Virtual Machine Server Management Tools

A Major Qualifying Project
Submitted to the Faculty of
Worcester Polytechnic Institute
In partial fulfillment of the requirement for the
Degree in Bachelor of Science
In

Computer Science

By

____________________________
Craig Nesbitt

December 30, 2014

Project Advisor:

____________________________
Prof. Craig Shue, Advisor
Abstract

In the Zoo Lab at WPI, the Computer Science Department has an isolated network with a Virtual Machine server on it. The network allows experimentation with new research and engineering projects, while also allowing students to practice cyber security defense and offense. The virtual machine server needs to host images for university courses along with images built by students for their practices. It also requires different levels of access to server resources for administrators, professors, and students. Students should be able to control their virtual machine environment without affecting course resources. A lack of clearly defined permission levels is a security hazard, as students could accidentally, or intentionally, impact server resources devoted to other users. Finally, the server needs to support a rapidly changing user base, as the student body changes each year. Manually providing students with access to the server creates a tedious, time consuming process for administrators of the server and disregards the purpose of the Central Authentication Service used by WPI. These feature requirements are addressed by virtual machine management and cloud management tools.

The best approach to WPI’s needs is the open source cloud management tool, Apache CloudStack, which uses a web interface to monitor and utilize the virtual machines. CloudStack has a three-tier permission system, grouping users into sections that it calls “domains”. System Administrators are able to monitor and control all resources on the server. Domain administrators apply to professors and TAs and are able to control resources within an allotted domain. Users are the lowest tier and are granted limited access to the domain’s resources. This applies to students who would be able to create, access, and delete their own virtual machines without being able to see other users’ virtual machines. CloudStack also has support for user authentication using Lightweight Directory Access Protocol which is compatible with the Central Authentication Service used by WPI.

While CloudStack is one solution to the needs of the virtual machine server, there are dozens of virtual machine managers available both in the open source community and commercially. These tools vary dramatically from the GUI to the backend. Virsh, for example, uses a command line interface while OpenStack has a web UI similar to CloudStack’s. VMware’s vSphere can cost up to $3,495.00 depending on the version purchased, while Proxmox VE and Eucalyptus are open-source and free. oVirt only fully supports the KVM hypervisor, while Archipel can support any hypervisor that uses the libvirt library. Among the dozens of management tools, OpenStack, Eucalyptus, and oVirt are CloudStack’s chief competitors, with only minor differences between OpenStack and CloudStack in particular.

There is a specific requirement of WPI’s server that none of these tools address; virtual machines within the isolated network need to be able to be accessed in order to configure and query for information. This access also needs to be achieved from outside of the isolated network without exposing the virtual machines themselves to the outside network. This is achieved by implementing an agent-based architecture containing an agent on each virtual machine, a controller running alongside CloudStack, and a webserver outside of the isolated network.
## Contents

1 Background Information ........................................................................................................... 5  
1.1 Virtual Machine Server .......................................................................................................... 5  
1.2 Hardware Virtualization and Paravirtualization ................................................................. 6  
1.3 Hypervisors .......................................................................................................................... 7  
1.4 Libvirt .................................................................................................................................. 9  
1.5 Migration .............................................................................................................................. 10  
1.6 Virtual Machine Management Tools .................................................................................. 11  
1.7 Central Authentication Service ............................................................................................ 12  
1.8 WPI's Computer Science Virtual Machine Server ............................................................. 12  
2 WPI's Management Tool Requirements ................................................................................... 13  
3 Current Implementation: Archipel ........................................................................................... 15  
4 Possible Solutions .................................................................................................................... 16  
5 Recommended Solution ............................................................................................................ 19  
6 Designing a Management Application .................................................................................... 20  
7 VMGMT Implementation .......................................................................................................... 22  
7.1 Common Objects and Protocol ............................................................................................ 22  
7.2 Agent ................................................................................................................................... 26  
7.3 Controller ............................................................................................................................. 29  
7.4 Web Server .......................................................................................................................... 32  
7.5 Summary of Network ........................................................................................................... 34  
7.6 Use Case .............................................................................................................................. 35  
8 Challenges Faced ...................................................................................................................... 36  
9 Future Work ............................................................................................................................. 37  
10 Conclusion .............................................................................................................................. 38
Figures and Tables

Figure 1: “A Type 1 or bare-metal hypervisor sits directly on the host hardware.” .............................. 8
Figure 2: “A type 2 hypervisor runs as an application on a host operating system.” .............................. 8
Figure 3: Example of how Archipel interacts with users and resources .......................................................... 15
Figure 4: Agent-Based Architecture in CSNW’s Isolated Network ................................................................. 20
Figure 5: Class Diagram of Common Classes ................................................................................................. 23
Figure 6: Packet Protocol ................................................................................................................................. 24
Figure 7: Class Diagram of Agent .................................................................................................................... 27
Figure 8: agentServer and agentServerClient Message Protocols ................................................................. 28
Figure 9: Class Diagram of Controller ........................................................................................................... 30
Figure 10: Class Diagram of Web Server ........................................................................................................ 33

Table 1: Summary of VMware’s x86 virtualization techniques ................................................................. 7
Table 2: List of Hypervisor Types .................................................................................................................... 9
Table 3: Libvirt Hypervisor Drivers ............................................................................................................... 10
Table 4: Comparison of Potential Virtual Machine Management Tools ....................................................... 17
1 Background Information

Before comparing different virtual machine management tools, it is first necessary to understand the fundamentals of what these tools can do. This section will introduce virtual machines and why they are useful, and then go into the details of the different hardware and software components involved in managing multiple virtual machines. Virtual machine management tools provide an interface between the user and these components and will be discussed in greater detail at the end of the section.

1.1 Virtual Machine Server

“A virtual machine (VM) is a software implementation of a physical machine, where the software implementation is constrained within another computer at a higher or lower level of symbolic abstraction.”

A virtual machine (VM) is a simulation of a physical machine within a software environment. The VM’s hardware is simulated by a piece of software, called a hypervisor, and is treated like any other computer program on the operating system (OS). While there are two types of hypervisors, this description is assuming a Type 2 hypervisor as the behavior of a VM is the same between both. This introduces the idea of a host operating system and one or more guest operating systems. The host OS runs the hypervisor as a program, granting it disk space, RAM, and access to external devices. The hypervisor then runs a guest OS within a VM and creates drivers out of the resources provided by the host OS. When the guest OS accesses the Internet, for example, it uses the driver provided by the hypervisor. This driver then passes the request via the hypervisor to the host OS, which uses the real network driver to complete the request. This methodology allows most VMs to be able to install and run any OS that could be installed on a physical machine.

The purpose of virtual machines is to provide a level of abstraction from the hardware, although the underlying motivation to do so can vary dramatically. On a personal computer, especially in academia, a VM allows the user to work within an operating system separate from their host OS without having to go through the process of installing it on their hardware. When working with malware, a VM’s environment is contained, preventing any viruses from permanently affecting the machine running it. If a guest OS becomes corrupt, the hypervisor can simply delete it, wiping the memory on that section of the disk, and create a new VM. Hypervisors are also able to manage more than one VM at a time, allowing for servers and clouds that manage hundreds of virtual machines.

A virtual machine server is simply a server that manages and provides access to virtual machines. A cloud on the Internet, for example, is a cluster of virtual machines servers that use the VMs to process requests. Having separate VMs execute each request provides a method of multitasking in which no process can interfere with the environment of another task. This also adds a level of security to
the servers as they cannot be accessed directly. If a VM crashes or becomes corrupt, the rest of the server is still fully functional and that given VM is rebooted.

WPI’s VM server not only provides access to the virtual machines for its users, but also provides the ability to create, delete, and modify them. Management software makes it possible to manage resources on a per-user basis, meaning that one student would not be able to use all of the available disk space or outgoing network traffic. The result is a cloud-like architecture for students to access and manage their work environments from any connected computer without affecting the environments of other students.

1.2 Hardware Virtualization and Paravirtualization

Hardware virtualization is the term used for creating a virtual machine. In other words, hardware virtualization is the act of creating a logical representation of a computer in software. While there are operating systems that can execute in the application layer of a machine, x86 operating systems are designed to run directly on the bare-metal hardware and therefore assume that they own the computer hardware. This means that many of the instructions executed by x86 operating systems do not behave properly in a virtualized environment. VMware has reported three techniques with which to address this challenge.

The first technique to virtualize the x86 OS is full virtualization using binary translation. Full virtualization completely abstracts the hardware from the guest OS, allowing any OS that could stand alone on hardware to be able to run on a VM. To achieve this, the VM’s hypervisor translates all of the guest OS instructions into their user-level equivalent. With full virtualization, the guest OS is not aware that it is in a VM. A good example of this is virtual memory, where the guest OS accesses its memory by the address starting at 0x0, which is then converted to by the hypervisor to the physical memory address located in user space. This is the only option that does not depend on assistance from the guest OS or the physical hardware.

The second approach is paravirtualization, in which the guest OS is modified such that all instructions that cannot be virtualized are replaced with corresponding calls to the hypervisor. With paravirtualization, the hypervisor no longer has the overhead of translating all of the instructions from the guest OS. However, there are several concerns with paravirtualization as well. The greatest problem is that unmodified operating systems cannot be run on a VM using paravirtualization. This causes problems with compatibility and portability between versions of the OS. “The performance advantage of paravirtualization over full virtualization can vary greatly depending on the workload.”

Most computers now support hardware assisted virtualization, where the hypervisor achieves full virtualization with help from the hardware. The processor grants the hypervisor root mode privileges. Instead of the guest OS’s instructions getting read by the hypervisor to be translated as needed, they go directly to the hardware. If the processor encounters any privileged or sensitive
instructions, it traps up to the hypervisor to obtain the translation for that instruction. Otherwise, the instructions are executed normally, saving the overhead of any binary translation from the hypervisor.

### Table 1: Summary of VMware's x86 virtualization techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Full Virtualization with Binary Translation</th>
<th>Hardware Assisted Virtualization</th>
<th>Paravirtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technique</td>
<td>Hypervisor translates binary from guest OS to execute on user-level</td>
<td>Guest OS runs on hardware and traps to hypervisor for specific instructions</td>
<td>Guest OS makes calls to hypervisor instead of assuming ownership of the hardware</td>
</tr>
<tr>
<td>Guest OS Modification Required?</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Performance</td>
<td>High</td>
<td>Fair, with potential to reach performance of Full Virtualization</td>
<td>High under specific conditions</td>
</tr>
<tr>
<td>Used By</td>
<td>VMware, Microsoft, Parallels</td>
<td>VMware, Microsoft, Parallels, Xen</td>
<td>VMware, Xen</td>
</tr>
</tbody>
</table>

### 1.3 Hypervisors

As mentioned earlier in this report, a hypervisor runs one or more virtual machines. Hypervisors are software or firmware components that virtualize system resources and distribute them across virtual machines. It is the hypervisor that determines how the amount of memory, processors, RAM, and external devices are available to each VM. It is also possible to limit the network available to each VM so that one VM cannot flood the entire network with activity. There are two different types of hypervisors: Type 1 and Type 2.

Type 1 hypervisors, also known as bare-metal hypervisors, run directly on the system hardware. Each VM on the hypervisor is found within a given domain number. Domain 0 is called the “control domain” and is a VM with special privileges to access the hardware and other virtual machines. This is the OS where the admin can control the hypervisor and other virtual machines. When the physical machine is turned on, the BIOS launches the Type 1 hypervisor, which in turn launches its control domain. From there, other virtual machines can be created in domains 1 or higher.
Type 2 hypervisors run as an application on the host OS. With a Type 2 hypervisor, the physical machine is turned on, the host OS is launched, then the hypervisor application is launched on top of the host OS. In this environment, there is no need for domains as all hypervisor and VM control is done from the host operating system. Due to the extra layer created by the host OS, Type 2 hypervisors tend to be slower that Type 1 hypervisors, but this is not always the case.

One unique hypervisor is KVM (Kernel-based Virtual Machine) that is exclusive to Linux. From an access perspective, KVM is a Type 2 hypervisor since it is launched as an application within a host Linux OS. At this point, KVM loads itself into the host OS as a kernel module, placing it on top of the hardware alongside Linux. Therefore, from hardware perspective, KVM is more of a Type 1 hypervisor since there is no OS layer between the hypervisor and the hardware. QEMU (Quick Emulator) is a Type 2 hypervisor that works in tandem with KVM. QEMU emulates external drivers and manages multiple virtual machines, passing base hardware activity to KVM. This allows virtual machines to run in a Type 2 environment with Type 1 hardware access. Other popular hypervisors are listed below.
### Table 2: List of Hypervisor Types

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Hypervisor Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMware ESXi</td>
<td>Type 1</td>
</tr>
<tr>
<td>VMware Player</td>
<td>Type 2</td>
</tr>
<tr>
<td>Microsoft Hyper-V</td>
<td>Type 1</td>
</tr>
<tr>
<td>Oracle VirtualBox</td>
<td>Type 2</td>
</tr>
<tr>
<td>Xen</td>
<td>Type 1</td>
</tr>
<tr>
<td>Wind River</td>
<td>Type 1</td>
</tr>
<tr>
<td>LXC (Linux Containers)</td>
<td>Type 2</td>
</tr>
<tr>
<td>OpenVZ</td>
<td>Type 2</td>
</tr>
<tr>
<td>UML (User Mode Linux)</td>
<td>Type 2</td>
</tr>
<tr>
<td>IBM PowerVM</td>
<td>Type 1</td>
</tr>
<tr>
<td>Parallels</td>
<td>Type 2</td>
</tr>
<tr>
<td>Bhyve</td>
<td>Type 2</td>
</tr>
</tbody>
</table>

#### 1.4 Libvirt

Many Linux-based hypervisors, including KVM and Xen, use a library called libvirt. The **libvirt** library is a hypervisor-agnostic Linux API over the virtualization capabilities of Linux. Libvirt allows management on a single machine or between many nodes, where a **node** is a physical machine either by itself or within a network of other nodes. Libvirt detects the local hypervisors of a node and sets up drivers for each one. A management application then uses libvirt to control the domains on the machine.

When multiple nodes are involved, one node has the management application and the rest, called remote nodes, have a daemon called libvirtd. “This daemon runs on host servers and performs required management tasks for virtualized guests. This includes activities such as starting, stopping and migrating guests between host servers, configuring and manipulating networking, and managing storage for use by guests.” The management application communicates with libvirtd remotely through a custom protocol in order to send commands and retrieve information from the remote nodes. This allows the management application to access all virtual machines across all nodes in one interface, rather than looking at each node individually.
In order to be hypervisor-agnostic, libvirt uses a driver-based architecture. Any hypervisor that implements libvirt therefore must have a corresponding driver that converts the hypervisors functions to match the libvirt’s API. This means that any functionality that a hypervisor has that is not listed in libvirt’s API is lost. This also means that, while libvirt is hypervisor-agnostic, any hypervisor without a corresponding driver will not work with libvirt. A table of supported hypervisors can be seen below.

Table 3: Libvirt Hypervisor Drivers

<table>
<thead>
<tr>
<th>Libvirt Hypervisor Drivers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LXC (Linux Containers)</td>
<td>OpenVZ</td>
</tr>
<tr>
<td>QEMU</td>
<td>Test (used for testing)</td>
</tr>
<tr>
<td>UML (user Mode Linux)</td>
<td>VirtualBox</td>
</tr>
<tr>
<td>VMware ESX</td>
<td>VMware Workstation/Player</td>
</tr>
<tr>
<td>Xen</td>
<td>Microsoft Hyper-V</td>
</tr>
<tr>
<td>IBM PowerVM (phyp)</td>
<td>Parallels</td>
</tr>
<tr>
<td>bhyve</td>
<td></td>
</tr>
</tbody>
</table>

1.5 Migration

When working with multiple nodes, it is possible to move a VM from one node to another. This action is called migration. If the VM is not running, this process simply involves copying the existing OS image from the source node to the destination node, then attaching the image to the appropriate drivers on the destination node. Live migration, on the other hand, is a more complicated process that transfers the VM from the source node to the destination node while the VM is still running. During live migration, the VM usually experiences a downtime of at most a few seconds. After receiving a migration request, live migration is done in five stages.

The first stage is the reservation stage, in which the destination node creates a VM container that meets the resource requirements of the VM on the source node. If the destination node cannot allocate the required resources, then the migration fails and the VM continues to run on the source node.

The second stage is the iterative pre-copy. In the first iteration, all memory pages are copied from the source node to the destination node. For future iterations, only pages that were dirtied during
the previous copy are sent to the destination node. This process generally continues until the rate at which pages are being dirtied is less than the rate at which they are copied. If the ratio is not achieved, it is possible to impose limit for write faults, which reduces the working set for processes and slows down the rate that pages get dirtied.

Third is the stop-and-copy stage. At this point, the VM is stopped on the source node and any remaining dirty pages as well as the CPU state are copied to the destination node. The downtime experienced by the VM occurs during this copy to prevent any other pages from being dirtied. At the end of this stage, both the source and destination nodes should have identical versions of the VM.

The fourth stage is the commitment, where the destination node confirms that it has successfully received the copy of the VM. At this point, the source node deletes its copy of the VM and the VM is considered to be hosted on the destination node.

The final stage is the activation of the VM on the destination node. The VM is turned on and device drivers are connected to it. The VM is also notified of the new IP address that it has on the node. Once this is completed, the migration is concluded.

1.6 Virtual Machine Management Tools

It has been established that hypervisors are used to manage virtual machines and their resources. However, hypervisors often do not have the desired interface for administrators to monitor the virtual machines. As a result, many virtual machine management tools have been created that provide a more user-friendly interface for hypervisors. These tools often have a graphical interface that allows users to quickly create and modify virtual machines as well as display the status and activity of each one. Simple command line interfaces are adequate for small scale virtualization, but are not practical when managing servers containing virtual machines for many users. This is where the graphic interfaces, especially web-based GUIs, excel. More elaborate tools also provide VM templates, user management systems, and one-click migration.
1.7 Central Authentication Service

The Central Authentication Service (CAS) is fairly self-explanatory in its name. CAS is an open-source tool used by WPI to allow authentication for web applications using a single username database. It uses a ticket-based protocol designed exclusively for CAS. When the user attempts to access a protected web application, they are redirected to the CAS login page. CAS checks the login information against a database using an access protocol such as Kerberos or Lightweight Directory Access Protocol (LDAP). If the user successfully logs in to CAS, then it creates a Single-signon (SSO) session and redirects the user back to the protected web application, passing along a service ticket. The application then contacts the CAS server to validate the ticket and, if the ticket is valid, successfully logs the user in to the application. While the user’s session is still active, it creates service tickets for all web applications using CAS, allowing the user to only have to sign in once.

1.8 WPI’s Computer Science Virtual Machine Server

WPI’s Computer Science Department VM Server (CSNW) is run on a single server in a mostly isolated network. Access to the server is provided through two general means. For students, or anybody else obtaining service from the server, the only method to access the CSNW is through the orange Ethernet cables available in the Computer Science labs in the basement of Fuller Laboratories on campus. This network does not allow any traffic to directly reach any other part of WPI’s network or the Internet. However, specific ports can be configured to allow traffic outside of the isolated network. For example, users within the isolated network have to query a server that is also inside the isolated network for software and updates from Ubuntu. That server has permissions to access the Internet to complete the user’s request, while the users themselves cannot connect to anything outside of the isolated network. The purpose of the isolated network is to allow experiments in cyber-security as well as advanced networking without the risk of affecting machines other than the test VMs.
2 WPI’s Management Tool Requirements

There are many factors to take into consideration when analyzing a VM management tool. The most obvious, as with any product, is price. High-end management tools can cost thousands of dollars, and many have a recurring subscription for support. At the time of this report, the price of the VMware vSphere license ranges from $995.00 to $3,495.00, depending on the version, in addition to an optional annual support subscription. On the other hand, there are several well developed open source management tools. This gives the tools that are not free a niche set of users, specifically those who will have large clusters and require 24-hour support for their system. WPI, on the other hand, has a single server in which the power and support of an expensive tool would be excessive. The variety of open-source tools will provide WPI with the resources adequate to achieve its goals in virtualization, as well as allow easy modification to the source code to add features exclusive to a university.

Different VM management tools support different hypervisors, and many tools are specific to only one hypervisor. QEMU/KVM is already set up on WPI’s VM server and, along with its unique behavior between Type 1 and Type 2 hypervisors, is the preferred hypervisor for managing VMs. This does not exclude other hypervisors as possibilities in the event that a tool is found with the best available features for WPI.

The third attribute worth considering the type of interface that the tool provides. The VM server is expected to handle several courses as well as computer science clubs, easily managing over a hundred virtual machines. Accessing and monitoring this user base is impractical from command line interface (CLI), disqualifying any CLI tools. There are also security concerns with students accessing the server directly, meaning that any tools involving SSH are also unacceptable solutions. Students will naturally have varying experience with computers. To accommodate the broad range of experience, as well as provide the easiest access both on and off campus, a web interface would be WPI’s best option.

Permissions are very important when working with any server, but a university environment creates a unique tiered access system. The highest access is the system administrators, who are made up of the staff that monitor the VM server. Next in line are the group administrators. In this context, a group refers to a specific course or club, and the group administrators are the professor and TAs. Group administrators can monitor and manipulate VMs within that group, but should not have any access to other groups. The lowest tier is the students within each group. The student should have a limited amount of resources with which to create and monitor their own VMs. Where team projects are very common at WPI, it is also important for students to be able to grant other students permissions to their VM.

In some courses, professors create VM images for the students to use for their projects. These images generally have specific software installed or a modified OS to allow the focus of the assignment to be on the lesson materials. With this in mind, the VM server’s management tool should have some form of template, allowing students to select a pre-made VM from a list of options.
One behavior does not impact the decision process of VM management tools, but is worth being aware of when using one. Some tools are aware of all VMs on the node, and can import the existing VMs to be managed on the same interface as those created from the tool. If the tool is capable of doing this, then it is important to ensure that the corresponding permissions to imported VMs are set accordingly.

With each year, a new set of students will need access to the VM server, meaning that the management tool will have to manage hundreds of registered users. Most tools assume a static, or at least less dynamic, set of users. The common method is to be able to create an individual user and grant them privileges to specific domains or VMs. In a university environment, this means that professors and TAs will be very busy before each class manually creating users for each attending student. An ideal management tool will be able to allow students to log in to the VM server via the CAS tool, removing the need to manually create users and set permissions. Recognizing that CAS is a unique tool, a management tool that supports authorization via LDAP would provide an adequate framework to implement CAS support.
3 Current Implementation: Archipel

WPI’s VM server is currently using the management tool, Archipel, to manage its VMs. Archipel is an open source project distributed under AGPL v3 that supports KVM, Xen, VirtualBox, and OpenVZ, using the libvirt library. It uses a graphical web interface and communicates with hypervisors and VMs across multiple nodes using XMPP (Extensible Messaging and Presence Protocol). This communication is accomplished via Jabber through the instant messaging server, ejabberd, distributed under the GPLv2 open source license. Each hypervisor and VM connected to Archipel has a unique Jabber ID (JID) which is in a similar format to an email address, as seen in the picture below. Archipel also recognizes any VMs created on the node and offers to import it into the interface.

![Diagram of Archipel interaction](image)

*Figure 3: Example of how Archipel interacts with users and resources. PIDs are shown. Lines represent XMPP communication.*

Archipel allows the distribution of VM templates via RSS feeds called VMCasts. Each hypervisor under Archipel has a VMCast feed that with which it can package any of its VMs and make them available to other hypervisors (or itself). Users that have access to the VMCast can then download the package to a new VM. One problem with this is that a hypervisor must download from its own VMCast in order to use its own templates.

Archipel supports QEMU/KVM through a web interface at no cost and provides a method for providing templates to students. However, a glaring problem that negates the convenience of web access and templates is the user management system. It is possible to create a template for different types of users, which is indeed helpful. However, users have to be manually created by an administrator one at a time. Therefore, at the beginning of any course that will use the VM server, an administrator
must take the time add a user for every registered student. This can easily exceed one hundred new users with more than one course using the server.

Alternatively, TAs would have to be granted administrator privileges to make the users. This is undesirable as TAs should not have administrator access to the server’s virtualization management system. In addition, adding students to VMs to work as a group is an additional manual process. There is no obvious place in the source code to add functionality to support CAS, either. Once all users are set up, Archipel is a very competitive VM management tool. However, in a university environment, users are constantly being created and moved around. Archipel would need heavy modification to the user management system in order to match what WPI is looking for in a VM management tool.

4 Possible Solutions

There are dozens of available VM management tools; KVM lists thirty six possibilities on their website, which do not include any tools exclusive to other hypervisors.\textsuperscript{14} These tools, as well as many others, were compared against the set of requirements described earlier in this report. In addition, GUI screenshots and tutorial videos were used to compare the how user-friendly each tool is. VMware’s vSphere, Red Hat Enterprise Virtualization, and VMmanager are all commercial management tools, making them unaffordable and therefore unviable options. Tools such as Virsh, kvmadm, and Ganeti are exclusive to the command line, which is unreasonable for the scale that the VM server will be addressing. Proxmox VE is open-source and has a web interface, but has worse user management capabilities than Archipel and is therefore also not an option. OpenQRM has both an open-source version and a commercial version, but LDAP user authentication is only available through the commercial version, making it more difficult support CAS. This left four tools that stood out as potential solutions: OpenStack, Apache CloudStack, Eucalyptus Cloud, and oVirt. These tools are compared side-by-side on the following page.
<table>
<thead>
<tr>
<th>OpenStack</th>
<th>Apache CloudStack</th>
<th>HP Eucalyptus Cloud</th>
<th>oVirt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>License</strong></td>
<td>Apache License Version 2.0</td>
<td>Apache License Version 2.0</td>
<td>GNU General Public License v3</td>
</tr>
<tr>
<td><strong>Support</strong></td>
<td>Community and forum</td>
<td>3rd party such as ShapeBlue</td>
<td>$149 - $299 / mo.</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td>Web</td>
<td>Web</td>
<td>Web, CLI, REST, SOAP</td>
</tr>
<tr>
<td><strong>Compatible Hypervisors</strong></td>
<td>KVM (most popular)</td>
<td>Hyper-V</td>
<td>KVM</td>
</tr>
<tr>
<td></td>
<td>Hyper-V</td>
<td>KVM</td>
<td>LXC</td>
</tr>
<tr>
<td></td>
<td>ESXi</td>
<td>LXC</td>
<td>vSphere</td>
</tr>
<tr>
<td></td>
<td>Xen</td>
<td>XenServer</td>
<td></td>
</tr>
<tr>
<td><strong>Compatible Operating Systems</strong></td>
<td>Red Hat Enterprise Linux</td>
<td>CentOS/RHEL 6.3+</td>
<td>CentOS 5</td>
</tr>
<tr>
<td></td>
<td>Scientific Linux</td>
<td>Ubuntu 12.04(.1)</td>
<td>RHEL 5</td>
</tr>
<tr>
<td></td>
<td>CentOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fedora</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Debian</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ubuntu</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Required Software</strong></td>
<td>Keystone</td>
<td>Eucalyptus-cloud</td>
<td>Eucalyptus-cloud</td>
</tr>
<tr>
<td></td>
<td>Glance</td>
<td>Eucalyptus-cc</td>
<td>Eucalyptus-cc</td>
</tr>
<tr>
<td></td>
<td>Nova</td>
<td>Eucalyptus-sc</td>
<td>Eucalyptus-sc</td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>Eucalyptus-walrus</td>
<td>Eucalyptus-walrus</td>
</tr>
<tr>
<td></td>
<td>Neutron</td>
<td>Common Utilities</td>
<td>Common Utilities</td>
</tr>
<tr>
<td></td>
<td>OpenStack Dashboard</td>
<td>MySQL</td>
<td>MySQL</td>
</tr>
<tr>
<td></td>
<td>Cinder API</td>
<td>MySQLdb</td>
<td>MySQLdb</td>
</tr>
<tr>
<td></td>
<td>Swift</td>
<td>Tomcat 6 (not 6.0.35)</td>
<td>Tomcat 6 (not 6.0.35)</td>
</tr>
<tr>
<td></td>
<td>And more...</td>
<td>Genisoimage</td>
<td>Genisoimage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rpmbuild of dpkg-dev</td>
<td>Rpmbuild of dpkg-dev</td>
</tr>
<tr>
<td><strong>User Management</strong></td>
<td>Environments are separated into “tenants”, which contain a group of users. Users within each tenant are assigned “roles”. Roles are a set of permissions a user can perform in a given tenant. Administrator can create specific groups of users and assign roles to the group as a whole. Supports LDAP authentication.</td>
<td>Three-tier user types with domains. - Administrator - Domain (group) Admin - User Users can make projects in domains and invite other users to their VM. User Authentication possible via connection to an LDAP Server</td>
<td>Tiered access: - Sys Admin - Account Admin - Users Has a registration request where sys admin is contacted to approve/reject new account (group). Users can belong to multiple groups for sharing VMs. Supports LDAP/AD integration.</td>
</tr>
<tr>
<td><strong>Template Management</strong></td>
<td>Contains several tools for managing VM images</td>
<td>Can Take snapshot of any VM to use as template. Can also create VM template from scratch.</td>
<td>Works with Hybridfox to create multiple instances from established images.</td>
</tr>
</tbody>
</table>
As seen in the table, Apache CloudStack, Eucalyptus Cloud, and oVirt have very similar functionality with respect to the requirements of WPI’s VM server. The best candidate of these three was therefore determined by comparing GUIs, available documentation, and ideal use case. Each candidate will be analyzed by order of elimination.

oVirt is a massive project with very thorough documentation. It has a quantity of features above and beyond the requirements for the VM server in question; so many, in fact, that it can clutter the UI with options that are not applicable to WPI’s system. From the table, it can be seen that both CloudStack and Eucalyptus have three levels of user permissions. oVirt, on the other hand, has six basic roles and thirteen advanced roles.\(^\text{15}\) oVirt’s GUI’s primary selection menu is a collapsible menu that easily reaches four levels deep, selecting the system, rack, cluster, and then host. It is very clear that oVirt is designed for large cloud systems rather than a single server and adds unnecessary levels of complexity for this project’s intended purpose.

Eucalyptus suffers a similar problem to that of oVirt in the complexity of the GUI. While it meets the requirements for managing users and templates, Eucalyptus’s GUI contains a plethora of tabs and menus to create, pause, monitor, and delete VMs across multiple hypervisors. While the complex interface is not ideal, the deal-breaker for Eucalyptus cloud is that the GUI does not provide a view of the guest instances.\(^\text{16}\) This defeats the purpose of the tool as students would not be able to use their VMs through the web interface.

This leaves the decision between OpenStack and CloudStack. Both of these tools have clean, professional GUls as well as thorough online documentation. The biggest difference between these two tools is the implementation and installation process. While CloudStack has several software pre-requirements, the project itself is monolithic. OpenStack, on the other hand, consists of several different components that must be separately installed, each step requiring software configuration. This makes OpenStack much more difficult to install and configure than CloudStack.

OpenStack’s user management lies between that of oVirt and CloudStack. It’s “tenants” fit the requirement for separating administration into different classes, and assigns roles within each tenant. This adds flexibility to have more than 3 levels of permissions, but requires configuration to reach the same behavior that is default for CloudStack. The final deciding factor between the two is the idea of team projects. In OpenStack, the administrator creates a group of users within a tenant and grants them equal access to resources. In CloudStack, users are able to create their own “projects” and invite other users to it, allowing them to share resources without needing an administrator to be involved. Among the tools reviewed, CloudStack’s relatively simple setup and user project feature make it the most applicable VM management tool for an academic VM server.
5  Recommended Solution

To conclude the VM management tool analysis, Apache CloudStack meets the initial requirements established for the CSNW as well as introducing a feature that may prove useful if implemented at WPI. It is an open-sourced project under the Apache License Version 2.0. CloudStack uses the libvirt API, meeting the CSNW’s requirement for KVM support. It has a web interface, allowing the possibility of students and professors limited control of VM’s from both on and off campus. It allows users to, depending on permissions, either download ISOs as templates or save existing VMs as templates, allowing professors to create an initial state of a VM for students. Rather than a direct login, CloudStack is able to connect to an LDAP server, which is the same type of server used at WPI by CAS, allowing CloudStack to synchronize its user-base with WPI’s. With respect to additional features, CloudStack provides the ability to specify project groups. In addition, virtual local area networks (VLAN) can be configured for VMs, which can prevent VMs from different groups or classes from communicating with each other. Finally, there is an additional tool that can be implemented alongside CloudStack for scripting and command line access, called Apache CloudMonkey.

Apache CloudMonkey is a command line interface tool for CloudStack versions 4.0 and higher and requires Python 2.6 or above to be implemented. Like CloudStack, CloudMonkey is distributed under the Apache License Version 2.0. This tool provides two significant use cases for CloudStack which would have otherwise been unavailable. First, it allows administrators to utilize scripting in order to quickly configure any number users, networks, and VMs. For example, being able to create all VMs for a certain class would be much faster with a command to “create n VMs” than it would be to create each one manually. The second application is to use CloudMonkey as a bridge between CloudStack and another application. Since CloudMonkey can be called from the command line, it can be accessed via the ‘system’ function. This means that a broader management application would be able to use CloudStack as one of its tools.
6 Designing a Management Application

While CloudStack supports basic VM configuration through the use of templates, it is more difficult for it, as well as any of the other VM management tools reviewed, to configure each VM individually. One example of this would be to assign a specific unique hostname to each VM. This would require opening the “view console” from the CloudStack UI and configuring each VM one at a time. Alternatively, it is possible to SSH into a VM in order to issue any necessary commands, but this still becomes a time-consuming process as the number of VMs increases. While this is not a large problem on a small scale application, WPI requires a method to quickly send commands to configure VMs that does not involve manually establishing a connection ahead of time. The solution is to provide a constant connection to every VM by using an agent-based architecture. This VM management system was not specifically named, but was written in the namespace VMGMT, and will therefore be identified as such throughout this report.

![Diagram of Agent-Based Architecture in CSNW's Isolated Network.](image)

The agent-based architecture implemented allows a user to connect to a web server to issue system commands to any VM as well as start, stop and connect to servers hosted on VMs. The client-server support on each VM provides an easy way to check their network configuration and confirm VMs’ ability to communicate with each other. VMGMT’s design also provides the capabilities to perform simple operations from CloudStack, such as start and stop VMs, through the use of CloudMonkey. It is based off of a similar application written in PHP by Professor Craig Shue at WPI, but provides the capabilities of public/private key authentication as well as maintained state between connections. VMGMT consists of three major components: multiple agents, a controller, and a web server. One agent
runs on every VM and listens for commands from the controller. The controller runs alongside CloudStack inside of WPI’s CSNW, providing access to CloudStack as well as distributing messages to and handling responses from agents. The web server runs outside of the CSNW and maintains connection with the controller and is used to issue commands to the management system. Users can connect to the web server using a specified API from on or off campus in order to configure and test their VMs inside the network.

VMGMT was written in object-oriented C++. There are specific protocols for the components to follow when communicating with each other and these protocols were easiest to encapsulate using an object-oriented language. The most prominent example of this is network packets sent between agents, the controller, and the web server. Standard getters and setters are used to specify each value of the packets header as well as set any message that is sent with the packet. When the packet is sent, all of the data is converted into a character vector, and then a packet constructor is used to convert the vector back into the individual fields when the packet is received. While this is possible with functional programming, it is not as intuitive as it is in object-oriented programming. In addition, the modularity of object-oriented programming allowed the same object type to be used track connections for both the controller’s server and the agent’s server. While, again, this is not exclusive to object-oriented programming, the organization provided by class diagrams made it easier, if not more efficient, to design this modularity and determine what features could be used in multiple places.

In regards to the decision of C++ as the language of choice for VMGMT, the reasoning is less objective. WPI’s Computer Science department’s networking classes are taught in C, with very little usage of C++ and no mention of Java. In addition, Java is an unnecessarily large tool to include on every VM in order to run the agent when a more lightweight alternative is available. As such, the most familiar language for networking, and most applicable, is C. However, with larger projects, the memory management challenges of C can cause many problems. By utilizing the Standard Template Library (STL) provided with C++, the majority of the memory management concerns can be avoided. C also does not provide the desired object-oriented approach, and therefore it was concluded that C++ would be the preferred language for VMGMT.

One concern when implementing VMGMT is what happens when an agent loses connection. In the current PHP implementation using Archipel, once an agent connects to the controller, it is assigned a unique ID to differentiate it from other agents. The controller uses this ID to be able to distribute commands from the web server to the appropriate agents. However, if an agent loses connection, its ID is lost. Therefore, if it reconnects to the controller, it could be assigned a new ID. The obvious problem with this is that a VM containing one student’s configuration, as well as intellectual material, could end up getting assigned to a different student if the agent loses connection. To avoid this, a VM whose agent loses connection had to be destroyed.

In order to address this loss of data between connections, VMGMT’s agents store their assigned IDs. When the agent connects to the controller, it sends an empty initialization packet to the controller to confirm the connection. If the agent has not previously connected to the controller, then it initializes its ID to all zeros. The controller checks the ID that is included in the packet header and compares it to
its record of existing IDs as well as existing agent connections. If the ID matches an existing ID and there is not an agent already connected under that ID, then the agent is added accepted and added to the record of connected agents. If the agent’s ID is not a valid existing ID, then it can be assumed that the agent is connecting for the first time and can be assigned a new ID. In which case, the controller will take an ID from its record of unused IDs, place it in the existing IDs, add the agent to the record of connected agents, and send a response packet with the new ID. The agent reads the response packet and stores the new ID in a file to be used for the next connection. This makes the assumption that an ID of all zeros will never be a valid ID for the controller and that agents will not try to lie about their ID to take the place of other agents.

7 VMGMT Implementation

VMGMT’s source code is divided into four directories: one for each component as well as a directory for common objects used by multiple components. The components share a Makefile, however they compile into three separate executables. The only requirement for the controller and agent executables is to be in the same directory as a “txt” folder that contains files to retain state between connections. Specific details regarding dependencies, as well as information on communication protocols and individual component designs, can be found in the following subsections.

7.1 Common Objects and Protocol

The objects that are shared between multiple components can all be described to fit some form of protocol. The packet and VMID objects provide a translation between network data and object fields. Network sockets are bound in seven unique places throughout VMGMT’s design, and the static networkProtocol object provides a quick standard for binding and connecting to servers. NetworkProtocol also provides functions to send and receive packets. ConnectedAgents provides a thread-safe map of VMID to connectedAgentEntry, which contains an agent’s connection socket, authentication secret, and public key.
The packet and VMID objects are used to specify the protocol for all network messages sent between VMGMT’s components. A packet header is forty-four bytes and specifies which agent the packet is correlated with, what action should be performed, and the response status of the action. The header also contains an authentication secret to be able to verify the identity of each component. Finally the header specifies the length of the packet’s data. These fields, as well as the message sent with the packet, can be assigned and retrieved using standard setters and getters, respectively. The exception to this is the message length, which is calculated and cannot be set.

There are three enumerations used in the packet to specify details and state of the request. The “type” declares the destination object that will process the packet; the value “VMCTRL” states that the packet contains a command for the controller to execute through CloudStack. The “INIT” value is used
when a connection is first being established and the packets contain information rather than configuration commands. The other values (CMDLINE, SERVER, and CLIENT) refer to each agent execution object, which will be covered in the next section. The “action” enumeration specifies the action that destination object should perform. It has individual values to start, stop, and restart a VM. The start and stop value are also used for commands to the agent server and client, and the execute value is used for the commandLine object. When initializing a connection, there are packets sent that have the value “NO_ACTION”, which are containing initialization information. The “acknowledgement” enumeration describes the direction and success status of the packet. A packet with the “REQUEST” value means that the packet contains a command that has not been executed yet and is traveling from the web server to the controller, or from the controller to an agent. When a packet’s command has been executed, or is determined unable to do so, the acknowledgement is set to either “SUCCESS” or “FAILURE,” depending on success state of the execution. At which point, the packet is traveling from an agent to the controller or from the controller to the web server.

Figure 6: Packet Protocol

To allow packets to be easily sent over a socket, the packet has a function called “toNetworkData” that converts all of the header fields, as well as the packet’s message, into a vector of 1-byte characters. To complement this, the packet has a constructor that accepts a character vector as its only parameter and attempts to convert the vector data into packet fields. This is achieved primarily with the aid of unions. In the network protocol, the enumerations are stored as unsigned 2-byte integers and the message length is an 8-byte unsigned integer. Creating a union of the integer type and character array of matching size allows a quick conversion between the two data types. The toNetworkData function stores its values in the union, and then pushes the converted characters into the back of the vector containing network data. String and character vector packet values are simply copied into the network data. The network constructor simply follows the reverse process to read each value from the vector. The VMID has the same network function/constructor pair. However, the constructor accepts a second parameter of the position in a vector to begin parsing the VMID. Therefore, the packet is able to pass the serialization and deserialization for the ID field to its VMID.

When an agent connects to the controller, the controller randomly generates an 8-byte secret that will be used by the agent and controller in order to verify that the message received is authentic. This is passed in the packet’s authentication field in the packet header of every packet. The only
exception is the first packet sent by the agent when connecting to the controller, in which case the authentication field is zeroed as the secret has not yet been generated. This process also applies to the controller connecting to the web server, and will be covered in more detail in later sections.

Each agent is identified by its VMID, which contains the six-byte MAC address of the VM running the agent, the class or club correlated with that VM, the year that the class was held, the team number of the team or individual in the class or club, and a number assigned to the VM on a given team. The MAC address is used to be able to link the agent to the correct VM id that is used by CloudStack. The rest of the fields are used to easily identify an agent’s owner in a university environment. The class is a string, at most ten bytes, that identifies the class and section number or the club name. An example of this is WPI’s standard class naming convention of “CS4000 A00” or a cyber-security club using “cyber-sec”. The year is used to be able to distinguish VMs of a given class from possible remaining VMs from classes with the same ID that met previous years. In a club environment, the year may be correlated with a student’s graduation date or the date that they joined the club. Each student, or group, gets assigned a team number, and a team may have multiple VMs in order to achieve a project. As such, the team number and VM number, respectively, achieve an ID fitting WPI’s structure such that no two ID’s can overlap if assigned correctly.

When an agent connects to a server, the server must store all of the network information of that agent in a manner that is thread-safe. The connectedAgents object provides a map that uses a pthread mutex in order to allow access from the server’s communication object as well as the threaded daemon that listens for and stores connections. The map’s key is a string representation of the agents VMID object, which concatenates every VMID field except the MAC address into a stringstream. The MAC address is excluded in order to provide abstraction from the web server without sacrificing functionality of the map key. The web server or user should not need to be aware of an agent’s MAC address in order to issue a command to it. Instead, the user specifies the class, year, team number, and VM number, which is enough information to perform a lookup to see if that agent is connected, regardless of which VM is running the agent. The object stored in the map is the connectedAgentEntry, which contains the socket that the agent is connected on and the authentication secret that was generated for that connection. If public/private key authentication had been fully implemented, then this object would also contain the public key of the connected agent.

As previously mentioned, the networkProtocol object contains static functions that handle network setup and communication for all clients and servers in VMGMT. Servers are established using the function, bindServer, which has parameters necessary to establish a socket and listen on a specified port. The process for binding a socket to the network differs slightly between IPv4 and IPv6; specifically, IPv4 uses a sockaddr_in structure and IPv6 uses a sockaddr_in6 structure. BindServer recognizes and accommodates these differences with a simple if-statement. Clients, on the other hand, use the connectToServer function in order to establish a socket and connect to a server. ConnectToServer has nearly the same parameters as bindServer, except that bindServer has a backlog variable to specify how many pending connections should be allowed and connectToServer has a variable to specify the hostname that the server is listening at. Both of these functions return the socket number that the server or client is bound to. To send a packet, the sendMessage function accepts a packet object and
socket number, obtains the network vector using the packet’s `toNetworkData` function, sends the serialized packet, then returns the number of bytes sent or negative-one if an error occurred trying to send the packet. `getMessage` reads the forty-four byte header, parses the message length located at the thirty-sixth byte, then reads the packet message based on the length specified in the header. The header and message are concatenated and then passed into the packet constructor to deserialize the packet and the packet is returned. `sendBytes` and `getBytes` send and receive a raw character vector rather than a packet and act as helper functions for `sendMessage` and `getMessage`, as well as a tool for the web server interface.

## 7.2 Agent

From the perspective of the controller and `agentInterface`, the front-end of the agent is the `agentClient` and the back-end is the execution objects that process requests. The `agentInterface` contains the main function that creates an `agentClient`, orders it to connect to the controller, and then tells it to enter a loop to listen for commands from the controller as long as the connection is open. If the connection with the controller is lost, then the agent safely closes and must be started again. The configuration of the controller’s hostname and port establish a connection is located in the `agentInterface`. 
The agentClient object handles all communication with the controller, using the agentKeyInfo and myVMID objects to assist in authentication and identification, respectively. MyVMID contains static functions to read and set the agent’s VMID that is stored in a text file. AgentKeyInfo is a plug for public/private key authentication with the controller, but it currently has no functionality. To be implemented, the agentKeyInfo would store the authentication secret provided by the controller as well as its public/private key and the controller’s public key. To authenticate a packet, the authentication field of the packet header is passed to agentKeyInfo, which would decrypt and compare the secret to the stored secret. Before packets are sent, they are passed to agentKeyInfo to have the authentication secret encrypted using the controller’s public key and then stored in the packet header. agentClient’s connectToController function uses the networkProtocol to establish a connection with the controller,
then creates an empty initialization packet with the VMID obtained from myVMID and sends it to the controller. This packet’s authentication field is buffered with zeroes as the controller has not assigned a secret for the agent yet. If the agent does not have a VMID assigned to it yet, then those fields, except the MAC address, are also buffered with zeroes and the controller will assign a VMID to the agent. The agent then listens for a response from the controller that contains the authentication secret and assigned VMID in the header fields. The authentication secret and the VMID are stored and the connection is considered to be successful. The listenForCommands function enters a loop that, while the agent is connected to the controller, it will read a packet form the controller, verify it, and pass it to the client’s agentCore. The response from the agentCore is signed and sent back to the controller, and the loop starts again.

The agentCore acts as a central distributor of packets to the agent’s execution objects. It checks the packet’s “type” and compares it to the available resources. Packets of type “SERVER”, “CLIENT”, or “CMDLINE” are passed to the agentServer, agentServerClient, or commandLine, respectively, to be processed. If the type does not match one of these three values, then the acknowledgement field of the packet is set to “FAILURE” and the packet message is filled with an explanation that the packet is of an unsupported request type, then the modified packet is returned to be sent to the web server. Otherwise, the packet returned from one of the execution objects is returned.

![AgentServer and AgentServerClient Message Protocols](image)

The agentServer and agentServerClient are used to confirm network configuration of agents. The serverCommand function checks the packet’s “action” and compares it to the corresponding private functions. If the action is not “START” or “STOP”, then the packet’s acknowledgement is set to “FAILURE” and returned with the reason in the packet message. When the action is “START”, the agentServer will start a thread that acts as a simple server for other packets to connect to, using the required parameters that are sent in the packet message to specify server setup. The server will listen for a connection, read an initialization packet, send a confirmation response, and store the connected agent’s information. Similarly, the agentServerClient will use the parameters in the message, according to the protocol shown above, to connect to an agentServer, send an initialization packet, and receive a confirmation packet. If the server is bound successfully or the client successfully connects and receives the confirmation packet, then the response packet for the web server has its acknowledgement set to “SUCCESS” and gets returned to the agentCore. If there are missing parameters in the request packet...
message, or the server or client fails to connect, then the response packet has its acknowledgement set to “FAILURE” instead.

TheCommandLine object takes the request packet message and executes it as a system command. This is the execution object to be used in order to configure the VM as well as get configuration information from the VM without having to SSH to it. The only applicable action for this is “EXECUTE”, and any other actions will return a response packet with a “FAILURE” acknowledgement. CommandLine uses the outputOfCommand object to obtain the standard output of the command. The standard output is stored in the response packet’s message, the acknowledgement is set to “SUCCESS”, and the response packet is returned to the agentCore. The outputOfCommand object has a static function that creates a pipe that issues the system command, and then reads data from the pipe into a stringstream until an end-of-file character is found. The outputOfCommand object is also used by myVMID in order to parse the VM’s MAC address from output of the “ifconfig” command.

### 7.3 Controller

Similar to the agentInterface, the main function to initialize the controller is the controllerInterface. When the controller is first started, it will attempt to connect to the web server. Once connected, the controller with bind its server to listen for agents and connect to the web server’s notification daemon to be able to inform the web server of connecting agents. After these connections have been established, the controller enters a loop to listen for commands from the web server as long as the connection with the web server is established.
The controllerClient, controllerKeyInfo, and controllerCore objects are also very similar to their agent counterparts. Rather than using a blank or preset VMID when connecting to the web server’s webserverNetwork, the controller simply identifies itself with “cntrl” in the VMID class field with all other fields set to zero. It sends the initialization packet and then receives and stores the authentication secret, but there is no exchange of public keys as it controller and web server would be configured to have each other’s keys prior to initialization.

Just as agentKeyInfo handles the agent’s public/private key as well as the controller’s public key, controllerKeyInfo handles its own public/private key as well as the web server’s public key. However, the difference is that the agent only needs to sign and send packets to the controller, but the controller signs packets sent to both the agent and the web server. As such, packets sent to the web server mirror...
the way they are sent to the controller from the agent, but controllerKeyInfo adds the function signAgentPacket and overloads the function checkAuthenticity to include the authentication secret as a parameter rather than using the privately stored authentication secret. If public/private keys were implemented, then the signAgentPacket would also take the given agent’s public key as a parameter to sign the packet.

The controllerCore distributes request packets between vmControls and the controllerServer, depending on the specified type in the packet header. Packets containing the type “VMCTRL” are passed to the vmControls object to be executed using CloudMonkey. All other types are passed to the controllerServer to be sent to the agent specified in the packet’s VMID. There is a trade-off with this design decision. By sending any packet that is not of type “VMCTRL” to an agent, the controller can waste network activity and delay the response time of packets that are simply invalid. If a packet has an invalid type, then the agent declares such and returns it to the controller, which will pass it to the web server. If the controller checked for invalid types in the controllerCore, then it would save the round trip to the agent. However, if the controller determined which packets were deemed invalid, then both the agent and controller have to be modified in order to modify the available packet types of an agent. With the current implementation, the agentCore and the agent’s back-end can be modified freely without modification to the controller. The controllerCore also contains initialization functions to be able to start the controllerServer.

The vmControls object is the only object on the controller that processes request packets. Its purpose is to provide a hook to CloudStack by using the system command to execute CloudMonkey commands. Due to configuration challenges faced in implementing CloudStack, vmControls was unable to be implemented by the conclusion of this project. As such, vmControls contains function plugs that respond to valid requests with a packet with the “FAILURE” acknowledgement and message stating “Not yet implemented.” If vmControls receives an invalid packet, it will set the failure state in the same way that the agent execution objects do.

The controllerServer and controllerDaemon share a connectedAgents object. On initialization, the controllerServer creates the connectedAgents object. Then, when starting the daemon, a pointer to the connectedAgents is passed. When the controllerServer receives a request packet, it uses the connectedAgents object to look up the socket and authentication secret of the agent specified by the VMID field of the request packet. The controllerServer then signs and sends the packet to that agent’s agentClient, and then reads and returns the response packet from the agent. If the lookup fails and there is not a connected agent that matches the request packet’s VMID, or if an error occurs while sending to or receiving from the agent, then the controllerServer creates and returns a response packet with the “FAILURE” acknowledgement and appropriate error message.

The controllerDaemon creates a threaded server that listens for agents to connect and notifies the web server when they do. Before creating the thread, the binds its server and connects to the web server’s notificationDaemon. Once both connections are successfully established, the pointer to the connectedAgents object and the sockets for the server and notificationDaemon are put into a structure to be passed in the pthread creation function. The threaded function in an unnamed namespace sets
itself to be able to be killed by its parent using pthread_setcancelstate, extracts the pointer and sockets from the struct, and then enters an infinite loop to listen for agents to connect. When an agent connects, the controllerDaemon reads the initialization packet, checks vmidNames as well as connectedAgents to confirm, deny, or assign a new VMID to the agent, then generates an authentication secret for the connection. If public/private keys were implemented, then controllerKeyInfo would also generate a new public/private key for the agent and store it in the response packet’s message field. The new VMID and the authentication secret are put into the response packet, and then the packet is signed and sent back to the agent. If the response is successfully sent, then a connectedAgentEntry is created with the agent’s socket, authentication secret, and key. The entry is added to connectedAgents via its pointer, and the web server’s notificationDaemon is sent a packet containing the agent’s VMID.

The vmidNames object is a more complex version of the agent’s myVMID. vmidNames handles two files: one which stores the VMIDs of agents that have connected to controller and therefore have a VM associated with them, and one for unused VMIDs for agents that are connecting for the first time. The unused VMIDs are added to the file prior to the initialization of the controller. When a new VMID is obtained via getAvailableID, the entry is removed from the unused agents file and concatenated to the existing agents. The function, makeAvailable, reverses this process, adding the VMID to the used agents and, if it is in the existing VMIDs, removes it from that file.

7.4 Web Server

While the webserviceInterface also contains the main function to initialize the web server, it provides much more purpose than the agentInterface and controllerInterface. The initialization process starts both the webserverNetwork and notificationDaemon. Once the controller has connected to the webserverNetwork, the webserviceInterface creates its own server to provide a network interface for users to issue requests to VMGMT. Rather than using packets to communicate, the interface uses a simpler protocol that mimics a command line. The first 8 bytes specify the length of the message, the rest of the message is a string that contains the type, action, VMID class, VMID year, VMID team, VMID VM number, and any information required to execute the request. Each value is separated by spaces and the VMID class is enclosed in quotations. The webserviceInterface parses the input to create the VMID and convert the action to the appropriate enumeration value, then makes an appropriate function call to the webserviceCore based on the type.
The controllerCore has a function corresponding to each request type. The parameters enforce whether or not the type needs the message field of the packet filled. The purpose of the webserverCore is to create a packet out of the data provided, pass the packet to the webserverNetwork to be sent to the controller, then unpack and interpret the response packet received. It is adequate for most calls to simply return a Boolean value, obtained from the response packet’s acknowledgement, to determine if the call was successful or not. The exception is the vmCommand function, which creates a packet for an agent’s commandLine. As the commandLine object can return information, such as values from “ifconfig”, the controllerCore returns a string. If the response packet’s acknowledgement is “FAILURE”, then vmCommand returns the string, “Received failing response.” If the acknowledgement is “SUCCESS”, but the packet does not contain any message, the “Received successful response” is returned. Otherwise, the packet’s message is converted to a string and returned.

All network-facing objects are straightforward, with components that were explained in the agent and controller. The webServerNetwork creates a server to allow one controller to connect, reads
the initialization packet, and then provide the controller with an authentication secret. The webServerKeyInfo contains the public/private key of the web server as well as the controller’s public key and signs and authentications packets. The notificationDaemon is similar to the controllerDaemon in that it is a threaded server. It will accept a single connection and then listen for packets. When a packet is received, the notificationDaemon converts the packet’s VMID into a human-readable string, and prints to standard output that an agent with the given VMID has connected to the controller.

7.5 Summary of Network

With seven sockets bound throughout VMGMT with one agent connected, incremented by two for each additional agent, it is beneficial to diagram and walk through where traffic occurs for initialization, sending a request, and getting a response.

![VMGMT Network Communication Diagram](image)

Figure 11: VMGMT Network Communication

On initialization, the web server is started first. The web server will start its webserverNetwork and notificationDaemon, and then wait for connection. Then the controller is started, in which the controllerClient connects to the webserverNetwork and initialization packets are exchanged between the two. After confirming the connection, the web server will initialize the server in the webserverInterface to listen for users to connect. The controller then starts its server, which starts the controllerDaemon. The controllerDaemon connects to the notificationDaemon, but no packets are exchanged. Finally, one or more agents can be started. The agentClient connects to the controllerDaemon and two initialization packets are exchanged to set the VMID and authentication secret. Once the initialization packets are transferred, the controllerDaemon sends a packet with the VMID to the notificationDaemon and stores the agent’s information to be accessed by the controllerServer. Once this has been done for all agents, initialization is concluded. However, requests can be issued to connected agents while other agents are initializing.

Before a request is issued, the agentClient, controllerClient, and webserverInterface are all listening for a packet or message. After a user connects to the webserverInterface, they send a request
following the specified format. The webServerInterface parses the request and passes the data to the webServerCore to create a request packet. The request packet is sent from the webServerNetwork to the controllerClient. The packet is distributed to either vmControls or the controllerServer by the controllerCore. If given to the controllerServer, the packet is then sent to the agentClient and distributed to the appropriate execution object by the agentCore. The execution objects process the request packet and create a response packet that is returned through the agentCore to the agentClient. The response packet is sent to the controllerServer and returned through the controllerCore to the controllerClient. If the request packet was sent to vmControls, then it is processed and the response packet is returned to the controllerClient in the same manner. The controllerClient responds to the webServerNetwork with the response packet, which is converted into a human-readable value in the webServerCore and sent to the user via the webServerInterface.

### 7.6 Use Case

While VMGMT can be used by manually issuing each request to the web server, its true potential is realized through the use of scripts and hooks to other tools. Scripting is a common practice for any repetitive task and is very popular for configuration. An easy implementation for VMGMT is to have a list of actions and a list of VMIDs, and use a script that applies the actions to every VMID. This means that the user only has to write the action one time instead of n times, where n is the number of VMs to configure. The script would report the success and failure of each call so that failures can be manually addressed by the user.

The second major use case of VMGMT is web tools. Professor Craig Shue has developed a tool, called InstructAssist, in place of WPI’s Blackboard to provide students with information and resources pertaining to his course. One of these resources is a VM console, which provides basic information of the VMs configuration and tests their ability to communicate with each other. Students can request the output from ifconfig, iptables, and route commands. This is currently achieved by using the existing PHP version of VMGMT, and can be modified to connect to VMGMT for maintained state and additional security. Student’s access to issue commands is limited by what InstructAssist’s webpage provides, and simply clicking a button on the webpage issues the appropriate number, which could be dozens depending on implementation and environment, of requests to be able to obtain the necessary information for the request.
8 Challenges Faced

Throughout the course of the project, there were several challenges which required a work-around or prevented implementation completely. These challenges occurred as either configuration issues or knowledge barriers. While CloudStack was generally regarded as an easy setup compared to several other VM management tools, there are still many steps involved in the configuration process where problems can arise. Separately, a lack in experience regarding databases and cyber security meant that VMGMT had to be implemented using flat files and plugs for future implementation.

CloudStack’s documentation has a fairly thorough guide on the installation and configuration. However, the guide focuses on a setup using multiple hosts, where the client is not necessarily on the same machine as the management server. This makes some of the steps for configuration on a single server unclear. Alternatively, a guide to create an instance of CloudStack on a single server was written Martijn Koster, who does not appear to have any direct association with Apache.17 This guide proved to work in creating an instance of CloudStack, but errors in the network configuration prevented CloudStack’s secondary storage server from being able to download ISO templates. The network configuration error may have been exclusive to the machine in this project was being tested on.

CloudStack’s template database is managed by a system VM called the Secondary Storage VM (SSVM). In order to provide the SSVM access to install an ISO, KVM required a bridge to the network device, eth0. When this bridge was created the host’s routing table would have two default routes. As a result, the first time that the SSVM queried an IP on the Internet, the request was successful, but all traffic following, including traffic from the host itself, would cause a conflict within the routing tables and be unable to reach the Internet. Removing Ubuntu’s Network Manager and attempting the set up the routing tables manually did not resolve this problem. However, this configuration was done with limited experience in networking and can most likely be achieved by somebody with intermediate to expert knowledge in networking.

Due to the inability to create a template, CloudStack was setup only to a point of testing the user interface. Therefore, in the implementation of CloudStack support in VMGMT had to be provided simply as a plug. The tools are available for such an implementation; however the configuration challenges made anything that could have been written untestable. One goal for VMGMT was to support public/private key authentication to add security to a system that currently does not have any. VMGMT was implemented with all of the plugs necessary to implement this degree of authentication, but with no previous experience in public/private keys, there was not enough time within this project to fully implement this. Finally, the lack of experience with databases was addressed by simply using flat files. Using flat files as storage is very inefficient, as removing an entry from the file requires copying every entry except for that one. This applies to the vmidNames object handled by the controller. With databases implemented, vmidNames would use a databased for unused VMIDs and a database for existing VMIDs.
9 Future Work

VMGMT introduces the core functionality of a powerful tool for WPI’s CSNW, but there are many improvements that can be made to improve its security, efficiency, and functionality. The most obvious features to be implemented to VMGMT are those mentioned in the previous section. Additionally, the use of shared pointers can improve the efficiency of function calls while removing concerns of memory leaks. The VMID is designed in such a way that requests could be sent to multiple VMs by specifying a subset of VMID fields.

As described in the challenges faced, there are three features that take priority in future work. First, with a fully functional CloudStack implementation, the plug for vmControls can be filled in to support the ability to up and down VMs. This can also be expanded to make modifications users and teams, depending on necessity for access. The second feature to address is VMGMT’s security. The plugs for authentication do not check the authentication secret, and public/private key encryption is not implemented. With understanding of cyber security, there is a lot of improvement that can be made to ensure that devices are not able to have unauthorized access to any of VMGMT’s components. Finally, replacing the flat files used by the controller would dramatically increase efficiency of the controller for connecting agents.

The only place that an object is created and deleted is the structures that are made as parameters for the threaded servers. If the object is not allocated, then its contents are lost by the time the threaded function tries to read its contents. By using Boost Shared Pointers, or C++ 11’s shared pointers, the need to manually delete any objects would be removed. This guarantees that the system would have no memory leaks. This can also be used to dramatically improve efficiency regarding packets. All objects that return vectors or packets mean that those objects have to be copied between functions. The current implementation prefers no memory leaks over guaranteed efficiency, as VMGMT should be able to run indefinitely. Shared pointers can be implemented to pass packets and objects between functions so that they are only constructed once per component with the guarantee of getting deleted when they are no longer in use.

The webServerInterface currently accepts one connection at a time, which is not ideal if multiple classes and clubs will want to use it. There are two solutions to distributing access to VMGMT. A separate distribution component can accept multiple connections and forward all requests to the webServerInterface or the webServerInterface itself can be modified to support multiple connections. This would best be done using a pthread mutex in order to provide first-come-first-serve access to VMGMT. A round-robin service approach can be achieved simply by cycling through all connected sockets and checking for requests. This avoids threading and mutexes and guarantees fairness between users, but decreases efficiency.

The final suggestion for future work is to add a mask to the VMID to be able to configure multiple agents at once. The API would need to be able to clarify whether the information received is a VMID mask or simply an incomplete VMID. The idea of the mask is that a user could issue a request to a
specific class and year, and have the request sent to all agents that match those fields. This would remove repetition from the user and build it into the controller. A concern with this is having a mask that is too vague that impacts more than just the target VMs, which could interrupt the work flow of individuals who were not intended targets of the command.

10 Conclusion

Of the plethora of virtual machine management tools available, Apache CloudStack meets the requirements of WPI’s CSNW without the configuration and user interface of large scale cloud tools. Additionally, the agent-based architecture of VMGMT provides limited access to CloudStack’s VMs from outside of the CSNW’s isolated network, without exposing the VMs themselves to WPI’s network and the Internet. This allows the use of scripts to configure VMs as well as access for tools such as InstructAssist to be able to provide students with information from their VMs. The VMID introduced for VMGMT is intuitive for a university environment and versatile for clubs and classes over multiple years. Its object-oriented approach clearly separates code into objects for communication, distribution, or execution, providing a strong base for expanding functionality if additional features are added in the future. The common objects are used to enforce network protocol across the three components and the all network configuration is established in each components interface, rather than across multiple different objects. The object-oriented approach also provides superior tools for the design and demonstration of VMGMT compared to functional programming by using class diagrams. Compared to the existing system in place, VMGMT and CloudStack dramatically improve the user management of the CSNW, allow agents to retain state between connections to give VMs more permanence, and provide the ability to improve the security of communication between configuration components.
References


   <http://archipelproject.org/index.php>

   <https://github.com/ArchipelProject/Archipel/wiki/User-manual:-vmcasts>


   <http://www.ovirt.org/OVirt_Administration_Guide#oVirt_User_Properties_and_Roles>


   <http://www.greenhills.co.uk/2013/08/30/cloudstack-single-server-on-ubuntu-with-kvm.html>