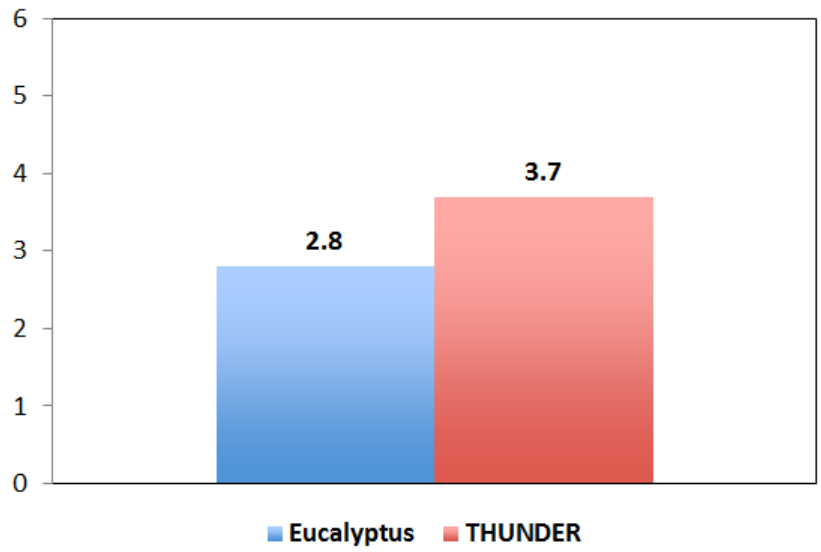



Figure 5.3: Quantitative Results: Clarity of Eucalyptus and THUNDER instructions.

official deployment instructions from Eucalyptus systems as well as instructions for deploying THUNDER. The first part of the study required that the participants deploy a minimal one-node Eucalyptus cloud. The second part of the study required that the participants deploy a one-node THUNDER cloud. After the deployment the participants were given a survey which rated their experience during the deployment process. The survey consisted of basic demographic questions followed by questions related to the deployment. At the end of the survey were fields which allowed participants to add any additional comments related to their experience in deploying the two architectures.

#### *5.1.1.1 Process*

Participants in this study first deployed Eucalyptus. The steps for installing Eucalyptus are quite lengthy and involve setting up RSA keys for authentication, installation of Eucalyptus components, the use of a text editor for modifying the default Eucalyptus configuration, and manually starting each component in the correct order. The Eucalyptus deployment consisted of ten

steps each requiring participants to complete multiple sub-tasks that resulted in one to two hours of labor. Some participants failed to successfully install Eucalyptus on their first attempt. These participants decided to restart the Eucalyptus deployment but gave up shortly after. After deploying Eucalyptus, the participants were asked to deploy THUNDER. The THUNDER deployment consisted of two steps. The first step involved downloading the installation script, and the second step involved the execution of the script by clicking a button. The THUNDER deployment resulted in five minutes of labor and on average fifteen minutes of automated tasks requiring no user intervention. Additionally, participants were encouraged to test of the installation by viewing the web interface in a web browser.

#### *5.1.1.2 Quantitative Results*

Based on the results it is clear that the participants had a better overall experience in deploying the THUNDER architecture. Figure 5.1 shows the time spent on deploying each system rated on a scale of one to five with one being the least amount of time and five being the most amount of time. Figure 5.2 shows the amount of difficulty perceived by the participant during the deployment process, one being the least difficult and five being the most difficult. Figure 5.3 shows the clarity of the instructions given to the participant, one being the least clear and five being the most clear. These results appear to be in favor of the deployment system of THUNDER for each of the three metrics evaluated.

#### *5.1.1.3 Qualitative Results*

In addition to the numeric scoring for each architecture, we performed a cluster analysis on the comments recorded by the participants. By clustering the comments we are able to determine the overall user experience for each architecture qualitatively. The survey participants were also given the option to leave any additional comments about Eucalyptus and THUNDER. The results

of the qualitative questions are shown in Figures 5.4 and 5.5. Based on the answers provided by the survey participants the following categories were derived:

**Question 1:**

1. None
2. Inaccuracies
3. Component Naming

**Question 2:**

1. None
2. GUI Interface
3. Improve Instructions
4. Automation
5. Speed

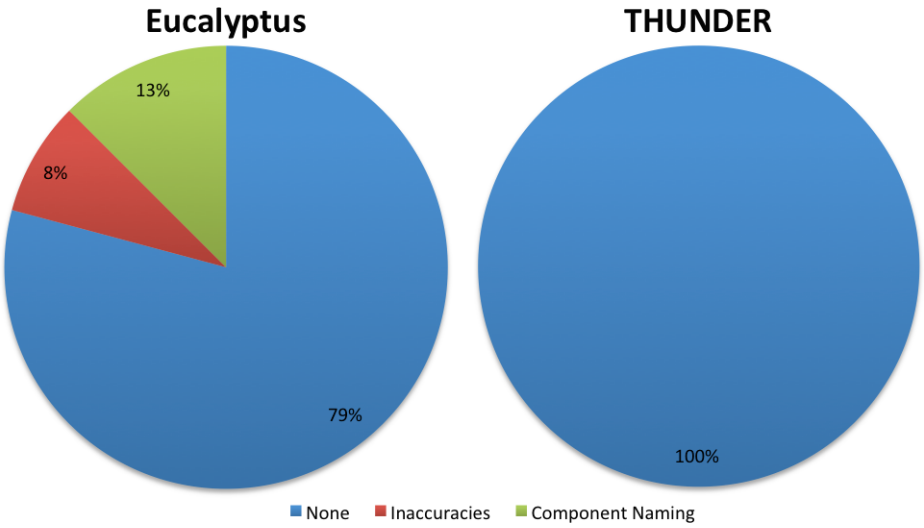



Figure 5.4: Qualitative Question 1: Eucalyptus Results.

### *5.1.1.4 Additional Comments*

The additional comments recorded by the survey participants indicate that the deployment system used by Eucalyptus is not particularly user friendly, but the deployment system used by

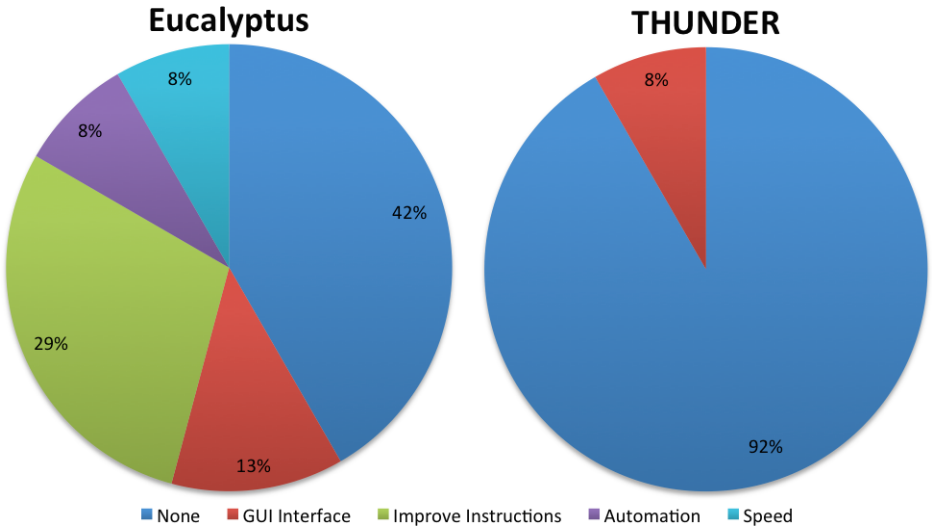



Figure 5.5: Qualitative Question 2: Eucalyptus Results.

THUNDER is easy and simple. We have selected a subset of comments about each architecture as an illustration.

**Eucalyptus**

- "*It was hard for a masters student in computer science, much less anyone else.*"
- "*It was too difficult and required too many commands.*"

**THUNDER**

- "*Very simple and much easier than deploying Eucalyptus.*"
- "*Very fast and lightweight.*"

The results from the additional comments show that 100% of the comments about THUNDER were positive and 100% of the comments about Eucalyptus were negative. However, only 54% of the participants left additional comments about THUNDER and only 37% of the participants left additional comments about Eucalyptus.

#### *5.1.1.5 Synthesis*

Data gathered from the empirical study on the deployment of Eucalptus and THUNDER show that the volunteers expressed far more concerns with the deployment process of Eucalyptus. As shown in Figure 5.4 8% of the volunteers experienced difficulty with the deployment instructions provided by Eucalyptus, and 13% of the volunteers experienced difficulty with naming of the components in Eucalyptus. The latter difficulty is explained by the fact that when deploying Eucalyptus the user must give each node a unique name such that the name may be used to identify nodes during the authentication process. If named improperly or if any typos occurred during the authentication process then authentication will fail. Additionally, in Figure 5.5 we see that only 42% of volunteers did not have major complaints with the deployment process of Eucalyptus. 29% of the volunteers had major complaints regarding the quality of the instructions for deploying Eucalyptus. 13% of volunteers had major complaints regarding the GUI interface of Eucalyptus. 8% of volunteers had major complaints about speed of deployment as well as frustration over the lack of automation. Only 8% of volunteers had major complaints about the GUI interface of THUNDER, while the rest of the volunteers had no complaints about THUNDER.

### 5.1.2 OpenStack Deployment

Using the same empirical methods used to compare the Eucalyptus deployment process to THUNDER, we also compare the deployment process of OpenStack [OpenStack Foundation, 2013a], which is another general-purpose cloud architecture. In this study, fourteen participants were given deployment instructions for OpenStack as well as instructions for deploying THUNDER. After deploying both architectures the participants were given an optional survey. The survey consisted of basic demographic questions followed by questions related to the deployment.

#### 5.1.2.1 *Quantitative Results*

The participants had a better overall experience in deploying the THUNDER architecture when compared to OpenStack. Figure 5.6 shows the time spent on deploying each system, one being the least amount of time and five being the most amount of time. Figure 5.7 shows the amount of difficulty perceived by the participant during the deployment process, one being the least difficult and five being the most difficult. Figure 5.8 shows the clarity of the instructions given to the participant, one being the least clear and five being the most clear. These results also appear to be in favor of the deployment system of THUNDER for each of the three metrics evaluated.

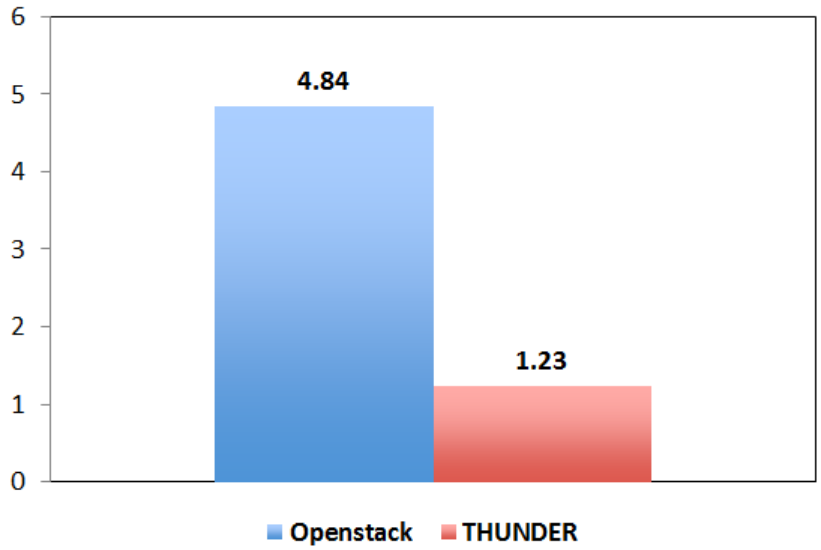



Figure 5.6: Quantitative Results: Time spent to deploy OpenStack and THUNDER.

#### 5.1.2.2 *Qualitative Results*

We performed cluster analysis on the comments recorded by the participants of the OpenStack deployment study. By clustering the comments we are able to determine the overall user experience for each architecture qualitatively. In addition to the two core sets of questions, the

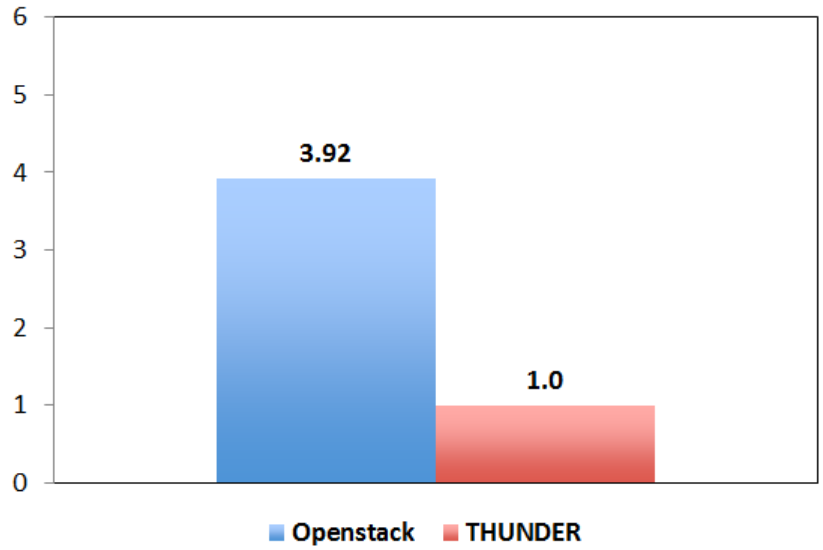



Figure 5.7: Quantitative Results: Perceived difficulty of OpenStack and THUNDER deployment.

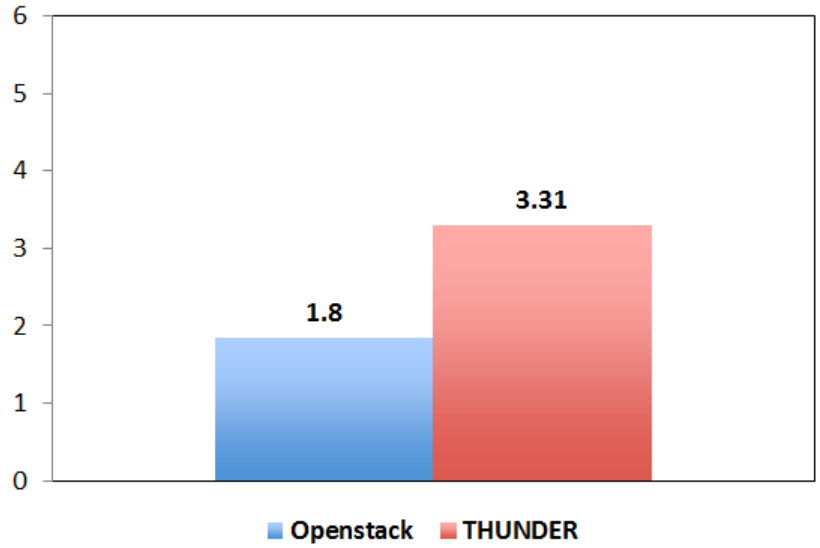



Figure 5.8: Quantitative Results: Clarity of OpenStack and THUNDER instructions.

survey participants were also given the option to leave any additional comments about OpenStack and THUNDER. The results of the qualitative questions are shown in Figures 5.9 and 5.10. Based on the answers provided by the survey participants the following categories were derived:

**Question 1:**

1. None
2. Configuration Editing
3. Networking
4. Authentication
5. Component Naming

**Question 2:**

1. None
2. Improve Instructions
3. Automation

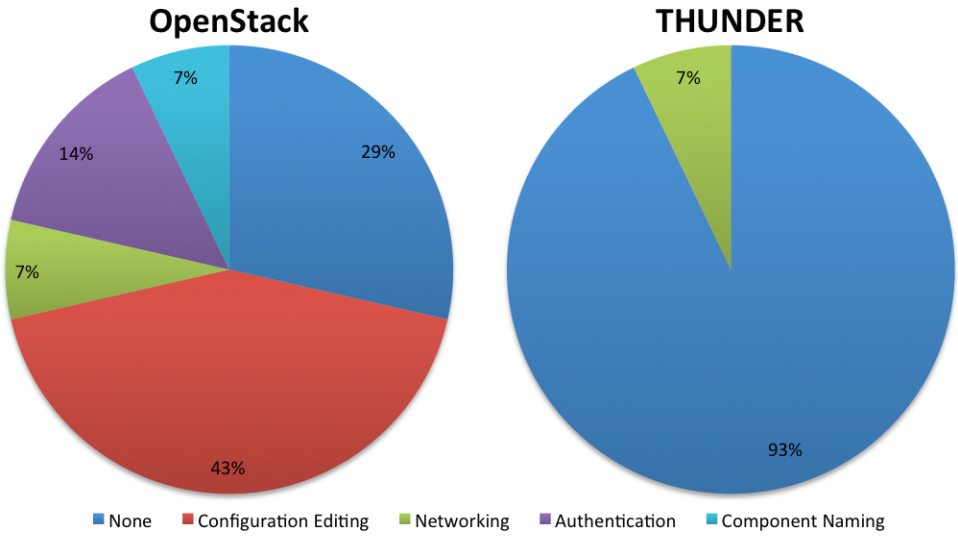



Figure 5.9: Qualitative Question 1: OpenStack Results.

### *5.1.2.3 Additional Comments*

The additional comments recorded by the survey participants indicate that the deployment system used by Eucalyptus is not particularly user friendly, but the deployment system used by THUNDER is easy and simple. We have selected a subset of comments about each architecture as an illustration.

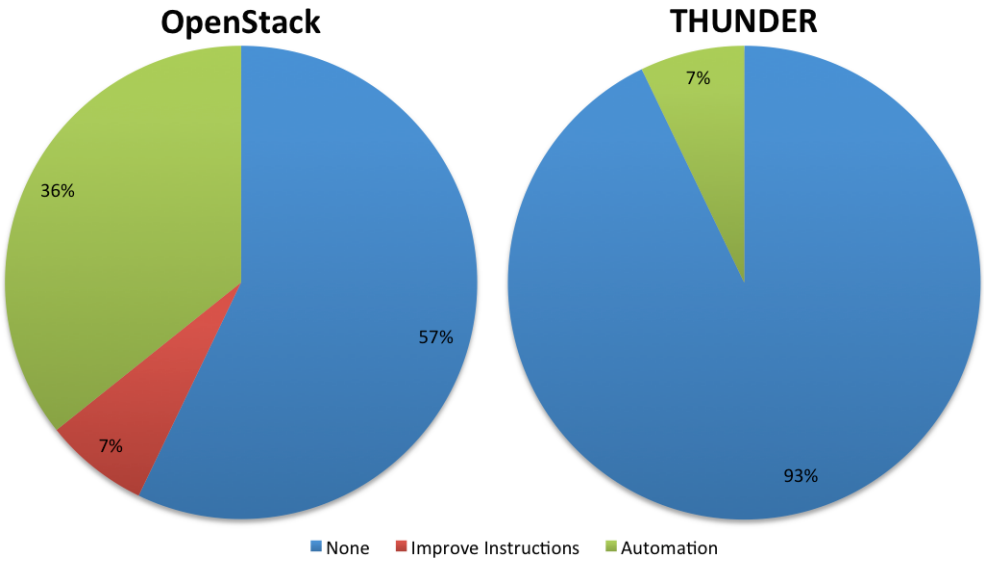



Figure 5.10: Qualitative Question 2: OpenStack Results.

**OpenStack**

- "*Much of what I did could have been automated.*"
- "*I wasn't able to get anything to work.*"

**THUNDER**

- "*Simple and easy. It just works.*"
- "*Only took a few minutes. Much better than OpenStack.*"

#### *5.1.2.4 Synthesis*

Data gathered from the empirical study on deployment of OpenStack and THUNDER show that the volunteers expressed far more concerns with the deployment process of OpenStack. As shown in Figure 5.10, 7% of the volunteers experienced difficulty with the deployment instructions provided by OpenStack, and 36% of the volunteers experienced frustration with the lack of

automation available during the deployment process. Additionally, as shown in Figure 5.9 we see that only 29% of volunteers did not have major complaints with the deployment process of OpenStack. 43% of the volunteers had major complaints regarding the configuration editing and the necessity to customize the configuration of each node. 14% of volunteers had major complaints regarding authentication. 7% of volunteers had major complaints about the naming of components. 7% of the volunteers also experienced difficulty with the networking configuration. Only 7% of volunteers had major complaints about the GUI interface of THUNDER, while the rest of the volunteers had no complaints about THUNDER.

## 5.2 Survey of Professional Users

Following the study of student users we wanted to ascertain how the results would differ if the survey targeted professionals that work for a non-profit organization. In this study we targeted the fields of education, medicine and law. The results of the survey show striking similarity to that of the previous results. The study had five volunteers, which consisted of three educators at the middle and high school levels, one medical doctor and one lawyer. The volunteers were asked the same questions as in the previous study, and their responses were aggregated and used to determine the ease of deployment of both OpenStack and THUNDER.

#### *5.2.0.5 Quantitative Results*

From the results in Figures 5.11 to 5.13 we see that the professional users tend to dislike the deployment process of OpenStack. The participants spent more time deploying OpenStack and had a more difficult time when compared to the deployment of THUNDER. None of the participants successfully deployed OpenStack, while all participants successfully deployed THUNDER. Additionally, the participants could not understand the instructions provided by OpenStack as they

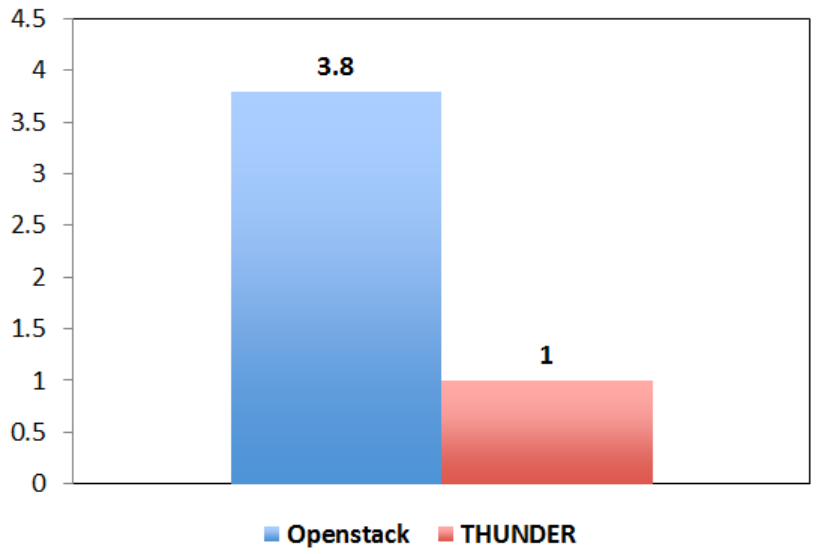



Figure 5.11: Quantitative Results: Time spent to deploy OpenStack and THUNDER.

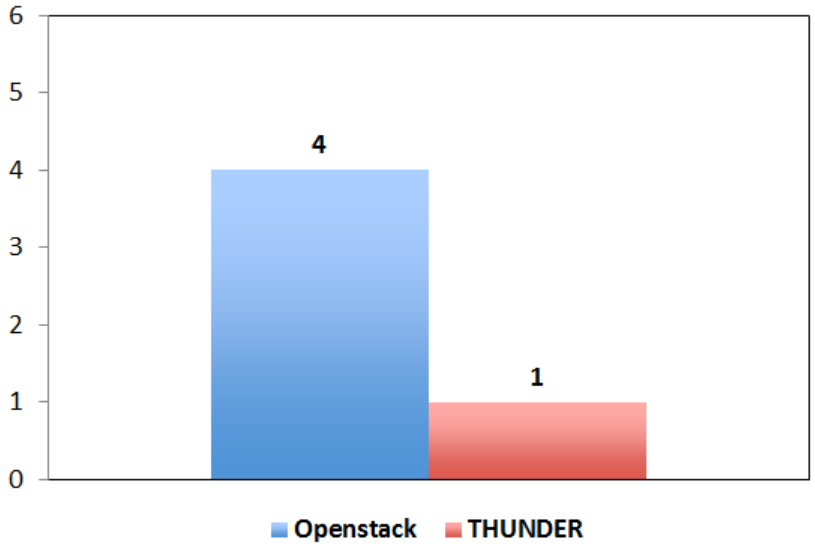



Figure 5.12: Quantitative Results: Perceived difficulty of OpenStack and THUNDER deployment.

require considerably more experience and conceptual knowledge than the instructions provided for deploying THUNDER.

#### *5.2.0.6 Qualitative Results*

Interestingly, all participants in the study had the same complaint with regard to OpenStack. The instructions for deploying OpenStack were too cumbersome and made no attempt to appease

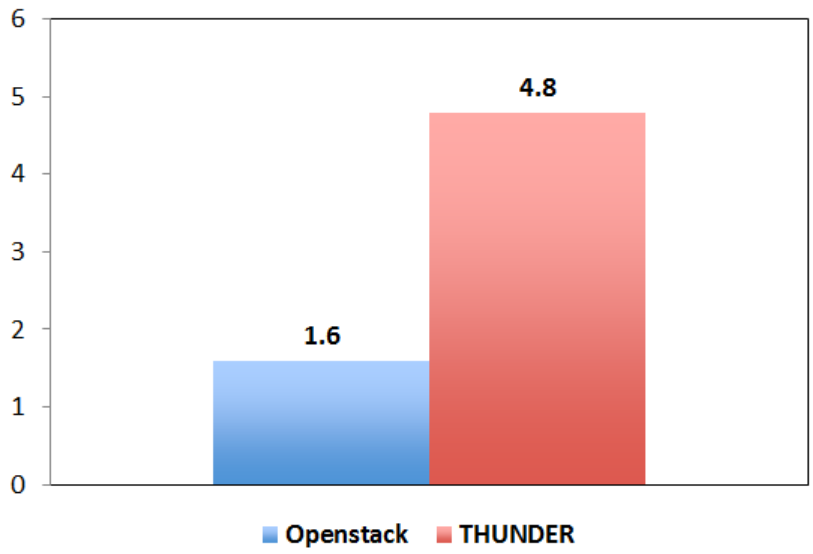



Figure 5.13: Quantitative Results: Clarity of OpenStack and THUNDER instructions.

inexperienced users. No complaints were made regarding the THUNDER deployment. All participants indicated that the deployment of OpenStack was non-intuitive and could have been improved with the use of some automation.

## 5.3 Summary

In total forty-three participants participated in the empirical study. Participants overwhelmingly preferred the deployment process of THUNDER over that of OpenStack and Eucalyptus. Additionally, the surveys helped to illuminate areas in which improvements could be made with regard to the THUNDER deployment system. We found automation to be the most requested feature with regard to all deployments in the study. Based on the results of the study further investigation into automation is planned for THUNDER and a discussion of this appears in Chapter 7.

# Chapter 6

## VIRTUAL MACHINE INSTANTIATION AND LOAD BALANCING

Virtual machine instantiation is the process by which a virtual machine image is transferred to a compute node, upon which the local hypervisor reads the virtual machine specification and allocates resources for the image. Before the virtual machine can be booted, it must be loaded into RAM. The amount of RAM required for a particular virtual machine is allocated by the local hypervisor, and the virtual machine image is then loaded into RAM. Once the image has been loaded into RAM it is considered to be an *instance*. The virtual machine instance starts the bootstrapping process, by which the local hypervisor is treated like physical hardware. However, before booting a virtual machine, we must consider several factors. Namely, we must consider where a virtual machine should be placed. Preferably, a cloud is deployed onto a homogeneous cluster of high performance computing hardware. Homogeneous deployments are preferable because load balancing is less crucial due to the fact that each compute node has the same load capacity. However, organizations may prefer to re-purpose workstations as cloud servers as a means to cut costs relating to hardware acquisition. The resulting heterogeneous cluster is composed to equipment for which the load capacity may vary greatly from node to node. Therefore, in heterogeneous clouds it is important to consider strategies that help to optimize virtual machine placement. In this chapter we examine virtual machine placement in a heterogenous cloud environment using a heuristic approach.

Virtual machine placement strategies vary depending on whether emphasis is on load bal-

ancing or load consolidation. We define virtual machine load balancing is the process by which the resource load produced by virtual machine instances is distributed across multiple compute nodes. This is accomplished primarily via node selection for new virtual machine instantiation, and virtual machine migration for prevailing virtual machine instances. Optimal node selection is an NP complete problem, and therefore a heuristic based approach is typically used to select compute nodes for new virtual machine instantiations. Round robin is a faire approach that is very common, but does not consider any priorities with regard to the selection process. In round-robin, the node selection process simply cycles over compute nodes and instantiates virtual machines as soon as a node has been found that can accommodate the virtual machine. In this paper we present a metrics-based load balancing strategy, called RAIN, which allows user defined resource priorities for determining close to optimal virtual machine placement. RAIN is an acronym for "Rating Assisted Instantiation Negotiator". In this strategy, resource allocation metrics consisting of RAM utilization, CPU load, and a count of active virtual machine instances are gathered from compute nodes. These metrics are combined with resource priority constants, which are defined by the user. These constants, $K_i$ are real numbers that range between 0.0 and 1.0, such that $\sum_{i=0}^{n} K_i = 1.0$. By multiplying the resource priority constants by the metrics, we form a ranking of compute nodes, where low ranking nodes are considered to be the most optimal. Algorithm 6.3.1 shows the rank algorithm for a single node.

## 6.1 Motivation

The primary goal of our research is to decrease the overhead involved in local private clouds for non-profit organizations, including aspects of deployment and management as well as overall usability. Virtual machine placement is the process by which a compute node is selected to host a particular virtual machine instance. In addition to round-robin, another common approach

attempts to place virtual machine instances onto the node currently hosting the fewest instances. After reviewing both approaches it is clear that neither approach allows for highly granular and fined tuned adjustments to node selection based on metrics other than a count of active virtual machines, or *AVM*.

Deployment and management of virtual machines consists of two types of costs: preparation costs and runtime costs [Sotomayor, Keahey, and Foster, 2006]. The preparation costs consist of creating the virtual machine with user defined constraints, developing the environment for hosting the virtual machine, and deploying the virtual machine on that environment. Runtime costs consists of physical resources required to host the virtual machine, such as virtual CPU cores, RAM, network interfaces, and storage. [Cherkasova, Gupta, and Vahdat, 2007b] notes that static allocation of physical cloud resources [Grosu, Chronopoulos, and Leung, 2008] becomes inefficient in larger cloud architectures with varying workloads. Dynamic resource allocation is used in the THUNDER architecture which allows for an increased number of successfully running virtual machine instances.

Based on our observations of common node selection algorithms we wanted to expand upon the former approach by allowing additional metrics to be used in the placement process. These metrics not only include AVM, but also RAM utilization and CPU load. With regard to the CPU load metric we introduce a granular approach, such that short-term CPU load(*STL*) and long-term CPU load(*LTL*) can have varying priorities. By allowing the end user to vary these resource priorities we believe that THUNDER may be more efficient at load balancing than that of the common strategies.

## 6.2 Theoretical Premise

In this section we describe the theoretical premise of the RAIN load balancing algorithm. Firstly, we define an upper bound resource constraint. This bounded resource constraint uses physical attributes of the machine to decide on an acceptable maximum amount of a particular resource that can be allocated. For example, the maximum number of VCores allowed to be allocated on a particular machine depends on the number of physical cores as well as the VCore/Core constraint. The maximum CPU load is also defined by the number of physical cores of the host machine, where a CPU load average greater than $N$ is considered to be overloaded on an $N$ core machine. Using proof by contradiction we show that RAIN strictly adheres to the bounded resources constraint, and therefore does not allow nodes to become overloaded as it is defined by Definition 6.2.2. Similarly, using proof by contradiction we show that round-robin does not consider physical attributes of the host machine beyond that which is reported by the hypervisor, and therefore, can violate the bounded resources constraint. Our rank algorithm for a single node is presented in Section 6.3.

Table 6.1: Symbol Descriptions

| Symbol | Description |
|---|---|
| $N$ | A selected node. |
| $K_i$ | A resource priority constant. |
| $R_{iv}$ | The amount of resource $i$ requested by virtual machine $v$. |
| $L_{iN}$ | The amount of resource $i$ currently allocated on Node $N$. |
| $\theta_{iN}$ | The resource capacity of Node $N$ for resource $i$. |
| $\#_{vmsN}$ | The number of virtual machines running on node $N$. |
| $\varepsilon$ | Ambient load of a node. |
| $\#_{nodes}$ | The number of compute nodes available in the cloud. |
| $\alpha$ | The number of resource types available. |

**Definition 6.2.1** (Node Rank)**.** *There exists some constants $K_i$ where $\sum_{i=0}^{\alpha} K_i = 1.0$, and resource utilization metrics $L_i$ s.t. $\sum_{i=0}^{\alpha} K_i \times L_{iN} \leq \sum_{i=0}^{\alpha} K_i \times L_{iN'}$ for some nodes N and $N'$.*

**Definition 6.2.2** (Bounded Resources)**.** *For each node there exists a capacity for each resource utilization metric, $\theta_i$. For each virtual machine instance request there exists a set of resource constraints $R_i$. The resource allocation allowed on node N during instantiation of virtual machine v is bounded by the inequality $\forall i \leq \alpha\ (L_{iN} + R_{iv} \leq \theta_{iN})$.*

*Violation of Bounded Resources in Round Robin.* If $s \bmod N$ denotes the next node in the node selection sequence, where $0 \leq s < \infty$, then the resource load produced by round-robin is modeled by the following:

$$L_{is \bmod N} = \sum_{v=0}^{\#_{vmsN}} R_{iv} + \varepsilon_i.$$

Let's assume that round-robin does not violate the bounded resources constraint. Therefore,

$$\forall i < \alpha \left( \sum_{v=0}^{\#_{vmsN}} R_{iv} + L_{is \bmod N} \right) \leq \theta_{s \bmod N i}$$

By definition, we know that if $R_{iv} + L_{is \bmod N} > \theta_{s \bmod N i}$ then round-robin has violated the bounded resources constraint for resource $i$. Considering that the round-robin selection mechanism, $s \bmod N \mid 0 \leq s < \infty$, does not utilize load attributes, it is not possible to say that overloading of a resource $i$ will never occur. Consider a selected machine $S'$ where the following constraints hold:

$$R_{iv} = 2048$$

$$L_{iS'} = 512$$

$$\theta_{iS'} = 1024$$

In this instance, the resource $i$ represents RAM. The virtual machine is requesting 2048MB, the current amount of used RAM on the node $S'$ is 512MB, and the maximum capacity of node $S'$ is 1024MB. In this instance round-robin will not reject the instantiation, which will cause the virtual machine to potentially cause the overutilization of the host hardware due to excessive context switching and memory swapping. Therefore, round-robin violates the bounded resource constraint if and only if:

$$R_{iv} > \theta_{s \bmod Ni} - L_{iv}.$$

That is, if the amount of requested resources is larger than the difference between the resource capacity and the current resource load then the bounded resource constraint is violated. In this case, round-robin will allow one or more resources to exceed the maximum acceptable resource capacity as defined by Definition 6.2.2 because $R_{iv}$ is allowed to grow without bound. □

*Strict Adherence to Bounded Resource Constraint in RAIN.* Node selection in RAIN utilizes a weighted sum ranking mechanism (definition 6.2.1). The weighted sum is computed for each node $N$ and the lowest ranked node takes priority in the selection process. The lowest ranked node is selected to host a virtual machine instance. This mechanism can be modeled by the following set:

$$\text{Let } \upsilon \rightarrow \left\{ \forall N < \#_{nodes} \mid \left( \sum_{i=0}^{\alpha} K_i \times L_{iN} \right) \leq \theta_{iN} \right\}.$$

Let the selected node $S$ be

$$\min_{\upsilon_1, \ldots \upsilon_n} \neq \phi \mid R_{iv} + L_{iS} \leq \theta_{iS}$$

Resource load produced by RAIN is modeled by the following:

$$L_{iS} = \sum_{v=0}^{\#_{vms\,S}} R_{iv} + \varepsilon_i$$

Let's assume that RAIN violates the bounded resources constraint. Therefore,

$$\forall i \leq \alpha \left( \sum_{v=0}^{\#_{vms}S} R_{iv} + L_{iS} \right) > \theta_{iS}.$$

By definition, it must be that case that $S \in \upsilon$, and therefore the following must be true for all resources $i < \alpha$:

$$R_{iv} + L_{iS} \leq \theta_{iS}.$$

However, this contradicts the assumption that RAIN violates the bounded resources constraint, and therefore, it never allows any particular resource to exceed the maximum acceptable resource capacity as defined by Definition 6.2.2. □

## 6.3 Experimental Model

During each experiment random workloads consisting of virtual machines configured with between 1 and 4 VCores and between 512MB and 2GB of RAM are selected at random to be instantiated in the cloud. Before each trial the random seed for selecting the virtual machine workloads is set using a pre-defined seed value such that the sequence of virtual machines instantiated is deterministic. The seed value used is 217645199, which is the $12000001^{st}$ prime number and was chosen in order to potentially maximize the entropy of the pseudo random number generator. Node metrics are gathered before and after instantiation. The experimental model depends on the resource priority constants where sets of priority constants are selected, and are presented in Appendix A. For each set of priority constants, we are interested in evaluating the efficiency of RAIN. Efficiency in this context is the ability for the load balancer to utilize the resources in such a way that ensures the most virtual machine instantiations are successful, without over-utilization. Over-utilization happens when so many virtual machines are placed on a particular compute node that the performance penalty reaches the upper limit for the particular system. The resource uti-

lization metric provided by the Linux kernel, referred to as the "load average" has an upper bound which is determined by the number of CPU cores. Therefore, an *N*-core system is considered to be *overloaded* when the load average exceeds *N*.

The hardware utilized during the experimentation consists of a heterogeneous cluster of compute nodes. The two-node trials consisted of two Intel Xeon quad-core processors, one with support for two threads per core and the other with support for one thread per core. The former is equipped with 32GB of RAM, while the latter is equipped with 16GB of RAM. The four-node trials consist of the hardware from the two-node trials with the addition of two Intel Core2-Duo nodes, each with 8GB of RAM.

---

**Algorithm 6.3.1:** RANK(*node*)

---

**comment:** Compute the rank of one node

$$S_{RAM} \leftarrow K_{RAM} \times \frac{RAM_{Used}(node)}{RAM_{Total}(node)}$$

$$S_{Load_{1min}} \leftarrow K_{Load_{1min}} \times \frac{Load_{1min}(node)}{NumCores(node)}$$

$$S_{Load_{5min}} \leftarrow K_{Load_{5min}} \times \frac{Load_{5min}(Node)}{NumCores(node)}$$

$$S_{Load_{15min}} \leftarrow K_{Load_{15min}} \times \frac{Load_{15min}(Node)}{NumCores(node)}$$

$$S_{ActiveVMs} \leftarrow K_{ActiveVMs} \times \frac{ActiveVMs(node)}{NumCores(node) \times VMsPerCore}$$

**return** $(S_{RAM} + S_{Load_{1min}} + S_{Load_{5min}} + S_{Load_{15min}} + S_{ActiveVMs})$

---

The rank algorithm is then applied to all of the compute nodes, which is represented by a vector of node objects. Algorithm 6.3.2 shows that the RAM, number of physical CPU cores, CPU load, and the number of active virtual machines determine the rank of the node.

**Algorithm 6.3.2:** GETRANKINGS(*nodes*)

**comment:** Compute vector of rankings from vector of nodes

$in \leftarrow nodes$
$rankings \leftarrow \{\}$
**while not** $in \cdot empty()$
**do** $\begin{cases} node \leftarrow in \cdot dequeue() \\ rankings \cdot \text{PUSH}(\text{RANK}(node)) \end{cases}$
**return** $(rankings)$

A ranking vector produced using this approach is unordered. That is, the most optimal node is not at the head of the vector. Therefore, before selecting a node to place a virtual machine, we first sort the node vector by the corresponding rankings, such that the nodes are ordered from most optimal to least optimal. Algorithm 6.3.3 shows the algorithm by which a node is selected to host a virtual machine.

**Algorithm 6.3.3:** SELECT(*nodes*, *vm*)

**comment:** Select a node based on a vector of nodes and VM requirements

$rankings \leftarrow \text{GETRANKINGS}(nodes)$
$sortedNodes \leftarrow \text{SORT}(nodes, rankings)$
**for each** $node \in sortedNodes$
**do** $\begin{cases} \textbf{comment:} \text{ Check if the VM fits on the node (RAM, CPU Cores, etc.)} \\ \textbf{if } node \cdot fits(vm) \\ \quad \textbf{then return } (node) \end{cases}$
**comment:** The virtual machine does not fit on any of the nodes!
**return** $(ERROR)$

## 6.4 Instantiation with Node Locking

A potential pitfall of using the RAIN can occur when many virtual machine instantiation requests are received within a short span of time, $\Delta t$. If $\Delta t$ is too small then the RAIN algorithm will potentially select the same node for each request because the metrics received by participating compute nodes are the same or very similar and the virtual machine has not had time for instan-

tiation yet. To mitigate this problem RAIN introduces a locking step such that nodes selected for instantiation will be locked until the virtual machine has either started or failed. During the locking phase any locked node that is selected for instantiation will force the instantiation request to wait, during which time new metrics will be generated. Instantiation locking guarantees that only one request will be processed on each node simultaneously. Algorithm 6.4.1 illustrates the instantiation algorithm with node locking.

---

**Algorithm 6.4.1:** INSTANTIATE($vm$)

---

$locks \leftarrow \{0,0,0,\ldots,0\}$
$ready \leftarrow false$
$node \leftarrow null$
**repeat**
  $utilization \leftarrow$ GETUTILIZATION($nodes$)
  $node \leftarrow$ SELECT($utilization, vm$)
  **if** $node \rightarrow ERROR$
    **then return** ($ERROR$)
  **if** $locks[node] \rightarrow 0$
    **then** $\begin{cases} locks[node] \leftarrow 1 \\ ready \leftarrow true \end{cases}$
    **else** $sleep$ 5 seconds
**until** $ready \rightarrow true$
$node \cdot$ INSTANTIATE($vm$)

---

## 6.5 Simulation of Node Selection

Before experiments on THUNDER were undertaken a simulation of RAIN was performed. Simulation of the RAIN algorithm shows favorable results when compared to that of random, round-robin, and consolidation strategies. For each strategy a node is selected and instantiation of a randomly selected virtual machine image is attempted on that particular node. The random strategy chooses a node at random. The round-robin strategy creates a sequence of nodes and selects the next node in sequential order starting with the first in the sequence and ending with

the last in the sequence. The consolidation strategy creates a sequence of nodes similar to the round-robin strategy, except that it always selects the first node in the sequence that has enough resources to fit the virtual machine. In each strategy the instantiation has the potential to either fail or be successful. Failure may be the result of one or more factors, including insufficient amount of free RAM, insufficient number of available virtual CPU cores, and unexpected system malfunction. Data gathered from the simulation of between one and twenty compute nodes and between zero and fifty instantiations show that the RAIN algorithm minimizes failed virtual machine instantiations as the number of compute nodes increase. Figure 6.1 shows a graph of failed simulated virtual machine instantiations with regard to the number of compute nodes. With the exception of random, the placement strategies show similar results when the number of compute nodes is greater than seven. When the number of compute nodes is less than or equal to seven, all four of the algorithms increase dramatically in the number of failed instantiations. As will be shown in the next section, simulation does not provide an accurate depiction of how an actual system will respond.

## 6.6 Bare Metal Performance Results

We compare the performance of our metrics-based load balancing strategy, RAIN with that of round-robin. In this comparison, metrics are gathered from the cloud servers before and after instantiation of virtual machines using the RAIN strategy for each case stud. Similarly, the load balancing algorithm is swapped out for round-robin and the metrics are gathered again. The goal of this study is to determine whether or not RAIN chooses optimal nodes, such that we minimize the amount of increased resource utilization overhead. The presumption is that non-optimal node selections may cause unwanted overhead in the form of CPU utilization that approaches a 100% load, and memory utilization that exceeds the physical memory size limitations. Exceeding mem-

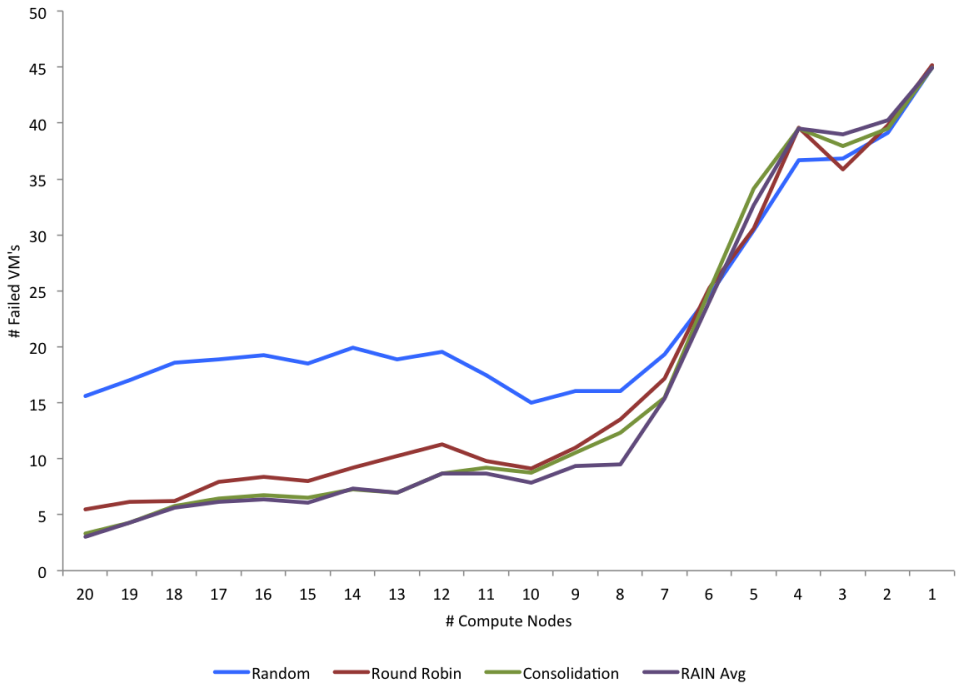



Figure 6.1: Simulated Average of Failed Virtual Machine Instantiations.

ory limitations can cause memory pages to be swapped out to disk, which causes increased I/O overhead, causing machines to die earlier than expected.

Nine case studies were conducted. Four of the case studies non-strictly emphasize each parameter of the RAIN algorithm, such that the resource allocation constants are higher for the emphasized parameter, but less than 1.0. Four of the case studies strictly emphasize each parameter of the RAIN algorithm, such that one resource allocation constant is set to 1.0, while others are set to 0.0. The final case study demonstrates the RAIN algorithm without emphasizing any parameter of the RAIN algorithm, such that all resource allocation constants are set to 0.0 for consolidation of virtual machines.

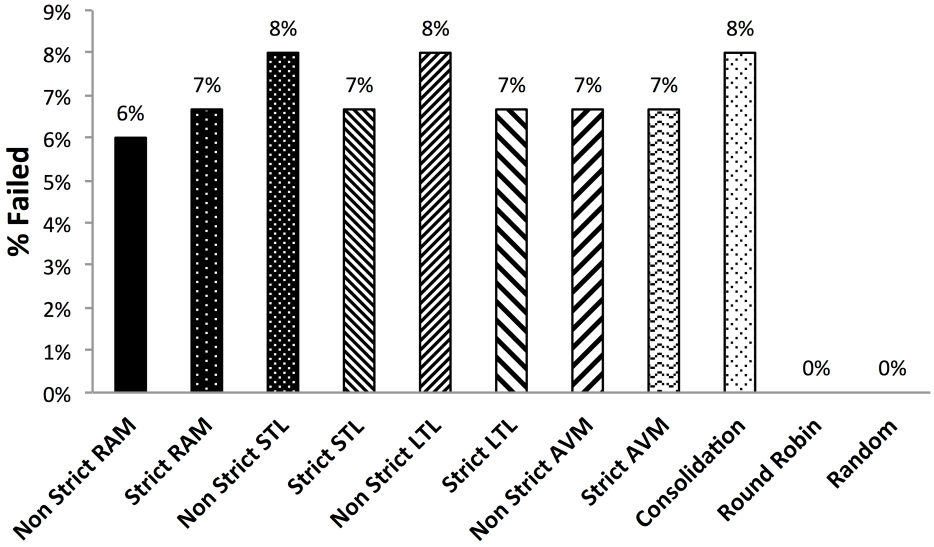
We tested the RAIN algorithm against round-robin with random selection as a control. Each experiment was conducted by attempting to instantiate $N$ virtual machines on an $M$ node cluster. Since RAIN allows for fine-tuned control over the max number of *VCores* allowed per physical

core, the *VCore/Core* constraint varies between three and six. Round-robin and random strategies do not have an explicit *VCore/Core* constraint except for what is imposed by the KVM hypervisor. Each experiment was executed five times and the results are averaged. In total we attempted to instantiate RAIN resource priority constraints for each case study are presented in Appendix A.

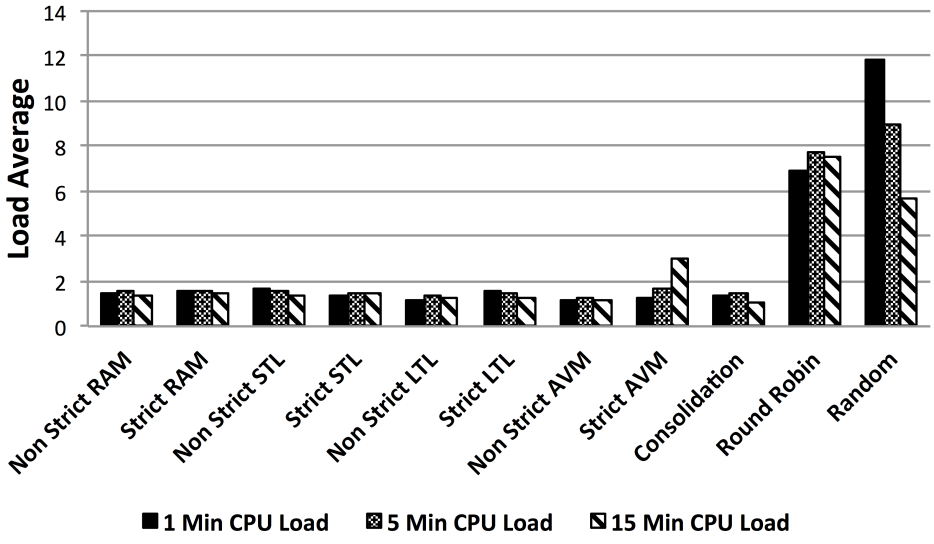
Figure 6.2 shows the results from the first experiment in which we attempt to instantiate fifteen virtual machines on two nodes with the *VCore/Core* constraint of three. We show Figure 6.2a that by using RAIN we achieved an overall failure rate of between 6% and 8%. Neither round-robin nor random sustained any failures. From these results we can see that RAIN rejected at most 8% of the virtual machine instantiation requests in order to avoid over-utilization. Although, neither round-robin nor random sustained any failures, it is very apparent in Figure 6.2b that both strategies produced resource loads that far exceeds what could be considered optimal. Round robin produced CPU load of between 7.1 and 7.6, while random produced CPU load of between 5.9 and 11.9. On the contrary, RAIN produced CPU load of between 1.8 and 3.1. Additionally, as shown in Figure 6.2c the RAM overhead of round-robin and random are 46% and 43% respectively. RAIN produced RAM overhead of between 38% and 41%. Based on what we have defined as the most optimal fit of virtual machines under the constraints of this trial, as shown in Figure 6.2d round-robin accepted 20% more instantiations than the desired maximum and random instantiated 4% more than the desired maximum. RAIN instantiated between 5% and 1% less than the desired maximum, meaning that RAIN strictly instantiated less than the upper bound as defined in Definition 6.2.2.

In an attempt to ascertain a limit by which round-robin and random would fail to instantiate virtual machines we performed an experiment by attempting to instantiate twenty virtual machines on a two-node cluster with the constraint of 3 VCores/Core. Results show that even with the

Figure 6.2: 15 VM, 2 Node, 3 VCore/Core Trial

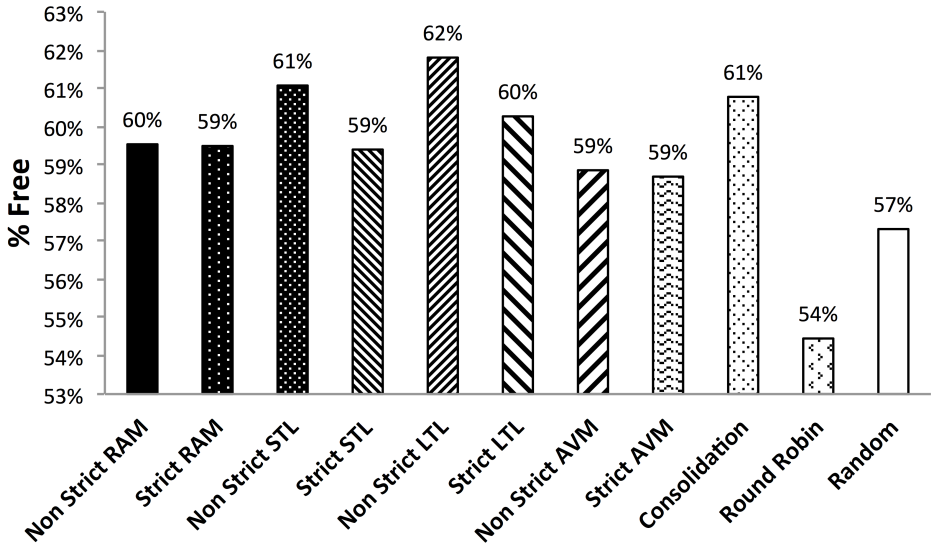



(a) Average Failed VM Instantiations

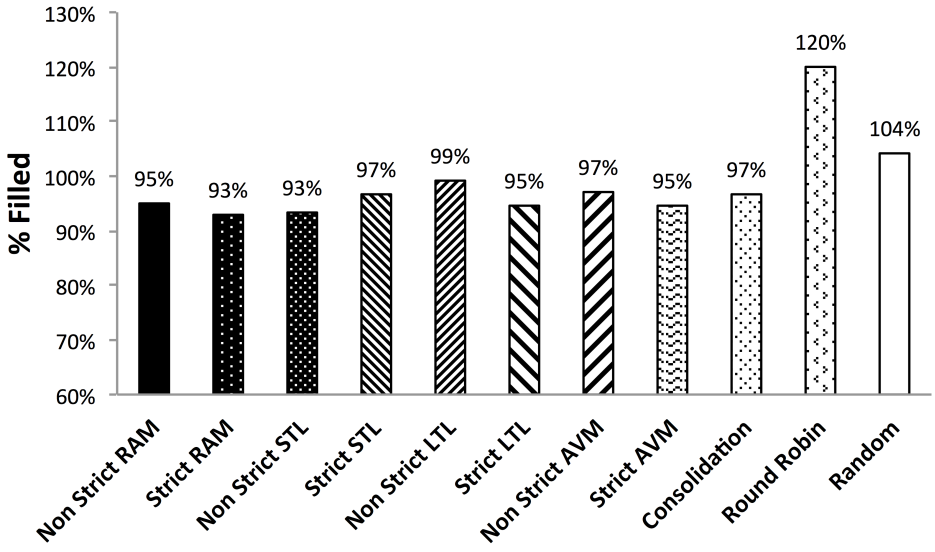



(b) Average Load

Figure 6.2: (Continued) 15 VM, 2 Node, 3 VCore/Core Trial

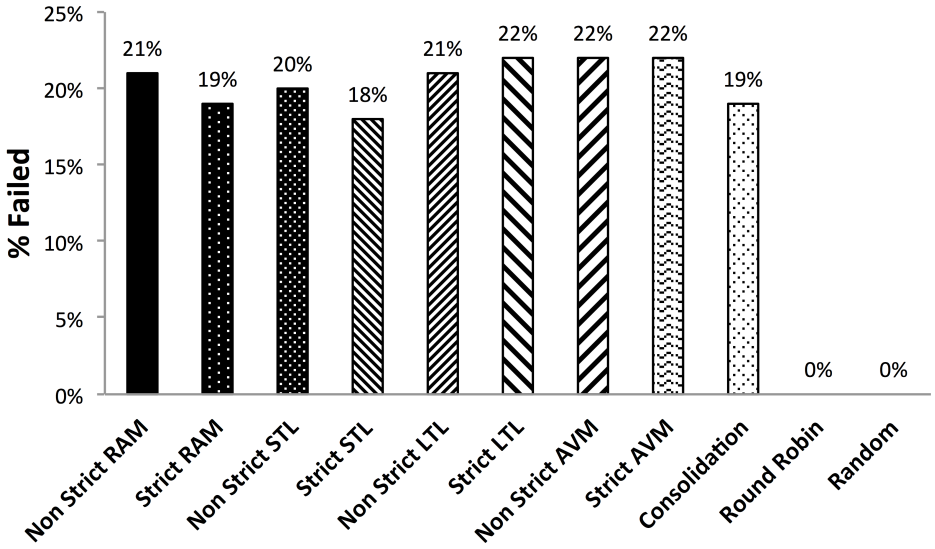



(c) Average RAM Utilization

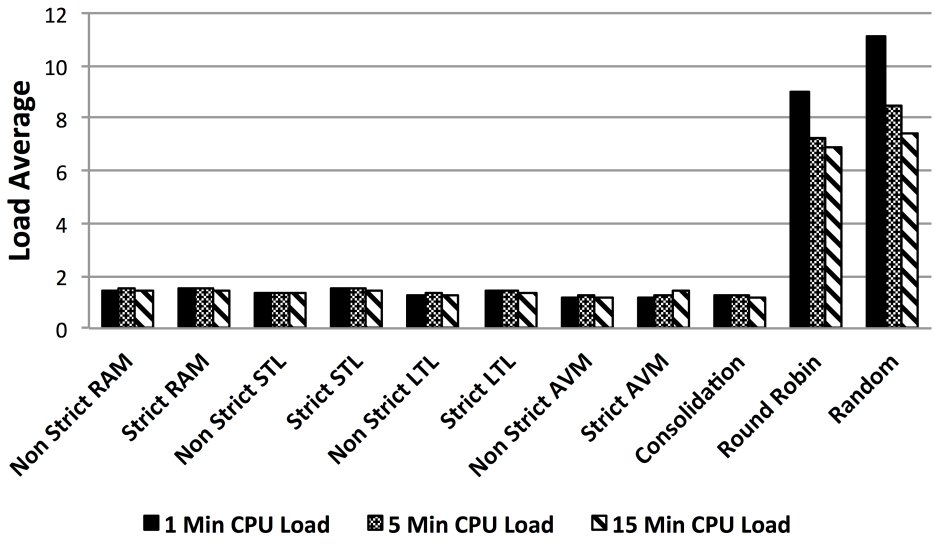



(d) Average VM Fill

Figure 6.3: 20 VM, 2 Node, 3 VCore/Core Trial

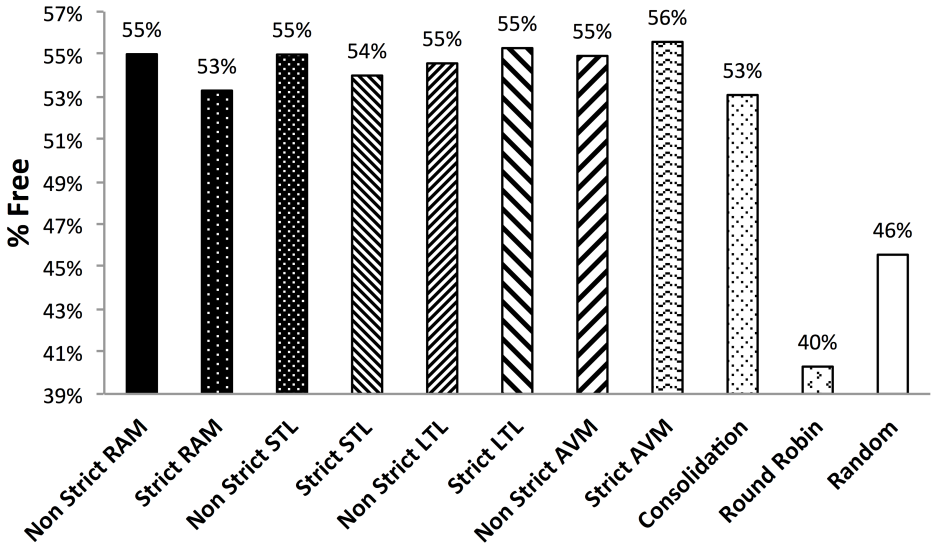



(a) Average Failed VM Instantiations

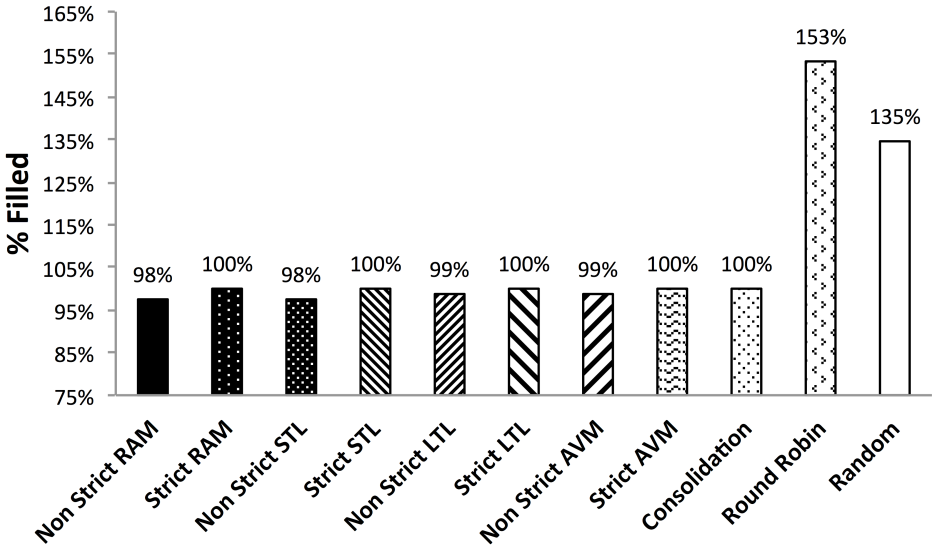



(b) Average Load

Figure 6.3: (Continued) 20 VM, 2 Node, 3 VCore/Core Trial




(c) Average RAM Utilization

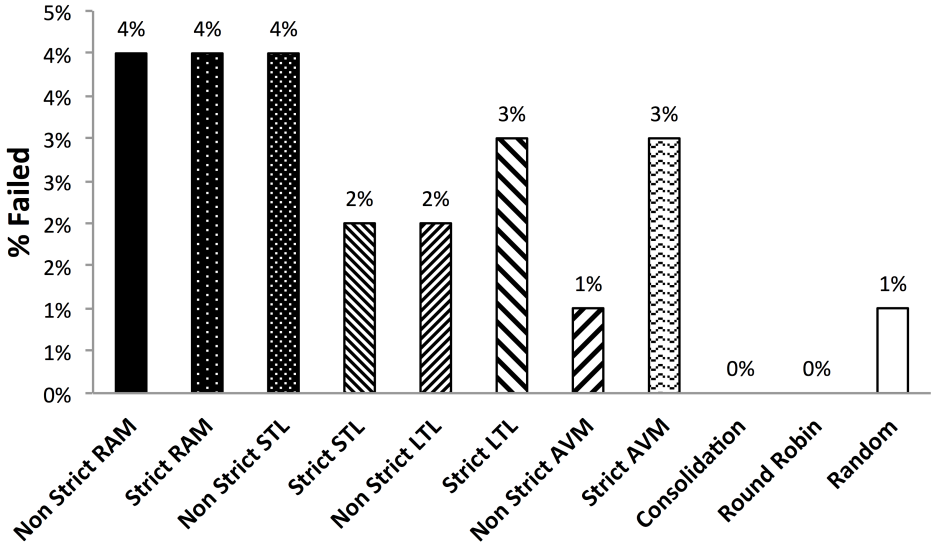



(d) Average VM Fill

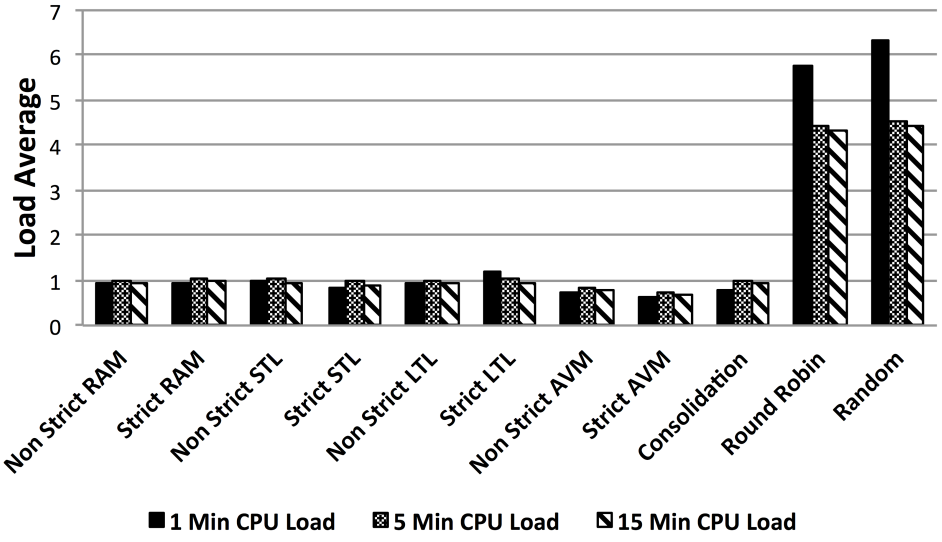
increased load of an additional five virtual machine instantiations neither round-robin nor random sustained any failures. As presented in Figure 6.3a RAIN sustained a failure rate of between 18% and 22%. Both round-robin and random produced excessive load during this experiment as shown in Figure 6.3b with a CPU load of between 7.2 and 9.1 for round-robin and between 7.7 and 11.2 for random. Additionally, as presented in Figure 6.3c RAM overhead of round-robin increased to 60% and RAM overhead of random increased to 54%. RAM overhead of RAIN fluctuated between 44% and 47%. We show in Figure 6.3d that round-robin accepted 53% more instantiations than the desired maximum and random instantiated 35% more than the desired maximum. RAIN instantiated between 2% and 0% less than the desired maximum.

By utilizing the elastic nature of the THUNDER cloud architecture, we increased the physical size of the cloud to four nodes in an attempt to mitigate the rejection of virtual machine instantiations in RAIN. In Figure 6.4 we show the results of twenty attempted virtual machine instantiations on a four-node cluster using the same constraint of 3 VCores per core. In this experiment the resource capacity has increased with the addition of two nodes. As a result, RAIN rejected between 0% and 4% of virtual machine instantiations, which is an improvement over previous experiments. As shown in Figure 6.4a round-robin sustained a failure rate of 0% while random sustained a failure rate of 1%. The additional nodes should theoretically help to decrease the average CPU load during instantiation, and in fact we do see a decrease in average CPU load for round-robin and random. As shown in Figure 6.4b round-robin sustained an average CPU load of between 4.2 and 5.7, while random sustained an average CPU load of between 4.3 and 6.2. RAIN kept the average CPU load close to 1.0. As shown in Figure 6.4c average RAM overhead produced by round-robin reached 55% and the average RAM overhead produced by random reached 51%. The average RAM overhead produced by RAIN was between 40% and 45%. Figure 6.4d RAIN

Figure 6.4: 20 VM, 4 Node, 3 VCore/Core Trial

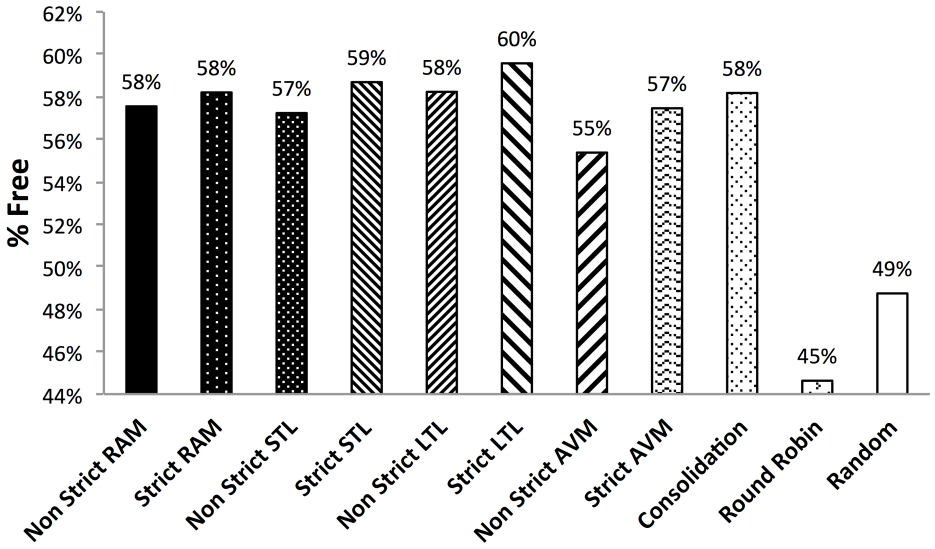



(a) Average Failed VM Instantiations

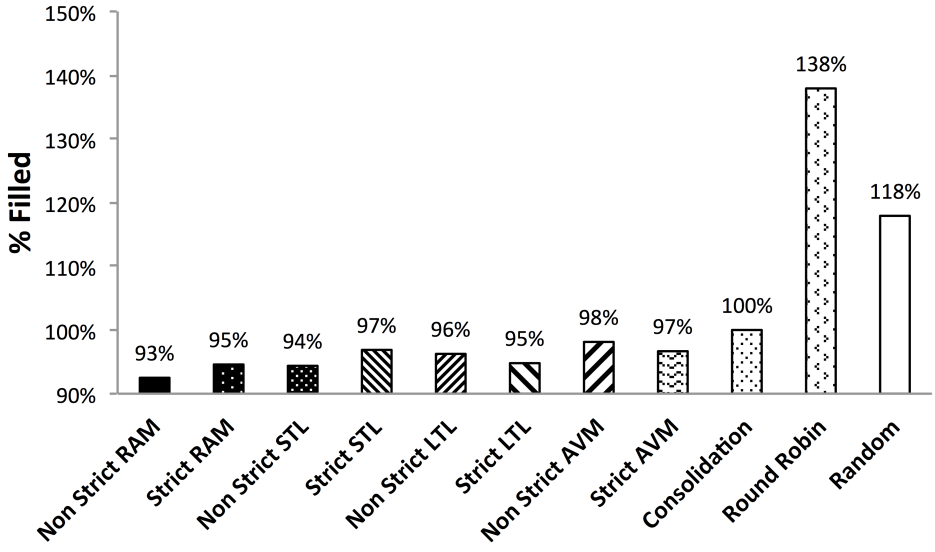



(b) Average Load

Figure 6.4: (Continued) 20 VM, 4 Node, 3 VCore/Core Trial

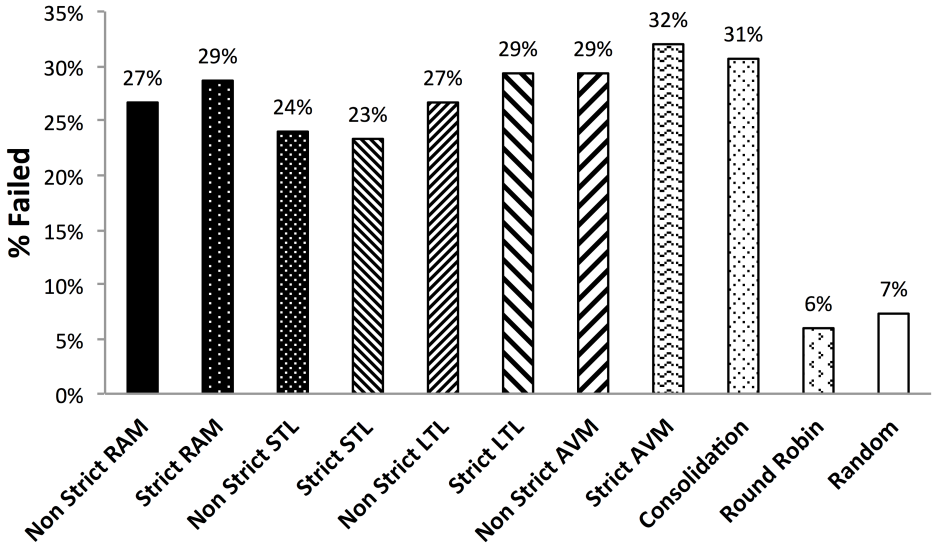



(c) Average RAM Utilization

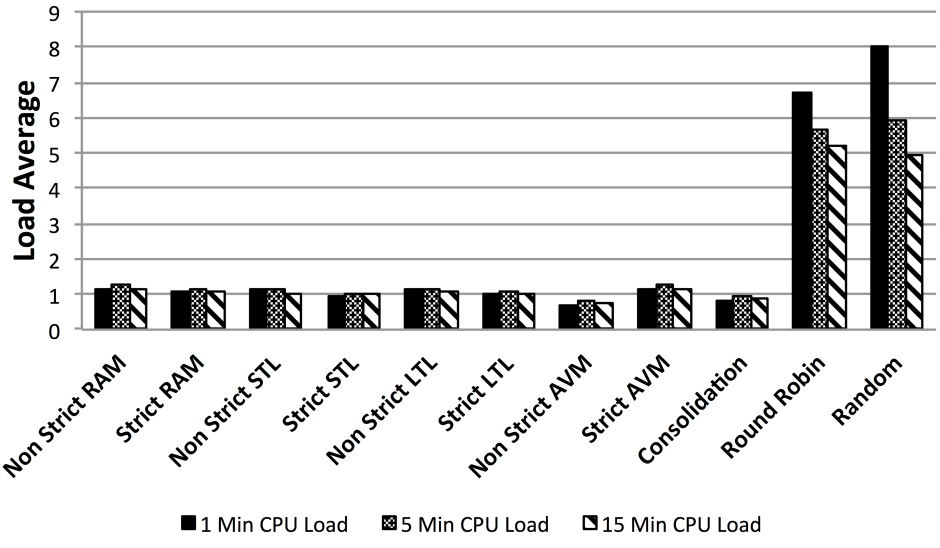



(d) Average VM Fill

Figure 6.5: 30 VM, 4 Node, 3 VCore/Core Trial

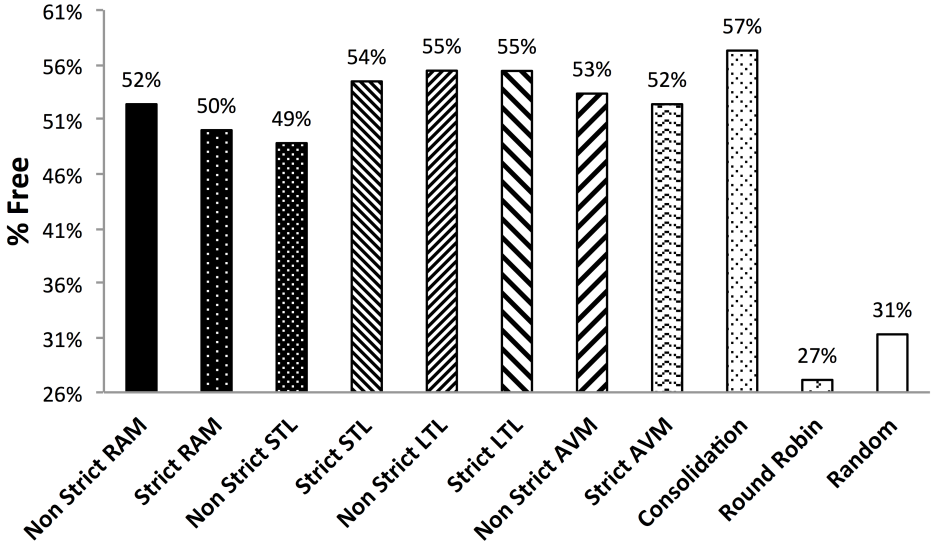



(a) Average Failed VM Instantiations

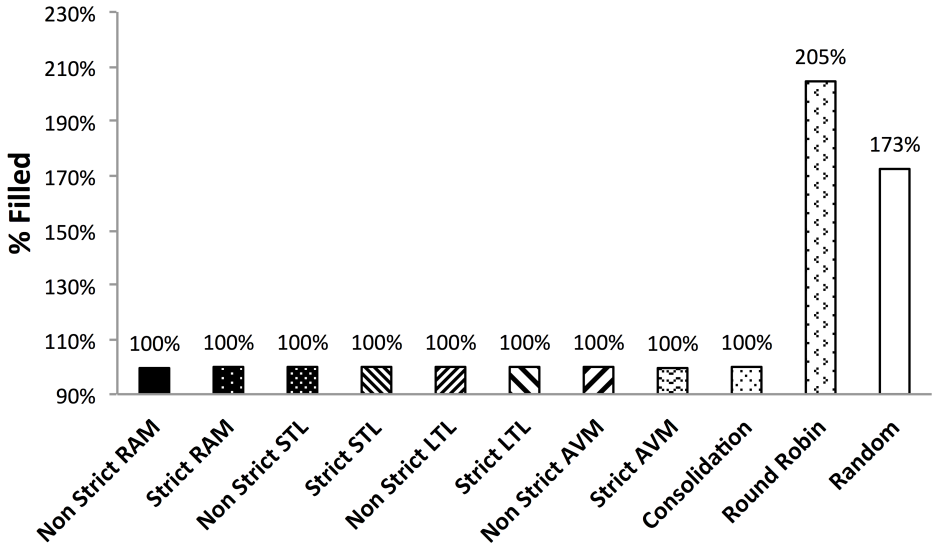



(b) Average Load

Figure 6.5: (Continued) 30 VM, 4 Node, 3 VCore/Core Trial




(c) Average RAM Utilization



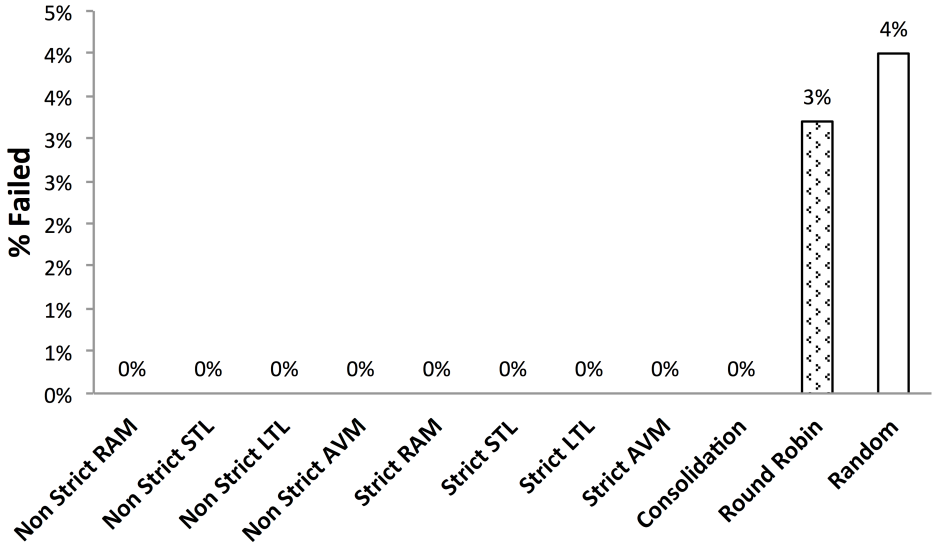

(d) Average VM Fill

instantiated between 7% and 0% less than desired maximum number of virtual machines allowed. Round robin instantiated 38% more than the desired maximum, while random instantiated 18% more than the desired maximum.

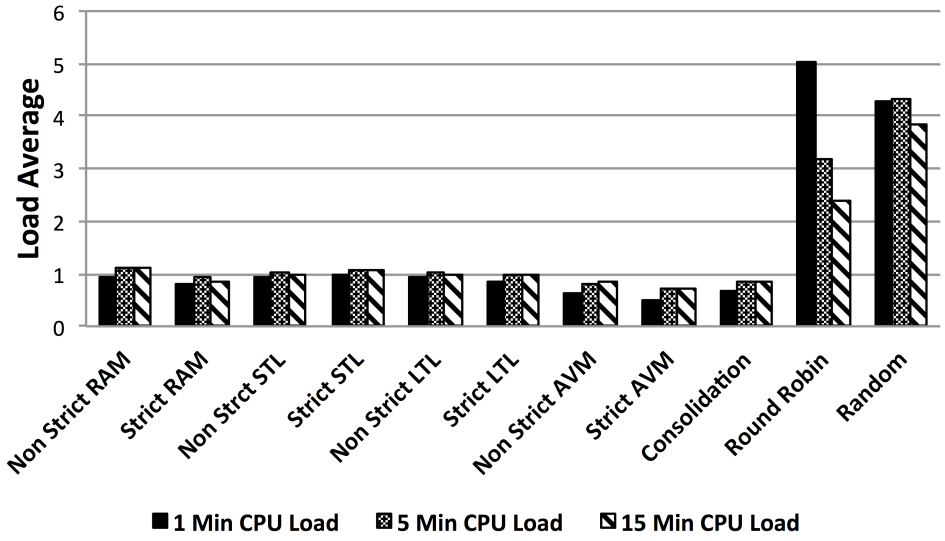
In order to attempt to find a point at which round-robin fails to instantiate virtual machines, we increased the number of attempted virtual machine instantiations to thirty in Figure 6.5. In this experiment we see the first evidence of round-robin failing to instantiate virtual machines. Figure 6.5a illustrates that round-robin sustained a failure rate of 6% while random sustained a failure rate of 7%. RAIN rejected between 23% and 32% of virtual machine instantiation requests. The average CPU load produced by round-robin is observed to be between 5.1 and 6.7. The average CPU load produced by random is between 5.0 and 8.0. RAIN sustained an average CPU load of between 0.75 and 1.2. Due to the increased instantiation attempts, round-robin produced RAM overhead of 72% and random produces a RAM overhead of 69%. As shown in Figure 6.5c RAIN produces RAM overhead of between 43% and 51%. On average round-robin exceeded the desired maximum number of instantiations per node as defined in Definition 6.2.2 by 105%. As shown in Figure 6.5d random exceeded the desired maximum number of instantiations per node by 73%. RAIN strictly kept the number of instantiations per node to 100% of the desired maximum.

After observing a point of failure with round-robin in Figure 6.5 we relaxed the VCore per core constraint of RAIN to five. By relaxing the constraint we are able to allow more instantiations to be accepted per node. In Figure 6.6 we present the results of twenty-five instantiation attempts on a four-node cluster using the relaxed constraint. In this experiment we see results that, for the first time show a higher number of successful instantiations with RAIN when compared to round-robin. As shown in Figure 6.6a round-robin failed to instantiate 3% of virtual machines and random failed to instantiate 4% of virtual machines. RAIN was successful in instantiating all virtual machines.

Figure 6.6: 25 VM, 4 Node, 5 VCore/Core Trial

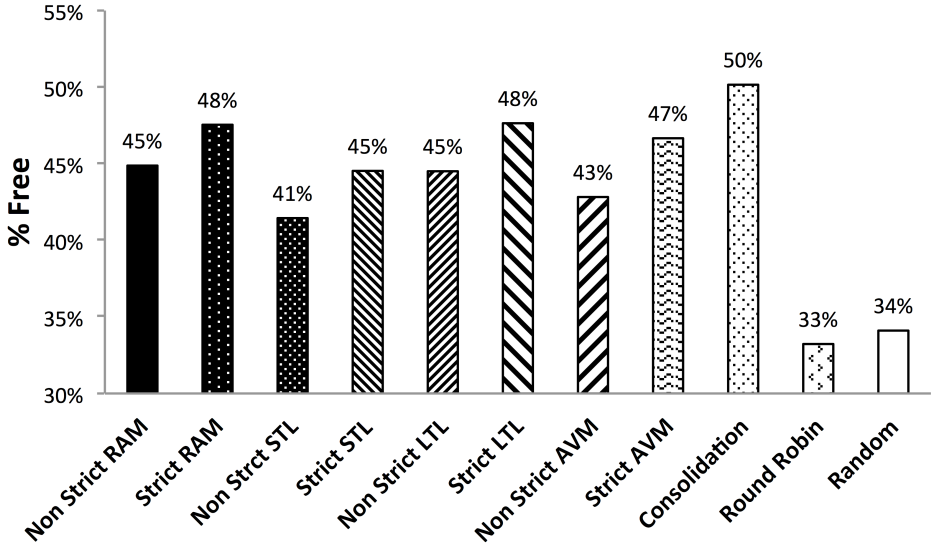



(a) Average Failed VM Instantiations

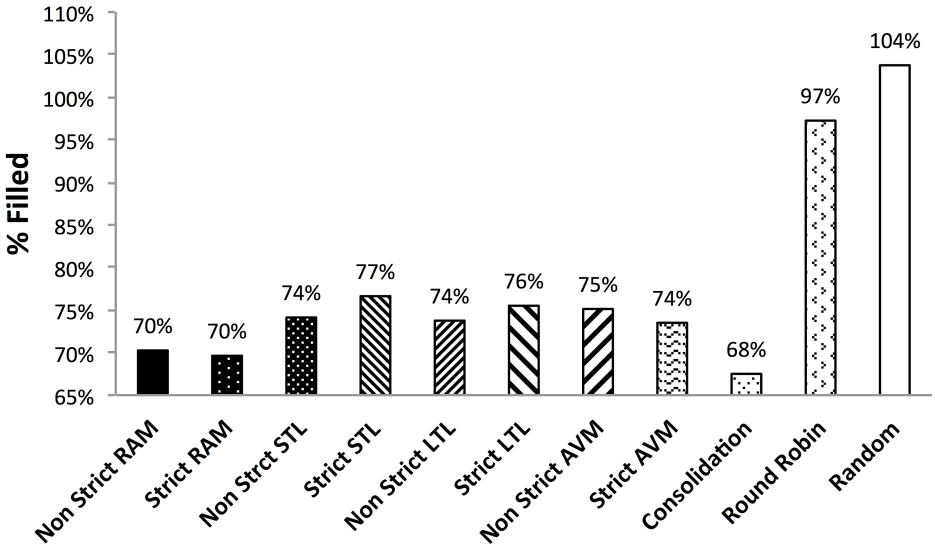



(b) Average Load

Figure 6.6: (Continued) 25 VM, 4 Node, 5 VCore/Core Trial

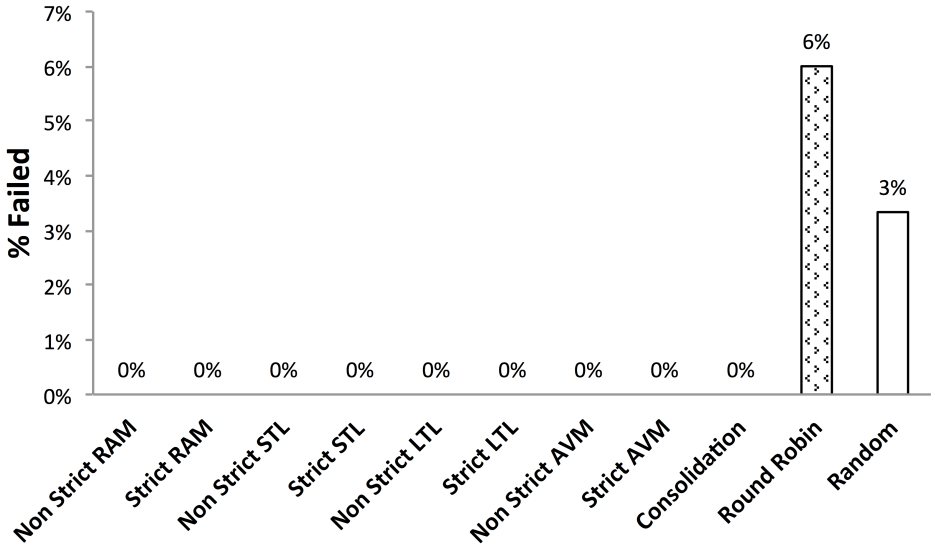



(c) Average RAM Utilization

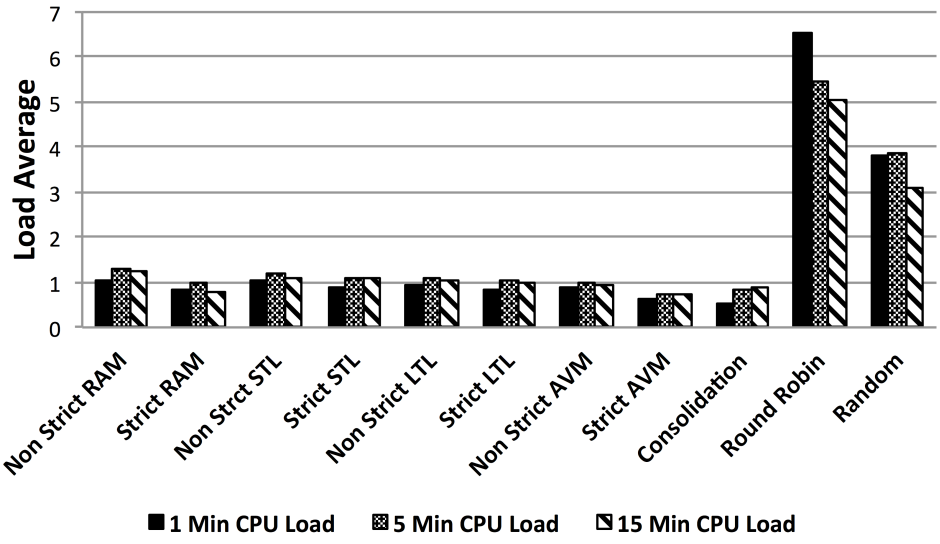



(d) Average VM Fill

Figure 6.7: 30 VM, 4 Node, 5 VCore/Core Trial

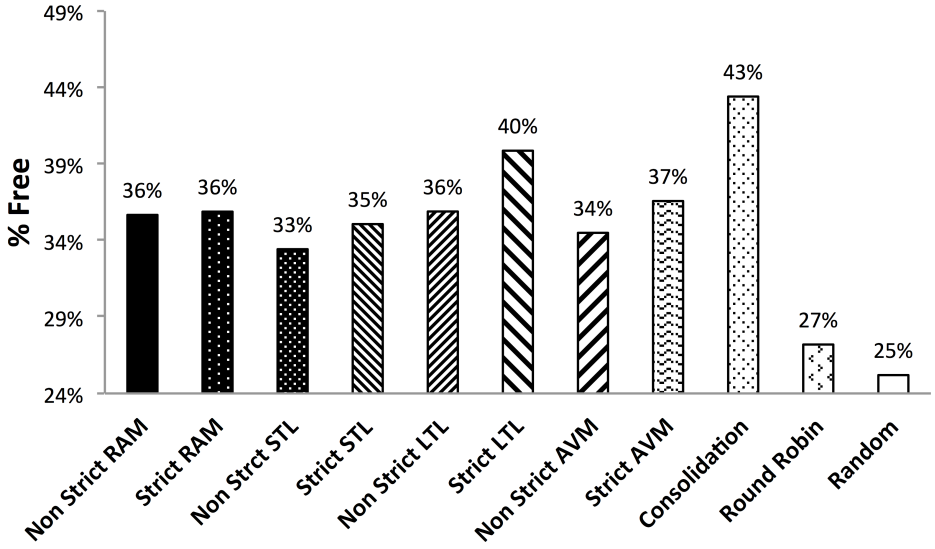



(a) Average Failed VM Instantiations

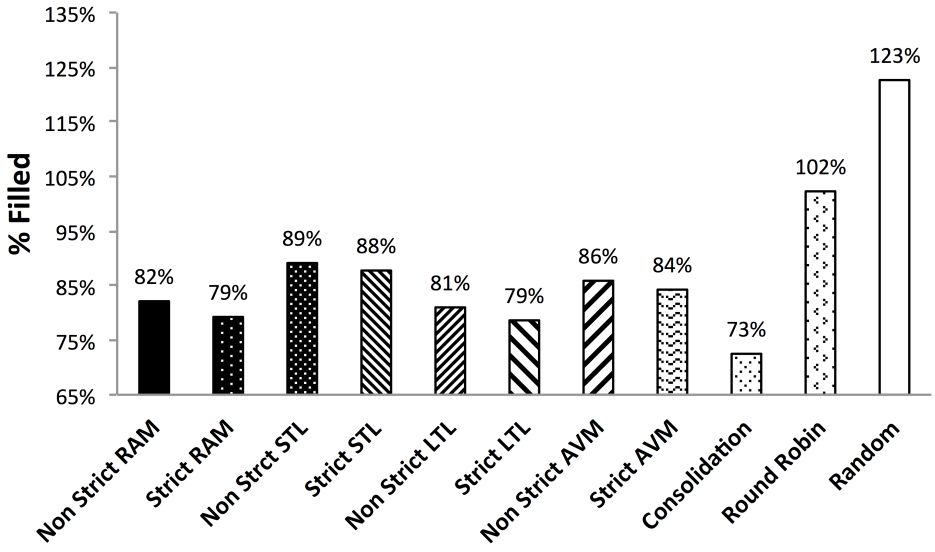



(b) Average Load

Figure 6.7: (Continued) 30 VM, 4 Node, 5 VCore/Core Trial




(c) Average RAM Utilization



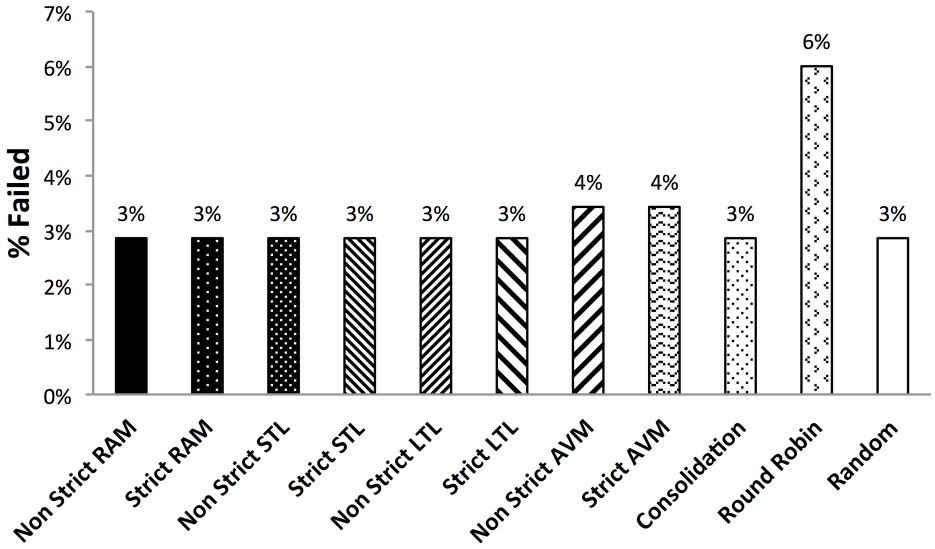

(d) Average VM Fill

Again, we see in Figure 6.6b there is a very high average CPU load under round-robin of between 2.2 and 5.0. Random produced CPU load of between 3.8 and 4.2. RAIN produced average CPU load of between 0.75 and 1.1. RAM overhead produced by RAIN is observed to be between 50% and 59%. In Figure 6.6c we see that RAM overhead produced by round-robin is observed to be 67% and RAM overhead produced by random is observed to be 66%. As shown in Figure 6.6d the average number of instantiations per machine in RAIN is observed to be between 68% and 77% of the desired maximum. The average instantiations per machine in round-robin is observed to be 97% of the desired maximum. Random instantiated on average 4% more virtual machines than the desired maximum.

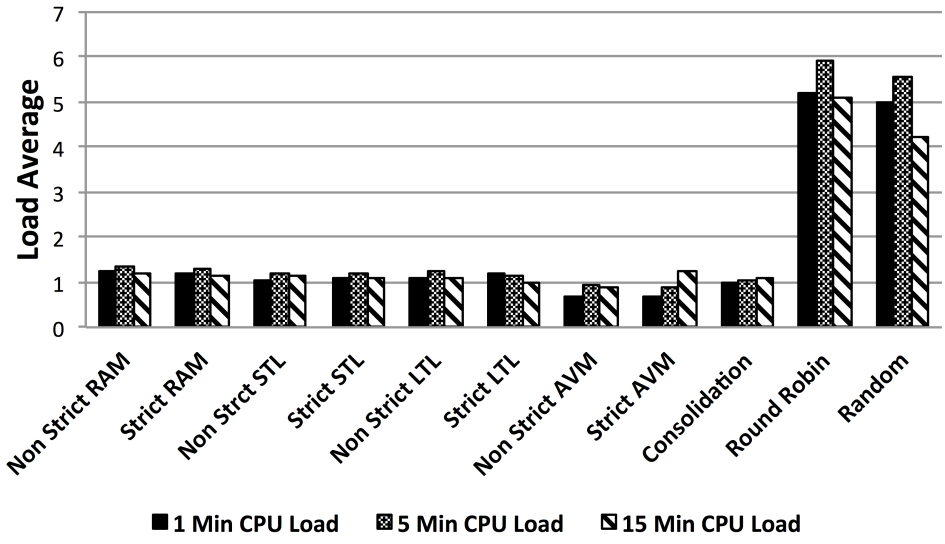
In Figure 6.7 we increase the number of virtual machine instantiation requests to thirty. As shown in Figure 6.7a RAIN is able to successfully instantiate all virtual machines, while round-robin failed to instantiate 6% of virtual machines and random failed to instantiate 3% of virtual machines. As shown in Figure 6.7b round-robin produced average CPU load of between 5.0 and 6.5 and random produced average CPU load of between 3.0 and 3.8. CPU load produced by RAIN is observed to be between 0.8 and 1.3. Average RAM overhead produced by round-robin reached 73% as shown in Figure 6.7c. Average RAM overheads produced by random reached 75%. Average RAM overhead produced by RAIN is observed to be between 57% and 67%. Round robin instantiated 2% more virtual machines than the desired maximum. As illustrated in Figure 6.7d random instantiated 23% more virtual machines than the desired maximum. RAIN instantiated between 11% and 27% less than the desired maximum.

In Figure 6.8 we increase the number of virtual machine instantiations from 30 to 35. Figure 6.8a shows that round-robin has the same failure rate as in the previous trial with 6% of the virtual machine instantiation attempts failing. RAIN has a rejection rate of between 3% and

Figure 6.8: 35 VM, 4 Node, 5 VCore/Core Trial

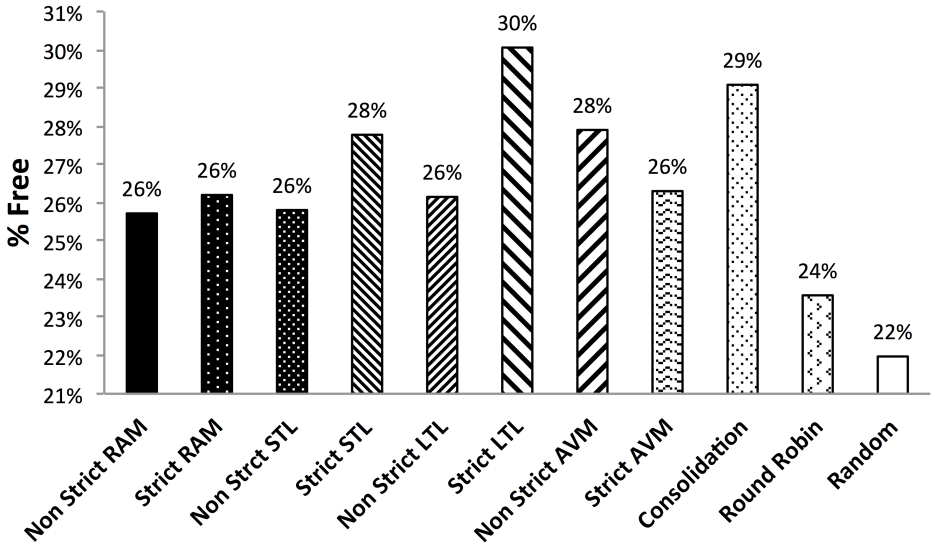



(a) Average Failed VM Instantiations

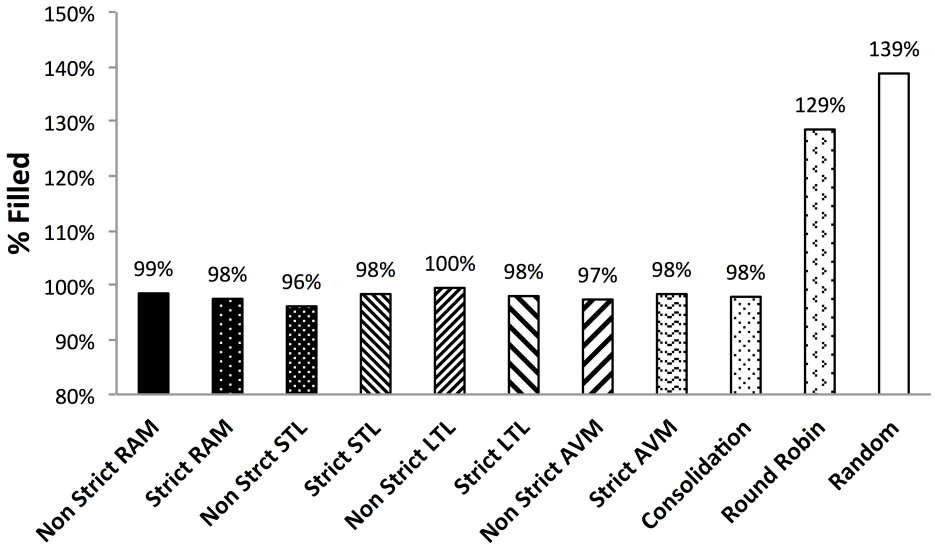



(b) Average Load

Figure 6.8: (Continued) 35 VM, 4 Node, 5 VCore/Core Trial




(c) Average RAM Utilization

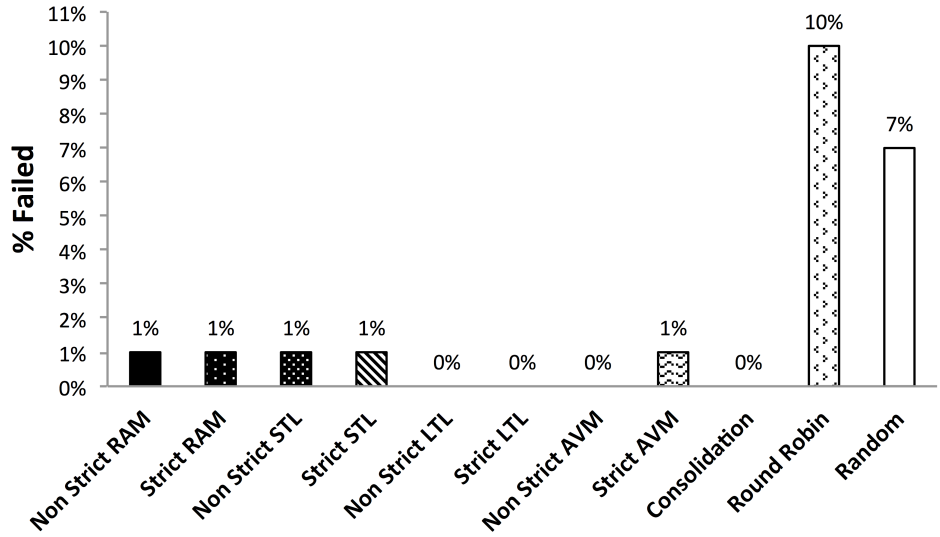



(d) Average VM Fill

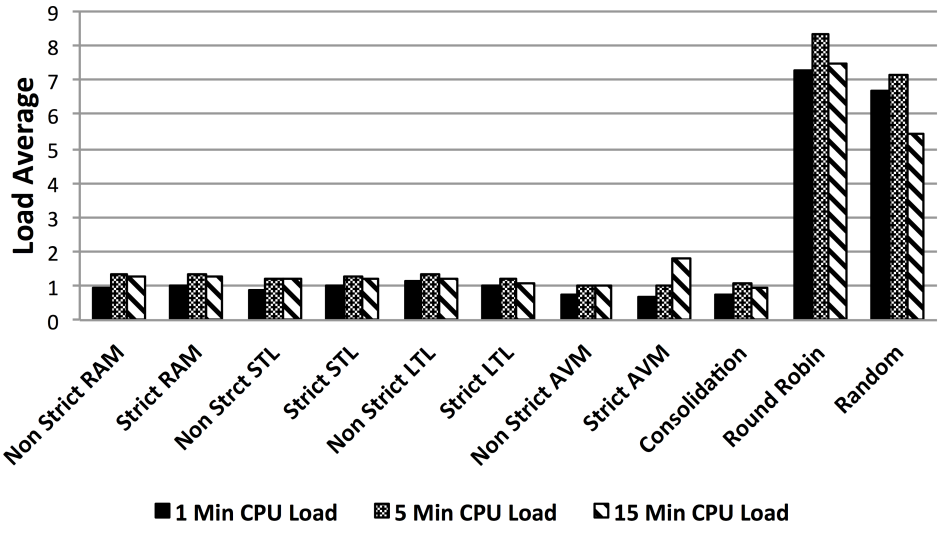
4%. We continue to see the excessive average CPU load as shown in Figure 6.8b with round-robin sustaining an average CPU load of between 5.0 and 5.9. RAIN sustained an average CPU load of between 0.8 and 1.2. As shown in Figure 6.8c the average RAM overhead in round-robin is observed to be 76%, while the average RAM overhead observed in RAIN is between 70% and 74%. In this experiment we observe that round-robin instantiated 29% more virtual machines than the desired maximum as illustrated in Figure 6.8d, while RAIN instantiated between 2% and 0% less than the desired maximum.

Finally, we attempted to instantiate 40 virtual machines with an even more relaxed constraint of 6 VCores/Core. In this experiment we wanted to determine how wide the failure rate gap could be made between round-robin and RAIN. By relaxing the constraint to 6 VCores/Core, more virtual machines may be instantiated per node in RAIN. However, this constraint does not affect round-robin. We see in Figure 6.9a that RAIN sustained a failure rate of between 0% and 1%. The failure rate of round-robin is observed to be 10%, while random sustained a failure rate of 7%. It is clear that the increase of virtual machine instantiation requests resulted in an increase of failure in round-robin. Additionally, resource utilization of round-robin is similar to previous experiments with round-robin reaching an average CPU load of between 7.1 and 8.2 as shown in Figure 6.9b. Conversely, the average CPU load observed in RAIN ranges between 0.8 and 1.9. As shown in Figure 6.9c this experiment is the first in which the RAM overhead produced by RAIN is higher than that of round-robin. This can be explained by the fact that RAIN had a higher success rate in instantiating virtual machines, and therefore, additional RAM is being allocated by the active virtual machines. We see in Figure 6.9d that on average round-robin instantiated 30% more than the optimal maximum per node and random instantiated 24% more than the optimal maximum per node. RAIN instantiated between 4% and 2% less than the optimal maximum per node.

Figure 6.9: 40 VM, 4 Node, 6 VCore/Core Trial

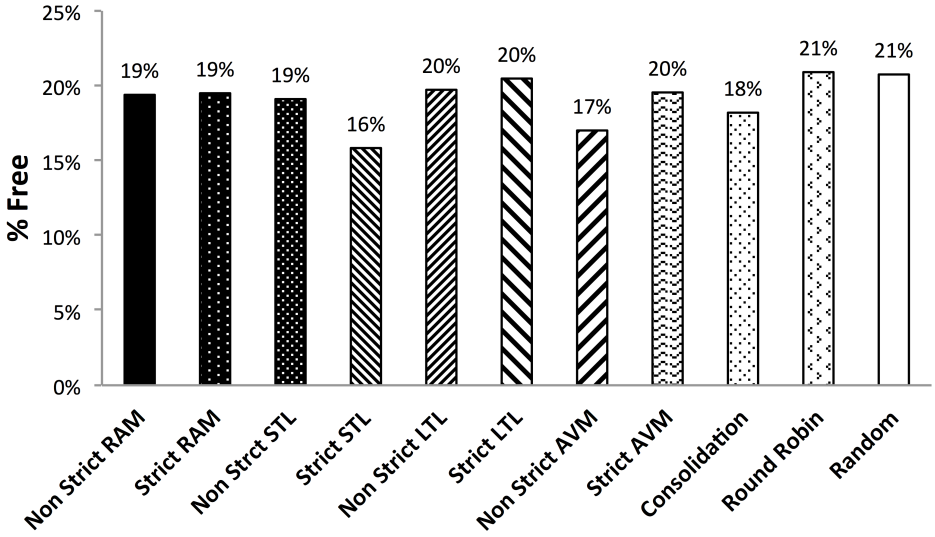



(a) Average Failed VM Instantiations

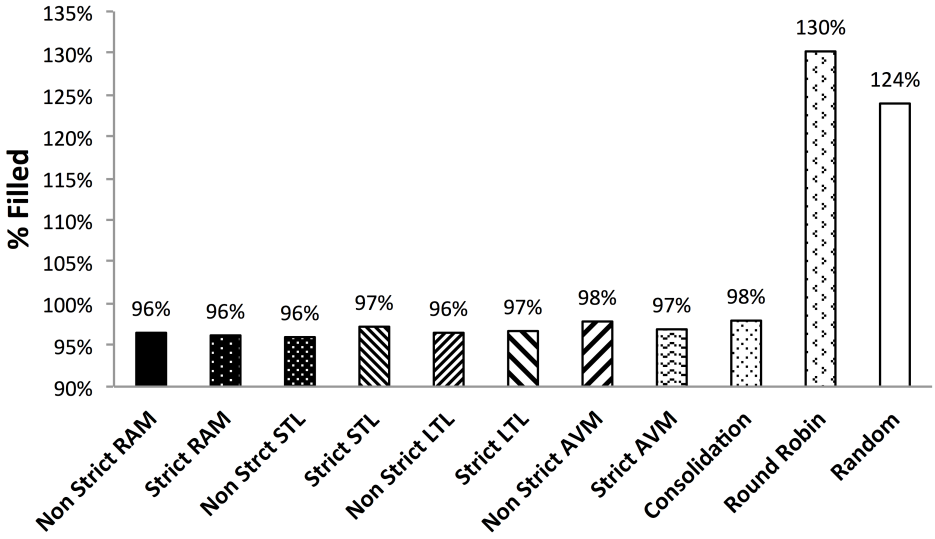



(b) Average Load

Figure 6.9: (Continued) 40 VM, 4 Node, 6 VCore/Core Trial

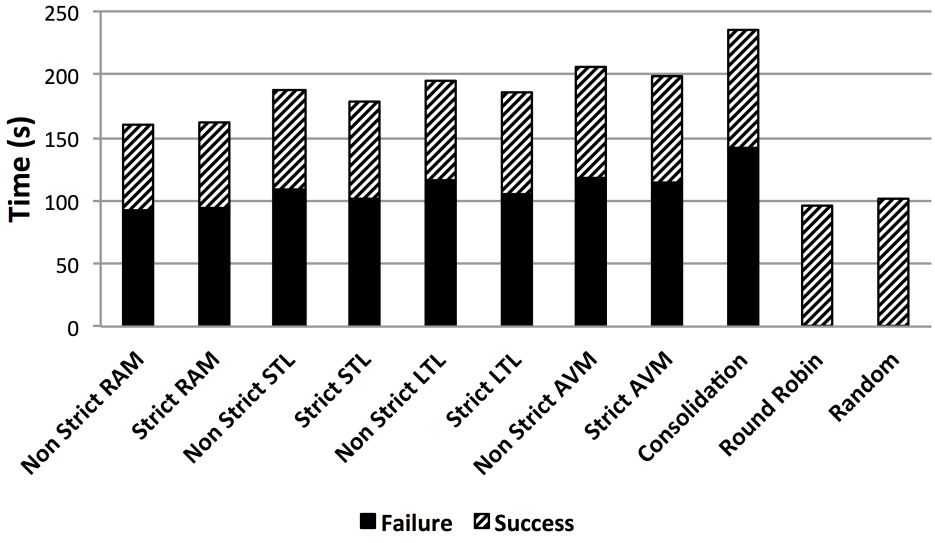



(c) Average RAM Utilization

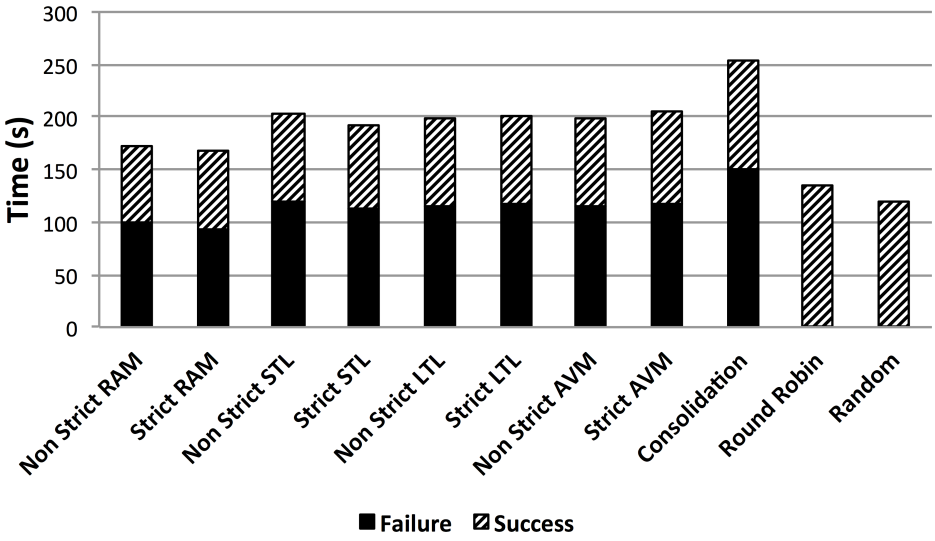



(d) Average VM Fill

Figure 6.10: Turnaround Times for 3 VCore/Core trials

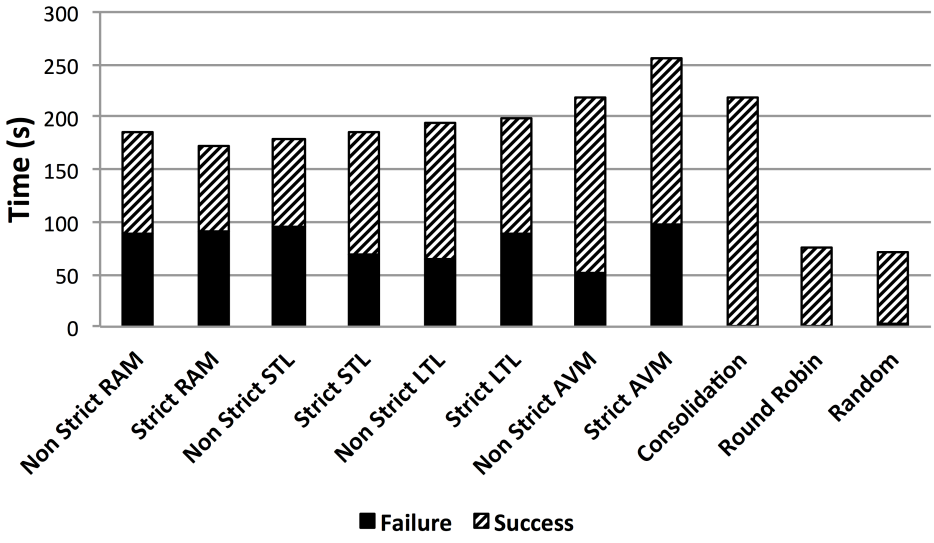



(a) 15 VM, 2 Nodes

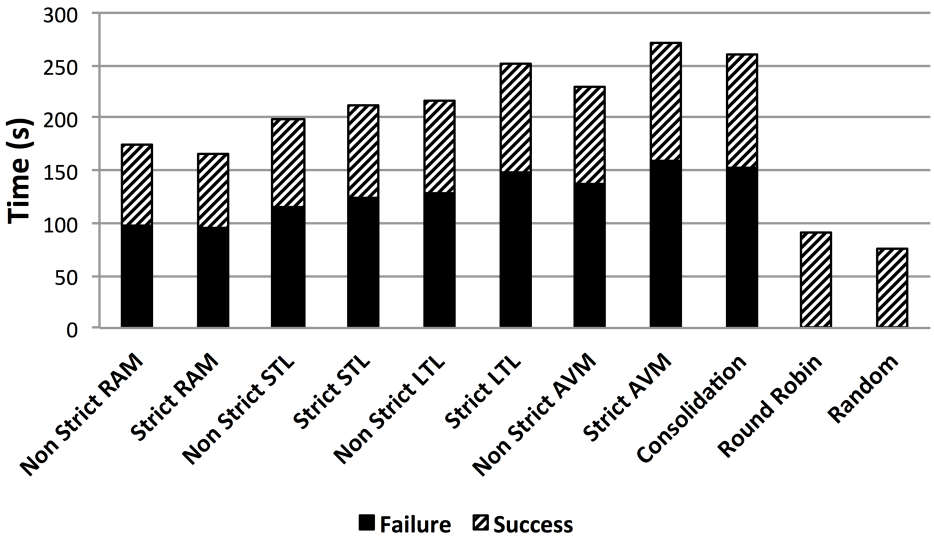



(b) 20 VM, 2 Nodes

Figure 6.10: (Continued) Turnaround Times for 3 VCore/Core trials

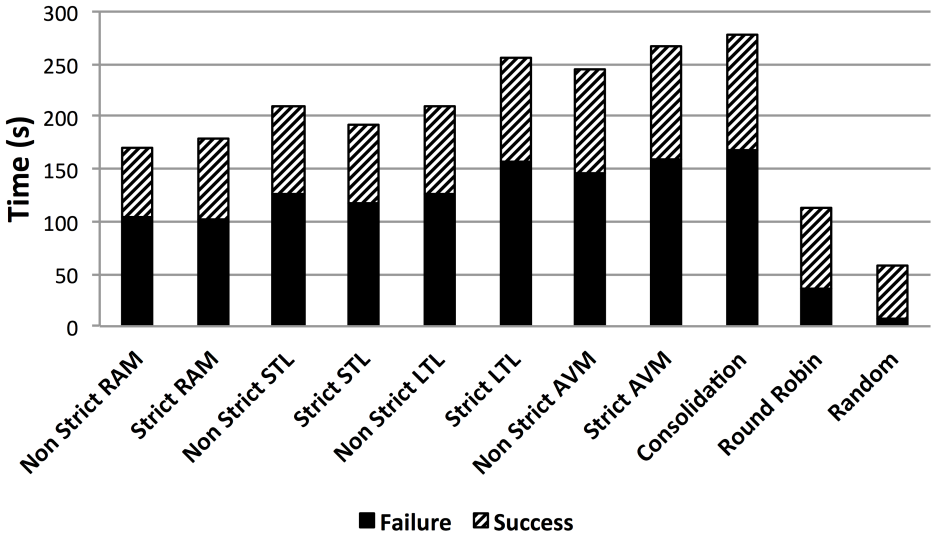



(c) 20 VM, 4 Nodes




(d) 25 VM, 4 Nodes

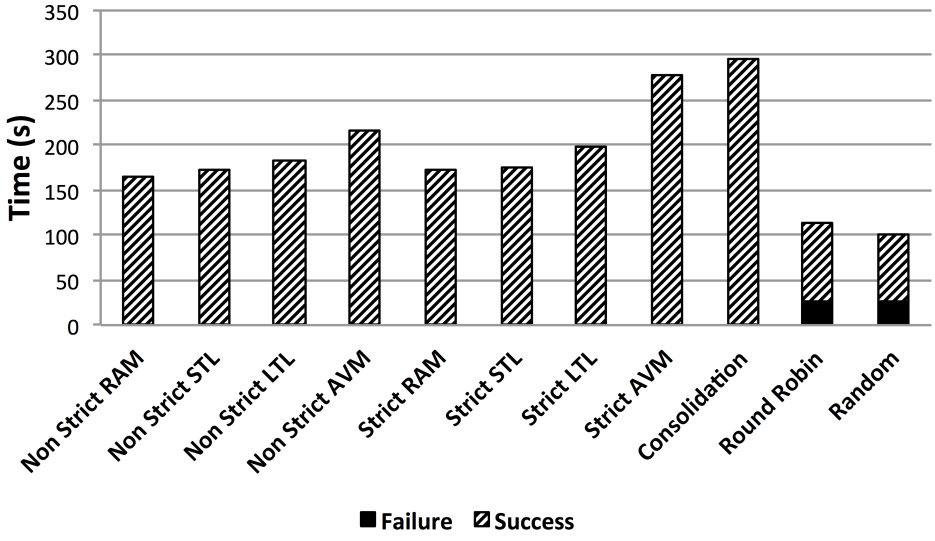
Figure 6.10: (Continued) Turnaround Times for 3 VCore/Core trials



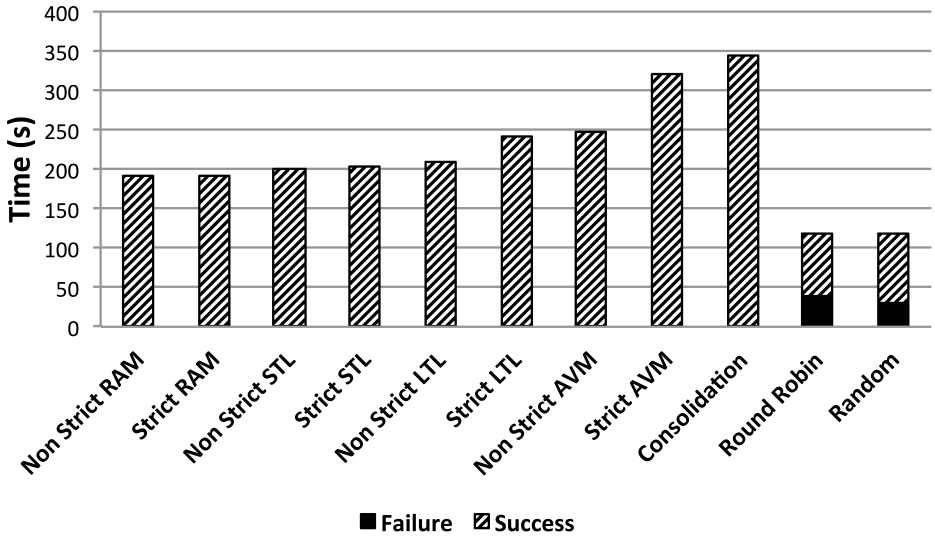

(e) 30 VM, 4 Nodes

Figures 6.10 to 6.12 show the average turnaround times for each experiment as we very them from 3 VCores/Core to 6 VCores/Core. This is the measured time between a virtual machine being requested to either success or failure of the virtual machine. It is clear that RAIN has the undesired side effect of an increased turnaround time over that of round-robin. This increase in turnaround time is due to the fact that RAIN must lock each node during instantiation of a virtual machine in order to ensure that the correct load metrics are acquired before attempting the next instantiation. As stated earlier, all of the virtual machines would be immediately placed on the most optimal node, but as virtual machines are instantiated the selected node quickly becomes non-optimal. This process produces increased turnaround time, but also produces a more optimal virtual machine placement to that of round-robin. The trade off here is therefore, between quick instantiation with round-robin, or better virtual machine placement with RAIN.

Figure 6.11: Turnaround Times for 5 VCore/Core Trials

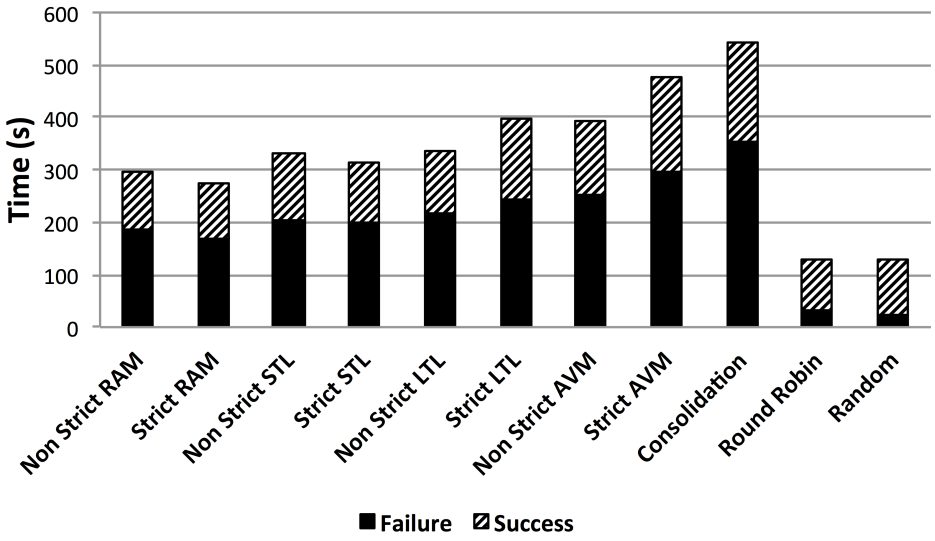



(a) 25 VM, 4 Nodes



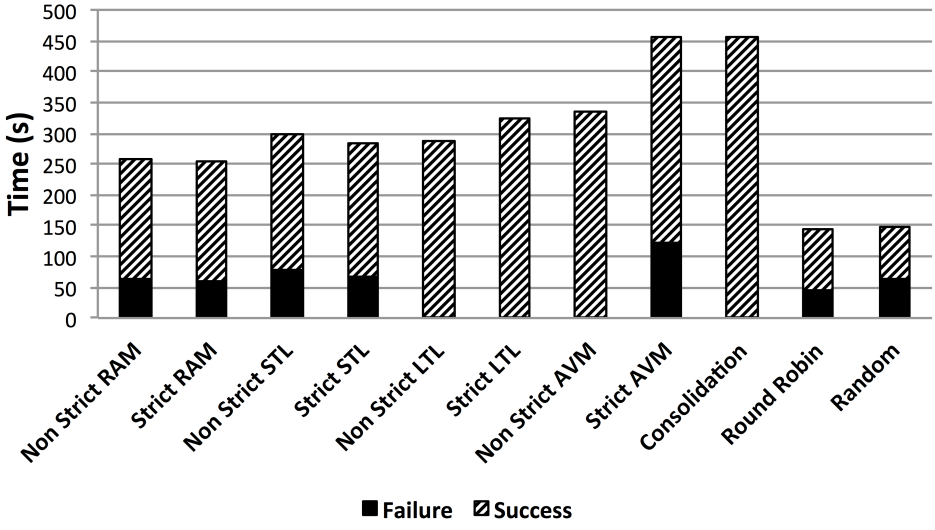

(b) 25 VM, 4 Nodes

Figure 6.11: (Continued) Turnaround Times for 5 VCore/Core Trials




(c) 35 VM, 4 Nodes

Figure 6.12: Turnaround Times for 6 VCore/Core Trial




(a) 40 VM, 4 Nodes

## 6.7 Synthesis

Several conclusions can be made from the experimental results. The RAIN strategy gives fine tuned control over the acceptable level of resource utilization on the host node during instantiation. This is evident by the normalized CPU load measurements as well as the virtual machine fill rate during the RAIN experiments. Conversely, round-robin is a strategy that takes no consideration of the resource load currently applied to compute nodes during the node selection process. This is evident in the results, which show that round-robin overutilizes some nodes, while underutilizing others. The implications of these results are that virtual machine placement based on resource utilization can help to alleviate spikes in CPU load and RAM utilization, which given the right circumstances would normally cause performance issues of colocated virtual machines.

By comparing like experiments we may also be able to see performance trends during phases of elastic growth as well as relaxation of the VCores/Core constraint. In Figure 6.3 and Figure 6.4 we see that increasing the number of compute nodes allows for greater flexibility in the placement of virtual machine instances. Similarly, in Figure 6.5 and Figure 6.7 we see that as we relax the VCores/Core constraint the failure rate drops from a best case of 20% to a best case of 3% for RAIN. Failures that occurred during round-robin fluctuate regardless, considering that the constraint applies only to RAIN. In Figure 6.9 we increase the constraint to 6 VCores/Core and see that the gap in performance between RAIN and round-robin further increases. This indicates that as the constraints of RAIN are relaxed, the performance continues to improve.

The optimal resource priority constants are dependent upon the sizes and types of virtual machines instantiated. However, based on the results we are able to determine the case study which produced the best results most consistently. For each experiment we rank each case study

where the best ranked case study is the one which minimizes resource utilization while maximizing success rate. We found that virtual machine consolidation produced the best overall results out of all strategies. This makes sense considering that with consolidation we attempt to keep virtual machines on a subset of the nodes, which allows some nodes to remain idle. The second best strategy observed in this particular set of experiments non-strictly emphasizes active virtual machines. What this means is that we may optimally choose nodes by selecting those that have the fewest number of active virtual machines but allowing the node choice to vary slightly based on other resource metrics. The worst performing strategy seems to be both strict and non-strict RAM emphasis. It seems as though selecting nodes with the most free RAM does not produce an optimal virtual machine placement, even if other resource metrics are allowed to alter the selection.

# Chapter 7

# FUTURE WORK

As THUNDER is a vertical cloud, it is designed to meet the demands of a particular group of potential users. This group is composed of individuals whose educational and professional experience may not have given the proper exposure to the complex task of managing a distributed system. Specifically, we are interested in helping non-profits in the integration of cloud computing within their respective organizations. The state of the THUNDER architecture as presented in this dissertation is such that we believe the computational demands of the groups of interest will be met sufficiently. However, there are some future improvements which we would like to make and which we believe will improve aspects of the THUNDER architecture for the end user as well as for developers.

Interfacing with the cloud is managed by a web interface written in HTML5 and supports communication between the users and the cloud by using WebSockets. This ensures that the integration between the web interface and the THUNDER communication mechanism is uniform because both WebSockets and the THUNDER communication mechanism are implemented on top of a TCP server. However, the industry standard for web services is to expose an API over a REST [Fielding and Taylor, 2000] interface. REST is the shortened form of Representational State Transfer, and is a standardized protocol for communication and data retrieval between a client and a server over standard HTTP. By implementing a REST interface it may be easier for developers to build alternative interfaces.

As shown in our empirical study on cloud deployment, automation is a feature that a majority of users want. We would like to introduce additional support for automation in the deployment system of THUNDER. Specifically, we are interested in reducing the amount of time required to deploy a node in THUNDER. Currently, the preboot execution environment and installation preseeding are used in the deployment process and normally required about twenty to thirty minutes to complete. This is due to the fact that the entire base operating system is reinstalled during deployment. We would like to be able to keep the same base operating system installation while being able to ensure that the THUNDER software stack may be installed.

More advanced architectures, such as OpenStack, typically have much more robust networking features when compared to that of THUNDER. For example, OpenStack has a dedicated networking controller called Neutron, which provides virtual networking services, thereby allowing a more scalable and dynamic networking configurations. The methodology used by THUNDER is static, such that virtual network interfaces are automatically created during deployment of a node. These interfaces are not able to be changed by the end user. We would like to expand upon the networking capabilities of THUNDER to provide a solution for users to customize aspects of the networking subsystem. For example, the ability to create new subnets on the local network would give the end user more control over the IP addressing of virtual machines.

Hypervisor support on THUNDER is limited to the open source Linux-based KVM solution. We would like to introduce support for other hypervisors such as XEN, Microsoft Hyper-V and ESXi. Performance of virtual machines is directly related to the type of hypervisor that is managing virtual machine instances. KVM is particularly well suited to manage Linux virtual machines. However, Microsoft Hyper-V is arguably a better choice when high performance of Mi-

crosoft Windows virtual machines is desired. Therefore, we are interested in supporting additional hypervisors and making the option to change the desired hypervisor available to the user.

With regard to load balancing and virtual machine placement, we would like to add features that make this process adaptive to the organizations needs. To do this, an artificial neural network may be used to optimize the constraints of RAIN in order to always produce the best strategy for the specific organization. We would also like to add support for live migration of virtual machines as a means to periodically re-balance the computational load of virtual machines in execution.

Additional studies would prove to be important in capturing some quantitative data regarding overall performance of THUNDER and that of virtual machines instantiated by THUNDER. Chapter 5 presents an empirical study on the deployment and management of THUNDER. However, this study focuses on qualitative data regarding the usability of THUNDER with regard to the experiences of the user. Quantitative data could be gathered from THUNDER regarding networking throughput, CPU and I/O overhead, turnaround time for virtual machine instantiation and more. These metrics would be useful in identifying potential bottlenecks and single points of failure.

# Chapter 8

# CONCLUSION

In this dissertation we have shown that cloud computing can be better developed to address the concerns of small organizations, especially non-profits. By reflecting upon the cultural and technical barriers introduced in the literature, and by evaluating the features of popular cloud architectures we built a new cloud solution called "THUNDER Helps Underfunded Nonprofits Distribute Electronic Resources", or simply THUNDER. We believe that THUNDER solves many of the technical challenges regarding deployment, management, and interfacing with cloud resources. By empirical study and the analysis of the results we found that users prefer the deployment solution of THUNDER over that of popular general purpose cloud solutions. We also introduce a novel load balancing strategy called RAIN that considers real time load metrics to make ad hoc decisions regarding virtual machine placement. The performance results of RAIN show that the process by which computational resources are selected to host virtual machine instances greatly affects the virtual machine capacity of the underlying hardware. By optimizing the virtual machine placement we show that we can scale up the capacity of compute nodes without sacrificing performance or introduce harmful wear on the underlying hardware.

From the literature review we discovered that one particular class of organization, non-profit organizations, seem to not be adopting cloud computing as an integral component of daily operations as quickly as other organizations. Explanations for the lack of adoption include lack of qualified staff to manage these systems, fear of new technology, lack of sufficient funding, and

lack of knowledge of the benefits of cloud computing. Based on these explanations we concluded that if a cloud architecture was to be adopted by a non-profit organization it would have to satisfy some minimal requirements in terms of usability. Namely, the system must be easily deployable, and intuitive to use once deployed. In order to address these requirements we proposed a set of questions to guide the research as to how cloud computing can be better presented to non-profit organizations. These challenge questions are as follows:

1. Is it possible for nodes to be discovered automatically on the local network?
2. How can communication between nodes be established without user intervention?
3. Is there a good solution to distributing the required software to each node automatically?
4. Is there an alternative to RSA key sharing with regard to authentication and security?
5. How does THUNDER differ from general purpose architectures?
6. Can we produce some qualitative or quantitative metrics to demonstrate that THUNDER is better for users of non-technical backgrounds?
7. Are there any other constraints to deployment that have not been uncovered?

In response to challenge questions 1 and 2, we employ a multicast based automatic node discovery mechanism such that machines on the local network may establish a connection with one another without human intervention. To address challenge question number 3, a number of approaches were considered but ultimately the most intuitive solution was found to be the use of the preboot execution environment and preseeding. This process allows nodes to be rebooted and imaged with a new filesystem. Preseeding is the process by which the imaging process becomes

automated and pre-selected software packages are installed without human intervention. The result of the preseeding process is a node with the THUNDER software stack installed along with all pre-requisite packages. Challenge 4 is solved in THUNDER with the use of pre-shared key challenge/response authentication. In this strategy a pre-shared key is used to encrypt a message by the authenticating node. This message is decrypted by a challenging node, upon which the decrpyted message is sent back as a response. The authenticating node then compares the decrypted message with the original challenge and permits authentication if and only if the decrypted message completely matches the original challenge. This allows for authentication without a human having to generate RSA keys for each node and sharing the keys among the rest of the nodes in the cloud.

THUNDER is defined as a vertical architecture such that not all layers of a traditional cloud architecture are implemented. What THUNDER does support is the basic infrastructure to allow organizations to reduce costs by utilizing virtual machines and low cost thin client devices as opposed to thick traditional desktop computers. Although, a general purpose cloud architecture would also support this infrastructure, we introduce additional measures to ensure that struggling organizations can successfully deploy a local cloud. During the course of this work we conducted several anonymous empirical studies in response to challenge question 5 and 6. In these studies we asked volunteers to deploy THUNDER as well as either Eucalyptus or Openstack. What we found as a result of these empirical studies is that the method of deployment used by THUNDER is more intuitive and less complicated when compared to that of popular general purpose cloud architectures. Additionally, in response to challenge question 7, the empirical studies helps to illuminate potential pitfalls with regard to the deployment process, which helped to further guide the research.

In addition to responding to the proposed challenge questions, we wanted to add robust

load balancing features. Virtual machine load balancing further enables cost saving measures by means of the ability to instantiate the most virtual machines without overutilization of hardware. One pitfall discovered in many widely general purpose architectures is that fact that round robin is typically used as a node selection algorithm. Optimal node selection in an $N$ node system is equivalent to the traditional bin packing problem and is therefore within the class of NP-complete problems. For this reason, typically a node selection algorithm will use a simple approach like round robin or first fit. We found that given a high enough VCore/Core constraint RAIN allows for more virtual machines to be instantiated without overutilization and helps to alleviate spikes in CPU load, which contributes to potentially poor performance of co-located virtual machines.

We accomplished our goal of producing a cloud architecture that better addresses the concerns of small organizations without sacrificing any of the cloud attributes, including scalability, elasticity, and on-demand virtualization. In addition, we show that the performance of RAIN improves upon common load balancing solutions, especially in heterogeneous configurations. The combined features of THUNDER make it a convenient, easy to use, and easy to deploy vertical cloud architecture.

REFERENCES

Ajit, M. and G. Vidya (2013, July). Vm level load balancing in cloud environment. In *Computing, Communications and Networking Technologies (ICCCNT),2013 Fourth International Conference on*, pp. 1–5.

Alabbadi, M. (2011). Cloud computing for education and learning: Education and learning as a service (elaas). In *Interactive Collaborative Learning (ICL), 2011 14th International Conference on*, pp. 589–594.

Alexander, S. and R. Droms (1993, October). Dhcp options and bootp vendor extensions. RFC 1533.

Alexander, S. and R. Droms (1997, March). Dhcp options and bootp vendor extensions. RFC 2132.

Amazon Web Services, inc. (2013, September). Amazon elastic compute cloud. http://aws.amazon.com/ec2/. Accessed: 9/27/2013.

An, K., S. Pradhan, F. Caglar, and A. Gokhale (2012). A publish/subscribe middleware for dependable and real-time resource monitoring in the cloud. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, SDMCMM '12, New York, NY, USA, pp. 3:1–3:6. ACM.

Apache Foundation (2013, April). Apache cloudstack. http://cloudstack.apache.org/. Accessed: 4/12/2013.

Apostol, E., I. Baluta, A. Gorgoi, and V. Cristea (2011, August). Efficient manager for virtualized resource provisioning in cloud systems. In *Intelligent Computer Communication and Processing (ICCP), 2011 IEEE International Conference on*, pp. 511 –517.

Azeez, A., S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle (2010, July). Multi-tenant soa middleware for cloud computing. pp. 458 –465.

Baev, S., C. Weaver, and C. Weaver (2011, December). Cost-efficient desktop vitualization experience at georgia southwestern state university. *J. Comput. Sci. Coll. 27*(2), 188–195.

Baliga, J., R. Ayre, K. Hinton, and R. Tucker (Jan.). Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE 99*(1), 149–167.

Banister, S. (2010). Integrating the iPod touch in k-12 education: Visions and vices. *Computers in the Schools 27*(2), 121–131.

Barrett, D. J., R. E. Silverman, and R. G. Byrnes (2005). *SSH, The Secure Shell: The Definitive Guide*. O'Reilly Media.

Behrend, T. S., E. N. Wiebe, J. E. London, and E. C. Johnson (2011, March). Cloud computing adoption and usage in community colleges. *Behav. Inf. Technol. 30*(2), 231–240.

Berson, A. (1992). *Client/server architecture*. New York, NY, USA: McGraw-Hill, Inc.

Birman, K. and T. Joseph (1987, November). Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev. 21*(5), 123–138.

Campbell, R., M. Mirko, and F. Reza (2011). A middleware for assured clouds. *Journal of Interent Services and Applications*.

Canonical Group Ltd. (2010). An introduction to cloud computing. Technical report, Canonical.

Canonical, Ltd. (2015, January). Init scripts for use on cloud images. `https://launchpad.net/cloud-init`. Accessed: 1/24/2015.

Cao, J., K. Li, and I. Stojmenovic (2014, Jan). Optimal power allocation and load distribution for multiple heterogeneous multicore server processors across clouds and data centers. *Computers, IEEE Transactions on 63*(1), 45–58.

Cappos, J., I. Beschastnikh, A. Krishnamurthy, and T. Anderson (2009). Seattle: a platform for educational cloud computing. In *Proceedings of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, pp. 111–115.

Cardellini, V. and S. Iannucci (2012). Designing a flexible and modular architecture for a private cloud: a case study. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, VTDC '12, pp. 37–44.

CBR Online (2013, January). Uk smes embrace cloud during recession: Survey. `http://www.cbronline.com/news/tech/software/appdev/uk_smes_embrace_cloud_during_recession_survey_230309`. Accessed: 1/7/2013.

Cherkasova, L., D. Gupta, and A. Vahdat (2007a, September). Comparison of the three cpu schedulers in xen. *ACM SIGMETRICS Performance Evaluation Review 35*(2), 42–51.

Cherkasova, L., D. Gupta, and A. Vahdat (2007b). When virtual is harder than real: Resource allocation challenges in virtual machine based it environments. Technical report, HP.

Ciardo, G., A. Blakemore, J. Chimento, Philip F., J. K. Muppala, and K. S. Trivedi (1993). Au-

tomated generation and analysis of markov reward models using stochastic reward nets. In C. Meyer and R. Plemmons (Eds.), *Linear Algebra, Markov Chains, and Queueing Models*, Volume 48 of *The IMA Volumes in Mathematics and its Applications*, pp. 145–191. Springer New York.

Constantine, M., B. Wallace, and M. Kindaichi (2005). Examining contextual factors in the career decision status of african american adolescents. *Journal of Career Assessment 13*(3), 307–319.

Cooper, B. (2010, November). The prickly side of building clouds. *Internet Computing, IEEE 14*(6), 64 –67.

Daniel, D. and S. Lovesum (2011, July). A novel approach for scheduling service request in cloud with trust monitor. In *Signal Processing, Communication, Computing and Networking Technologies (ICSCCN), 2011 International Conference on*, pp. 509–513.

Delgado, M. (2011). The evolution of health care it: Are current u.s. privacy policies ready for the clouds? In *Services (SERVICES), 2011 IEEE World Congress on*, pp. 371–378.

Distributed Redundant Block Devices (2013). Drbd. http://www.drbd.org/. Accessed: 4/12/2013.

Doelitzscher, F., A. Sulistio, C. Reich, H. Kuijs, and D. Wolf (2011, January). Private cloud for collaboration and e-learning services: from iaas to saas. *Computing 91*(1), 23–42.

ECMA International (2013, October). Ecma-404 the json data interchange format. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`. Accessed: 4/8/2015.

Edmonds, A., S. Johnston, T. Metsch, and G. Mazzaferro (2010, March). Open Cloud Computing Interface - Core and Models. Technical report, OCCI.

Ercan, T. (2010). Effective use of cloud computing in educational institutions. *Procedia- Social and Behavioral Sciences 2*(2), 938–942.

Eucalyptus Systems (2013a, February). Ec2 tools. http://www.eucalyptus.com/eucalyptus-cloud/tools/ec2. Accessed: 4/12/2013.

Eucalyptus Systems (2013b, February). Eucalyptus cloud. http://www.eucalyptus.com/. Accessed: 4/12/2013.

Eucalyptus Systems (2014, August). Eucalyptus faststart. https://www.eucalyptus.com/eucalyptus-cloud/get-started. Accessed: 9/15/2014.

Feng, W.-c. (2003, October). Making a case for efficient supercomputing. *Queue 1*(7), 54–64.

Fette, I. and A. Melnikov (2011, December). The websocket protocol. RFC 6455.

Fielding, R. T. and R. N. Taylor (2000). Principled design of the modern web architecture. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, New York, NY, USA, pp. 407–416. ACM.

Foster, I., C. Kesselman, and S. Tuecke (2001, August). The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl. 15*(3), 200–222.

Foster, I. T. and C. Kesselman (2000). Computational grids. In *VECPAR*, pp. 3–37.

Galloway, J. M., G. Loewen, and S. V. Vrbsky (2014). Deploying a cost efficient geographically distributed private cloud architecture. *International Journal of Cloud Computing 3*(2), 125–142.

Ghosh, R., F. Longo, F. Frattini, S. Russo, and K. Trivedi (2014, Jan). Scalable analytics for iaas cloud availability. *Cloud Computing, IEEE Transactions on 2*(1), 57–70.

Glyptodon LLC. (2015, January). Guacamole - html5 clientless remote desktop. `http://guac-dev.org/`. Accessed: 1/24/2015.

Goto, Y. (2011). Kernel-based virtual machine technology. *Fujitsu Science Technology Journal 47*, 362–368.

Grandison, T., E. M. Maximilien, S. Thorpe, and A. Alba (2010). Towards a formal definition of a computing cloud. *Services, IEEE Congress on 0*, 191–192.

Grosu, D., A. T. Chronopoulos, and M. Y. Leung (2008, November). Cooperative load balancing in distributed systems. *Concurr. Comput. : Pract. Exper. 20*(16), 1953–1976.

Hu, J., J. Gu, G. Sun, and T. Zhao (2010, Dec). A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. In *Parallel Architectures, Algorithms and Programming (PAAP), 2010 Third International Symposium on*, pp. 89–96.

IBM (2019, July). The benefits of cloud computing - a new era of responsiveness, effectiveness and efficiency in it service delivery. *Dynamic Infrastructure*.

Iperf team (2013). Iperf. `http://iperf.sourceforge.net/`. Accessed: 4/12/2013.

Jamshidi, P., A. Ahmad, and C. Pahl (2013, July). Cloud migration research: A systematic review. *Cloud Computing, IEEE Transactions on 1*(2), 142–157.

Kansal, K. J. and I. Chana (2012). Existing load balancing techniques in cloud computing: A systematic review. *Journal of Information Systems and Communication 3*(1), 87–91.

Khalidi, Y. (2011, March). Building a cloud computing platform for new possibilities. *Computer 44*(3), 29 –34.

Kim, D., H. Kim, M. Jeon, E. Seo, and J. Lee (2008). Guest-aware priority-based virtual machine scheduling for highly consolidated server. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, Berlin, Heidelberg, pp. 285–294. Springer-Verlag.

Koomey, J. (2007). Estimating total power consumption by servers in the u.s. and the world. Technical report, Lawrence National Laboratory, Stanford University.

Lanjewar, S. M., S. S. Surwade, S. P. Patil, and P. S. Ghumatkar (2014, February). Load balancing in public cloud. *Journal of Computer Engineering 16*(1), 82–87.

Lee, S.-Y., D. Tang, T. Chen, and W.-C. Chu (2012, July). A qos assurance middleware model for enterprise cloud computing. In *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pp. 322 –327.

Licklider, J. C. R. and R. W. Taylor (1968). The computer as a communication device. *Science and Technology 76*, 21–31.

Linux Terminal Server Project (2013). Ltsp. http://www.ltsp.org/. Accessed: 4/12/2013.

Liu, F., J. Tong, J. Mao, R. Bohn, J. Messina, M. Badger, and D. Leaf (2011). Nist cloud computing reference architecture. Technical report, U.S. Department of Commerce.

Loewen, G. and S. Vrbsky (2014, November). Study on cloud development: General purpose vs. vertical architecture. IRB # 14-OR-040.

Mahajan, K. and D. Dahiya (2014, Aug). A cloud based deployment framework for load balancing policies. In *Contemporary Computing (IC3), 2014 Seventh International Conference on*, pp. 565–570.

Masud, M. and X. Huang (2011). Esaas: A new education software model in e-learning systems. In M. Zhu (Ed.), *Information and Management Engineering*, Volume 235 of *Communications in Computer and Information Science*, pp. 468–475. Springer Berlin Heidelberg.

Metz, C. (1998). Ip routers: new tool for gigabit networking. *Internet Computing, IEEE 2*(6), 14–18.

Nitoh, S., H. Nagakura, and A. Sakurai (2011, October). Middleware for supporting private clouds. *FUJITSU Science and Technology 47*(4), 451–458.

OpenNebula (2013). Open source data center virtualization. http://www.opennebula.org/. Accessed: 4/12/2013.

openQRM (2013). openqrm - leading open source data center and cloud computing platform. http://www.openqrm.com/. Accessed: 4/12/2013.

OpenStack Foundation (2013a, February). Openstack cloud software. http://www.openstack.org/. Accessed: 4/12/2013.

OpenStack Foundation (2013b, February). Openstack nova. http://nova.openstack.org/. Accessed: 4/12/2013.

Openstack Foundation (2014, August). Devstack. http://devstack.org/. Accessed: 9/15/2014.

Pew Research Center (2010, April). Tracking survey. pew research center internet and american life project. http://www.pewinternet.org/. Accessed: 3/8/2013.

Pew Research Center (2012, April). Digital differences. pew research center internet and american life project. http://www.pewinternet.org/. Accessed: 3/9/2013.

Puppet Labs (2015, January). Puppet labs: It automation software for system administrators. `http://puppetlabs.com/`. Accessed: 1/24/2015.

Ranabahu, A. and M. Maximilien (2009). A best practice model for cloud middleware systems. In *Best Practices in Cloud Computing: Designing for the Cloud workshop*, pp. 41–51.

Raspberry Pi Foundation (2013). Raspberry pi. `http://www.raspberrypi.org/`. Accessed: 4/12/2013.

Redhat, Inc. (2013). Kvm - kernel-based virtual machine. http://www.redhat.com/resourcelibrary/whitepapers/doc-kvm. Accessed: 4/12/2013.

Richardson, T. (2010, November). The rfb protocol. Technical report.

Samba team (2013). Samba. `http://www.samba.org/`. Accessed: 4/12/2013.

Shin, D., Y. Wang, and W. Claycomb (2012). A policy-based decentralized authorization management framework for cloud computing. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, New York, NY, USA, pp. 465–470. ACM.

Sotomayor, B., K. Keahey, and I. Foster (2006, Nov). Overhead matters: A model for virtual resource management. In *Virtualization Technology in Distributed Computing, 2006. VTDC 2006. First International Workshop on*, pp. 5–5.

Sotomayor, B., K. Keahey, and I. Foster (2007). Enabling cost-effective resource leases with virtual machines. In *Hot Topics session in ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC 2007)*.

Sourceware (2013). Lvm2 resources. http://sourceware.org/lvm2/. Accessed: 4/12/2013.

Stanford University (2011). Folding@home distributed computing. `http://folding.stanford.edu`. Accessed: 4/12/2013.

Stein, S., J. Ware, J. Laboy, and H. E. Schaffer (2013). Improving k-12 pedagogy via a cloud designed for education. *International Journal of Information Management 33*(1), 235 – 241.

Stevens, W., B. Fenner, and A. Rudoff (2004). *UNIX Network Programming*. Number v. 1 in Addison-Wesley professional computing series. Addison-Wesley.

Sultan, N. (2010). Cloud computing for education: A new dawn? *International Journal of Information Management 30*(2), 109 – 116.

Sun, H., X. Wang, C. Zhou, Z. Huang, and X. Liu (2010, September). Early experience of building a cloud platform for service oriented software development. pp. 1 –4.

Surhone, L., M. Timpledon, and S. Marseken (2010). *Remote Desktop Protocol*. VDM Publishing.

Syslinux (2013). Pxelinux. http://www.syslinux.org/pxe.php. Accessed: 4/12/2013.

The jQuery Foundation (2015, January). jquery. `http://www.jquery.com/`. Accessed: 1/24/2015.

The National Computing Group (2015, January). The national computing centre. `http://www.ncc.co.uk/`. Accessed: 1/24/2015.

Tian, W. (2008, December). Three ways to improve the efficiency of virtual/cloud computing lab. In *Apperceiving Computing and Intelligence Analysis, 2008. ICACIA 2008. International Conference on*, pp. 160–163.

Toby Velte, Anthony Velte, R. E. (2009, September). *Cloud Computing, A Practical Approach* (1 ed.). McGraw-Hill.

Tsai, C.-W., W.-C. Huang, M.-H. Chiang, M.-C. Chiang, and C.-S. Yang (2014, April). A hyper-heuristic scheduling algorithm for cloud. *Cloud Computing, IEEE Transactions on 2*(2), 236–250.

Van, H. N., F. Tran, and J.-M. Menaud (July). Performance and power management for cloud infrastructures. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pp. 329–336.

Vereecken, W., L. Deboosere, P. Simoens, B. Vermeulen, D. Colle, C. Develder, M. Pickavet, B. Dhoedt, and P. Demeester (2009). Energy efficiency in thin client solutions. In A. D.

Doulamis, J. Mambretti, I. Tomkos, and T. A. Varvarigou (Eds.), *Networks for Grid Applications - Third International ICST Conference, GridNets 2009, Athens, Greece, September 8-9, 2009, Revised Selected Papers*, Volume 25, pp. 109–116. Springer.

Vrbsky, S. V., M. Galloway, R. Carr, R. Nori, and D. Grubic (2013, July). Decreasing power consumption with energy efficient data aware strategies. *Future Gener. Comput. Syst. 29*(5), 1152–1163.

Weiss, A. (2007, December). Computing in the clouds. *netWorker 11*(4), 16–25.

Zeng, W., Y. Zhao, K. Ou, and W. Song (2009). Research on cloud storage architecture and key technologies. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ICIS '09, New York, NY, USA, pp. 1044–1048. ACM.

Zhang, L.-J. and Q. Zhou (July). Ccoa: Cloud computing open architecture. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pp. 607–616.

Zhang, Q., M. Zhani, R. Boutaba, and J. Hellerstein (2013, July). Harmony: Dynamic heterogeneity-aware resource provisioning in the cloud. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pp. 510–519.

Zhang, Q., M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, and J. L. Hellerstein (2012). Dynamic energy-aware capacity provisioning for cloud computing environments. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, New York, NY, USA, pp. 145–154. ACM.

Zhang, Z., L. Xiao, Y. Tao, J. Tian, S. Wang, and H. Liu (2013, Oct). A model based load-balancing method in iaas cloud. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pp. 808–816.

Zhu, X., L. Yang, H. Chen, J. Wang, S. Yin, and X. Liu (2014, April). Real-time tasks oriented energy-aware scheduling in virtualized clouds. *Cloud Computing, IEEE Transactions on 2*(2), 168–180.

# Appendices

# Appendix A
# RESOURCE PRIORITY CONSTANTS

Table A.1: Resource Priority Constants for Case Study on Non-Strict RAM Emphasis.

| Metric | Priority Constant | Value Used |
|---|---|---|
| RAM Utilization | $K_{RAM}$ | 0.8 |
| 1 Minute Avg. Load | $K_{Load_1}$ | 0.05 |
| 5 Minute Avg. Load | $K_{Load_5}$ | 0.05 |
| 15 Minute Avg. Load | $K_{Load_{15}}$ | 0.05 |
| Active VM Instances | $K_{ActiveVMs}$ | 0.05 |

Table A.2: Resource Priority Constants for Case Study on Non-Strict Long Term CPU Load (LTL) Emphasis.

| Metric | Priority Constant | Value Used |
|---|---|---|
| RAM Utilization | $K_{RAM}$ | 0.05 |
| 1 Minute Avg. Load | $K_{Load_1}$ | 0.2 |
| 5 Minute Avg. Load | $K_{Load_5}$ | 0.3 |
| 15 Minute Avg. Load | $K_{Load_{15}}$ | 0.4 |
| Active VM Instances | $K_{ActiveVMs}$ | 0.05 |

Table A.3: Resource Priority Constants for Case Study on Non-Strict Short Term CPU Load (STL) Emphasis.

| Metric | Priority Constant | Value Used |
|---|---|---|
| RAM Utilization | $K_{RAM}$ | 0.05 |
| 1 Minute Avg. Load | $K_{Load_1}$ | 0.4 |
| 5 Minute Avg. Load | $K_{Load_5}$ | 0.3 |
| 15 Minute Avg. Load | $K_{Load_{15}}$ | 0.2 |
| Active VM Instances | $K_{ActiveVMs}$ | 0.05 |

Table A.4: Resource Priority Constants for Case Study on Non-Strict Active VM (AVM) Emphasis.

| Metric | Priority Constant | Value Used |
|---|---|---|
| RAM Utilization | $K_{RAM}$ | 0.05 |
| 1 Minute Avg. Load | $K_{Load_1}$ | 0.05 |
| 5 Minute Avg. Load | $K_{Load_5}$ | 0.05 |
| 15 Minute Avg. Load | $K_{Load_{15}}$ | 0.05 |
| Active VM Instances | $K_{ActiveVMs}$ | 0.8 |

Table A.5: Resource Priority Constants for Case Study on Strict RAM Emphasis.

| Metric | Priority Constant | Value Used |
|---|---|---|
| RAM Utilization | $K_{RAM}$ | 1.0 |
| 1 Minute Avg. Load | $K_{Load_1}$ | 0.0 |
| 5 Minute Avg. Load | $K_{Load_5}$ | 0.0 |
| 15 Minute Avg. Load | $K_{Load_{15}}$ | 0.0 |
| Active VM Instances | $K_{ActiveVMs}$ | 0.0 |

Table A.6: Resource Priority Constants for Case Study on Strict Long Term CPU Load (LTL) Emphasis.

| Metric | Priority Constant | Value Used |
|---|---|---|
| RAM Utilization | $K_{RAM}$ | 0.0 |
| 1 Minute Avg. Load | $K_{Load_1}$ | 0.0 |
| 5 Minute Avg. Load | $K_{Load_5}$ | 0.1 |
| 15 Minute Avg. Load | $K_{Load_{15}}$ | 0.9 |
| Active VM Instances | $K_{ActiveVMs}$ | 0.0 |

Table A.7: Resource Priority Constants for Case Study on Strict Short Term CPU Load (STL) Emphasis.

| Metric | Priority Constant | Value Used |
|---|---|---|
| RAM Utilization | $K_{RAM}$ | 0.0 |
| 1 Minute Avg. Load | $K_{Load_1}$ | 0.9 |
| 5 Minute Avg. Load | $K_{Load_5}$ | 0.1 |
| 15 Minute Avg. Load | $K_{Load_{15}}$ | 0.0 |
| Active VM Instances | $K_{ActiveVMs}$ | 0.0 |

Table A.8: Resource Priority Constants for Case Study on Strict Active VM (AVM) Emphasis.

| Metric | Priority Constant | Value Used |
|---|---|---|
| RAM Utilization | $K_{RAM}$ | 0.0 |
| 1 Minute Avg. Load | $K_{Load_1}$ | 0.0 |
| 5 Minute Avg. Load | $K_{Load_5}$ | 0.0 |
| 15 Minute Avg. Load | $K_{Load_{15}}$ | 0.0 |
| Active VM Instances | $K_{ActiveVMs}$ | 1.0 |

Table A.9: Resource Priority Constants for Case Study Without Resource Emphasis (Consolidation).

| Metric | Priority Constant | Value Used |
|---|---|---|
| RAM Utilization | $K_{RAM}$ | 0.0 |
| 1 Minute Avg. Load | $K_{Load_1}$ | 0.0 |
| 5 Minute Avg. Load | $K_{Load_5}$ | 0.0 |
| 15 Minute Avg. Load | $K_{Load_{15}}$ | 0.0 |
| Active VM Instances | $K_{ActiveVMs}$ | 0.0 |

# Appendix B
# IRB CONSENT FORM



**UNIVERSITY OF ALABAMA**
**Informed Consent for a Research Study**

You are being asked to take part in a research study. This study is called *Study on Cloud Deployment: General Purpose vs. Vertical Architectures.* The study is being done by *Gabriel Loewen* who is a PhD *student* at the University.

**What is this study about?**

This study is being conducted to compare the complexity of system deployment of Euclayptus and THUNDER.

**Why is this study important--What good will the results do?**

This study will help us better understand the implications of deploying the THUNDER architecture, and whether or not it is better than the deployment of Eucalyptus.

**Why have I been asked to take part in this study?**

You have been asked to be in this study because you are enrolled in CS491/CS591. However, you must be 19 years or older to participate in this study.

**How many people besides me will be in this study?**

About 30 other people will be in this study.

**What will I be asked to do in this study?**

If you decide to be in this study, you will be asked to do these things:

> *You will be asked to fill out a short survey containing 1) demographic questions: your work experience, major and class level and 2) questions evaluating THUNDER. If you consent to the study, you are giving permission for the researchers to analyze the information provided during the testing exercises in the cloud lab.*

**How much time will I spend being in this study?**

Participation in the study will require no extra time on your part except for filling out a short survey containing demographic questions and a questionnaire evaluating THUNDER. This should task about 5 minutes.

**Will I be paid for being in this study?**

No. You will receive class credit for successful completion of the in-class exercises whether or not you consent to participate in the study.

**Will being in this study cost me anything?**

There will be no cost to you.

**Can the researcher take me out of this study?**

UNIVERSITY OF ALABAMA IRB
CONSENT FORM APPROVED: 2-12-14
EXPIRATION DATE: 2-11-15

The researcher may take you out of the study if you drop the course.

**What are the benefits (good things) that may happen to me if I am in this study?**

There are no direct benefits to you from being in this study.

**What are the benefits to scientists or society?**

This study will help the researchers understand which directions to take with regard to the development of the THUNDER architecture.

**What are the risks (dangers or harm) to me if I am in this study?**

There are no risks to participating in this study. All data will be kept confidential (See below).

**How will my confidentiality (privacy) be protected? What will happen to the information the study keeps on me?**

All information you provide will be anonymized. Therefore, your name will not be associated with your data.

**What are the alternatives to being in this study? Do I have other choices?**

The alternative/other choice is not to participate.

**What are my rights as a participant?**

Taking part in this study is voluntary—it is your free choice. You may choose not to take part at all. If you start the study, you can stop at any time. Leaving the study will not result in any penalty or loss of any benefits you would otherwise receive.

**Who do I call if I have questions or problems?**

If you have questions about the study right now, please ask them. If you have questions about the study later on, please call Dr. Vrbsky at (205)-348-6363. If you have questions about your rights as a person taking part in a research study, make suggestions or file complaints and concerns, you may call Ms. Tanta Myles, the Research Compliance Officer of the University at (205)-348-8461 or toll-free at 1-877-820-3066. You may also ask questions, make suggestions, or file complaints and concerns through the IRB Outreach Website at http://osp.ua.edu/site/PRCO_Welcome.html. You may email us at participantoutreach@bama.ua.edu.

I have read this consent form. The study has been explained to me. I understand what I will be asked to do. I freely agree to take part in it. I will receive a copy of this consent form to keep.

_____ _____

Signature of Research Participant Date

_____ _____

Investigator Date

UNIVERSITY OF ALABAMA IRB
CONSENT FORM APPROVED: 2-12-14
EXPIRATION DATE: 2-11-15

# Appendix C
# IRB PROTOCOL

**Protocol for "Study on Cloud Deployment: General Purpose vs. Vertical Architectures"**

**Procedures:**
The purpose of this study is to determine how the deployment (installation) systems for an open source general purpose cloud architecture named Eucalyptus differ from that of our own architecture, THUNDER. Additionally, we would like to understand how THUNDER may be improved in order to be as easy to install for people of all technical backgrounds.

This study will be conducted in CS491/CS591 (Special Topics: Cloud Computing) at the University of Alabama. This course has an enrollment of approximately 31 subjects who will be solicited for participation.

As a part of their regularly assigned coursework, the student will complete two in-class exercises. The students will deploy an open source general purpose cloud architecture called Eucalyptus, and asked to record the time and difficulty of the task as well as any confusion with the documentation.

In the next class, the subjects will be asked to deploy a new cloud architecture called THUNDER. The subjects will be asked to record the time and difficulty of the task as well as any confusion with the documentation.

At the conclusion of both exercises, the researchers will explain the purpose of the study, using the attached recruitment script and ask for the students' consent to analyze the data submitted during the two exercises. If they consent, they will also be asked to complete a short survey to provide basic demographic information including: work experience, major, and class.

Because the bulk of the data will come from a required course assignment, the only extra time required for participation in the study will be the time to complete the consent form and the demographics questionnaire.

The students will receive no incentive for consenting to allow their data to be analyzed. They will receive credit for completing the in-class exercises that are part of the normal course work.

**Informed Consent:**
Informed consent will be obtained by reading the attached recruitment script at the conclusion of the two exercises. The students will be allowed to ask any questions they may have. They will then be given the attached consent form to complete. The instructor of the course will not know whether students consent or not.

**Risks and Benefits:**
This study contains no known risks. This study will produce information that will benefit future computer science students. The researchers will gain insight into how to better teach software testing in the computer science curriculum.

**De-identification, Storage and Disposal of Study Data**
After the paper forms are collected during the study, the information will be recorded into a spreadsheet or word processing document. At this point, each subject will be given an identifier and the names will not be retained. Any analysis and reporting done will be done either in the aggregate or anonymously so that no subject can be identified. The data will be stored on a password protected computer in the possession of one or more of the investigators. Any paper forms will be kept locked in Dr. Vrbsky's office until such time as the data has been encoded electronically. At this point, the sheets will be shredded.

# Appendix D
# IRB RECRUITMENT SCRIPT

Subject Recruitment Script

My name is Dr. Susan Vrbsky. On behalf of myself and my PhD student, Gabriel Loewen, I am here to talk to you about a study that I am conducting. We are interested in understanding the level of complexity for deploying the Eucalyptus cloud architecture in relation to the level of complexity for deploying our cloud architecture, THUNDER. You have just participated in two classroom exercises related to cloud deployment. During the first one, you were asked to deploy Eucalyptus. During the second exercise, you were asked to deploy THUNDER.

Now that you have completed the exercises, I am here to ask you if you will consent to participation in a study. This study will require no extra effort on your part. I am only asking you for permission to analyze the data that you provided during the two exercises. This analysis will be done anonymously and your name will not be associated with the data or with the results. Our main interest is understanding whether or not the deployment process of Eucalyptus was more or less difficult than the deployment process of THUNDER.

Any information gathered from this study will not be used for grading purposes. Your grade on these two in-class exercises will be based solely on whether you followed instructions and completed the assignment.

Are there any questions?

# Appendix E
# DEPLOYMENT QUESTIONNAIRE

**Experience Questionnaire**

Before taking this class, what was your previous experience with cloud deployment? (Check all that apply.)
__ I have never deployed a cloud.
__ I have deployed a cloud in my spare time.
__ I have deployed a cloud as part of a class requirement.
__ I have deployed a cloud as part of a job requirement.

**Please explain your answer**. List any difficulties that you experienced while deploying Eucalyptus.

_____
_____

**Please explain your answer**. List any difficulties that you experienced while deploying THUNDER.

_____
_____

**Education and Training**
Please provide your educational background (i.e. list degrees and Majors – B.S. in Chemistry; M.S in Chemistry, etc…)

_____
_____

How many years have you worked/took courses in the above field (or related fields)?

_____

Please describe any other significant work experience in fields other than your educational background.

_____
_____

**Did you successfully deploy Eucalyptus?** ___ yes. ___ no.
(If you answer no, please skip the next section)

**Eucalyptus Deployment**
Please rate your experience in deploying Eucalyptus. **1 – Lowest; 5 – Highest.**
(Please include any relevant comments at the end)

- Amount of time spent 1 2 3 4 5
- Perceived difficulty of deployment 1 2 3 4 5
- Clarity of instructions 1 2 3 4 5

Specifically, which instructions were unclear (if any):

What could be done (if anything) to simplify the deployment process?

**Did you successfully deploy THUNDER?** ___ yes. ___ no.
(If you answer no, please skip the next section)

**THUNDER Deployment**
Please rate your experience in deploying Eucalyptus. **1 – Lowest; 5 – Highest.**
(Please include any relevant comments at the end)

- Amount of time spent 1 2 3 4 5
- Perceived difficulty of deployment 1 2 3 4 5
- Clarity of instructions 1 2 3 4 5

Specifically, which instructions were unclear (if any):

What could be done (if anything) to simplify the deployment process?

**Eucalyptus comments:**

**THUNDER comments:**

# Appendix F
# IRB APPROVAL

February 12, 2014

Office for Research

Institutional Review Board for the Protection of Human Subjects

Gabriel Loewen, MS
Department of Computer Science
College of Engineering
The University of Alabama

THE UNIVERSITY OF ALABAMA RESEARCH

Re: IRB # 14-OR-040 "Study on Cloud Deployment: General Purpose vs. Vertical Architecture"

Dear Mr. Loewen:

The University of Alabama Institutional Review Board has granted approval for your proposed research

Your protocol has been given expedited approval according to 45 CFR part 46. Approval has been given under expedited review category 7 as outlined below:

*(7) Research on individual or group characteristics or behavior (including, but not limited to, research on perception, cognition, motivation, identity, language, communication, cultural beliefs or practices, and social behavior) or research employing survey, interview, oral history, focus group, program evaluation, human factors evaluation, or quality assurance methodologies.*

Your application will expire on February 11, 2015. If your research will continue beyond this date, complete the relevant portions of the IRB Renewal Application. If you wish to modify the application, complete the Modification of an Approved Protocol Form. Changes in this study cannot be initiated without IRB approval, except when necessary to eliminate apparent immediate hazards to participants. When the study closes, complete the appropriate portions of the IRB Study Closure Form.

Please use reproductions of the IRB approved stamped informed consent form to obtain consent from your participants.

Should you need to submit any further correspondence regarding this proposal, please include the above application number.

Good luck with your research.

Sincerely,

Signature Redacted

358 Rose Administration Building
Box 870127
Tuscaloosa, Alabama 35487-0127
(205) 348-8461
FAX (205) 348-7189
TOLL FREE (877) 820-3066

# Appendix G
# MAIN THUNDER MODULE

```
#!/usr/bin/env python3

'''
thunder.py
-----------------
Main THUNDER service class
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Imports
import auth, threading, socket, socketserver, sys, platform, struct
from interface import interface
from dictionary import *
from websocket import *
from mysql_support import mysql
from time import sleep
from config import *
from uuid import uuid1, getnode
import events as builtinEvents
from networking import *

# ensure that the address used by the service can be reused if it crashes.
socketserver.TCPServer.allow_reuse_address = True

# parse thunder config file
parseConfig('thunder.conf')

# default port of the server
SERVER_PORT = constants.get('server.port')

class ThunderRPC(threading.Thread):
    # local imports
    import rpc

    '''
    constructor ---
        Expected action:
           Sets up the object with role and group memberships
           for the local node.  Also starts the threads for
           both RPC request handlers and the multicast server
           for fault tolerance.

        Expected positional arguments:
           role - String representing the role of this node.
                  Must be either "PUBLISHER" or "SUBSCRIBER"
           group - Name of group to join as a member.  Groups
                   represent logical clusters.

        Expected return value:
           None
    '''
    def __init__(self, role = constants.get('default.role'),                  \
                 group = constants.get('default.group')):

        # invoke the constructor of the threading superclass
        super(ThunderRPC, self).__init__()
```

```
59
60          # check the role of the service to determine the proper configuration
61          if (role == 'PUBLISHER'):
62              iface = constants.get('server.interface')
63              port = SERVER_PORT
64          elif (role == 'SUBSCRIBER'):
65              iface = constants.get('default.interface')
66              port = constants.get('default.port')
67          else:
68              print('Invalid role identifer.  Must be either ' +                  \
69                    '"PUBLISHER" or "SUBSCRIBER"')
70              sys.exit(-1)
71
72          # set the interface variable, which is the interface that we want to
73          # bind the service to
74          self._interface = iface
75
76          # set the role ('PUBLISHER' | 'SUBSCRIBER')
77          self._role = role
78
79          # set the default nonce value.  this is autogenerated, so the initial
80          # value does not matter.
81          self._nonce = 'DEADBEEF'
82
83          iface_tool = interface()
84          # get the IP address of the system
85          self._IP = iface_tool.getAddr(iface)
86
87          # create a TCP server, and bind it to the address of the desired
88          # interface
89          self._server = self.rpc.ThunderRPCServer((self._IP, port),              \
90                                                   self.rpc.RequestHandler)
91          self._server._ThunderRPCInstance = self
92          self._server.daemon = True
93
94          # workaround for getting the port number for auto-assigned ports
95          # (default behavior for clients)
96          self._port = self._server.server_address[1]
97
98          # create a dictionary mapping an event name to a function
99          self._events = Dictionary()
100
101         # create a dictionary mapping group name to an array of tuples (IP,Port)
102         # these tuples represent the clients that are connecting to this service
103         # if the role of this service is SUBSCRIBER then this dictionary should
104         # always be empty.
105         self._clients = Dictionary()
106
107         print('Starting ThunderRPC Service (role = %s)' % role)
108         print('---------------------------------------------------')
109         print('Binding to IP address - %s:%s' % (self._IP, self._port))
110
111         # register some builtin events for metadata aggregation
112         self.registerEvent('UTILIZATION', builtinEvents.utilization)
113         self.registerEvent('SYSINFO', builtinEvents.sysInfo)
114
115         # associate with a group
116         self._group = group
117
118         # give this node an arbitrary name
119         self._name = getMAC(getnode())
120
121         # set the default publisher to None
122         self._publisher = None
123
124         if (role == 'PUBLISHER'):
125             # startup a multicasting thread to respond to multicast messages
126             self.mcastThread = threading.Thread(target = self.multicastThread)
```

```
127             self.mcastThread.start()
128             if (self.clients.contains(self.group)):
129                c = self.clients.get(self.group)
130                c.append((self.addr[0], int(self.addr[1])))
131             else:
132                self.clients.append((self.group, [(self.addr[0],              \
133                                     int(self.addr[1]))]))
134                myConnector = mysql(self.addr[0], 3306)
135                myConnector.connect()
136                myConnector.insertNode(self.name, self.addr[0] + ':' +        \
137                                       str(self.addr[1]), self.group)
138                myConnector.disconnect()
139             # recreate preseeding files from template
140             generatePreseed(self._IP)
141
142         # start the thread
143         self.start()
144
145         return
146
147     '''
148     Property mutator/accessors
149     '''
150     # accessors and mutators
151     @property
152     def clients(self):
153         return self._clients
154
155     @property
156     def events(self):
157         return self._events
158
159     @property
160     def interface(self):
161         return self._interface
162
163     @property
164     def role(self):
165         return self._role
166
167     @property
168     def name(self):
169         return self._name
170
171     @property
172     def addr(self):
173         return (self._IP, self._port)
174
175     @property
176     def group(self):
177         return self._group
178
179     @group.setter
180     def group(self, value):
181         self._group = value
182
183     @property
184     def publisher(self):
185         return self._publisher
186
187     @publisher.setter
188     def publisher(self, value):
189         self._publisher = value
190
191     @property
192     def publisherSubnet(self):
193         return self._publisherSubnet
194
```

```
195     '''
196     getClusterList() ---
197         Expected action:
198            Retrieves the list of clusters/groups that
199            have active members.
200
201         Expected positional arguments:
202            None
203
204         Expected return value:
205            An array of cluster/group names
206     '''
207     def getClusterList(self):
208         ret = []
209         self.cleanupClients()
210         for cluster in self.clients.collection():
211            ret += [cluster]
212         return ret
213
214     '''
215     getClientList(group) ---
216         Expected action:
217            Retrieves the list of clients that are member of
218            a particular cluster/group
219
220         Expected positional arguments:
221            group - The name of a group
222
223         Expected return value:
224            An array of client IP/port
225     '''
226     def getClientList(self, group):
227         listString = ''
228         clist = []
229         self.cleanupClients()
230         for client in self.clients.get(group):
231            clist += [client[0] + ':' + str(client[1])];
232         return clist
233
234     '''
235     cleanupClients() ---
236         Expected action:
237            Checks each group for inactive clients and removes them
238
239         Expected positional arguments:
240            None
241
242         Expected return value:
243            None
244     '''
245     def cleanupClients(self):
246         # For each group in the list of clients, find the unavailable address
247         # and remove it.
248         emptyGroups = []
249         for grp in self.clients.collection():
250             addresses = self.clients.get(grp)
251             for addr in addresses:
252                 try:
253                     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
254                     s.settimeout(5)
255                     s.connect(addr)
256                     s.close()
257                 except:
258                     print('Removing inactive client "%s:%d"' % (addr[0],addr[1]))
259                     addresses.remove(addr)
260                     myConnector = mysql(self.addr[0], 3306)
261                     myConnector.connect()
262                     myConnector.deleteNodeByIP(addr[0]+':'+str(addr[1]))
```

```
                    myConnector.disconnect()
                    if (len(self.clients.get(grp)) == 0):
                        emptyGroups += [grp]
        # Any groups with no clients should be removed
        for grp in emptyGroups:
            print('Removing empty group "%s"' % grp)
            del self.clients.collection()[grp]
    '''
    getMulticastingSender() ---
        Expected action:
           Sets up a socket for outbound communication over multicast

        Expected positional arguments:
           None

        Expected return value:
           The socket object
    '''
    def getMulticastingSender(self):
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
        sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 2)
        return sock

    '''
    getMulticastingReceiver() ---
        Expected action:
           Sets up a socket for inbound communication over multicast

        Expected positional arguments:
           None

        Expected return value:
           The socket object
    '''
    # get a bound socket for receiving multicast messages
    def getMulticastingReceiver(self):
        MCAST_GRP = constants.get('default.mcastgrp')
        MCAST_PORT = constants.get('default.mcastport')
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
        try:
            sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        except AttributeError:
            print('Attribute Error')
            pass

        # bind to all adapters
        sock.bind(('0.0.0.0', MCAST_PORT))
        mreq = struct.pack('4sl', socket.inet_aton(MCAST_GRP), socket.INADDR_ANY)
        sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
        return sock

    '''
    multicastThread() ---
        Expected action:
           Wait for requests over multicast and respond immediately.
           This is used to autoamtically locate publishers on the network.

        Expected positional arguments:
           None

        Expected return value:
           None
    '''
    def multicastThread(self):
        MCAST_GRP = constants.get('default.mcastgrp')
        MCAST_PORT = constants.get('default.mcastport')
        receiver = self.getMulticastingReceiver()
        while 1:
```

```
331             try:
332                 data, addr = receiver.recvfrom(1024)
333                 msg = parseMessage(data)
334                 if ('cmd' in msg and msg['cmd'] == 'ROLE'):
335                     # pack the role up with the client IP
336                     response = createMessage(role=self.role, ip=self.addr[0])
337                     receiver.sendto(response, addr)
338             except:
339                 print('Error during multicast receive')
340                 pass
341 
342     '''
343     findPublisher() ---
344         Expected action:
345             Send a request over multicast and wait for a response.
346             Once the publisher is found on the network then we
347             will register with that publisher as a subscriber.
348 
349         Expected positional arguments:
350             None
351 
352         Expected return value:
353             None
354     '''
355     def findPublisher(self):
356         MCAST_GRP = constants.get('default.mcastgrp')
357         MCAST_PORT = constants.get('default.mcastport')
358         sender = self.getMulticastingSender()
359         sender.settimeout(constants.get('multicast.timeout'))
360         found = False
361         address = None
362         while(not found):
363             sender.sendto(createMessage(cmd='ROLE'), (MCAST_GRP, MCAST_PORT))
364             try:
365                 data, addr = sender.recvfrom(1024)
366                 response = parseMessage(data)
367                 if ('role' in response and response['role'] == 'PUBLISHER'):
368                     address = (response['ip'],SERVER_PORT)
369                     found = True
370             except KeyboardInterrupt:
371                 raise
372             except:
373                 print('Publisher not found, retrying.')
374                 continue
375         self.registerClient(address, self.group)
376 
377     '''
378     publishToHost() ---
379         Expected action:
380             Send a message to a particalur host (IP, port).
381             This can be a blocking or non-blocking operation
382             depending on whether the caller of this function
383             expects that the response will need to timeout
384             or not.
385 
386         Expected positional arguments:
387             host - The (IP, port) tuple of the receiving node
388             data - The message to send to the node
389             timeout - Sets blocking (timeout=False) or nonblocking
390                       communication.
391 
392         Expected return value:
393             The response from the receiving node in the form:
394             "<hostIP>:<message>"
395     '''
396     def publishToHost(self, host, data, timeout = True):
397         retryLimit = constants.get('default.maxConnectionRetries')
398         count = 0;
```

```
399         while (count < retryLimit):
400             try:
401                 # open a socket connection
402                 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
403                 if (timeout):
404                     s.settimeout(constants.get('default.messageTimeout'))
405                 else:
406                     s.settimeout(None)
407                 s.connect(host)
408                 # encode the data before sending
409                 s.send(data)
410                 # wait for a response from the host
411                 response = s.recv(1024).decode().strip()
412                 s.close()
413                 # return a colon delimited string containing the IP address of
414                 # the host and its response
415                 if (response == ''):
416                    ret = {}
417                 else:
418                    ret = parseMessage(response)
419                 ret['_HOST_'] = host
420                 return ret
421             except:
422                 print('Host', host[0] + ':' + str(host[1]),                    \
423                       'didn\'t respond.  Trying again.')
424             count += 1
425         raise Exception("Timeout")
426 
427     '''
428     publishToGroup() ---
429         Expected action:
430            Send a message to a particalur group/cluster.
431            This can be a blocking or non-blocking operation
432            depending on whether the caller of this function
433            expects that the response will need to timeout
434            or not.
435 
436         Expected positional arguments:
437            group - The name of the group to send the message to
438            data - The message to send to the group
439            timeout - Sets blocking (timeout=False) or nonblocking
440                      communication.
441 
442         Expected return value:
443            The responses from the nodes in the group separated
444            by semicolons or None if nothing is returned.
445     '''
446     def publishToGroup(self, group, data, timeout = False):
447         ret = []
448         if (self.clients.contains(group)):
449             addresses = self.clients.get(group)
450             for addr in addresses:
451                 try:
452                     val = self.publishToHost(addr, data, timeout)
453                     if (val != None):
454                          ret += [val]
455                 except:
456                    print("Timeout in communicating with", addr)
457         return ret
458 
459     '''
460     registerEvent() ---
461         Expected action:
462            Register an event by adding a mapping between logical name
463            and function to a global dictionary of events.
464 
465         Expected positional arguments:
466            name - The name of the event
```

```
467             event - The function being mapped to
468
469         Expected return value:
470             None
471     '''
472     def registerEvent(self, name, event):
473         print('Registering event - "%s (%s)"' % (name, event.__name__))
474         self.events.append((name, event))
475
476     '''
477     registerClient() ---
478         Expected action:
479             Registers a node (IP, port) with a particular publisher
480             and assigns group membership of the node.
481
482         Expected positional arguments:
483             host - The (IP, port) tuple of the subscriber
484             group - The group that the subscriber is joining
485
486         Expected return value:
487             None
488     '''
489     def registerClient(self, host, group):
490         # send an authorization request to the server.  The server should
491         # return a nonce value.
492         ret = self.publishToHost(host, createMessage(cmd='AUTH'))
493         nonce = ret['nonce']
494
495         # encrypt the nonce with pre-shared key and send it back to the host.
496         decVal = auth.encrypt(nonce).decode('UTF8')
497
498         message = createMessage(cmd='SUBSCRIBE',
499                                 ip=self.addr[0],
500                                 port=self.addr[1],
501                                 group=self.group,
502                                 nonce=decVal)
503
504         # TODO: What to do with ret?
505         ret = self.publishToHost(host, message)
506
507         # save the IP of the publisher.
508         self._publisher = host
509
510         # start a heartbeat for fault tolerance
511         self._heartBeatThread = threading.Thread(target = self.heartBeat)
512         self._heartBeatThread.start()
513
514         # Insert the ndoe into the database
515         myConnector = mysql(str(host[0]), 3306)
516         myConnector.connect()
517         myConnector.insertNode(self.name, self.addr[0]+':'+str(self.addr[1]), self.group)
518         myConnector.disconnect()
519
520     '''
521     heartbeat() ---
522         Expected action:
523             Thread that periodically will send out a heartbeat
524             request to the publisher to make sure that it is
525             still alive.  If the publisher does not respond then
526             the node will attempt to reconnect to another publisher
527             on the network.
528
529         Expected positional arguments:
530             None
531
532         Expected return value:
533             None
534     '''
```

```
535     def heartBeat(self):
536         # if there is no publisher to ping then something bad happened.
537         if (self._publisher == None):
538             self.findPublisher()
539             return
540 
541         message = createMessage(cmd='HEARTBEAT', ip=self.addr[0], port=self.addr[1])
542         alive = True
543         while (alive):
544             sleep(constants.get('heartbeat.interval'))
545 
546             try:
547                 ret = self.publishToHost(self._publisher, message)
548             except:
549                 continue
550             if (ret == None):
551                 alive = False
552             else:
553                 if (ret['result'] == 'SUBSCRIBE'):
554                     alive = False
555         self.findPublisher()
556         return
557 
558     '''
559     run() --- inherited from threading.Thread
560         Expected action:
561             Continuously handle requests coming in over the
562             socket server.
563 
564         Expected positional arguments:
565             None
566 
567         Expected return value:
568             None
569     '''
570     def run(self):
571         self.running = True
572         self._server.serve_forever()
573         self._server.shutdown()
574         return
```

# Appendix H
# THUNDER RPC SERVER COMPONENT

```
#!/usr/bin/env python3

'''
rpc.py
-----------------
Remote procedure call handler for THUNDER
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Imports
import auth, socket, socketserver, threading, libvirt, load_balancer
import subprocess, shutil, fileinput, threading, os, urllib.request
import threadpool
from websocket import *
from networking import *
from mysql_support import *
from time import sleep
from enum import Enum

locks = [0] * 5

class LBMode(Enum):
    RAIN=0
    ROUNDROBIN=1
    CONSOLIDATE=2
    RANDOM=3

'''
ThunderRPCServer(ThreadPoolMixIn, TCPServer) ---
Dummy class used to ensure per-request threading
'''
class ThunderRPCServer(threadpool.ThreadPoolMixIn, socketserver.TCPServer):
    pass

'''
RequestHandler(socketserver.BaseRequestHandler) ---
Each request should be given an independent thread, each handled by
the RequestHandler class
'''
class RequestHandler(socketserver.BaseRequestHandler):

    '''
    setup() ---
        Expected action:
           Sets up the socket for blocking communication on default
           also enables background process (daemon) mode for threads.

        Expected positional arguments:
           None

        Expected return value:
           None
    '''
    def setup(self):
        self.daemon_threads = True
        self.request.setsockopt(socket.IPPROTO_TCP,socket.TCP_NODELAY, True)
```

```
        self.request.setblocking(1)
        try:
            fp = open('./lb.conf', 'r')
            self.lbMode = LBMode[fp.read()]
            fp.close()
        except:
            self.lbMode = LBMode.RAIN
        return

    '''
    handle() ---
        Expected action:
            Reads data from the socket server, determines whether it
            is a raw message or if it is wrapped in a HYBI websocket
            frame.  Calls the appropriate handler to process the data.

        Expected positional arguments:
            Nonde

        Expected return value:
            None
    '''
    def handle(self):
        self.container = self.server._ThunderRPCInstance
        self.data = self.request.recv(2048)
        websock = websocket(self.request)
        decodedData = ''
        ws = False

        # Check to see if the data is coming over a websocket connection
        # (cloud interface)
        if (websock.isHandshakePending(self.data)):
            handshake = websock.handshake(self.data.decode())
            self.request.sendall(handshake)
            d = self.request.recv(2)
            decodedData = websock.decode(d).strip()
            ws = True
        # Else, it could be coming from a peer (cloud server)
        else:
            decodedData = self.data.decode('UTF8').strip()

        # if there is data waiting to be processed then process it!
        if (decodedData != ''):
            decodedData = parseMessage(decodedData)
            if (ws):
                self.processWebsocketRequest(self.request, decodedData, websock)
            else:
                self.processTraditionalRequest(self.request, decodedData)
        return

    '''
    processWebsocketRequest(data, websock) ---
        Expected action:
            Process incoming messages from the web interface
            relay these messages to other nodes if need
            be and return a response to the caller

        Expected positional arguments:
            data - the raw message after websocket headers have been removed
            websock - the websock object used to re-encode responses

        Expected return value:
            None
    '''
    def processWebsocketRequest(self, request, data, websock):
        clients = self.container.clients
        # check if the client is requesting data from a group
        if (data['cmd'] == 'EXECGROUP'):
```

```
127             group = data['group']
128             message = createMessage(cmd=data['remote_cmd'], args=data['remote_args'])
129             res = self.container.publishToGroup(group, message)
130             request.sendall(websock.encode(Opcode.text, res))
131 
132         # check if the client is requesting data from a node
133         elif (data['cmd'] == 'EXECNODE'):
134             node = (data['ip'], int(data['port']))
135             message = createMessage(cmd=data['remote_cmd'], args=data['remote_args'])
136             res = self.container.publishToHost(node, message)
137             if ('result' in res):
138                 result = createMessage(result=res['result'])
139             else:
140                 print("Result not in res? ", res)
141                 result = str(res).encode('UTF8')
142             request.sendall(websock.encode(Opcode.text, result))
143 
144         # check if the client is requesting a list of clusters available
145         elif (data['cmd'] == 'GROUPNAMES'):
146             result = websock.encode(Opcode.text,                                    \
147                      createMessage(clusters=self.container.getClusterList()))
148             request.sendall(result)
149 
150         # check if the client is requesting a list of nodes in a particular
151         # cluster
152         elif (data['cmd'] == 'NODESINGROUP'):
153             clients = self.container.getClientList(data['cluster'])
154             msg = createMessage(clients=clients)
155             request.sendall(websock.encode(Opcode.text, msg))
156 
157         # check if the client is requesting the server to poll for mac
158         # addresses, used for deployment
159         elif (data['cmd'] == 'POLLMACS'):
160             p = subprocess.Popen('./capturemacs.sh', stdout=subprocess.PIPE,    \
161                                  stderr=subprocess.PIPE)
162             out, err = p.communicate()
163             macs = out.decode().rstrip().split('\n')
164             result = []
165             for mac in macs:
166                 if (not mac.startswith(constants.get('default.vmMACPrefix'))):
167                     result+=[mac.strip()]
168             msg = createMessage(macs=result)
169             request.sendall(websock.encode(Opcode.text, msg))
170 
171         elif (data['cmd'] == 'INSTANTIATE'):
172             global locks
173             self.container.cleanupClients()
174             nodes = clients.get('COMPUTE')
175             # Message style:
176             # INSTANTIATE <VM_NAME> <USER_NAME>
177             message = createMessage(cmd=data['cmd'], vm=data['vm'], user=data['user'])
178             utilization = createMessage(cmd='UTILIZATION')
179             error = createMessage(ip=None)
180 
181             myConnector = mysql(self.container.addr[0], 3306)
182             myConnector.connect()
183             if (self.lbMode == LBMode.CONSOLIDATE):
184                 weights = [0] * 5
185             elif (self.lbMode == LBMode.RAIN):
186                 weights = myConnector.getWeights('balance')
187             vm = myConnector.getProfileData(data['vm'])
188             myConnector.disconnect()
189             load = [self.container.publishToHost(nodes[0], utilization)]
190             for node in nodes[1:]:
191                 tmp = self.container.publishToHost(node, utilization)
192                 load += [tmp]
193 
194             selected = None
```

```
            index = -1
            if (self.lbMode == LBMode.RAIN or self.lbMode == LBMode.CONSOLIDATE):
               while (1):
                  # [memTotal, memFree, 1min, 5min, 15min, maxVCore, activeVCore]
                  selected = load_balancer.rain_select(load, weights, vm)
                  if (selected == None):
                     # Couldn't find a node to instantiate the vm
                     request.sendall(websock.encode(Opcode.text, error))
                     return

                  for i in range(0, len(nodes), 1):
                     if (nodes[i][0] == selected[0]):
                        index = i

                  if (index == -1):
                     request.sendall(websock.encode(Opcode.text, error))
                     return

                  # If we don't have enough locks, double it
                  if (index > len(locks)):
                     locks += [0] * len(locks)

                  if (locks[index] == 0):
                     locks[index] = 1
                     break
                  else:
                     load = [self.container.publishToHost(nodes[0], utilization)]
                     for node in nodes[1:]:
                        load += [self.container.publishToHost(node, utilization)]

            elif (self.lbMode == LBMode.ROUNDROBIN):
               selected = load_balancer.rr_select(load, vm)
               if (selected == None):
                  # Couldn't find a node to instantiate the vm
                  request.sendall(websock.encode(Opcode.text, error))
                  return

               for i in range(0, len(nodes), 1):
                  if (nodes[i][0] == selected[0]):
                     index = i
                     break

            if (index == -1):
               request.sendall(websock.encode(Opcode.text, error))

            selectedNode = nodes[index]
            response = self.container.publishToHost(selectedNode, message, False)
            locks[index] = 0

            response = response['result']
            ip = ''
            if ('mac' in response and response['mac'] != ''):
               ip = getIPFromARP(response['mac'])
            myConnector.connect()
            myConnector.updateInstanceIP(response['domain'], ip)
            myConnector.disconnect()
            request.sendall(websock.encode(Opcode.text, createMessage(ip=ip)))

         elif (data['cmd'] == 'GETUSERINSTANCES'):
            username = data['user']
            myConnector = mysql(self.container.addr[0], 3306)
            myConnector.connect()
            instances = myConnector.getUserInstances(username)
            for instance in eval(instances):
               node = myConnector.getNodeByName(instance[2])
               nodeAddr = node[1].split(':')
               nodeAddr = (nodeAddr[0],int(nodeAddr[1]))
               message = createMessage(cmd='CHECKINSTANCE', domain=instance[0])
```

```
263                 response = self.container.publishToHost(nodeAddr, message)
264                 if (response['result'] == 'error' and instance[1] != '-1'):
265                     myConnector.deleteInstance(instance[0])
266             instances = myConnector.getUserInstances(username)
267             myConnector.disconnect()
268             message = createMessage(user=username, instances=instances)
269             request.sendall(websock.encode(Opcode.text, message))
270 
271         # the user has requested that an instance be destroyed.
272         # a message should be relayed to the node hosting the instance.
273         elif (data['cmd'] == 'DESTROYINSTANCE'):
274             username = data['user']
275             domain = data['domain']
276             myConnector = mysql(self.container.addr[0], 3306)
277             myConnector.connect()
278             instances = myConnector.getUserInstances(username)
279             result = 'error'
280             for instance in eval(instances):
281                 if (instance[0] == domain):
282                     node = myConnector.getNodeByName(instance[2])
283                     nodeAddr = node[1].split(':')
284                     nodeAddr = (nodeAddr[0],int(nodeAddr[1]))
285                     message = createMessage(cmd='DESTROY', domain=instance[0])
286                     self.container.publishToHost(nodeAddr, message)
287                     result = 'success'
288                     break
289             message = createMessage(result=result)
290             myConnector.disconnect()
291             request.sendall(websock.encode(Opcode.text, message))
292 
293         # retrieve the RAIN constants from the database and send them to the
294         # caller
295         elif (data['cmd'] == 'RAINCONSTANTS'):
296             myConnector = mysql(self.container.addr[0], 3306)
297             myConnector.connect()
298             weights = myConnector.getWeights('balance')
299             result = []
300             for weight in weights:
301                 result += [weight]
302             myConnector.disconnect()
303             message = createMessage(result=result)
304             request.sendall(websock.encode(Opcode.text, message))
305 
306         elif (data['cmd'] == 'GETLBMODE'):
307             try:
308                 fp = open('lb.conf', 'r')
309                 mode = fp.read()
310                 fp.close()
311                 message = createMessage(lbmode=mode);
312             except:
313                 fp = open('lb.conf', 'w')
314                 fp.write("RAIN")
315                 fp.close()
316                 message = createMessage(lbmode="RAIN");
317             request.sendall(websock.encode(Opcode.text, message))
318 
319         elif (data['cmd'] == 'CHANGELBMODE'):
320             mode = data['mode']
321             try:
322                 self.lbMode = LBMode[mode]
323                 fp = open('lb.conf', 'w')
324                 fp.write(mode)
325                 fp.close()
326             except:
327                 print(mode,"is not a valid load balance mode.")
328             message = createMessage(result=0)
329             request.sendall(websock.encode(Opcode.text, message))
330 
```

```
331         elif (data['cmd'] == 'UPDATERAIN'):
332             myConnector = mysql(self.container.addr[0], 3306)
333             myConnector.connect()
334             myConnector.updateWeights('balance', data['constants'])
335             myConnector.disconnect()
336             # TODO: Failutes?
337             request.sendall(websock.encode(Opcode.text, createMessage(result=0)))
338 
339         elif (data['cmd'] == 'IMAGELIST'):
340             myConnector = mysql(self.container.addr[0], 3306)
341             myConnector.connect()
342             images = myConnector.getImages()
343             myConnector.disconnect()
344             result = []
345             for image in images:
346                 result += [image[0]]
347             message = createMessage(result=result)
348             request.sendall(websock.encode(Opcode.text, message))
349 
350         elif (data['cmd'] == 'SAVEPROFILE'):
351             #name = data[1]
352             #title = data[2]
353             #desc = data[3]
354             #ram = data[4]
355             #vcpu = data[5]
356             #image = data[6]
357             #myConnector = mysql(self.container.addr[0], 3306)
358             #myConnector.connect()
359             #myConnector.insertProfile(name, title, desc, ram, vcpu, image)
360             #myConnector.disconnect()
361             print(data)
362 
363         elif (data['cmd'] == 'IMPORTIMAGE'):
364             url = data['url']
365             myConnector = mysql(self.container.addr[0], 3306)
366             myConnector.connect()
367             storageNodes = myConnector.getStorageNodes()
368             myConnector.disconnect()
369             # Figure out what to do with multiple storage nodes
370             nodeAddr = storageNodes[0][1].split(':')
371             nodeAddr = (nodeAddr[0],int(nodeAddr[1]))
372             message = createMessage(cmd='IMPORTIMAGE', url=url)
373             res = self.container.publishToHost(nodeAddr, message, False)
374 
375         elif (data['cmd'] == 'PROFILEINFO'):
376             myConnector = mysql(self.container.addr[0], 3306)
377             myConnector.connect()
378             profile = myConnector.getProfile(data['profile'])
379             myConnector.disconnect()
380             result = []
381             for datum in profile:
382                 result += [datum]
383             message = createMessage(result=result)
384             request.sendall(websock.encode(Opcode.text, message))
385 
386         elif (data['cmd'] == 'DEPLOY'):
387             role = data['role']
388             macs = data['macs']
389             for mac in macs:
390                 oldDHCPTime = getDHCPRenewTime(mac)
391                 tftpDir = constants.get('default.tftpdir')
392                 shutil.copyfile('pxetemplate.cfg', tftpDir +                    \
393                                 '/pxelinux.cfg/01-' + mac)
394                 fname = tftpDir + '/pxelinux.cfg/01-' + mac
395                 for line in fileinput.input(fname, inplace=True):
396                     if '<ROLE>' in line:
397                         print(line.replace('<ROLE>', role), end='')
398                     elif '<SERVER_IP>' in line:
```

```
399                     print(line.replace('<SERVER_IP>',                          \
400                                        self.container.addr[0]),                \
401                                        end='')
402                 else:
403                     print(line, end='')
404             t = threading.Thread(target = self.detectDHCPRenew,                \
405                                  args = (mac, oldDHCPTime, ))
406             t.start()
407         message = createMessage(result=1)
408         request.sendall(websock.encode(Opcode.text, message))
409
410     elif (data['cmd'] == "REBOOTNODE"):
411         p = subprocess.Popen(['sudo','reboot'], stdout=subprocess.PIPE,        \
412                              stderr=subprocess.PIPE)
413         out = p.communicate()
414
415     else:
416         print('DATA:',data)
417
418     request.close()
419     return
420
421 '''
422 detectDHCPRenew(mac, time) ---
423     Expected action:
424        Read the DHCP log to determine if a particular
425        MAC address has been assigned a new IP address.
426        Once a new IP has been issued then it is safe
427        to remove any PXE configurations for this node.
428
429        Explanation: This is used for deployment.  We assume
430        That after the node boots and gets configured over
431        PXE then it is okay to discard the configuration
432        so that the node does not get redeployed on the next
433        boot
434
435     Expected positional arguments:
436        mac - The MAC address of the node in question
437        time - The previous time that the node received an IP address
438
439     Expected return value:
440        None
441 '''
442 def detectDHCPRenew(self, mac, time):
443     changed = False
444     while (not changed):
445        newTime = getDHCPRenewTime(mac)
446        if newTime != time:
447           changed = True
448           print('Detected DHCP renew of MAC:', mac)
449     sleep(120)
450     os.remove(constants.get('default.tftpdir') + '/pxelinux.cfg/01-' + mac)
451     print('Removed PXE configuration')
452
453 '''
454 processTraditionalRequest(data) ---
455     Expected action:
456        Process incoming messages from another physical node
457        relay these messages to other nodes if need
458        be and return a response to the caller
459
460     Expected positional arguments:
461        data - the raw message
462
463     Expected return value:
464        None
465 '''
466 def processTraditionalRequest(self, request, data):
```

```
        client = self.client_address
        events = self.container.events
        clients = self.container.clients

        # check if the request is an event call
        if (events.contains(data['cmd'])):
            func = events.get(data['cmd'])

            response = func(data)
            # send the result to the caller
            request.sendall(createMessage(result=response))

        # check if the request is a query for the service role
        # (PUBLISHER | SUBSCRIBER)
        elif (data['cmd'] == 'ROLE'):
            message = createMessage(role=self.container.role)
            request.sendall(message.encode('UTF8'))

        # check if the caller is requesting a nonce for authorization
        elif (data['cmd'] == 'AUTH'):
            nonce = auth.generateNonce()
            self.container._nonce = nonce
            request.sendall(createMessage(nonce=nonce))

        # check if the caller is requesting authorization for subscription
        elif (data['cmd'] == 'SUBSCRIBE'):
            r = data['nonce'].encode('UTF8')
            m = auth.decrypt(r).decode('UTF8')
            if (m == self.container._nonce):
                # we can consider this subscriber to be authentic
                if (len(data) == 5): # should be 5 values
                    # data[3] is the group name
                    if (clients.contains(data['group'])):
                        c = clients.get(data['group'])
                        c.append((data['ip'], data['port']))
                    else:
                        c = [(data['ip'], int(data['port']))]
                        clients.append((data['group'], c))
            request.sendall(createMessage(result=0))

        # check if the caller is sending a heartbeat
        elif (data['cmd'] == 'HEARTBEAT'):
            # check if the client is still registered
            response = data['cmd']
            if (len(clients.collection()) == 0):
               response = 'SUBSCRIBE'
            else:
               found = False
               for group in clients.collection():
                  for ip in clients.get(group):
                     if (ip[0] == data['ip'] and ip[1] == data['port']):
                        found = True
               if (not found):
                  response = 'SUBSCRIBE'
            message = createMessage(result=response)
            request.sendall(message)

        elif (data['cmd'] == 'UPDATERAIN'):
            myConnector = mysql(self.container.addr[0], 3306)
            myConnector.connect()
            myConnector.updateWeights('balance', data['constants'])
            myConnector.disconnect()
            message = createMessage(result=0)
            request.sendall(message)

        elif (data['cmd'] == 'CHANGELBMODE'):
            mode = data['mode']
            try:
```

```
535                 self.lbMode = LBMode[mode]
536                 fp = open('lb.conf', 'w')
537                 fp.write(mode)
538                 fp.close()
539             except:
540                 print(mode,"is not a valid load balance mode.")
541             message = createMessage(result=0)
542             request.sendall(message)
543 
544         # check if an instance is running
545         elif (data['cmd'] == 'CHECKINSTANCE'):
546             virtcon = libvirt.openReadOnly(None)
547             error = createMessage(result='error')
548             success = createMessage(result='success')
549             if (virtcon == None):
550                 request.sendall(error)
551             try:
552                 virtcon.lookupByName(data['domain'])
553                 request.sendall(success)
554             except:
555                 request.sendall(error)
556             virtcon.close()
557 
558         else:
559             print('DATA:',data)
560 
561         request.close()
562         return
```

# Appendix I
# THUNDER METRICS AGGREGATION EVENTS

```
#!/usr/bin/env python3

'''
events.py
-----------------
Default events for metadata aggregation
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Imports
import os
import platform
import libvirt
from networking import createMessage, parseMessage

'''
utilization(*params) ---
    Expected action:
        Determine the utilization of RAM and CPU

    Expected positional arguments:
        None

    Expected return value:
        Colon delimited string of data in the following form:
        "<memtotal>:<memfree>:<load1>:<load5>:<load15>:<#Cores>:<activeCores>"
        Where loadX denotes the average CPU load over X minutes
'''
def utilization(*params):
   # Get memory utilization
   memInfo = open('/proc/meminfo','r')
   memtotal = memInfo.readline().split(':')[1].strip().split()[0]
   memfree = memInfo.readline().split(':')[1].strip().split()[0]
   memInfo.close()

   # Get CPU load averages
   loadAvg = open('/proc/loadavg', 'r')
   data = loadAvg.readline().split()
   loadAvg.close()
   oneMin = data[0]
   fiveMin = data[1]
   fifteenMin = data[2]

   nproc = open('/proc/cpuinfo', 'r')
   num_proc = 0
   for line in nproc:
      if (line[:9] == 'processor'):
         num_proc += 1

   num_proc *= constants.get('default.vcoresPerCore')

   activeVCores = 0

   conn = libvirt.open()
   for dom in conn.listAllDomains():
      activeVCores += int(dom.info()[3])
```

```
    conn.close()

    retVal = [memtotal,memfree,oneMin,fiveMin,                                  \
              fifteenMin,str(num_proc),str(activeVCores)]

    return retVal

'''
sysInfo(*params) ---
    Expected action:
        Determine the operating system, kernel, and architecture
        of the machine

    Expected positional arguments:
        None

    Expected return value:
        A string containing the information in the following form:
        "<osName>:<kernel>:<architecture>"
'''
def sysInfo(*params):
    fullOsName = ''
    for i in platform.dist():
       fullOsName += i + ' '
    fullOsName = fullOsName[:-1]
    shortOsName = os.uname()[0]
    kernel = os.uname()[2]
    arch = os.uname()[4]

    retVal = [shortOsName, fullOsName, kernel, arch]
    return retVal
```

# Appendix J
# AUTHENTICATION MODULE

```
#!/usr/bin/env python3

'''
auth.py
-----------------
Shared key authorization module for THUNDER
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''
import base64
from Crypto import Random
from Crypto.Random import random
from Crypto.Cipher import AES

# Shared key!  This should probably be more secure...
k = b'YM{<GC\xdaQ\x80\xe8\xd8\xcf\xf1\xf7\xdaM'

# Blocksize for crypto and padding to make sure the length is as expected
BS = 16
pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
unpad = lambda s : s[0:-ord(chr(s[-1]))]

# generateNonce - Generates a random 32-bit value
def generateNonce():
   return str(random.getrandbits(32))

# encrypt - Encrypts the raw data passed into the function using AES
def encrypt(raw):
   raw = pad(raw)
   iv = Random.new().read( AES.block_size )
   cipher = AES.new( k, AES.MODE_CBC, iv )
   return base64.b64encode( iv + cipher.encrypt( raw ) )

# decrypt - Decrypts the encrypted data using the shared key
def decrypt(enc):
   enc = base64.b64decode(enc)
   iv = enc[:16]
   cipher = AES.new(k, AES.MODE_CBC, iv)
   return unpad(cipher.decrypt(enc[16:]))
```

# Appendix K
# MYSQL SUPPORT MODULE

```
#!/usr/bin/env python3

'''
mysql_support.py
-----------------
Support some basic mysql operations for THUNDER
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Imports
import pymysql
from time import sleep

# Encapsulate functions for mySQL access using embedded SQL
class mysql():
   def __init__(self, server, port):
      self.server = server
      self.port = port

   '''
   connect(self) ---
       Expected action:
           Connect to the mySQL server

       Expected positional arguments:
           None

       Expected return value:
           None
   '''
   def connect(self):
      while (1):
          try:
              self.conn = pymysql.connect(user='thunder', passwd='thunder', db='thunder', \
                                          host=self.server, port=self.port)
              break
          except:
              print('Couldn\'t connect to the database. Trying again!')
              sleep(1)
              continue
   '''
   disconnect(self) ---
       Expected action:
           Commit and disconnect from the mySQL server

       Expected positional arguments:
           None

       Expected return value:
           None
   '''
   def disconnect(self):
      self.conn.commit()
      self.conn.close()

   '''
```

```
59    Begin mySQL accessors and mutators.
60    Context of each function should be self explanatory.
61    '''
62
63    def getNASAddress(self, name):
64       cur = self.conn.cursor()
65       cur.execute("SELECT address FROM nodes WHERE type='STORAGE' " +          \
66                   "AND name='" + name+"';")
67       res = cur.fetchone()
68       cur.close()
69       return res
70
71    def getImageData(self, name):
72       cur = self.conn.cursor(pymysql.cursors.DictCursor)
73       cur.execute("SELECT * FROM images WHERE name='"+name+"';")
74       res = cur.fetchone()
75       cur.close()
76       return res
77
78    def getProfileData(self, name):
79       cur = self.conn.cursor(pymysql.cursors.DictCursor)
80       cur.execute("SELECT * FROM profiles WHERE name='"+name+"';")
81       res = cur.fetchone()
82       cur.close()
83       return res
84
85    def getComputeNodeAddresses(self):
86       cur = self.conn.cursor()
87       cur.execute("SELECT address FROM node WHERE type='COMPUTE';")
88       res = cur.fetchall()
89       cur.close()
90       return res
91
92    def getUserInstances(self, user):
93       cur = self.conn.cursor()
94       cur.execute("SELECT domain, ip, node, profile FROM instances " +         \
95                   "WHERE owner='"+user+"';")
96
97       res = cur.fetchall()
98       res = [list(row) for row in res]
99       cur.close()
100      return str(res)
101
102   def deleteInstance(self, domain):
103      cur = self.conn.cursor()
104      cur.execute("DELETE FROM instances WHERE domain='"+domain+"';")
105      cur.close()
106
107   def insertInstance(self, domain, ip, node, owner, profile):
108      cur = self.conn.cursor()
109      cur.execute("INSERT INTO instances VALUES ('"+domain+"','"+ip+           \
110                  "','" +node + "','" + owner + "','" + profile +              \
111                  "') on duplicate key UPDATE ip='" + ip + "', node='" +       \
112                  node + "', profile='" + profile + "';")
113      cur.close()
114
115   def updateInstanceIP(self, domain, ip):
116      cur = self.conn.cursor()
117      cur.execute("UPDATE instances SET ip='" + ip + "' WHERE domain='" +      \
118                  domain +"';")
119      cur.close()
120
121   def insertNode(self, name, address, tpe):
122      cur = self.conn.cursor()
123      res = cur.execute("INSERT INTO nodes VALUES ('" + name + "','" +         \
124                        address + "','" + tpe + "') on duplicate key " +       \
125                        "UPDATE address='" + address + "';")
126      cur.close()
```

```
127
128     def deleteNodeByName(self, name):
129         cur = self.conn.cursor()
130         cur.execute("DELETE FROM nodes WHERE name='" + name + "';")
131         cur.close()
132
133     def getNodeByName(self, name):
134         cur = self.conn.cursor()
135         cur.execute("SELECT * FROM nodes WHERE name='" + name + "';")
136         res = cur.fetchone()
137         cur.close()
138         return res
139
140     def deleteNodeByIP(self, ip):
141         cur = self.conn.cursor()
142         cur.execute("DELETE FROM nodes WHERE address='" + ip + "';")
143         cur.close()
144
145     def updateWeights(self, name, weights):
146         cur = self.conn.cursor()
147         ram = weights[0]
148         load1 = weights[1]
149         load5 = weights[2]
150         load15 = weights[3]
151         activevms = weights[4]
152         cur.execute("UPDATE weights SET ram='" + ram + "', load1='" + load1 +    \
153                     "', load5='" + load5 + "', load15='" + load15 +               \
154                     "', activevms='" + activevms+"' WHERE id='" + name + "';")
155         cur.close()
156
157     def getWeights(self, name):
158         cur = self.conn.cursor()
159         cur.execute("SELECT ram, load1, load5, load15, activevms " +             \
160                     "FROM weights WHERE id='" + name + "';")
161         res = cur.fetchone()
162         cur.close()
163         return res
164
165     def getImages(self):
166         cur = self.conn.cursor()
167         cur.execute("SELECT name FROM images;")
168         res = cur.fetchall()
169         cur.close()
170         return res
171
172     def getProfile(self, name):
173         cur = self.conn.cursor()
174         cur.execute("SELECT * FROM profiles WHERE name='" + name + "';")
175         res = cur.fetchone()
176         cur.close()
177         return res
178
179     def insertProfile(self, name, title, desc, ram, vcpu, image):
180         cur = self.conn.cursor()
181         cur.execute("INSERT INTO profiles " +                                    \
182                     "(name,title,description,ram,vcpus,image) " +                \
183                     "VALUES ('" + name + "','" + title + "','" + desc +          \
184                     "','" + ram + "','" + vcpu + "','" + image + "');")
185         cur.close()
186
187     def getStorageNodes(self):
188         cur = self.conn.cursor()
189         cur.execute("SELECT * FROM nodes WHERE type='STORAGE';")
190         res = cur.fetchall()
191         cur.close()
192         return res
```

# Appendix L
# NETWORKING SUPPORT MODULE

```
#!/usr/bin/env python3

'''
networking.py
-----------------
Support some networking operations for THUNDER
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Imports
import subprocess
import shutil
import fileinput
import json
from time import time, sleep

def createMessage(**kwargs):
    return json.dumps(kwargs).encode('UTF8')

def parseMessage(msg):
    if (type(msg) == type(bytes())):
        return json.loads(msg.decode('UTF8'))
    else:
        return json.loads(msg)

'''
getMAC(node) ---
    Expected action:
        Converts the hex value of the node name to proper MAC

    Expected positional arguments:
        node - Hex value returned from getNode()

    Expected return value:
        The proper MAC address of the node
'''
def getMAC(node):
    hexMac = hex(node)
    mac = ''
    for i in range(2,len(hexMac),2):
        mac+=hexMac[i]+hexMac[i+1]+'-'
    return mac[:-1]

def getDHCPRenewTime(mac):
    result = None
    fp = open(constants.get('default.dhcpdLeases'), 'r')
    entries = fp.readlines()
    fp.close()
    for i in range(len(entries)-1, -1, -1):
        line = entries[i]
        if mac.replace('-', ':') in line:
            foundTime = False
            while (not foundTime and i > -1):
                i -= 1
                line = entries[i]
                if 'starts' in line:
```

```
                foundTime = True
            if foundTime:
                result = line.split()[-1]
                break
    return result

def getIPFromARP(mac):
    mac = mac.replace('-', ':')
    # Remove any previous entry for this mac address
    cmd = "/usr/sbin/arp -d `arp -an | grep " + mac + " | awk '{print $2}' | tr -d '()'`"
    subprocess.call(cmd + " > /dev/null 2>&1", shell=True)
    cmd = "./iplookup.sh"
    out = ""
    start_time = int(round(time()*1000))
    cur_time = start_time
    max_time = constants.get('default.maxIPWait')

    while (out == "" and start_time - cur_time < max_time):
        p = subprocess.Popen([cmd, mac], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        out, err = p.communicate()
        out = out.decode().strip()
        cur_time = int(round(time() * 1000))

    return out

def generatePreseed(ip):
    wwwDir = constants.get('default.wwwdir')
    template = wwwDir  + '/preseed_template.cfg'
    template_bare = wwwDir  + '/preseed_bare_template.cfg'
    shutil.copyfile(template, wwwDir + '/preseed_compute.cfg')
    shutil.copyfile(template, wwwDir + '/preseed_storage.cfg')
    shutil.copyfile(template, wwwDir + '/preseed_controller.cfg')
    shutil.copyfile(template_bare, wwwDir + '/preseed_bare.cfg')
    configs = ['compute', 'storage', 'controller', 'bare']
    for config in configs:
        for line in fileinput.input(wwwDir + '/preseed_' + config + '.cfg', inplace=True):
            if '<ROLE>' in line or '<SERVER_IP>' in line:
                print(line.replace('<ROLE>', config).replace('<SERVER_IP>', ip), end='')
            else:
                print(line, end='')
    shutil.chown(wwwDir + '/preseed_compute.cfg', user='thunder', group='thunder')
    shutil.chown(wwwDir + '/preseed_storage.cfg', user='thunder', group='thunder')
    shutil.chown(wwwDir + '/preseed_bare.cfg', user='thunder', group='thunder')
    shutil.chown(wwwDir + '/preseed_controller.cfg', user='thunder', group='thunder')
```

# Appendix M
# WEBSOCKET SUPPORT MODULE

```
#!/usr/bin/env python3

'''
websocket.py
-----------------
Simple websocket support for ThunderRPC
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Imports
from hashlib import sha1
from base64 import b64encode, b64decode
from struct import pack, unpack

def Enum(**enums):
    return type('Enum', (), enums)

# Enum of opcodes for simplicity
Opcode = Enum(
   continuation = 0x00,
   text         = 0x01,
   binary       = 0x02,
   terminate    = 0x08,
   ping         = 0x09,
   pong         = 0x10
)

# Handle websocket upgrade requests and send/receive hybi-13 websocket frames
class websocket:
   def __init__(self, request):
      self.socket = request
      self.magic = '258EAFA5-E914-47DA-95CA-C5AB0DC85B11'

   # Test for a websocket upgrade request.  If request is valid return True.
   # Otherwise return False.
   def isHandshakePending(self, data):
      pending = False
      numKeys = 0
      dataArr = data.splitlines()
      for line in dataArr:
         lineArr = line.decode('UTF8').split(': ')
         if (lineArr[0].lower() == 'upgrade' and \
             lineArr[1].lower() == 'websocket'):
            pending = True
         elif (lineArr[0].lower() == 'sec-websocket-key'):
            numKeys+=1

      if (numKeys != 1):
         pending = False

      return pending

   # Parse the incoming request, and produce the correct handshake response.
   # Key algorithm == base64 encoded sha1 hashed incoming key prepended to
   # websocket magic value.
   def handshake(self, data):
```

```
59      response = 'HTTP/1.1 101 Switching Protocols\r\nUpgrade: websocket' +    \
60                 '\r\nConnection: Upgrade\r\n'
61      dataArr = data.splitlines()
62      key = ''
63      for datum in dataArr:
64         if (datum.startswith('Sec-WebSocket-Key')):
65            key = datum.split(': ')[1]
66      if (key == ''):
67         return False
68      retKey = b64encode(sha1((key+self.magic).encode('UTF8')).digest())
69      response += 'Sec-WebSocket-Accept: ' + retKey.decode() + '\r\n\r\n'
70      return response.encode('UTF8')
71
72   # Wrap data inside of a hybi-13 websocket frame.  Does not require masking
73   # from server->client.
74   def encode(self, opcode, data):
75      encodedData = data
76
77      # Header will be 10000001
78      # 1 - Finalize bit (we wont have a message that exceeds 126 characters)
79      # 000 - 3 reserved bits
80      # 0001 - Text opcode (UTF-8 encoded)
81      # 129 in base10
82      header = pack('!B', (128 | Opcode.text))
83      payloadLen = len(encodedData)
84      if (payloadLen < 126):
85         header += pack('!B', payloadLen)
86      elif (payloadLen < (1 << 16)):
87         header += pack('>B', 126) + pack('>H', payloadLen)
88      elif (payloadLen < (1 << 63)):
89         header += pack('>B', 127) + pack('>Q', payloadLen)
90      return bytes(header + encodedData)
91
92   # Read a hybi-13 websocket frame and unmask the payload.
93   def decode(self, data):
94      payloadLen = data[1] & 127
95      if (payloadLen == 126): # Two more bytes indicate length.  16-bits.
96         payloadLen = unpack('>H', self.socket.recv(2))[0]
97      elif (payloadLen == 127): # Eight more bytes indicate length.
98         payloadLen = unpack('>Q', self.socket.recv(8))[0]
99
100     # Get an array of bytes from the key
101     maskingKey = self.socket.recv(4)
102     mask = [byte for byte in maskingKey]
103
104     # Get the masked data
105     maskedData = self.socket.recv(payloadLen)
106
107     # We use the masking key with mod 4 indexing to unmask the payload.
108     unmasked = ''
109     for byte in maskedData:
110        unmasked += chr(byte ^ mask[len(unmasked) % 4])
111
112     return unmasked
```

Appendix N
LOAD BALANCING SUPPORT MODULE

```
#!/usr/bin/env python3
import random
import threading

'''
load_balancer.py
-----------------
RAIN implementation and generic load balancing functions
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Get the default number of vcores allocated per physical core
VCORES_PERCORE = constants.get('default.vcoresPerCore')
rr_index=0
rr_lock = threading.Lock()

'''
rain_select(nodes, weights, vm) ---
    Expected action:
        Select a node to instantiate using RAIN

    Expected positional arguments:
        nodes - array of node metadata
        weights - array of RAIN weights selected by user
        vm - virtual machine descriptor (RAM, #cores, etc,)

    Expected return value:
        The selected node descriptor if one is suitable, None otherwise
'''
def rain_select(nodes, weights, vm):
   rankings = getRankings(nodes, weights)
   sortedRankings = sorted(rankings, key=lambda rank: rank[2])
   for rank in sortedRankings:
      if (fits(rank[0], vm)):
         return rank[1]
   return None

'''
getRankings(nodes, weights) ---
    Expected action:
        Rank each node using RAIN algorithm

    Expected positional arguments:
        nodes - array of node metadata
        weights - array of RAIN weights selected by user

    Expected return value:
        Vector of nodes and corresponding ranks
'''
def getRankings(nodes, weights):
   vec = []
   for node in reversed(nodes):
      nodeArr = node['result']
      vec.append((nodeArr, node['_HOST_'], rank(nodeArr, weights)))
   return vec

```

```
'''
rank(node, weights) ---
    Expected action:
        Rank one node using RAIN algorithm

    Expected positional arguments:
        node - node metadata
        weights - array of RAIN weights selected by user

    Expected return value:
        Real number rank of one node
'''
def rank(node, weights):
   # RAM
   memTotal = int(node[0])
   memFree = int(node[1])
   memUsed = memTotal - memFree

   # Load
   loadOne = float(node[2])
   loadFive = float(node[3])
   loadFifteen = float(node[4])

   # VCores
   maxVCores = int(node[5])
   activeVCores = int(node[6])

   # Get the actual number of cores
   numCores = maxVCores / VCORES_PERCORE

   # Calculate terms of rank
   sRAM = weights[0] * (memUsed / memTotal)
   sLoad1 = weights[1] * (loadOne / numCores)
   sLoad5 = weights[2] * (loadFive / numCores)
   sLoad15 = weights[3] * (loadFifteen / numCores)
   sActiveVMs = weights[4] * (activeVCores / maxVCores)

   return sRAM + sLoad1 + sLoad5 + sLoad15 + sActiveVMs

'''
fits(node, vm) ---
    Expected action:
        Checks if a virtual machine can fit on a node

    Expected positional arguments:
        node - node metadata
        vm - virtual machine descriptor (RAM, #cores, etc,)

    Expected return value:
        True or False
'''
def fits(node, vm):
   freeVCores = int(node[5]) - int(node[6])
   freeRam = int(node[1])
   if (freeVCores >= int(vm['vcpus']) and freeRam >= int(vm['ram'])):
      return True
   return False

'''
rr_reset() ---
    Expected action:
        Resets the node array index for round robin

    Expected positional arguments:
        None

    Expected return value:
        None
```

```
'''
def rr_reset():
   global rr_index
   rr_lock.acquire()
   rr_index=0
   rr_lock.release()

'''
rr_select() ---
    Expected action:
        Selects a node using round robin

    Expected positional arguments:
        nodes - array of node metadata
        vm - virtual machine descriptor (RAM, #cores, etc,)

    Expected return value:
        The node, if one exists.  None otherwise.
'''
def rr_select(nodes, vm):
   global rr_index
   orig_index = rr_index
   retval = None
   touched = 0
   while (touched < len(nodes)):
      if (rr_index > len(nodes)-1):
         rr_reset()

      rr_lock.acquire()
      node = nodes[rr_index]
      rr_lock.release()
      touched += 1

      if (fits(node['result'], vm)):
         retval = node['_HOST_']
         break
      rr_lock.acquire()
      rr_index += 1
      rr_lock.release()
   rr_lock.acquire()
   rr_index += 1
   rr_lock.release()
   return retval

'''
rand_select() ---
    Expected action:
        Selects a random node

    Expected positional arguments:
        nodes - array of node metadata
        vm - virtual machine descriptor (RAM, #cores, etc,)

    Expected return value:
        The node, if one exists.  None otherwise.
'''
def rand_select(nodes, vm):
   r = random.randrange(0, len(nodes), 1)
   node = nodes[r]['result']
   if (fits(node, vm)):
      return nodes[r]['_HOST_']
   else:
      return None
```

# Appendix O
# ZEUS NODE COMPONENT

```
#!/usr/bin/env python3

'''
server.py
-----------------
Cloud Controller (Zeus) component of Thunder
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Imports
from thunder import *

server = ThunderRPC(role = 'PUBLISHER', group = 'CONTROLLER')
```

# Appendix P
# THOR NODE COMPONENT

```
#!/usr/bin/env python3

'''
computenode.py
-----------------
Compute Controller (Thor) component of THUNDER
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Imports
import tempfile, urllib.request, shutil, os, subprocess
from mysql_support import mysql
from thunder import *
from smb.SMBHandler import SMBHandler
from uuid import uuid1
from time import sleep
from networking import createMessage

'''
instantiate(*params) ---
    Expected action:
        This function should copy the virtual machine image
        from the storage node, unpack the image, generate
        a unique domain name, and instantiate it.

    Expected positional arguments:
        args[0] - Virtual machine name
        args[1] - Username of requester

    Expected return value:
        MAC Aaddress of virtual machine instance with colonds replaced with
        dashesand and the domain name in the format: "<MAC>:<DOMAIN>"
'''
def instantiate(data):
   # Get posiotional arguments
   name = data['vm']
   username = data['user']

   # Get the IP of the publisher
   publisher = client.publisher[0]

   # Get image information
   myConnector = mysql(publisher, 3306)
   myConnector.connect()
   profile = myConnector.getProfileData(name)
   image = myConnector.getImageData(profile['image'])
   serverName = image['node']

   # Lookup name in database to find network location of image
   nas = myConnector.getNASAddress(serverName)[0].split(':')[0]
   domain = str(uuid1()).replace('-','')
   myConnector.insertInstance(domain, '-1', client.name, username, name)
   myConnector.disconnect()

   # transfer the archive over
   archive = image['archive']
```

```
59    copyFromNAS(archive, archive, nas, domain)
60
61    # collect data about the archive contents
62    disk = image['disk']
63    overlay = image['overlay']
64    config = image['config']
65    ram = str(profile['ram'])
66    vcpus = str(profile['vcpus'])
67    dest_dir = constants.get('default.imagedir')
68
69    # clone the image and install into virsh
70    virtHelper = subprocess.Popen(['./cloneAndInstall.sh', archive, domain,      \
71                                   disk, overlay, config, ram, vcpus,            \
72                                   dest_dir], stdout=subprocess.PIPE,            \
73                                   stderr=subprocess.PIPE)
74    out, err = virtHelper.communicate()
75    mac = out.decode().rstrip().replace(':','-')
76    result = {}
77    result['mac'] = mac
78    result['domain'] = domain
79    return result
80
81 '''
82 destroy(*params) ---
83     Expected action:
84         This function should stop and destroy a running
85         virtual machine
86
87     Expected positional arguments:
88         args[0] - Domain name of running virtual machine instance
89
90     Expected return value:
91         0 - successful
92         1 - unsuccessful
93 '''
94 def destroy(data):
95    domain = data['domain']
96    vmDestructor = subprocess.Popen(['./destroyVM.sh', domain],                  \
97                                    stdout=subprocess.PIPE)
98    out, err = vmDestructor.communicate()
99    if (err != ""):
100      return 1
101   publisher = client.publisher[0]
102   myConnector = mysql(publisher, 3306)
103   myConnector.connect()
104   myConnector.deleteInstance(domain)
105   myConnector.disconnect()
106   return 0
107
108 '''
109 destroyAll(*params) ---
110     Expected action:
111         This function should stop and destroy all running
112         virtual machine
113
114     Expected positional arguments:
115         None
116
117     Expected return value:
118         0 - successful
119         1 - unsuccessful
120 '''
121 def destroyAll(data):
122    vmDestructor = subprocess.Popen('./destroyAll.sh',                           \
123                                    stdout=subprocess.PIPE)
124    out, err = vmDestructor.communicate()
125    if (err != ""):
126       return 1
```

```
    publisher = client.publisher[0]
    myConnector = mysql(publisher, 3306)
    myConnector.connect()
    myConnector.deleteInstance(domain)
    myConnector.disconnect()
    return 0

'''
copyFromNAS(imageName, name, server) ---
    Expected action:
        This function should copy a file from the storage node
        to the local images directory

    Expected positional arguments:
        imageName - Name of the archive to copy (ex: ubuntu.tar.xz)
        name - Name to save the image as locally (ex: myimage.tar.xz)
        server - IP and port of storage node to copy data from                \
                 (ex: (10.10.0.1:48574))

    Expected return value:
        0 - successful
        1 - unsuccessful
'''
def copyFromNAS(imageName, name, server, domain):
    # Create a temp file for the image
    opener = urllib.request.build_opener(SMBHandler)
    src = opener.open('smb://' + server + '/share/images/' + imageName)

    # Save the image to the correct location
    dest_dir = constants.get('default.imagedir') + '/' + domain
    if (not os.path.exists(dest_dir)):
        os.mkdir(dest_dir)

    fname = dest_dir + '/' + name
    dest = open(fname, 'wb')
    shutil.copyfileobj(src,dest)

    # Close files
    src.close()
    dest.close()

# Start up the service and register events
client = ThunderRPC()
client.registerEvent('INSTANTIATE', instantiate)
client.registerEvent('DESTROY', destroy)
client.registerEvent('DESTROYALL', destroyAll)
client.findPublisher()
```

# Appendix Q
# INDRA NODE COMPONENT

```
#!/usr/bin/env python3

'''
storagenode.py
-----------------
Storage Controller (Indra) component of Thunder
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Imports
import subprocess
from thunder import *
from networking import createMessage

def importImage(data):
   url = data['url']
   imageDir = constants.get('default.imagedir')
   # Download the image from the url
   p = subprocess.Popen(['./getImage.sh', url, imageDir], stdout=subprocess.PIPE)
   out,err=p.communicate()
   if (err != ''):
      return 1
   print(out)
   # Untar the image

   # Put the data from the YAML into the database
   # Store the image tar in the share
   return 0

client = ThunderRPC(group = 'STORAGE')
client.registerEvent('IMPORTIMAGE', importImage)
client.findPublisher()
```

# Appendix R
# PYTHON DICTIONARY WRAPPER

```python
#!/usr/bin/env python3

'''
dictionary.py
-----------------
Dictionary wrapper with additional functions
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

class Dictionary:
   # Constructor
   def __init__(self):
      self.dict = {}

   # Append key/value pair to dictionary
   def append(self, keyval):
      self.dict[keyval[0]] = keyval[1]

   # Remove a key from the dicionary
   def remove(self, key):
      del self.dict[key]

   # Removes all data from dictionary
   def clear(self):
      self.dict.clear()
      del self.dict
      self.dict = {}

   # Retrieves a value from a key/value pair
   def get(self, key):
      return self.dict[key]

   # Returns True if the dictionary contains the specified key
   def contains(self, key):
      return key in self.dict

   # Returns the length of the dictionary
   def length(self):
      return len(self.dict)

   # Returns the raw Python dictionary collection
   def collection(self):
      return self.dict
```

# Appendix S
# CONFIGURATION LOADING MODULE

```
#!/usr/bin/env python3

'''
config.py
-----------------
Module for parsing the Thunder configuration file
Developed by Gabriel Jacob Loewen
The University of Alabama
Cloud and Cluster Computing Group
'''

# Imports
from dictionary import *
import sys
import builtins

# Create a dictionary to store results
builtins.constants = Dictionary()

'''
parseConfig(fname) ---
    Expected action:
        Read the file, parse the configuration, and store
        the data in a dictionary of key-value mappings

    Expected positional arguments:
        fname - Filename of configuration file

    Expected return value:
        None
'''
def parseConfig(fname):
    fp = open(fname, 'r')
    lines = fp.readlines()
    for line in lines:
        # remove any leading or ending spaces
        line = line.strip()
        if (line != '' and line[0] != '#'):
            keyval = line.split('&')
            key = keyval[0].strip()
            val = keyval[1].strip()
            cIndex = len(val)
            # remove any ending comments
            for i in range(0, len(val), 1):
                ch = val[i]
                if (ch == '#'):
                    cIndex = i
                    break
            val = val[:cIndex].strip()
            if (val[0] == '"' and val[-1] == '"'):
                val = val[1:-1]
            else:
                try:
                    val = int(val)
                except:
                    print('Could not parse config on line:\n\t' + line)
                    sys.exit(-1)
            constants.append([key,val])
```

```
59      fp.close()
```

Appendix T
DEFAULT CONFIGURATION SPECIFICATION

```
# defaults
default.interface               & "eth0" # ("eth0" | "eth1" | "ethN")
default.port                    & 0 # 0 chooses any available port
default.group                   & "COMPUTE" # ("COMPUTE" | "STORAGE" | "CONTROLLER" | "OTHER")
default.role                    & "SUBSCRIBER" # (PUBLISHER | SUBSCRIBER)
default.mcastgrp                & "224.0.0.1"
default.mcastport               & 5007
default.vcoresPerCore           & 6
default.tftpdir                 & "/srv/tftp"
default.imagedir                & "/srv/images"
default.wwwdir                  & "/var/www/html"
default.dhcpdLeases             & "/var/lib/dhcp/dhcpd.leases"
default.maxConnectionRetries    & 10
default.messageTimeout          & 30
default.vmMACPrefix             & "52:54:00"
default.maxIPWait               & 300000

# server settings
server.port                     & 49687
server.interface                & "eth0" # ("eth0" | "eth1" | "ethN")

# heartbeat settings
heartbeat.interval              & 30 # seconds

# timeout settings for publisher lookups
multicast.timeout               & 1 # seconds
```

# Appendix U
# VIRTUAL MACHINE INSTANTIATION SCRIPT

```
#!/bin/bash

{
archive=$1
domain=$2
disk=$3
overlay=$4
config=$5
ram=$6
vcpus=$7
dest=$8

cd $dest/$domain
tar xf $archive
rm $archive
# create a storage pool for this domain
virsh pool-define-as $domain dir - - - - $dest/$domain
virsh pool-build $domain
virsh pool-start $domain

# if there is an overlay image then copy it, otherwise use the base image
if [[ -n "$overlay" ]]; then
   imageName=$overlay
else
   imageName=$disk
fi
virsh pool-refresh $domain
virt-install -r $ram -n $domain --vcpus=$vcpus --hvm --autostart --noautoconsole --vnc --force --
    ↪accelerate --memballoon virtio --boot hd --disk vol=$domain/$imageName,format=qcow2,bus=
    ↪virtio --disk vol=$domain/$config,bus=virtio
sleep 5
MAC=`virsh dumpxml $domain | grep 'mac address' | cut -d\' -f2 | tr -d ' '`
} > INSTANTIATE_LOG 2>&1

echo $MAC
```

# Appendix V
# DATABASE CREATION SCRIPT

```
#!/bin/bash

# Preseed the database with the initial data for THUNDER

echo "Preseeding the MySQL database with values."
echo "-------------------------------------------------"
`cat preseed.sql | mysql -f -uroot -pthunder`
echo "Done."
```

# Appendix W
# DATABASE DUMP

```
-- MySQL dump 10.13  Distrib 5.5.40, for debian-linux-gnu (x86_64)
--
-- Host: localhost    Database: thunder
-- ------------------------------------------------------
-- Server version 5.5.40-0ubuntu0.14.04.1

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `images`
--

CREATE DATABASE thunder;

CREATE USER 'thunder'@'%' IDENTIFIED BY 'thunder';
CREATE USER 'thunder'@'localhost' IDENTIFIED BY 'thunder';
CREATE USER 'thunder'@'0.0.0.0' IDENTIFIED BY 'thunder';
GRANT ALL PRIVILEGES ON thunder.* TO 'thunder'@'%';
GRANT ALL PRIVILEGES ON thunder.* TO 'thunder'@'localhost';
GRANT ALL PRIVILEGES ON thunder.* TO 'thunder'@'0.0.0.0';

USE thunder;

DROP TABLE IF EXISTS `images`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `images` (
  `name` varchar(200) NOT NULL,
  `directory` varchar(200) NOT NULL,
  `disk` varchar(200) NOT NULL,
  `config` varchar(200) NOT NULL,
  `type` varchar(200) NOT NULL,
  `node` varchar(200) NOT NULL,
  PRIMARY KEY (`name`,`type`,`node`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `images`
--

LOCK TABLES `images` WRITE;
/*!40000 ALTER TABLE `images` DISABLE KEYS */;
/*!40000 ALTER TABLE `images` ENABLE KEYS */;
UNLOCK TABLES;

--
-- Table structure for table `instances`
--
```

```
59
60DROP TABLE IF EXISTS `instances`;
61/*!40101 SET @saved_cs_client     = @@character_set_client */;
62/*!40101 SET character_set_client = utf8 */;
63CREATE TABLE `instances` (
64  `domain` varchar(200) NOT NULL,
65  `ip` varchar(15) NOT NULL,
66  `node` varchar(200) NOT NULL,
67  `owner` varchar(200) NOT NULL,
68  `profile` varchar(200) NOT NULL,
69  PRIMARY KEY (`domain`,`owner`),
70  UNIQUE KEY `domain_2` (`domain`),
71  KEY `ip` (`ip`),
72  KEY `domain` (`domain`),
73  KEY `ip_2` (`ip`)
74) ENGINE=InnoDB DEFAULT CHARSET=latin1;
75/*!40101 SET character_set_client = @saved_cs_client */;
76
77--
78-- Dumping data for table `instances`
79--
80
81LOCK TABLES `instances` WRITE;
82/*!40000 ALTER TABLE `instances` DISABLE KEYS */;
83/*!40000 ALTER TABLE `instances` ENABLE KEYS */;
84UNLOCK TABLES;
85
86--
87-- Table structure for table `nodes`
88--
89
90DROP TABLE IF EXISTS `nodes`;
91/*!40101 SET @saved_cs_client     = @@character_set_client */;
92/*!40101 SET character_set_client = utf8 */;
93CREATE TABLE `nodes` (
94  `name` varchar(100) NOT NULL DEFAULT '',
95  `address` varchar(100) DEFAULT NULL,
96  `type` varchar(100) DEFAULT NULL,
97  PRIMARY KEY (`name`)
98) ENGINE=InnoDB DEFAULT CHARSET=latin1;
99/*!40101 SET character_set_client = @saved_cs_client */;
100
101--
102-- Dumping data for table `nodes`
103--
104
105LOCK TABLES `nodes` WRITE;
106/*!40000 ALTER TABLE `nodes` DISABLE KEYS */;
107/*!40000 ALTER TABLE `nodes` ENABLE KEYS */;
108UNLOCK TABLES;
109
110--
111-- Table structure for table `profiles`
112--
113
114DROP TABLE IF EXISTS `profiles`;
115/*!40101 SET @saved_cs_client     = @@character_set_client */;
116/*!40101 SET character_set_client = utf8 */;
117CREATE TABLE `profiles` (
118  `name` varchar(200) NOT NULL,
119  `image` varchar(200) NOT NULL,
120  `title` varchar(200) NOT NULL,
121  `description` varchar(2000) NOT NULL,
122  `packages` varchar(2000) NOT NULL,
123  `ram` int(11) NOT NULL DEFAULT '1024',
124  `vcpus` int(11) NOT NULL DEFAULT '1',
125  PRIMARY KEY (`name`)
126) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `profiles`
--

LOCK TABLES `profiles` WRITE;
/*!40000 ALTER TABLE `profiles` DISABLE KEYS */;
/*!40000 ALTER TABLE `profiles` ENABLE KEYS */;
UNLOCK TABLES;

--
-- Table structure for table `user`
--

DROP TABLE IF EXISTS `user`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `user` (
  `fname` varchar(100) DEFAULT NULL,
  `lname` varchar(100) DEFAULT NULL,
  `role` varchar(100) DEFAULT NULL,
  `password` varchar(100) DEFAULT NULL,
  `username` varchar(100) NOT NULL DEFAULT '',
  `vmquota` int(11) NOT NULL DEFAULT '1',
  PRIMARY KEY (`username`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `user`
--

LOCK TABLES `user` WRITE;
/*!40000 ALTER TABLE `user` DISABLE KEYS */;
INSERT INTO `user` VALUES ('Default','Admin','SUPERUSER','49
    ↪eff747f7b66f70133bfe00aa8ac2d6b0fbee5be80e52537b0163f147d20418','admin',99),('Test','Test','
    ↪REGULAR','9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08','test',1);
/*!40000 ALTER TABLE `user` ENABLE KEYS */;
UNLOCK TABLES;

--
-- Table structure for table `weights`
--

DROP TABLE IF EXISTS `weights`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `weights` (
  `ram` float NOT NULL,
  `load1` float NOT NULL,
  `load5` float NOT NULL,
  `load15` float NOT NULL,
  `activevms` float NOT NULL,
  `id` varchar(64) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `cpu` (`ram`),
  KEY `cpu_2` (`ram`),
  KEY `cpu_3` (`ram`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `weights`
--

LOCK TABLES `weights` WRITE;
/*!40000 ALTER TABLE `weights` DISABLE KEYS */;
```

```
193 INSERT INTO `weights` VALUES (40,30,10,10,10,'balance');
194 /*!40000 ALTER TABLE `weights` ENABLE KEYS */;
195 UNLOCK TABLES;
196 /*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
197
198 /*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
199 /*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
200 /*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
201 /*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
202 /*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
203 /*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
204 /*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;
205
206 -- Dump completed on 2014-12-09 18:45:46
```

# Appendix X
# ZEUS SERVICE UPSTART CONFIGURATION

```
description "ThunderRPC Server"
author "gloewen@crimson.ua.edu"

start on runlevel [2345]
stop on runlevel [!2345]

respawn
expect fork

script
   cd /srv/thunder/ThunderRPC
   exec setuidgid thunder python3 server.py
end script
```

# Appendix Y
# THOR SERVICE UPSTART CONFIGURATION

```
description "ThunderRPC Server"
author "gloewen@crimson.ua.edu"

start on runlevel [2345]
stop on runlevel [!2345]

respawn

script
    cd /srv/thunder/ThunderRPC
    exec \
    setuidgid thunder \
    python3 computenode.py
end script
```

# Appendix Z
# INDRA SERVICE UPSTART CONFIGURATION

```
description "ThunderRPC Server"
author "gloewen@crimson.ua.edu"

start on runlevel [2345]
stop on runlevel [!2345]

respawn

script
    cd /srv/thunder/ThunderRPC
    exec \
    setuidgid thunder \
    python3 storagenode.py
end script
```