Summer 2016

Vertical Implementation of Cloud for Education (V.I.C.E.)

Travis S. Brummett
Western Kentucky University, travis.brummett757@toppermail.com

Follow this and additional works at: http://digitalcommons.wku.edu/theses

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
http://digitalcommons.wku.edu/theses/1643

This Thesis is brought to you for free and open access by TopSCHOLAR®. It has been accepted for inclusion in Masters Theses & Specialist Projects by an authorized administrator of TopSCHOLAR®. For more information, please contact topscholar@wku.edu.
VERTICAL IMPLEMENTATION OF CLOUD FOR EDUCATION (V.I.C.E.)

A Thesis
Presented to
The Faculty of the Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science

By
Travis Brummett
August 2016
Date Recommended

Dr. Michael Galloway, Director of Thesis

Dr. James Gary

Dr. Zhonghang Xia

7/5/2016

Dean, Graduate Studies and Research Date
DEDICATION

To my mother, father, sister, and friends.

&

To the faculty of the Computer Science Department at Western Kentucky University.
ACKNOWLEDGMENTS

My appreciation goes to my professor and adviser, Dr. Michael Galloway. I am grateful for his support and advice throughout my Masters program. I truly appreciate the sacrifices he has made to meet with me, to help me attend conferences, and to provide his students with a place and supplies to do research. I have never seen someone so invested in his students as Dr. Galloway. He strives to find his students funding and built a lab for them to work. On one occasion he even took me to tour a potential school in which I could pursue my Ph.D. I consider him to be a friend as well as a mentor and hope to keep in touch throughout the future. Thank you Dr. Galloway for all that you have done for the Students at WKU.

To Dr. James Gary, head of the Computer Science Department and the first contact I made at WKU, thank you. Dr. Gary always did whatever he could to help students out however he could. Dr. Gary helped me transition into the undergraduate program and did the same when I began the Masters program. He helped me secure the funding to go to a conference in New York City giving me a life experience that I will never forget. On top of all that he is one of the funniest people I know. Thank you Dr. Gary for everything you’ve done over these past many years.

To Dr. Xia, the graduate student adviser or the Computer Science Department, Thank you for your help and advice through my graduate career. I would also like to
express my appreciation for giving me the opportunity to be a graduate assistant. Thank you!

To the rest of the Western Kentucky University faculty, thank you! I would not be where I am today without you. I apologize that I cannot thank everyone of you individually. Thank you all for what you have done for me. I truly appreciate it.

I would also like to acknowledge My friends and family. Thank you for your continued support. I appreciate everything you do. Special thanks to my friends and fellow students Pezhman Sheinidashtegol, David Beverly, Harinivesh Donepudi, and the rest of the cloud group. Thank you for everything!

A special thank you to my friend Chris Goulet. I appreciate the time he spent in the lab helping me better my understanding of PHP and web programming. It was through his help that I was able to build my interface. Thank you for your tutoring and your friendship.

Finally, I would like to thank the financial support provided by the Department of Computer Science, the graduate school, and Ogden College. Thank you for the opportunities you have provided me at WKU.
## CONTENTS

1 INTRODUCTION ................................................................. 1

2 BACKGROUND ................................................................. 4

3 V.I.C.E. HARDWARE .......................................................... 16
   3.1 Architecture ............................................................ 16
      3.1.1 Compute Nodes ................................................... 17
      3.1.2 Head Node ......................................................... 18
      3.1.3 Storage Node ..................................................... 19
   3.2 Advantages .............................................................. 20
      3.2.1 Initial Purchase of Equipment ............................... 20
      3.2.2 Costs to Maintain ............................................... 22
      3.2.3 Power Consumption ............................................. 22
   3.3 Disadvantages .......................................................... 25

4 V.I.C.E. MIDDLEWARE ...................................................... 27
   4.1 Load Balancing ........................................................ 27
   4.2 Scalability .............................................................. 28
   4.3 Middleware .............................................................. 29
      4.3.1 Overview .......................................................... 29
      4.3.2 Architecture ...................................................... 30
      4.3.3 Compute Nodes .................................................. 31
LIST OF FIGURES

1.1 Local, Public, and Hybrid Clouds. [Galloway, 2015] .......................... 2

3.1 V.I.C.E. Lab with Raspberry Pis as thin clients. ............................... 18

3.2 V.I.C.E. Hardware. ........................................................................ 19

3.3 Daily Watt Consumption. ................................................................. 23

3.4 kWh Consumption. ......................................................................... 24

4.1 /etc/network/interfaces File. .............................................................. 32

4.2 network.xml File. ............................................................................ 32

4.3 VMBuilder Command. ...................................................................... 33

4.4 Results from Select * from nodes; .................................................. 36

4.5 Results from Select * from com1; .................................................... 36

5.1 Results from Select * from ids; ....................................................... 45

5.2 Login Page. ..................................................................................... 48

5.3 Failed Login. ................................................................................... 49

5.4 Student Page. .................................................................................. 49

5.5 Student VM Started. ........................................................................ 50

5.6 Terminal. ........................................................................................ 51

5.7 Terminal Post Log In. ..................................................................... 51

5.8 Program Written In VM. ................................................................. 52

5.9 Program Results. ............................................................................ 52
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>CC</td>
<td>Cluster Controller</td>
</tr>
<tr>
<td>CD-ROM</td>
<td>Compact Disc Read Only Memory</td>
</tr>
<tr>
<td>CLC</td>
<td>Cloud Controller</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheet</td>
</tr>
<tr>
<td>DHCP</td>
<td>Dynamic Host Configuration Protocol</td>
</tr>
<tr>
<td>fos</td>
<td>Factored Operating System</td>
</tr>
<tr>
<td>GB</td>
<td>Gigabyte</td>
</tr>
<tr>
<td>Gbps</td>
<td>Gigabits per second</td>
</tr>
<tr>
<td>HDD</td>
<td>Hard Disk Drive</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper-text Markup Language</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>ID</td>
<td>Identity</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based Virtual Machine</td>
</tr>
<tr>
<td>kWh</td>
<td>KiloWatt-hours</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>NC</td>
<td>Node Controller</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a Service</td>
</tr>
<tr>
<td>PHP</td>
<td>PHP: Hypertext Preprocessor</td>
</tr>
<tr>
<td>PVM</td>
<td>Para-Virtualized Machine</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>S3</td>
<td>Simple Storage Service</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>SATA</td>
<td>Serial Advanced Technology Attachment</td>
</tr>
<tr>
<td>SC</td>
<td>Storage Controller</td>
</tr>
<tr>
<td>SSH</td>
<td>Secure Shell</td>
</tr>
<tr>
<td>US</td>
<td>United States</td>
</tr>
<tr>
<td>USD</td>
<td>United States Dollar</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VT</td>
<td>Virtualization Extensions</td>
</tr>
<tr>
<td>W</td>
<td>Watt</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
There are several different implementations of open source cloud software that organizations can utilize when deploying their own private cloud. Some possible solutions are OpenNebula, Nimbus, and Eucalyptus. These are Infrastructure-as-a-Service (IaaS) cloud implementations that ultimately gives users virtual machines to undefined job types. A typical IaaS cloud is composed of a front-end cloud controller node, a cluster controller node for controlling compute nodes, a virtual machine image repository node, and many persistent storage nodes and compute nodes. These architectures are built for ease of scalability and availability.

Interestingly, the potential of such architectures could have in the educational field remains vastly underutilized. Large labs filled with costly machines could be replaced by an IaaS implementation of a cloud. The purpose of this thesis is to propose such an implementation for use in Computer Science courses.

The vertical architecture that I propose is known as V.I.C.E. which stands for Vertical Implementation of a Cloud for Education. It consists of a head node which will control the other nodes and handle the operations required to launch or terminate virtual machines (VMs), and five heterogeneous compute nodes on which VMs can be launched.

The most important features of my architecture are its scalability and simplicity. A middle-ware I developed is launched by the head node and draws the compute node information from a database and uses it to carry out operations. The user will access these operations through a web interface meaning that all the complexity is hidden from them. To add compute nodes one would simply hook up the machine and add its information to the database on the head node making the entire architecture highly scalable.
The goal of this research is to replace large costly computer labs with a vertical IaaS cloud architecture. This system would use thin clients to launch VMs from the cloud and allow students to a complete Operating System(OS) at a vastly reduced cost. More machines could also be added to the local cloud giving it the potential to support many more users.
Chapter 1

INTRODUCTION

There is no specific definition defining the paradigm of cloud computing. However, the National Institute of Standards and Technology (NIST) defines cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable resources that can be rapidly provisioned and released with minimal management effort or service provider interaction."

The NIST also identifies five core characteristics of cloud computing. The first of these characteristics is on-demand self service, meaning that users can acquire access to resources without interaction with the service provider. This leads directly into the second characteristic, broad network access. The cloud must be accessible over a network using standard access methods. Resource pooling is also important to a cloud architecture. The resources must be available to multiple users at the same time. The cloud also needs to be able to grow and shrink in the number of resources in can provision based on the demands it must meet. This characteristic is known as rapid elasticity. Finally, since cloud computing resources are generally provided on a pay-per-use model, like a utility, the service must be measurable.

There are also multiple service models. The three most commonly used are Infrastructure-as-a-Service(IaaS), Platform-as-a-Service(PaaS), and Software-as-a-Service(SaaS). An IaaS system gives the user the ability to access computing and storage resources on demand
through virtual machines. This model allows the user to access complete operating systems giving them the most freedom of the three models. A SaaS model allows the user access to one application that the service manager controls. It provides the user with the least amount of control. A PaaS model will allow end users to access the application and deployment frameworks. The PaaS model also allows the user to utilize the operating system.

In addition to the different service models, there are also different deployment models. A public cloud is available to the general public or a large group. In the public deployment model the service is provided by an external source. A private cloud, on the other hand, is provided by either a internal or external source for a specific group or organization. A amalgam of these two deployment models is referred to as a hybrid cloud. The final deployment model is called a community cloud. A community cloud can be made up of both public and private clouds and usually provides services to multiple organizations supporting the same community. [Bahga, 2014]

![Figure 1.1: Local, Public, and Hybrid Clouds.](Galloway, 2015)
The architecture proposed in this thesis will be a vertical IaaS cloud known as V.I.C.E. which stands for Vertical Implementation of a Cloud for Education. The purpose of the proposed architecture is to provide a simple yet more cost efficient alternative to traditional computer labs. This will be done through the use of the cloud architecture and thin clients which will consume less energy. Chapter two will discuss the research being done in regards to cloud computing. In chapter 3 a discussion on the hardware used will be presented. This chapter will also have a cost and power consumption comparison between the traditional computer lab and a lab using the proposed architecture. Chapter four will discuss the middleware used to manage the architecture. It will explain the load balancing process and scalability. Chapter five will go into details regarding the user interface. Chapter six will outline future work. Finally, chapter seven will present the conclusion.
Chapter 2

BACKGROUND

The idea of what a cloud should be is different to everyone. The authors of [Ubuntu, 2015] introduces cloud computing as a combination of cultural, commercial, economical and technological conditions. They feel that while the cloud has many benefits there are three main risks associated with it. These risk include doing nothing, transitional risks related a disruptive change in the industry caused by cloud computing, and the general risks associated with outsourcing. The authors feel that open source technology will allow users to work with the cloud as needed behind a firewall and that the competitive market will then be based around standards. To that extent, Canonical proposes the use of Ubuntu Enterprise Cloud which allows users to build their own private clouds based around the standards of those set by other popular vendors.

The authors of [Systems, 2010] also discuss a private cloud architecture. They specifically explain the Eucalyptus architecture. The authors suggests that there are three concepts that one must consider when developing a cloud. The first of which is operational changes. This means that users and IT personal need to adapt to the changes required to operate the cloud. Governance is the second consideration. Due to the clouds dynamic nature the security policies associated with it must also be dynamic. Cost is the third concept and is a very important aspect of private clouds.

[Systems, 2010] also discusses five steps to developing a private cloud. The first
The first step is to adopt a technology for machine virtualization. The second step is to implement performance requirements as well as memory, compute, and storage usage for profile applications. Step three is to design a VM design consultancy. This will provide users with the help they require when developing or debugging on the virtual machine. The fourth step is to develop accounting and recharge policies adapted to the self-service model. This means that the cloud provider can develop an incentive to get people to release resources that they are not currently using by charging them. Otherwise, a user might hoard the resources. The fifth and final step is to build a deployment and deploy a private cloud.

[Systems, 2009] and [Nurmi, 2009] both discuss the Eucalyptus architecture as well. The name Eucalyptus stands for "Elastic Utility Computing Architecture Linking Your Programs To Useful System". The open-source software is made up of multiple components. The first of which is the storage controller (SC). It implements a block-access network storage. The cloud controller (CLC) is the entry point to the cloud. It queries the other node controllers (NC) and uses the information it gets from them to make scheduling decisions. The cluster controller (CC) provides the link between NCs and the CLC. It takes stock of the virtual machines and schedules virtual machine execution. NCs are executed on every node that is meant to host a virtual machine. They inspect, launch, and terminate the virtual machines on the node. The NCs are also responsible for locating and cleaning local instance images. The next component is called Walrus. It allows users to create, delete, and list buckets. It also allows users to put, get, and delete objects. The Walrus can be used to set access control policies as well. It is compatible with Amazon’s S3. The final component is the management platform which provides an interface to various Eucalyptus services and modules.
The authors of [Systems, 2009] also discuss three steps to deploy a Eucalyptus cloud. They say first you must recognize projects that utilize virtualization. Then, you must select an application to run on the cloud. Finally, one must set-up a test-bed and deploy Eucalyptus. Eucalyptus is just one of many open source architectures.

The author of [Chappell, 2008] discusses how cloud platforms allow users to create applications to run in the cloud. The author claims that cloud platforms do not offer the full spectrum of an on-premises environment yet. The author says an example of this is that business intelligence as part of the platform is not common, and neither is support for business process management technologies such as full-featured workflow and rules engines. The author looks for this to change and foresees that the attraction of scalability and low costs will play a big role in application development within the cloud.

[Weiss, 2007] discusses how cloud computing is both a new concept and a return to the computer’s roots. The author states that the idea of centralizing computing power has been around for a long time and that the cloud concept draws from previous architectures. The author does state that the concept is new in the fact that it incorporates utility computing, distributed computing, and software as a service. This causes a shift in power from the processing unit to the network.

[Armbrust, 2009] also discusses the architecture of cloud computing. According to the authors cloud computing is a long held dream. They state that there are three primarily new concepts introduced in cloud computing. The first is the illusion of infinite resources on demand. This eliminates the need for users to plan far ahead for provisioning. The second concept is the elimination of commitment. This allows companies to increase
resources only when there is a need. The third concept is the ability to pay for use of computing resources on a short-term basis as needed.

The authors of [Armbrust, 2009] also argue that cloud computing is an old idea given new life. They attribute this to new business models and technological trends. The authors also outline several obstacles. The first is availability of service. Organizations worry that utility computing will not have adequate availability. The authors propose multiple cloud providers to combat this particular issue. The second obstacle is data lock-in. This is a reference to potential lost data and how best to resolve the issue and who is to be held accountable. The solution to which is a standardized API. This would allow customers to extract data from one site to another easily. The third obstacle is data confidentiality and auditability. The authors argue that a service provider would easily give up data requested by authorities and suggest encryption as the solution to this obstacle. The next two obstacles are data bottle-necking and performance unpredictability. The authors also cite scalable storage and scaling to quickly as obstacles that have been discussed before. Bugs in the large scale system would also be problematic. Interestingly, the authors also note reputation sharing as an obstacle and point out that the bad behavior of one user could reflect on others. The issue then becomes a question of legal liability. The final obstacle mentioned is software licensing. However, despite all this the authors still see a lot of promise in the further of cloud computing.

The authors of [Hayes, 2008] discuss the roots of cloud computing and how not all the questions about confidentiality and privacy have been addressed. They claim the idea of cloud computing is a resurgence of an idea the came about 50 years ago. Back then the idea took the form of service and time-sharing systems. In the 1980s the appeal of personal
computers was that they liberated users. The authors note that this resurgence is in direct contradiction to this liberation. They claim the difference this time is that instead of using a hub as was the case in the past, the machines are connected through the internet. This adds to the popularity.

The authors of [Hayes, 2008] also discuss the concern of privacy and confidentiality. They provide a hypothetical situation in which a government agency presents a subpoena or search warrant to the third party that has possession of the users. The authors claim that if the user had the physical copy they would be able to decide for themselves if they wanted to surrender the data. In the cloud scenario the user would be denied this choice because the third party provider would most likely be unwilling to go to court for the user. The authors believe that most digital information will be on a cloud before such issues are addressed.

[Haeberlen, 2010] argues that clouds should be made accountable to their customers, and have argued that both the customers and the cloud providers stand to benefit from it. The customer can check whether their computations are being performed correctly, and the provider can more easily handle complaints and resolve disputes. The author outlines several requirements for an accountable cloud. A cloud is considered accountable if faults can be reliably detected, and each fault can be undeniably linked to at least one faulty node. The author specifically states that an accountable cloud has four concepts. The first is identities. This means every action is linked to the node that preformed it. The second feature is a secure record. This way the system maintains a record of past actions such that nodes cannot secretly omit, falsify, or tamper with its entries. The third characteristic is auditing which allows the record to be searched for faults. Finally, there needs to be evidence
meaning that when an auditor detects a fault, it can obtain evidence of the fault that can be verified independently by a third party.

The author of [Haeberlen, 2010] goes on to propose the implementation of a primitive called AUDIT(A,S,t1,t2) on the cloud that can be invoked by the customer to determine whether the cloud has fulfilled an agreement A for a service S during the interval [t1 ...t2]. This proposed primitive would return either OK, to indicate that the service has conformed to the agreement in the specified time interval, or it would provide some evidence that the service had failed to conform to the agreement. The author states that if accountability is to be adopted, it must strike a balance between the requirements of the customer and those of the provider. Thus, it must avoid both false negatives which are bad for the customer and false positives which are bad for the provider. It would also need to provide evidence to resolve disagreements. A third party could also be useful in this resolution. The author proposes a series of guarantees. The first is completeness which means that if the agreement is violated, AUDIT will report this eventually and will produce evidence. Secondly, it must be accurate. AUDIT should not report a violation if none exist. Finally, it must be verifiable. Any evidence of an alleged violation should be checked independently by a third party even if the third party trusts neither the customer nor the provider. The author does mention that these techniques could be used to make a cloud accountable for correctness and for performance. However, it is not yet clear how to achieve accountability for other properties, such as confidentiality. The author also states that this is not meant to be final.

The authors of [Jensen, 2009] also describe cloud computing security. Their work outlines several different types of security threats. Within their work the authors classify these threats under four categories. The first category is XML signature related threats.
The authors state most of the attacks under this category are mostly theoretical. The second category are those threats related to browser security. The threats under this category are the legacy Same Origin Policy and attacks of Browser-based cloud authentication. To combat these issues the authors suggest four methods to combat these threats. They are TLS Federation, SAML 2.0 Holder-of-Key Assertion Profile, Strong Locked Same Origin Policy, and TLS session binding.

The third category mentioned by the authors of [Jensen, 2009] is the threats associated with cloud integrity and binding issues. The first threat mentioned in this category is cloud malware injection attacks. The authors claim that a countermeasure approach to this threat consists in the cloud system performing a service instance integrity check prior to using a service instance for incoming requests. The second threat listed in this category is a metadata spoofing attack. Flooding attacks are also a concern. Such threats include both direct and indirect denial of service. The final category is made up of threats related to accounting and accountability. The authors mention that given the nature of the pay per use model of cloud computing that a flooding attack would drive up the cost. They argue that in such a scenario a service provider would need to know where such an attack originated so they could hold the guilty party accountable rather than the victim.

There are other aspects of cloud computing to explore as well. The authors of [Wentzlaff, 2009] propose a solution to the challenges of constructing operating systems for manycore and cloud systems in the form of an factored operating system (fos). The authors point out the four major challenges the fos must overcome. The fos must be designed to handle scalability, elasticity, and faults in hardware and software. The authors also mention the difficulty of programming large systems. The authors list some design principles
for their fos. First, space multiplexing replaces time multiplexing. This means scheduling becomes a layout problem not a time multiplexing problem. The operating system runs on distinct cores from applications, and working sets are spatially partitioned. The operating system does not interfere with an application’s cache. The second design principle is that the operating system is factored into function specific services. According to the authors, this means that servers collaborate and communicate only via message passing, are bound to a core, leverage ideas from internet servers, and that applications communicate with servers via message passing. The third principle presented is that the operating system adapts resource utilization to changing system needs. This would mean that each service utilization is measured and highly loaded system servers are provisioned more cores. The operating system would have to closely manages resources utilized. Finally, the last principle is that faults are detected and handled by the operating system. The operating system services are monitored by a watchdog process and a name server reassigns the communication channel on the detected faults.

The authors of [Wentzlaff, 2009] claim that the fos provides solutions for the above challenges. To enable applications and the operating system to scale the fos incorporates several approaches. First, the operating system is factored by service being provided. Scalability is increased because each service can run independently. Also, fos’s messaging API is transparent with respect to whether a message’s destination is on the same machine or on a different machine. This makes it easy to move OS services to different machines as needed. Finally, the authors claim that fos’s operating system APIs contain core and machine specifiers thereby enabling the operating system server on one machine to manage the resources for cores on another. To combat elasticity the fos has a adaptable scheduler. The
scheduler receives feedback from the operating system services and adapts accordingly.
The authors also claim that the fos addresses elasticity of demand by providing a single
system image operating system which can grow and shrink across a cloud. According to
the authors the fos is designed to be fault tolerant by using feedback from running servers to
detect that a fault has occurred and takes advantage of the replicated design to recover from
such faults. This counters the challenges posed by faults. The authors provide a solution to
the difficulty of programming large systems. They claim that fos is unlike traditional cloud
and cluster systems, it provides a single system image to the application. This means that
the application interface is that of a single machine while the operating system implements
this interface across several machines in the cloud. fos has already been implemented Xen
Para-Virtualized Machine (PVM) operating system.

The authors of [Paradowski, 2014] tested the performance of two open source
clouds known as OpenStack and CloudStack. The authors say that for their experiments
both OpenStack and CloudStack were deployed and configured as virtual instances using
Oracle VM VirtualBox 4.2.10 on a host machine configured with an Intel Core i72630QM
running with full Virtualization Technology (VT) enabled alongside 8GB DDR3 memory
and 700GB of hard drive storage. The authors then tested both clouds under six conditions.
The first condition was to test processor/RAM effect on deployment times. The authors
found that OpenStack showed a more efficient machine deployment time in comparison to
CloudStack. The second condition was processor/RAM effect on deletion times. This time
the authors found that OpenStack once again was the faster of the two. Next, the authors
tested the effect of hard drive size on deployment times. The authors found that hard disk
size increases deployment time. The next test was the effect of hard drive size on deletion
times. The result of this test was OpenStack once again had better performing averages in comparison to CloudStack, but in this test the differences were very minimal. The authors next tested the effect of hard drive size deployment on host CPU utilization. The authors found that there is no correlation between the hard drive size and the host CPU utilization during deployment. Effect of hard drive size deletion on host CPU utilization was the final test. Again no correlation was found. The authors conclude that generally overall OpenStack offers the best performance. However, they state this benchmarking process warrants further investigation.

The authors of [Vogels, 2009] discuss data consistency within a large-scale reliable distributed systems. The authors state that there are two ways of looking at consistency. One of these is from the developer/client point of view. This is how they observe data updates. The other is from the server side. According to the authors the server side view is how updates flow through the system and what guarantees systems can give with respect to updates. The authors discuss several types of consistencies. The first is strong consistency. This is when a process can access the storage after an update and receive the updated information. The second type, weak consistency, occurs when the system cannot guarantee a subsequent access will yield the updated data. Eventual consistency is a form of weak consistency. In eventual consistency the storage system guarantees that if no further updates are made to the object, eventually all accesses will return the last updated value. The authors then break eventual consistency into even more categories. The first is casual consistency. In this form if one process has communicated with another process in which it has updated a data item, a subsequent access by the second process will return the updated value, and a write is guaranteed to override the earlier write. However, a subsequent access
by a third process is subject to the general rules of eventual consistency. The third type, read-your-writes consistency, is a special type of casual consistency. In it a process, after having updated a data item, always accesses the updated information and never sees the outdated value. Session consistency is the next type of eventual consistency. It is session based in that as long as the session exists the system guarantees read-your-writes consistency. Monotonic read consistency is the next type. In this if a process has seen a particular value for the object, any subsequent accesses will never return any previous values. The final type is monotonic write consistency. In this case, the system guarantees to serialize the writes that are made by the same process. The author states that there are two reasons for data inconsistency to be tolerated. They are improving read and write performance under highly concurrent conditions and handling partition cases where a majority model would render part of the system unavailable even though the nodes are up and running. Developers must be aware that consistency guarantees are provided by the storage systems and must take that into account when developing applications for the system.

The authors of [Zeng, 2009] propose a cloud storage architecture. They believe that the key technologies of cloud storage include many facets from servers, networks, clients, and related control measures. The authors discuss the deployment, virtualization, availability, data organization, data migration, load balance, redundant data deletion, storage, and security. The authors claim that although the applications of cloud storage have been developed practically and rapidly, the integration and operation mechanism in business and commerce should still need unified specifications and standards. They briefly delve into operation mechanism, ecology chain, game theory, ant colony optimization, storage resource convergence and evolution mechanisms and how they affect the development of
these standards. According to the authors these mechanisms warrant further study. The authors propose a layered architecture.

The research discussed in this paper centers around the development of a proof of concept cloud architecture. Thus, while the architecture previously discussed may influence it, security and consistency will be largely ignored. However, the main focus of the proposed architecture is load balancing across nodes, the costs due of the architecture’s development, energy consumption, and the design of a web interface. All of which will be discussed in detail in the following chapters.
Chapter 3

V.I.C.E. HARDWARE

The purpose of this chapter is to introduce the Vertical Implementation of Cloud for Education (V.I.C.E.) architecture and outline both the advantages and disadvantages of creating a computer lab with this approach versus a traditional computer lab. V.I.C.E. is designed to be used in schools and other educational facilities as a cost effective alternative to the traditional approach. The V.I.C.E. could allow institutes which do not have a great deal of funding to have a quality computer lab. This chapter will first discuss the hardware used in my cloud architecture. Then it will examine the advantages in building a lab based on a cloud architecture by discussing the comparison in cost to build such a lab versus the cost of a traditional lab. This chapter will also discuss the energy usage of each approach. Finally, the chapter will conclude with a discussion on the disadvantages of such a lab.

3.1 Architecture

One of the important aspects of the V.I.C.E. architecture is that it can be built using heterogeneous computing components. In the event that new nodes are added, a heterogeneous architecture will perform the same regardless of the memory or processing power of the newly added machine. This means that no two machines within the cluster have to be exactly the same as far as amounts memory or types of CPUs. Another important characteristic is that it is easily scalable. This means that new machines and more virtual machines
can be added at any point in the future simply by connecting the new machine into the network, adding the new machine’s name and IP to the database, and following the steps outlined in the next chapter to setup the software. Currently, the V.I.C.E. architecture is made up of six machines. Five of the machines serve as compute nodes and one serves as a head node. The compute nodes host Virtual Machines which users can utilize like a regular operating system. The compute nodes supply up to 50 virtual machines (VMs) that can be accessed via Raspberry Pis, tiny low cost computers, which will serve as thin clients. This can be seen in figure 3.1. The architecture’s network is managed by Ubiquiti ERPOE-5 EDgeRouter PoE Advanced 5-Port Router and a TP-LINK TL-SG1016 10/100/1000 Mbps Unmanaged 16-Port Gigabit rack mountable switch. Cat 6 Gigabit cable is used to connect the machines. The purpose of this architecture is to provide a both a cost and energy efficient alternative to a traditional computer lab. The switch to the V.I.C.E. architecture would not only save money on the initial costs of a lab but also on maintaining it overtime.

3.1.1 Compute Nodes

The V.I.C.E. architecture has five compute nodes. Compute nodes serve as the point were the work is actually accomplished in the local cloud architecture. It is the compute nodes job to launch and host the VMs. The compute nodes within the V.I.C.E. architecture are referred to as com1, com2, com3, com4, and com5. Com1, com2, and com3 are comprised of ECS BAT-I(1.2) J1800 Intel Celeron J1800 2.41 GHz Mini ITX Motherboard/CPU/VGA Combo. Com2 and com3 each possess 4GB RAM. Com1 has an additional 4GBs of RAM and a 64GB SATA III MLC internal Solid State Drive (SSD). These three each can host at least 6 Virtual Machines(VMs).

Com4 and com5 are more powerful machines than their counterparts. They have
Intel Core i5 quad-core processors, 128GB SATA III Internal SSD, and 16GB RAM. This allows them to support at least 16 VMs each. The actual amount of possible VMs may vary based on instance characteristics such as number of CPU cores, amount of RAM, local disk space, and other factors. Because the information of a new node is entered into a database, the middleware can access the database and get the information needed to access the new node.

3.1.2 Head Node

A head node serves as the point in which scheduling and load balancing are handled. In the V.I.C.E. architecture the head node serves as the home for the loadbalancer. It also
hosts the web interface in which users can connect to the architecture and launch VMs. The head node has an Intel Core i3 Dual-Core processor and 8GB of RAM.

3.1.3 Storage Node

Typically within a cloud architecture a storage node is used to save individual users data and to store images used for creating VMs. However, in the current V.I.C.E. design a storage was built but not implemented. The node was built for under 500 USDs. Hopefully, it will be implemented in the future iterations of the V.I.C.E. cloud architecture. Currently, VMs within the V.I.C.E. architecture are built on each node beforehand and managed through the loadbalancer. The V.I.C.E. hardware can be seen in figure 3.2

![V.I.C.E. Hardware](image-url)

Figure 3.2: V.I.C.E. Hardware.
3.2 Adavantages

According to the authors of [Emerson, 2016] two of the main ways to increase power efficiency are through virtualization and cooling. Both of these aspects are addressed by the V.I.C.E. architecture. V.I.C.E. requires less cooling through the use of small machines, Raspberry Pi 2 thin clients, which consume less power. Virtualization is addressed through hosting VMs on compute nodes. Another way in which the V.I.C.E. contributes to cost efficiency is through the initial purchase and maintenance of lab equipment.

3.2.1 Initial Purchase of Equipment

Typically when a school or university buys equipment for a computer lab they buy new machines from a company such as Dell which has a deal with institute in question. Currently, the labs at Western Kentucky University are made up of Dell 745s. For the sake of comparison the assumption is made that a traditional lab is made up of new Dell 745. The lab would need 50 of these machines to accommodate the same amount of users as the proposed architecture in its current state. It is also important to note that both methods will require the same amount of monitors, keyboards, and computer mice. Therefore such peripherals well not be discussed. It is important to note that in a traditional lab the user to machine ratio increases linearly. Thus, one user would mean one Dell 745, ten users would require ten machines, and so on. Similarly, the V.I.C.E. architecture requires a Raspberry Pi for each user. However, there only has to be one of each node in the V.I.C.E. architecture. This means that only one of these expensive components has to be purchased. The rest of the lab is made up of thin clients were as in the traditional approach an expensive desktop has to be purchased for each user.
Table 3.1: Initial Cost of V.I.C.E. Architecture.

<table>
<thead>
<tr>
<th>V.I.C.E. Hardware</th>
<th>Cost (in USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 Raspberry Pis</td>
<td>4,000</td>
</tr>
<tr>
<td>com1</td>
<td>224</td>
</tr>
<tr>
<td>com2</td>
<td>224</td>
</tr>
<tr>
<td>com3</td>
<td>224</td>
</tr>
<tr>
<td>com4</td>
<td>660</td>
</tr>
<tr>
<td>com5</td>
<td>660</td>
</tr>
<tr>
<td>head</td>
<td>430</td>
</tr>
<tr>
<td>Total</td>
<td>6,422</td>
</tr>
</tbody>
</table>

Table 3.2: Initial Cost of Traditional Lab Architecture.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Cost (in USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 Dell 745s</td>
<td>24,000</td>
</tr>
<tr>
<td>Total</td>
<td>24,000</td>
</tr>
</tbody>
</table>

According to [Amazon.com, 2016], a Dell Optiplex 745 is 480 USD. When that cost is multiplied by 50 the cost for the traditional lab becomes 24,000 USD. It can be concluded that for a traditional lab to match the V.I.C.E. architecture in terms of user support it would cost at least 24,000 USD. It is not uncommon for the machines in such labs to be replaced once every four to six years. Meaning that this would be a reoccurring cost.

However, on the other hand, the V.I.C.E. architecture costs significantly less. The cost for the head node on Newegg [Newegg.com, 2016] with a case came to 430 USD. Com4 and com5 came to 660 USD each with cases included. A high-end Raspberry pi 3 with accessories is sold for 80 USD. In order to support 50 users at a time, 50 Raspberry Pis would need to be purchased. Com1, com2, and com3 can all be purchased for 224 USD each. When added up the total cost of the V.I.C.E. architecture hardware comes to 6,422 USD. As indicated in table 3.1 and table 3.2, the V.I.C.E. architecture costs barely over a fourth of the price to setup a traditional lab.
3.2.2 Costs to Maintain

Costs are not only incurred from initial setup, but also from maintaining the lab. As previously mentioned, the machines used for either approach would need to be replaced every four to six years. This means that the costs discussed in the section above would reoccur every four to six years. For a traditional lab replacing all the equipment would cost 24,000 dollars where as replacing the V.I.C.E. would only cost 6,422 USD. The equipment would have to be replaced at least every six years assuming there are no failures before then. Cooling would also cost more for a traditional lab because the Optiplex 745s produce more heat than the Raspberry Pis. This is due to the fact that Raspberry Pis consume less energy.

3.2.3 Power Consumption

Initial setup is not the only cost benefit offered by the V.I.C.E. architecture. It’s power consumption is significantly less when compared to that of a traditional lab. To prove this a Dell Optiplex 745, the V.I.C.E. head node, com1, com4, and a Raspberry Pi were all connected to a Watt meter for a day. Over the course of the day the Watt meter recorded several types of information including the number of Watts used at each moment data was recorded, the average kiloWatt hour(kWh) at each moment, and the accumulated cost based on a 0.12 USD per kWh model.

According to the University of North Carolina the formal definition of a Watt is "the SI unit of power" and power is "the rate at which work is done, or (equivalently) the rate at which energy is expended." They go on to say that a single watt is equal to a power rate of one joule of work per second of time. The University of North Carolina defines a kiloWatt
hour as "a commercial unit of electric energy. One kiloWatt hour represents the amount of energy delivered at a rate of 1000 watts over a period of one hour." [UNC, 2016] Most electric companies charge by kiloWatt hour.

The Watt meter indicated that the Dell computers consumed more Watts than any of the V.I.C.E. hardware. As seen in figure 3.3, the Dell 745 requires far more Watts than the others. A traditional lab that could match the V.I.C.E. architecture would be made up of at least 50 of these computers. However, the V.I.C.E. architecture itself would require only one of each node and 50 Raspberry Pis which require very low amount of energy. This also means that the Dell machines also use more kiloWatt hours over time. This can be seen in figure 3.4.

![Daily Watt Consumption](image)

Figure 3.3: Daily Watt Consumption.
As stated earlier, most electric companies charge by kiloWatt hours. The Watt meter used in this experiment was set to calculate the daily cost of each machine on a scale of the national average of 0.12 USD per kiloWatt hour. Table 3.3 shows the daily cost for each type of hardware. It is important to note that in a traditional lab there would be 50 Dell 745s, meaning this value would be multiplied by 50. The V.I.C.E. architecture would be totaled by adding the cost for each of the five nodes to the that of the head node and 50 Raspberry Pis. However, this cost pales in comparison to that of the traditional lab. Table 3.4 shows the total daily cost for each architecture. As indicated, the traditional approach costs almost 16 times more money daily. To further put this into perspective, table 3.5 shows how much each approach would cost if ran all 365 days of a year. Since most lab equipment has a four to six year lifespan, Tables 3.6 and 3.7 show the accumulated costs over four and six years respectively.
Table 3.3: Daily Cost of Hardware.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Cost(in USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dell 745</td>
<td>0.16</td>
</tr>
<tr>
<td>Raspberry Pis</td>
<td>0.002</td>
</tr>
<tr>
<td>com1/com2/com3</td>
<td>0.044</td>
</tr>
<tr>
<td>com4/com5</td>
<td>0.091</td>
</tr>
<tr>
<td>Head Node</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Table 3.4: Daily Cost of Each Lab.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Cost(in USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional Lab</td>
<td>8.00</td>
</tr>
<tr>
<td>V.I.C.E. Architecture</td>
<td>0.514</td>
</tr>
</tbody>
</table>

The traditional approach is more expensive even without taking into account the amount of energy it would take to cool such a lab. However, it can be deduced that since the hardware for the traditional lab consumes more Watts than the V.I.C.E. architecture it also generates more heat. More heat means more cooling is required. Therefore the traditional architecture would be more expensive in that respect as well.

3.3 Disadvantages

However, V.I.C.E. is not without its faults. It relies entirely on a network connection. In fact this is a core characteristic of cloud architectures. A network failure would render the V.I.C.E. architecture totally useless until the network was repaired. However, in an educational setting it is arguable how useful any machine would be without a network connection. It would depend entirely on the intentions of the user.

Table 3.5: Yearly Cost of Each Lab.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Cost(in USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional Lab</td>
<td>2,920.00</td>
</tr>
<tr>
<td>V.I.C.E. Architecture</td>
<td>187.61</td>
</tr>
</tbody>
</table>
Another disadvantage is that if a compute node failed, the number of VMs on that node would be unavailable. For example, if *com1* failed, the six VMs it hosts would be down as well. This means six less users could use the resources. However, in a traditional lab one machine supplies only one user with resources. Thus, if a machine fails in that scenario only one user is affected. However, the cost of maintaining the V.I.C.E. architecture is so much cheaper that it more than compensates for this minor inconvenience.

Despite these draw backs an excellent case can be made for the benefits of the V.I.C.E. architecture. The V.I.C.E. architecture costs less to build initially and costs less monthly to maintain. However, it is limited to requiring network access. This is a small price to pay for the energy efficiency that the architecture offers. The next chapter will describe the middleware that manages the V.I.C.E. architecture.
Chapter 4

V.I.C.E. MIDDLEWARE

The purpose of this chapter is to explore the design of the middleware used to manage the V.I.C.E. cloud architecture. In the first section, this chapter will discuss the purpose of a middleware and the various approaches to virtual machine load balancing. The following section will explore the importance of scalability and its execution within the proposed middleware. In section 3, focus will shift to the middleware itself. The topics will include a brief overview and the setup of the hardware and related software that is required to build the middleware. The server program which was installed on each each compute node will then be discussed. Following that, the primary functions within the head node’s client program will be discussed in detail. Section 4 will outline the functions available to the cloud administrator.

4.1 Load Balancing

The authors of [Katyal, 2014] state that the two main tasks of load balancing are resource allocation and scheduling in a distributed environment. The authors then compare several load balancing algorithms. They conclude that a static load balancing scheme is the easiest to simulate and monitor but due to it requiring prior knowledge about the resources, all the resources must be homogeneous. This means that such a structure couldn’t scale to the heterogeneous model of the cloud. Thus, the authors of [Katyal, 2014] conclude that dynamic load balancing techniques in distributed or hierarchical environment provide
better performance. They also state that performance of the cloud computing environment can be further maximized if dependencies between tasks are modeled using workflows.

[Kansal, 2012] states that there are five metrics of load balancing. The authors say that most current load balancing techniques adequately focus on performance, response time, scalability, throughput, resource utilization, fault tolerance, migration time and associated overhead. However, the authors of [Kansal, 2012] and the authors of [Megharaj, 2013] both feel that energy consumption and carbon emission should also be considered. These authors claim that this is necessary in a continued pursuit to develop green load balancing methods.

[S, 2013] discusses various load balancing algorithms before suggesting one of their own. The authors discuss using a greedy algorithm for load balancing. They say that while the greedy algorithm is simple and requires little processing, it leads to poor resource utilization. This is due to the fact that it uses the first available resources it comes across which means that all the resources are used on one node at a time. The authors then discussed the Round Robin, a method in which nodes are selected in a predetermined order. According to the authors the advantages to this method are that it utilizes all the resources in a balanced order and that virtual machines (VMs) are evenly distributed across nodes. The authors propose the use of a weighted Round Robin algorithm. This proposed algorithm uses a weighted table to help select a node. The idea behind this method seems to be similar yet slightly different to the load balancing method discussed in section 3.

4.2 Scalability

Scalability is an important characteristic within cloud computing architectures. In fact, one of the main ideas of cloud computing is having resources to provision on-demand.
Thus, it becomes necessary to be able to expand or reduce the number of machines within a particular architecture. This means that a middleware also has to be able to adjust to a change in the number of compute nodes within an architecture.

While there are a multitude of ways to accomplish this, the use of a database was the best solution and is the approach used within this middleware. If the need arises for more machines, an administrator using this architecture and middleware could simply connect the machine into the router and get an IP address by using DHCP. It does not matter what size memory the machine has or how many CPU cores it runs as long as it supports resource virtualization. The administrator just needs to decide how many virtual machines (VMs) that the machine will allow to be launched per each CPU core. The number that the administrator selects cannot be greater than 8 VMs per core due to a limitation within KVM itself. This number along with the new node’s name and IP are then simply added to a table in a database. This allows the middleware to access the information it requires. A more detailed discussion of the database will follow.

4.3 Middleware

4.3.1 Overview

The middleware discussed in this chapter was developed in Python and uses sockets for communication. Python was chosen for the middleware’s development due to how easily sockets can be implemented. Python also offers the ability to use a list of lists. This is important because the middleware uses a list to hold dynamic information regarding each compute node. The ability to easily parse strings that are returned from commands passed to the compute nodes also aides in the load balancing process. Comparisons are also easy to do using these Python lists. Python has yet another benefit, it allows more than one
value of varying types to be returned from methods which made it the ideal language for
developing the middleware.

The middleware is comprised of two parts. The first is the viceclient.py which
accesses a database to get a list of compute nodes. It uses the list of nodes to poll each one
for information by passing a series of commands through the socket. It then uses a load
balancing algorithm to select a virtual machine to launch and a node to launch it on. If
there is a free VM that is started, viceclient.py prints out the IP associated with the VM. If
all nodes are full, a message is printed instead. The viceclient.py is executed on the head
node.

The second half of the middleware is called viceserver.py. It runs on the compute
nodes and listens for commands from viceclient.py. The compute nodes are always running
the viceserver.py program. Thus, they are always listening for the client-side’s commands.
When the viceserver.py receives a command it passes back the results to the viceclient.py.
These two programs will be explained in more depth later.

4.3.2 Architecture

Before we can discuss the load balancing software we must first describe the archi-
tecture upon which it was deployed. Our architecture is made up of a head node and five
heterogeneous compute nodes. The head node has an Intel Core i3 dual core processor and
8GBs of RAM. The two larger compute nodes have Intel Core i5 quad core processors and
16GBs of RAM. The last three compute nodes have an Intel Celeron dual core processor
and varying amounts of memory. Ubuntu 14.04 LTS Server was installed on each machine.
The networking is handled by a gigabit LAN using CAT6 copper ethernet cables.
4.3.3 Compute Nodes

The compute nodes serve as the machines that launch and host the virtual machines. It is the resources found on these nodes that the virtual machines utilize. Therefore, they need to have virtualization extensions enabled on their CPUs. Compute nodes also need to be able to receive commands from the head node. To do this, the middleware requires that the python script entitled *viceserver.py* which causes each compute node to listen for commands from the *viceclient.py* program executed from the head node. However, to adequately use the commands that are required by the middleware some software had to be set up first.

4.3.3.1 Setup

The setup process was the same across all compute nodes. After the hardware itself was hooked into the network, Python3, vm-builder [VMBuilder, 2015], libvirt and kvm [KVM/Virsh, 2015] were installed. Yet another tool, arp-scan, was downloaded using the command `sudo apt-get install arp-scan` [Arp-Scan, 2015]. This tool will later be used by the *viceclient.py* program to get the IP addresses assigned to VMs. However, to be useful the sudoers on each compute node had to be given passwordless privileges. The line `ALL=(ALL) NOPASSWD:ALL` is placed in each node’s sudoers file. The command to start the *viceserver.py* was added to the `rc.local` file on each compute node.

4.3.3.2 Networking

At this point any virtual machines created on these nodes will only be accessible on the local node. The virtual machines need to be able to access the router and be assigned an IP through DHCP. Thus it is necessary to create a virtual bridge and define a new network
setting for the virtual machines. The guides provided by [Bridge, 2015] provided instructions to add a network bridge to the `/etc/network/interfaces` file. With the new modifications the file looked like figure 4.1. Next, the resources of [Libvirt, 2015] where used to create a XML file for a new default network. The contents of the file can be seen in figure 4.2. Then the commands to create a new network for virtual machines were issued. They were `virsh net-destroy default`, `virsh net-undefine default`, `virsh net-define network.xml`, `virsh net-start default`, and `virsh net-autostart default`. The first two commands stopped the network that would be utilized the virtual machines and undefined the previous network settings. The next command defined new network settings for the VMs using the `network.xml` file. The final two commands started the network defined by the previous command and switched on its auto-start option.

```bash
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto p2p1
iface p2p1 inet dhcp

# The Bridge
auto br0
iface br0 inet dhcp
bridge_ports p2p1
bridge_fd 0
```

Figure 4.1: `/etc/network/interfaces` File.

```xml
<network>
  <name>default</name>
  <forward mode="bridge"/>
  <bridge name="br0"/>
</network>
```

Figure 4.2: `network.xml` File.
4.3.3.3 Virtual Machine Creation

After the network has been setup and arp-scan has been installed, Virtual Machines are defined on each compute node using a variation of the commands found at [VMBuilder, 2015] and [Petersson, 2015]. This means the middleware does not have to wait to define the VM and that it can simply start it. At this point the process changes slightly between each compute node. This is due to the fact that the limit of virtual machines per core on each node was decided before hand and the amount of cores vary between machine. However, each node had the vmbuilder command ran on it once for every VM the node hosts. Each VM has a unique name starting at vm1 and ending at vm50 with a directory of the same name. This naming scheme allowed each VM and directory to be uniquely named across the entirety of the cloud. The command was slightly modified each time. Figure 4.3 shows the command used. It was modified on "vmname" and "dir" to reflect the VM’s name and the directory’s name. Once all the VMs had been defined the viceserver.py file was moved onto each node and started.

```bash
sudo vmbuilder kvm ubuntu --suite trusty --flavour virtual \
--addpkg=linux-image-generic --addpkg=unattended-upgrades \
--addpkg=openssh-server --addpkg=acpid --arch amd64 \
--libvirt qemu:///system --user ubuntu --name ubuntu \
--hostname=vmname --dest dir --addpkg=python3 --mem 512 \
--addpkg=openjdk-7-jdk
```

Figure 4.3: VMBuilder Command.

4.3.3.4 viceserver.py

The purpose of the viceserver.py program is to open a Python socket up to listen for commands from the viceclient.py program. When commands are received by the viceserver.py program the results are passed back to the viceclient.py socket and the server
simply resumes waiting. After the program is launched on each compute node, they are simply left in their listening state indefinitely. Algorithm 1 shows the pseudocode of `viceserver.py` and the code can be seen in Appendix figure B.1.

### Algorithm 1 viceserver.py

```python
port ← 12345
s ← socket
s·bind(host, port)
s·listen(5)

while True do
    conn, addr ← ·accept()
    data ← cmdline(conn·rcv(1024), decode())
    conn·send(data)
    conn·close()
```

4.3.4 Head Node

The head node serves to manage compute nodes by communication through the middleware. It issues commands to the compute nodes and launches virtual machines on the compute node that is best suited to host it. However, it does not host directly host any virtual machines. The head node also stores the database containing the information related to the compute nodes and hosts the web server that is used to create the cloud architecture’s user interface. After the `viceserver.py` program is waiting for commands on all the compute nodes, the head node can issue commands from the `viceclient.py`. However, to maintain scalability, the head node also requires some setup.

4.3.4.1 Head Node Setup

The head node also requires that Python3 be installed. In addition the head node needs a way to manage a database using Python code. Thus, `Psycopg` was installed using the `sudo apt-get install python3-psycopg2`. `Psycopg` is a PostgreSQL adapter for the Python programming language [Psycopg, 2015]. After this installation was complete, `Psycopg`
could be imported into our Python program allowing it to access the *vice* database and run queries on it.

### 4.3.4.2 Head Node Database

In order to keep the load balancing algorithm dynamic and scalable, the *vice-client.py* program was designed to read information from a database. Thus, PostgreSQL was installed onto the head node. Using the SQL commands found at [PostgreSQL, 2015], a database called *vice* was created. Within the *vice* database a table called *nodes* was created. The nodes *table* consisted of three attributes, the node name, the node IP, and the number of VMs allotted per each core(npc). The last of which cannot exceed 8 due to KVM limitations. Each compute node was entered into this table with its own related information. Figure 4.4 shows the result of a select statement ran on the *nodes* table. Whenever a new node is added all an administrator would have to do is set it up as discussed previously and add its information into the *nodes* table.

In addition to the *nodes* table, the *vice* database also has a table named after each individual compute node. These tables consist of the names and MAC addresses associated with the VMs on that compute node. After each VM was created, the command `sudo virsh dumpxml vmname | grep 'mac address'` where vmname was actually the VM’s name was ran on the host compute node. This command retrieved the MAC address from the XML file. These were the values used to fill the database tables. An example of a select statement from such a table can be found in figure 4.5. This allows for resource expansion within previously utilized hardware and since the MAC addresses assigned to the VMs are static they can be used with *arp-scan* to find the machine’s IP address.
Now that the database is created, `viceclient.py` can be ran from the head node. `Vice-client.py` can be seen in Appendix figure B.2. The first thing the program does is call the `loadBalance` method which can be seen in algorithm 2. The first thing it does is call the `poll` method to get information about each node. The purpose of `poll` is to build a list of lists which contains information regarding each node. It can be seen in algorithm 3. The `poll` method first calls `sqlops`. It passes the string "nodes" to the function call. This allows the `sqlops` method access to the `nodes` table where it builds a list of tuples containing each entry regarding the information for each node entered into the `nodes` table. The `sqlops` method can be seen in figure 4. The (sqlops) returns a list containing the node name, IP address, and number of VMs per core (npc). The (poll) method then uses the IPs in this returned list to expand upon the information regarding each node.

The `poll` method next calls `free`, `lscpu`, and `virshCount`. It uses these methods to add
Algorithm 2 Method: `loadBalance()`

```plaintext
glist ← poll()
for (i=0, len(glist), 1) do
    for (j=0, len(glist), 1) do
        if glist[i][9]<glist[j][9] then
            sglist[i], glist[j] ← glist[j], glist[i]
        if glist[i][9]==glist[j][9] and glist[i][5]>glist[j][5] then
            sglist[i], glist[j] ← glist[j], glist[i]
    if glist[0][9]==1 then
        print("allresourcesinuse")
    else
        vmList ← sqlops(glist[0][0])
        rL ← virshList(glist[0])
        vm ← vmList[0][0] · strip()
        mac ← vmList[0][1] · strip()
        for (i=0, len(vmList), 1) do
            mac ← False
            for (j=2, len(rL)-1, 1) do
                if vmList[i][0].strip()==rL[j][1] then
                    flag ← True
                    break
            if flag==False then
                vm ← vmList[i][0] · strip()
                mac ← vmList[i][1] · strip()
                break
        virshStart(glist[0][1], vm)
        print(vm)
        flag2 ← True
        while flag2 do
            time · sleep(2)
            arpL ← arpScan(gList[0])
            for (i=2, len(arpL)-3, 1) do
                if mac==arpL[i][1] then
                    print(arp[i][0])
                    flag2 ← False
                    break
```

onto to the inner lists that represent each individual node within the larger list. These three functions are extremely similar except for the location of the information added to the list.

The pseudocode for the `free` method can be seen in algorithm 5 and the `lscpu` method can
Algorithm 3 Method: poll()

\[
\text{method: poll } \\
\text{list } \leftarrow \text{sqllops("nodes")}
\]

for (i=0, len(vmList), 1) do
  \[
  \text{port } \leftarrow 12345 \\
  \text{total, used, fm } \leftarrow \text{free(port, lst[i])} \\
  \text{cores, vms } \leftarrow \text{lscpu(port, lst[i])} \\
  \text{VMcount } \leftarrow \text{virshCount(port, lst[i])} \\
  \text{node } \leftarrow \text{list(lst[i])} \\
  \text{node } \cdot \text{append(total, used, fm, cores, vms, VMcount, (VMcount/vms))} \\
  \text{lst[i] = node}
\]

return lst

Algorithm 4 Method: sqllops(s)

\[
\text{connection } \leftarrow \text{psycopg.connect(database, user, password, host, port)} \\
\text{cursor } \leftarrow \text{connection.cursor()} \\
\text{cursor.execute("select * from" + s)} \\
\text{data } \leftarrow \text{cursor.fetchall()} \\
\text{return data}
\]

be seen in algorithm 6. The \textit{free} method returns the amount of total memory, used memory, and free memory. The \textit{lscpu} method returns the number of cores and the max number of virtual machines. The latter is the number of cores multiplied by the npc entered into the \textit{nodes} table. Finally the \textit{virshCount} method returns the number of virtual machines listed as running. \textit{VirshCount} is only slightly different then the other methods mentioned and can be seen in algorithm 7. The \textit{poll} method adds these items to the list at each index along with a usage percentage in decimal form determined by the number of running VMs divided by the maximum VM total. \textit{Poll} then returns a list in which each index is a list of information regarding one compute node. For example, index zero of the returned list would refer to a list containing the first compute node’s name, IP, npc, total memory, used memory, free memory, maximum number of VMs, number of VMs running, and usage percentage in decimal form.

Now that the \textit{loadBalance} method has all of this information, it can start comparing
Algorithm 5 Method: free(port, lst)

\[
\text{host} \leftarrow \text{lst}[1]
\]
\[
\text{s} \leftarrow \text{socket}
\]
\[
\text{s} \cdot \text{connect(} \text{host}, \text{port)}
\]
\[
\text{data} \leftarrow \text{"free"}
\]
\[
\text{s} \cdot \text{send(data)}
\]
\[
\text{retdata} \leftarrow \text{s} \cdot \text{recv()}
\]
\[
\text{t} \leftarrow \text{retdata} \cdot \text{splitlines()}
\]
\[
\text{ft} \leftarrow \text{t}[1] \cdot \text{split()}
\]
\[
\text{return ft[1], ft[2], ft[3]}
\]

Algorithm 6 Method: lscpu(port, lst)

\[
\text{host} \leftarrow \text{lst}[1]
\]
\[
\text{s} \leftarrow \text{socket}
\]
\[
\text{s} \cdot \text{connect(} \text{host}, \text{port)}
\]
\[
\text{data} \leftarrow \text{"lscpu"}
\]
\[
\text{s} \cdot \text{send(data)}
\]
\[
\text{retdata} \leftarrow \text{s} \cdot \text{recv()}
\]
\[
\text{t} \leftarrow \text{retdata} \cdot \text{splitlines()}
\]
\[
\text{ct} \leftarrow \text{t}[3] \cdot \text{split()}
\]
\[
\text{maxvm} \leftarrow \text{lst}[2] \ast \text{ct}[1]
\]
\[
\text{return ct[1], maxvm}
\]

Algorithm 7 Method: virshCount(port, lst)

\[
\text{host} \leftarrow \text{lst}[1]
\]
\[
\text{s} \leftarrow \text{socket}
\]
\[
\text{s} \cdot \text{connect(} \text{host}, \text{port)}
\]
\[
\text{data} \leftarrow \text{"virshlist"}
\]
\[
\text{s} \cdot \text{send(data)}
\]
\[
\text{retdata} \leftarrow \text{s} \cdot \text{recv()}
\]
\[
\text{t} \leftarrow \text{retdata} \cdot \text{splitlines()}
\]
\[
\text{VMCount} \leftarrow \text{len(t)} - 3
\]
\[
\text{VMCount}
\]

the nodes based on their usage percentage. The method uses a simple sorting pattern and sorts the lists on their usage percentage found at the 9th index. If the usage percentage is the same for two nodes the method looks at the 5th index where free memory is located and gives priority to the node with the most free RAM. After the nodes are sorted, the node that is best suited to launch a VM is located in position 0 of the main list. This node’s usage
percentage is then compared against 1. If the percentage is equal to 1 that means all the virtual machines are in use and a message is printed to the user.

However, if the node is not full, the loadBalance calls the sglps method, seen in algorithm 4 with the name of the selected node. This creates a list of virtual machines and MAC addresses from the database table with the same name as the node. The loadBalance method then calls virshList with the list related to the selected node. The virshList method can be seen in algorithm 8. This method creates a list of currently running VMs. A for loop then uses a flag to compare the two new lists. Whenever a VM is found in the list from the database but not in the list from the virshList that VM is selected to be started. The loops are broken and the virshStart method is called with the node IP and the VM name as arguments. The virshStart method is called with the node IP and the VM name as arguments. The virshStart method simply passes the command to start the VM it is given.

The virshStart method can be seen in algorithm 9.

---

**Algorithm 8 Method: virshList(lst)**

```plaintext
port ← 12345
host ← lst[1]
s ← socket
s·connect(host, port)
data ← "virshlist"
s·send(data)
retdata ← s·recv()
t ← retdata·splitlines()
for i=0, len(t)-1, 1 do
t[i] ← t[i]·split()
return t
```

**Algorithm 9 Method: virshStart(h, d)**

```plaintext
port ← 12345
s ← socket
s·connect(h, port)
data ← "virshstart" + d
s·send(data)
```
After the start command has been sent, the `loadBalance` sets a new flag to the boolean value True. It then enters a while loop for as long as this flag remains True. Inside the while loop the `loadBalance` method waits two seconds before calling the `arpScan` method. The delay is to give the virsh start command time to run. The `arpScan` method is passed the list containing the node information and uses it to send the `arp-scan` command to that node. A list containing all the IPs that are in use and the MACs that are using them is returned. *ArpScan* can be seen in algorithm 10. This list of MACs is then compared against the MAC associated with the VM that was launched. If the MAC is not found in the list, the VM does not have an IP yet and the while loop iterates once more. If the MAC is found the while loop is broken and the IP is printed. The user can now SSH into the VM using the given IP.

**Algorithm 10** Method: `arpScan(lst)`

\[
\begin{align*}
\text{port} & \leftarrow 12345 \\
\text{host} & \leftarrow \text{lst}[1] \\
\text{s} & \leftarrow \text{socket} \\
\text{s} & \cdot \text{connect(host, port)} \\
\text{data} & \leftarrow "\text{sudoarp --scan --interface = br0 --localnet}" \\
\text{s} & \cdot \text{send(data)} \\
\text{retdata} & \leftarrow \text{s} \cdot \text{recv}() \\
\text{t} & \leftarrow \text{retdata} \cdot \text{splitlines()} \\
\text{for} (i=0, \text{len(t)-1}, 1) & \text{do} \\
\quad & t[i] \leftarrow t[i] \cdot \text{split()} \\
\text{return} t
\end{align*}
\]

4.4 Administrator Functions

Due to the fact that the cloud architecture will support many users, it will need to be overseen by an administrative user. *Viceclient.py* contains several functions that are called through the user interface discussed in the next chapter. These functions handle the various tasks that will be required by the administrator. The first of which is viewing the
information pertaining to a specific node. A method known as `adminPoll` takes in the string containing the name of a specific node and compares it to the list created from the `poll` method discussed earlier. The method then compares the string that was passed in to the node names in the list. When a match is found the information is printed out. It then uses the method `adminVirsh` to pass the command that prints all the virtual machines and their current state on that node regardless if they have been started or not. This allows the admin to see the information for the node and what VMs are available or in use on that machine.

The `adminPoll` and `adminVirsh` methods can be seen in algorithms 11 and 12 respectively.

**Algorithm 11** Method: \textit{adminPoll}(s)

\begin{verbatim}
Lst ← poll()
for (i=0, len(Lst), 1) do
    if s==Lst[i][0] then
        print(Lst[i][0])
        print(Lst[i][1])
        print(Lst[i][3])
        print(Lst[i][4])
        print(Lst[i][5])
        print(Lst[i][6])
        print(Lst[i][7])
        print(Lst[i][8])
        print(Lst[i][9])
        adminVirsh(Lst[i])
\end{verbatim}

**Algorithm 12** Method: \textit{adminVirsh}(Lst)

\begin{verbatim}
port ← 12345
host ← Lst[1]
s ← socket
s·connect(host, port)
data ← "virshlist --all"
s·send(data)
retdata ← s·recv()
t ← retdata·splitlines()
for (i=0, len(t), 1) do
    print(t[i])
\end{verbatim}
The admin also has the ability to start a VM through *loadBalance* and stop any VM at any time. A method called *virshDestroy* uses the user interface to select a node’s IP and take in a VM name from the admin to send a destroy command to the selected node. The command stops the running VM. This allows the admin the freedom to free up previously allocate resources. The method *virshDestroy* can be seen in algorithm 13.

**Algorithm 13 Method: virshDestroy(h, d)**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>port ← 12345</code></td>
</tr>
<tr>
<td>2</td>
<td><code>s ← socket</code></td>
</tr>
<tr>
<td>3</td>
<td><code>s · connect(h, port)</code></td>
</tr>
<tr>
<td>4</td>
<td><code>data ← &quot;virshdestroy&quot; + d</code></td>
</tr>
<tr>
<td>5</td>
<td><code>s · send(data)</code></td>
</tr>
</tbody>
</table>

### 4.5 Interface Interaction

The middleware only interacts with the web interface through the *viceclient.py*. The program *viceclient.py* contains `eval(sys.argv[1])` which allows the interface to call its functions directly as if from the command line. More information on how the middleware and user interface interact can be found in the following chapter.
Chapter 5

V.I.C.E. INTERFACE

The V.I.C.E. user interface was designed to make usage simple for the users. The web interface was written in CSS, HTML, and PHP using bootstrap [Bootstrap, 2016]. It is hosted using apache [Apache, 2016]. The interface uses the Gate One web terminal emulator to SSH into VMs [One, 2016]. This chapter will delve into the details behind the V.I.C.E. architecture’s web interface. It will begin with a section on preparing the head node, *chead*, to host the necessary services. The following sections will outline the code used to make the interface and how it works. Finally, a rundown of the functions provided by the interface and instructions on using them will be provided.

5.1 Setup

The first step to build the interface is to prepare the head node to host the web interface. Apache 2 and PHP 5 were installed on the head node using the commands `sudo apt-get install apache2` [Apache, 2016] and `sudo apt-get install PHP5` respectively. These provided a HTTP server on which to host the web interface and the ability to use PHP. Gate One, a terminal emulator and SSH client that the interface starts for the user, was installed next. The terminal emulator allows the web interface to open up a terminal and connect to a VM for the user. This installation was done by following the Gate One documentation [One, 2016]. First, *git* was installed using the command `sudo apt-get install git`. This allowed the use of the command `git clone https://github.com/liftoff/GateOne.git`
which installs Gate One. Next, pip for python2 was installed using the `sudo apt-get install python-pip` and `sudo ./setup.py install` commands. Finally, Gate One was started using the `/run_gateone.py` from the the `GateOne/` directory. This command was added to the `rc.local` file so that Gate One would be ran automatically when the cloud head node is started.

Another function required by the interface was the ability to authenticate users and their roles. A table was added to the previously mentioned PostgreSQL database, `vice`. This new table called `ids` contains three columns. The columns `name`, `password`, and `type` contain a user’s username, password, and role respectively. `Type` should only ever be set to "student" or "admin". If this is not the case the user will not be able to log in. This table is used by a program, `conn.php`, to compare the information entered by the user to the credentials in the database. This will be explained further later. New users will need to be added to this table in order to gain access to the web interface. An example of the table can be seen in figure 5.1.

```
<table>
<thead>
<tr>
<th>name</th>
<th>password</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin</td>
<td>admin</td>
<td>admin</td>
</tr>
<tr>
<td>Student</td>
<td>student</td>
<td>student</td>
</tr>
<tr>
<td>Travis</td>
<td>vice</td>
<td>admin</td>
</tr>
<tr>
<td>chris</td>
<td>iscool</td>
<td>admin</td>
</tr>
</tbody>
</table>
```

Figure 5.1: Results from `Select * from ids;`.

### 5.2 File Overview

#### 5.2.1 main.css

After the environment is setup, the web interface was developed. The code is maintained in multiple files. The most basic of which is the `main.css` file. It’s purpose is to control the design of all the webpages. `Main.css`, like much of the code used for the in-
terface, was build from a bootstrap [Bootstrap, 2016] template. The code can be found in Appendix figure C.1.

5.2.2 footer.php

*Footer.php* is also included by all the webpages. It is completely based on a boot-
strap design [Bootstrap, 2016]. The inclusion of *footer.php* causes a footer containing text
to be displayed at the bottom of each webpage. *footer.php* can be seen in Appendix figure
C.3

5.2.3 navbar.php

Another file that is included on all the web pages is the *navbar.php*. This creates a header that is found at the top of the actual web pages and controls login authentication. *Navbar.php* includes and uses *conn.php* to check to see if a user is valid when a username
and password is entered. *Navbar.php* can be seen in Appendix figure C.4.

5.2.4 conn.php

*Conn.php* connects to the database and compares what was entered to the data from the *ids* table. If it is successful the user is logged in and shown a web page depending on their whether they are an "admin" or "student" user. *Conn.php* can be seen in Appendix figure C.2

5.2.5 logout.php

If the values of username and password are successfully set *navbar* displays a "logout" button and includes *logout.php*. If the "Logout" button is clicked *logout.php* ter-
minates the session. *logout.php* can be seen in Appendix figure C.9.
5.2.6 index.php

Index.php is the first webpage a user sees after navigating to the V.I.C.E. interface. This file is also built using a bootstrap template. It serves as the homepage and incorporates the navbar.php. From this page the user can use navbar.php here to log in. Depending on if the credentials are valid or not and whether the user is a "student" or an "admin" They are taken to a different version of the landing.php. The index.php can be seen in Appendix figure C.5.

5.2.7 landing.php

The page that contains all the functions is called landing.php. This page is the one that interacts with the middleware directly. It first checks to see if any functions have been selected by evaluating the values of variables which are assigned when a user tries to do something on the landing.php page. If an "admin" user has used the poll function by selecting a node name from a drop down menu and hitting the "poll" button, the adminPoll will be called from viceclient.py. When an "admin" enters a node IP and VM name into the text boxes and clicks the "Stop VM" landing.php will call viceclient.py's virshDestroy function. If either a "student" or "admin" clicks on the "Start VM" button, the loadBalance function is called. After the the VM is started, landing.php connects to it through Gate One and returns a button for the user to access it. This code can be seen in Appendix figure C.6.

If a user is an "admin" landing.php loads with three options. These are to poll a node through a drop down list, to stop a VM, or to start a VM. If any of these options are utilized the page reloads displaying the results if there are any. The "student" user can only start VMs. This works the same for both "admins" and "students". If a user fails to log
in, landing.php displays a message informing the user they failed to log in. The code for the "admin" page can be seen in Appendix figure C.7. The other two cases can be seen in Appendix figure C.8. The next section will describe how users can utilize each function.

5.3 Utilization

5.3.1 Logging In

This section will contain detailed instructions on how to use the interface and functions. Pointing a web browser to the interface’s IP address, which is that of the head node, will take a user to the login page. This page can be seen if figure 5.2. Here users can enter their username and password. If the credentials cannot be verified against the ids table the attempted login will fail and the user will see the page shown in figure 5.3. Otherwise, the user will be shown a page based on whether they are an "admin" or "student" user.

Figure 5.2: Login Page.
5.3.2 Student Functions

Upon successful log in as a student, the user will directed to the page shown in figure 5.4. The student can only do one thing aside from log out and start a VM. This is accomplished by clicking the "Start VM" button.

5.3.2.1 Start VM

Upon clicking the "Start VM" button, the interface uses the \textit{loadBalance} function to launch a VM. A new page is displayed that contains a "Launch SSH" button and information regarding the launched VM as seen in figure 5.5. If the user clicks on this button the Gate One terminal emulator opens in a new tab. The terminal is displayed in figure 5.6.
After the user connects to the terminal and enters their giving password, they can program in the terminal and use the Raspberry Pi to use other features such as a web browser. Figure 5.7 shows the terminal after log in. A simple Python program that was written within the VM is shown in figure 5.8. Figure 5.9 shows the VM after the program has been ran. However, if all VMs are in use the user will be shown the screen depicted in figure 5.10.

5.3.3 Admin Functions

"Admin" users are directed to a different page. They have the ability to poll each compute node to see information regarding the node and the state of each VM hosted on it. The "admin" users can also stop any given VM. "Admin" users can also start VMs just as a "student" user does. Figure 5.11 shows the homepage that the "admin" user is shown.

5.3.3.1 Poll Individual Compute Nodes

An "admin" can poll a compute node for information regarding it and the status of the VMs hosted on it. This is done by selecting a compute node from the drop down and
clicking the "Run" button. When this is done landing.php will call adminPoll from the viceclient.py. The page will then be reloaded with the information about the selected and
a listed of VMs hosted on it. This list will indicate if a VM is running or not. Figure 5.12 shows the results of this action.
5.3.3.2 Stop Individual Virtual Machines

Another ability offered exclusively to "admin" users is to stop a specific VM on any node. This is done by entering the IP of the node that host the VM that is to be turned
off and the VM’s name into the text boxes and hitting the "Stop VM" button. The required information can be found through use of the polling function. This triggers `landing.php` to call the `virshDestroy` function from `viceclient.py` passing the entered values as parameters. This causes the page to reload. However, no additional information prints. A use of the poll function can reveal that a VM has been shut down.

5.3.3.3 Start Individual Virtual Machines

The "admin" user can start and connect to VMs just like a "student" user. The process is exactly the same as it is described above. Like a "student", the "admin" can only start a VM through use of the load balancer and connects to the VM through Gate One. This gives the users the functions to make use of the V.I.C.E. architecture’s full potential.
Chapter 6

FUTURE WORK

While the V.I.C.E. architecture could be used as it is, there are a few improvements that will be made to future versions. Future iterations will incorporate a storage node. This means that user's data will be saved unto a storage node rather than inside the VM. Users will be allowed to access their data regardless of which VM they launch allowing user specific data to be stored more securely.

Another improvement to be implemented is VM migration. The purpose of this is to give the V.I.C.E. architecture the ability to migrate the number of running VMs from one compute node to another. For example, if one compute node was running five VMs and another compute node was only running one, the compute node with one VM could stop its current VM and restart one on the other node. The end goal would be to have as many VMs running on as few compute nodes as possible allowing compute nodes to be shutdown and restarted as needed. Inactive compute nodes will be turned off making the V.I.C.E. architecture even more cost effective.
Chapter 7

CONCLUSION

The term "cloud computing" means different things to different people. However, it is a way to offer scalable resources on demand. This goes under utilized in education where the elasticity and scalability of the cloud could be beneficial in a multitude of ways. One such way is as an alternative to computer labs comprised of many different machines.

The objective of this research is to propose a cost and energy efficient alternative to traditional computer labs. The V.I.C.E. architecture uses a vertical cloud architecture and thin clients to provide users access to virtual machines. The cost to set up and maintain the V.I.C.E. architecture is cheaper then that to do the same for a traditional lab. It also consumes less power meaning it is cheaper to cool.

On top of this the V.I.C.E. architecture utilizes a middleware built for easy scalability and load balancing. New nodes can be added by merely connecting the machines into the network and adding information to a database. After that, the middleware does the rest and disburses active VMs across the nodes evenly.

These VMs can be started through a simplistic web interface. The interface was designed to be easy to access and use. It offers users all the functions necessary to use the architecture. The process of starting a VM automatically provides a SSH client for the user making it more accessible to user.

The V.I.C.E. will continue to improve over time. Future iterations will consist of a
storage node offering user persistent storage and more security. VM migration will also be offered in the future. This will allow compute nodes that are hosting a few VMs to migrate them to more populated compute nodes and then shut off. This means as few compute nodes as possible will be on at a time making V.I.C.E. even more cost effective.

In short, the V.I.C.E. architecture serves as a better alternative to a traditional computer lab. It is more cost effective than a traditional lab. This means that it would be affordable to educational institutes that do not have the funds for a traditional computer lab. The V.I.C.E. cloud architecture’s ease of use and low cost make it ideal for the educational setting.
REFERENCES


Foely, M. J. (2014, February). Microsoft’s azure cloud team moves toward blurring the iaas/paas lines.


One, G. (2016, June). Gate one documentation.


Appendices
Appendix A

APPENDIX A: SETTING UP THE ARCHITECTURE

A.A Preparing Compute Node

The following steps are used to set up the compute node. This assumes that virtual extensions are enabled on the machine.

1. Install Ubuntu 14.04 LTS
2. Setup hardware.
3. Install Python3 using `apt-get Python3`
4. Install vm-builder [VMBuilder, 2015]
5. Install libvirt and kvm[KVM/Virsh, 2015].
6. Download arp-scan using the command `sudo apt-get install arp-scan` [Arp-Scan, 2015].
7. Give passwordless privileges to sudoers by placing the line `ALL=(ALL) NOPASSWD:ALL` in each node’s sudoers file.
8. Put `viceserver.py` on the node.
9. Add command to launch `viceserver.py` to `rc.local` file.

A.B Setting Up The Network Files

The following steps are to build the network.

1. Vim into the `/etc/network/interfaces` file.
2. Change it to look like the file shown below.

```plaintext
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto p2p1
iface p2p1 inet dhcp

# The Bridge
auto br0
iface br0 inet dhcp
  bridge_ports p2p1
  bridge_fd 0
```

Figure A.1: `/etc/network/interfaces` File.

3. Create a XML file for a new default network. The contents of the file can be seen in figure A.2.
4. Then use the command `virsh net-destroy default` to stop the current network.
5. Use the command `virsh net-undefine default` to undefine the current network.
6. Use `virsh net-define network.xml` to define a new network settings for the VMs using the `network.xml` file.
7. Use `virsh net-start default`
8. Use `virsh net-autostart default` to make the network autostart.

### A.C Building VMs

Build VMs using a variation of the command below.

```
sudo vmbuilder kvm ubuntu --suite trusty --flavour virtual \ 
--addpkg=linux-image-generic --addpkg=unattended-upgrades \ 
--addpkg=openssh-server --addpkg=acpid --arch amd64 \ 
--libvirt qemu:///system --user ubuntu --name ubuntu \ 
--hostname=vmname --dest dir --addpkg=python3 --mem 512 \ 
--addpkg=openjdk-jdk \ 
```

### A.D Setting Up the Head Node

The following steps are used to setup the compute node. This assumes that virtual extensions are enabled on the machine.

1. Install Ubuntu 14.04 LTS
2. Setup hardware.
3. Install Python3 using `apt-get Python3`
4. Install PostgreSQL
5. Create the Database.
6. Install Psycopg using the `sudo apt-get install python3-psycopg2`.
7. Give passwordless privileges to sudoers by placing the line `ALL=(ALL) NOPASSWD:ALL` in each node’s sudoers file.
8. Put `viceclient.py` on the node.
9. Use `sudo apt-get install git` command. 10. Use `git clone https://github.com/liftoff/GateOne.git`.
11. Use `sudo apt-get install python-pip`
12. Use `sudo ./setup.py install`
13. Add `./gateone.py` to `rc.local` file.
14. Use `sudo apt-get install apache2`
15. Use `sudo apt-get install php5`
import socket
import time
from subprocess import PIPE, Popen

def main():
    host = ''  # Symbolic name meaning all available interfaces
    port = 12345  # Arbitrary non-privileged port
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((host, port))
    s.listen(5)

    while True:
        conn, addr = s.accept()
        print('Connected by', addr)
        data = cmdline(conn.recv(1024).decode())
        #if not data: break
        conn.send(data)
        conn.close()

def cmdline(c):
    process = Popen(  
        args=c,  
        stdout=PIPE,  
        shell=True)
    return process.communicate()[0]

main()
#Description: This function is used to cycle through the nodes to get the need information from the database and combining it with the information gathered by the system calls to make a master list.
#Input: None
#Output: List

def poll():
    lst=sqlops("nodes")#lst is filled with the info placed into the db table "nodes"
    for i in range(0, len(lst),1):#for loop goes through each element in the list which represents each row of a db.
        total,used,free=sqlops(port, lst[i])#info from free command placed in variables
        cores, vms=sqlops(port, lst[i])#same as above only from lscpu function
        VMCount=virshCount(port, lst[i])
        node=lst[lst[i]]
        node.append(int(total))#the info gathered from the previous two lines is added one by one to the node element in list
        node.append(int(used))
        node.append(int(free))
        node.append(int(cores))
        node.append(int(vms))
        node.append(int(VMCount))
        node.append(VMCount/vms)
        lst[i]=node
    return lst

#Description:Uses sockets to call the lscpu and gets the number of cores it then uses that and the info passed in from the database to calculate the max number of vms allotted for that machine. These values are returned.
#Input: String, List
#Output: String, Integer

def lscpu(port, lst):
    host=lst[1]#get host ip from passed in tuple
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)#open and connect with socket
    s.connect((host, port))
    data = "lscpu" #data takes the string of the command
    s.send(data.encode('ascii'))#sends and sends the command
    retdata = s.recv(9999).decode('ascii')#the returned info is saved into retdata
    retdata = retdata.splitlines()#retdata is split by lines
    ct=[3].split()#the fourth line is split between spaces
    maxvm=ct[1]#max vms is calculated by multiplying the number of cpu cores gotten by the command with the npc value passed through the tuple from the database
    s.close()#socket is closed
    return ct[1],maxvm#two values number of cpu cores and max amount of vms are return.

#Description:Uses sockets to call the free command and gets the amount of total, used, and free memory
#Input: String, List
#Output: 3 Strings

def free(port, lst):
    host=lst[1]#get host ip passed in through the tuple
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)#open and connect with socket
    s.connect((host, port))
    data = "free"#data recieves string "free" which will be the used command
    s.send(data.encode('ascii'))#The command is sent and encode
    retdata = s.recv(9999).decode('ascii')#the returned data is stored in retdata
    retdata = retdata.splitlines()#the returned data is split between lines
    ft=[1].split()#the second line is split along
    s.close()#socket is closed
    return ft[1],ft[2],ft[3]#values are returned they are the total,used, and free memory.

#Description:Takes in a string which is the name of the table and returns the data from that table.
#Input: String
#Output: List of Tuples

def sqlops(s):
    connection=psycopg2.connect(database="vice",user="postgres",password="vice", host= "127.0.0.1",port="5432")#creates a connection object
    cursor=connection.cursor()#creates a cursor
    cursor.execute("select * from "+table_name+";")#calls the execute command
Appendix C
APPENDIX C: INTERFACE
C.A main.css

```css
# cloud {
  font-size: 500px;
}

# footer {
  background: rgba(255, 255, 255, 1);
  background: -moz-linear-gradient(top, rgba(255, 255, 255, 1) 0%, rgba(246, 246, 246, 1) 47%, rgba(237, 237, 237, 1) 100%);
  background: -webkit-gradient(left top, left bottom, color-stop(0%, rgba(255, 255, 255, 1)), color-stop(47%, rgba(246, 246, 246, 1)), color-stop(100%, rgba(237, 237, 237, 1))) 100%;
  background: -webkit-linear-gradient(top, rgba(255, 255, 255, 1) 0%, rgba(246, 246, 246, 1) 47%, rgba(237, 237, 237, 1) 100%);
  background: -o-linear-gradient(top, rgba(255, 255, 255, 1) 0%, rgba(246, 246, 246, 1) 47%, rgba(237, 237, 237, 1) 100%);
  background: linear-gradient(top, rgba(255, 255, 255, 1) 0%, rgba(246, 246, 246, 1) 47%, rgba(237, 237, 237, 1) 100%);
  filter: progid:DXImageTransform.Microsoft.gradient( startColorstr=#ffffff , endColorstr=#ededed , GradientType=0 );
  box-shadow: 0px 1px 6px 0px rgba(0, 0, 0, 0.75);  
  box-shadow: -webkit-box-shadow: 0px 1px 6px 0px rgba(0, 0, 0, 0.75);  
  box-shadow: -moz-box-shadow: 0px 1px 6px 0px rgba(0, 0, 0, 0.75);  
  color: black;
}
```

Figure C.1: main.css.

C.B conn.php

```php
<?php
// Connecting, selecting database
$dbconn = pg_connect("host=localhost dbname=vice user=postgres password=vice")
or die('Could not connect: ' . pg_last_error());

if (!isset($_SESSION)){
  session_start();
}
?>
```

Figure C.2: conn.php.

C.C footer.php, navbar.php and index.php
C.D landing.php and logout.php

Figure C.3: footer.php.
<?php
include_once("login/conn.php");
if(isset($_SESSION['username'])){?


<nav class="navbar navbar-inverse" style="border-radius: 0 !important;">
    <div class="container">
        <div class="col-md-4"><a href="landing.php"><h1 style="color:white" >V. I. C. E.</h1></a></div>
        <div class="col-md-8">
            <?php
                if(isset($_SESSION['username'])){?
                    <a href="logout.php"><button type="submit" class="btn btn-success pull-right">Logout</button></a>
                } else {?
                    <form class="form-inline pull-right" action="landing.php" method="post">
                        <div class="form-group">
                            <input type="text" class="form-control add-margin" id="exampleInputName2" placeholder="Username" name="username">
                        </div>
                        <div class="form-group">
                            <input type="password" class="form-control add-margin" id="exampleInputName2" placeholder="Password" name="password">
                        </div>
                        <button type="submit" class="btn btn-success">Login</button>
                    </form>
                } else {?
                    <form class="form-inline pull-right" action="index.php" method="post">
                        <div class="form-group">
                            <input type="text" class="form-control add-margin" id="exampleInputName2" placeholder="Username" name="username">
                        </div>
                        <div class="form-group">
                            <input type="password" class="form-control add-margin" id="exampleInputName2" placeholder="Password" name="password">
                        </div>
                        <button type="submit" class="btn btn-success">Login</button>
                    </form>
                }?
            </div>
        </div>
    </div>
</nav>
<?php }?>
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
<!-- The above 3 meta tags *must* come first in the head; any other head content must come *after* these tags -->
<title>V.I.C.E</title>
<!-- Bootstrap --
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css" integrity="sha384-1q8mTJOAsx8j1Au+ a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7" crossorigin="anonymous">
<!-- Main Style -->
<link rel="stylesheet" type="text/css" href="css/main.css">
<!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries --
<!-- WARNING: Respond.js doesn't work if you view the page via file:// --
<!--[if lt IE 9]>
<script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
<script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
<![endif]-->
</head>
<body>
<?php include_once "elements/navbar.php"; ?>

<div class="container">
  <div class="row">
    <div class="col-md-12">
      <div class="jumbotron">
        <h1>Welcome to V.I.C.E</h1>
        <p>Vertical Implementation of a Cloud for Education</p>
        <p><a class="btn btn-primary btn-lg pull-right" href="#" role="button">Learn more</a></p>
        <hr>
      </div>
      <div class="text-center">
        <span class="glyphicon glyphicon-cloud aria-hidden="true"
          id="cloud"></span>
      </div>
    </div>
  </div>
</div>

<?php include_once "elements/footer.php"; ?>

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) --
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
<!-- Include all compiled plugins (below), or include individual files as needed --
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js" integrity="sha384-o+sXQW+OnWkcr/D8G5u2dZq47N9taxsZPi7GeXqLQrXQ+X6zjWOQME4j0Q87j77" crossorigin="anonymous"></script>
</body>
</html>

Figure C.5: index.php.
<?php
include_once("login/conn.php");
if( isset($_POST["username"] ) and isset($_POST["password"] ) ) {
$query = "SELECT * FROM ids WHERE name='". $_POST["username"] ."' and password='". $_POST["password"] ."';
$result = pg_query($query) or die("Query failed: ". pg_last_error());
if(pg_num_rows($result) == 1)
    $line = pg_fetch_array($result);
$_SESSION["username"] = $line["name"];$_SESSION["type"] = $line["type"];}
if( isset($_POST) && isset($_POST["nodes"] ) )
    $outputInfo1 = shell_exec("python3 /home/chead/viceclient.py ".(adminPoll(‘". $_POST["nodes". ‘"'))");
if( isset($_POST) && isset($_POST["nodeIP"] ) && isset($_POST["VMName"] ) )
    $outputInfo2 = shell_exec("python3 /home/chead/viceclient.py ".(virshDestroy(‘". $_POST["nodeIP". ‘"'],". $_POST["VMName". ‘"'])");
if( isset($_POST) && isset($_POST["button1"] ) )
    $outputInfo3 = shell_exec("python3 /home/chead/viceclient.py ".(loadBalance())");
$arrayOfTest = explode("SSH at: ", $outputInfo3);
    try {
        $sshCred = ‘"’;
        $sshCred = $arrayOfTest[1];
    catch (Exception $e) {
        echo ‘Caught exception: ‘, $e->getMessage(), ‘
    }
}}
?>
<!DOCTYPE html >
<html lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
<!-- The above 3 meta tags *must* come first in the head; any other head content must 
    come *after* these tags -->
<title>V.I.C.E<title />
</head>
<body>
<?php include_once "elements/navbar.php"; ?>
<?php
if( isset($_SESSION["type"] ) ){
    echo "*".$_SESSION["type"]."*; if($_SESSION["type"] == "admin "){

Figure C.6: landing.php.
// This will display if you are an admin

<?php
if (isset($outputInfo1)) {
    echo '<h4>Results</h4>
    <pre>'. $outputInfo1 .'</pre>
};
?>

<h1>You are logged in as an Admin</h1>
<form action="landing.php" method="post">
    <select name="nodes">
        <option value="com1">com1</option>
        <option value="com2">com2</option>
        <option value="com3">com3</option>
        <option value="com4">com4</option>
        <option value="com5">com5</option>
    </select>
    <br>
    <input type="submit" value="Run">
</form>

<h2>Stop a VM</h2>
<form action="landing.php" method="post">
    <label>Node IP Address</label>
    <br><input type="text" name="nodeIP"><br>
    <label>VM Name</label>
    <br><input type="text" name="VMName"><br>
    <br>
    <input type="submit" value="Stop VM">
</form>

<h2>Start a VM</h2>
<form action="landing.php" method="post">
    <br>
    <input type="submit" name="button1" value="Start VM">
</form>

<?php
if (isset($outputInfo3)) {
    echo '<h4>Results</h4>
    <pre>'. $outputInfo3 .'</pre>
};
?>

<?php
if (strlen($sshCred) > 0) {
    ?>
    <h1>Password: ubuntu</h1>
    <!-- Standard button -->
        <button type="button" class="btn btn-primary btn-lg">Launch SSH</button>
    </a>
    <?php
} ?>

</div>
</div>
</div>
<?php
else if ($_SESSION['type'] == "student

Figure C.7: landing.php-admin.
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <h1>You Are Logged in as a Student</h1>
      <form action="landing.php" method="post">
        <br />
        <input type="submit" name=button1 value="Start VM">
      </form>
    </div>
  </div>
</div>

<?php
  if(isset($outputInfo3)){
    echo "<h4>Results</h4>";
    echo "<pre>".$outputInfo3."</pre>";
    if(strlen($sshCred) > 0){
      ?><br>Password: ubuntu><?php
    }
  } else {
    echo("Something has gone terribly wrong with our backend if you see this.");
  }
?>

<?php
//Login Failed Case
elseif(?
</div>
</div>
<?php
//include_once "elements/footer.php"; ?>
<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
</body>
</html>

Figure C.8: landing.php-student.
<?php
include_once("login/conn.php");
if(isset($_SESSION)){
    $_SESSION = array();
    session_destroy();
}
header('Location: http://192.168.2.45/index.php')
?>

Figure C.9: logout.php.