

THE GRID OVERLAY SYSTEM MODEL

by

Levi Daniel Junkert

A thesis submitted in partial fulfillment  
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY  
Bozeman, Montana, U.S.A.

April 2009

©COPYRIGHT

by

Levi Daniel Junkert

2009

All Rights Reserved

APPROVAL

of a thesis submitted by

Levi Daniel Junkert

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the Division of Graduate Education.

Dr. Rockford Ross

Approved for the Department of Computer Science

Dr. John Paxton

Approved for the Division of Graduate Education

Dr. Carl Fox

STATEMENT OF PERMISSION TO USE

In presenting this dissertation in partial fulfillment of the requirements for a Master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. I further agree that copying of this thesis is allowable only under the Creative Commons License, or also known as "Copy Left". Requests for extensive copying or reproduction of this thesis should be referred to at the Creative Commons group website at [Creativecommons.org](http://creativecommons.org).

Levi Daniel Junkert

April 2009

DEDICATION

To my Mom and Dad,  
Thank you for all your support and love;  
I could not have done it without you.

## ACKNOWLEDGMENTS

First I would like to thank Dr. Rockford “Rocky” Ross. He has been an integral part of my education here at Montana State University, and has been a wonderful mentor and advisor who has never seen my ideas as “crazy” and has never viewed them as impossible and unreachable. His years of experience not only in computer science, but also his experience as an editor in the computer science publication community has helped me shape this thesis from a mound of mud to a well structured concrete thesis. Without him this thesis would be impossible to follow and read.

I would also like to thank Ivan Judson for his help on the more technical issues and the large mountain of problems I had to climb to complete this thesis. Without his knowledge and help on this subject, the problems faced would have been like climbing Mt. Everest without a guide.

Third, I would like to thank the Center for Computational Biology, Computer Science Department, Academic Computing Group, Global Student Labs, and Information Technology Center of Montana State University. Without access to their expenditures, computational resources, and the occasional, “can you get this done me?”, I would have not been able to even imagine completing this thesis. There are too many names to list here, so I thank everyone in the departments listed above for their help, patience, and graciousness.

Last, I would like to thank my family. I know for certain that they played an indispensable role in my mental, physical, and spiritual strength while I worked to complete this thesis. Every time I was in the middle of a storm and did not know who to turn to, they were always there. Thank you for the love and encouragement that I am so grateful for.

## TABLE OF CONTENTS

1. INTRODUCTION . . . . .	1
2. PREVIOUS WORK . . . . .	3
3. METHODOLOGY . . . . .	7
Problem Definition . . . . .	7
The Grid Overlay System Model . . . . .	9
Overlay Layers . . . . .	11
Resource Layer . . . . .	14
Operating System Layer . . . . .	16
Network Layer . . . . .	17
Services Layer . . . . .	18
Application Layer . . . . .	20
User Layer . . . . .	20
4. RESULTS . . . . .	21
The Montana State University Grid . . . . .	21
Resources . . . . .	21
Operating System Layer . . . . .	21
Network Layer . . . . .	22
Service Layer . . . . .	23
Application Layer . . . . .	24
User Layer . . . . .	24
Run-time Statistics . . . . .	25
5. CONCLUSIONS . . . . .	27
Future Work . . . . .	27
REFERENCES . . . . .	29

LIST OF TABLES

Table	Page
4.1 3D Studio Max Render Times . . . . .	25

## LIST OF FIGURES

Figure	Page
3.1 Grid Overlay System Model . . . . .	10
3.2 GOS Overlay Layers . . . . .	11
3.3 Example VLAN Topology (Red Nodes) . . . . .	12
3.4 Example NAT Topology . . . . .	13
3.5 Example Bridged Topology . . . . .	13
3.6 Operating System Overlay Layer . . . . .	16
3.7 Network Overlay . . . . .	18
3.8 Service Overlay Layer . . . . .	19
4.1 Sample Building . . . . .	26
4.2 Moving Robot . . . . .	26
4.3 Simple Quick Scene . . . . .	26

## ABSTRACT

The grid overlay system model is a new technique for forming a grid computing model for research computing. In this method we construct a grid that is dynamically allocated from a set of resources in a unique and progressive manner. This new system model allows for construction of virtual environments for execution of applications on many diverse shared resources. The system can dynamically scale to create a range of resources from a single machine to a virtual cluster of machines. This model provides a virtual container that can run legacy and customized software in an emulated environment or directly on the host's hardware through virtualization. Using this model on current consumer hardware allows for a unique blend of software containment with dynamic resource allocation. Our model, in combination with commercial off the shelf (COTS) hardware and software, is able to create a large grid system with multiple combinations of hardware and software environments. In our model we propose a unique set of abstraction layers for systems. The combination of our model with current consumer hardware and software provides a unique design principle for addressing grid implementation, hardware reusability, operating system deployment and implementation, virtualization in the grid, and user control techniques. This provides a robust and simple framework that allows for the construction of computational research solutions in which performance can be traded for flexibility, and vice versa. Our model can be applied to computational research grids, service oriented grids, and even scales to collections of mobile or embedded system grids.

## INTRODUCTION

Large computational systems are located all over the world in places such as governments, universities, international businesses, and even small to medium sized businesses. Most of these systems are excellent examples of computational resources that are idle for most of their lives. Using these computational systems in a grid architecture will add resources to current research at many institutions. Many of these institutions use a frequent upgrade schedule that is on a triannual, biannual, or even annual replacement schedule. With this short technology life cycle comes the application of exciting theories in computer science. We can see these techniques being applied in technologies today, such as virtual machines, computational grids, and virtual networks [13]. With the latest advances in electronic and microcomputer architecture, we have seen the addition of microprocessors in everything from wi-fi capable wrist watches to IBM's Roadrunner Cluster topping out with over 1 petaflops [7]. The scalability of networked systems has evolved to an interconnection of all electronic devices through the Internet and wireless/wired providers and in return created a global interconnection of systems of all sizes. These new technologies allow the interconnection of remote locations of our solar system to even personal cellular telephones and personal data assistants that we use every day. With these advancements, new systems are being built around virtual operating systems and hardware technologies like Intel's Virtualization Technology(VT) and Analog Micro Device's Virtualization (AMD-V) [2].

With the advent of the virtualized x86 architecture in hardware, which is in most of today's modern workstations and servers, we can now propose a new solution for extracting unused cycles from idle workstations and servers, securely. We propose a simplified process for harvesting the maximum use of computational resources at Montana State University from student labs across campus. In our solution we use grid computing software in

combination with virtual networks and our proposed multi-layer grid technique. This technique allows one to run multiple operating systems on multiple architectures at the same time with minimal overhead and in a more secure environment. The method proposed includes a simplified layering technique that might not be useful in just our example case, but networked systems of any size. By using this method we believe one will be able to build a network, or virtual network, of resources already in hand.

In developing our technique we recognized that freely distributed software is often a requirement of the research community. Therefore we chose to use software that is freely obtainable for non-profit and research use. By using this collection of proposed software, one is able to build a sample network on a single workstation (or multiples) with an Intel or AMD virtualization enabled processor with minimal system and software cost. Even if VT or AMD-V is not available, one is still able to run the virtualized system. This method is possible at the cost of a significant increase in running time and memory size. By utilizing software packages such as Condor, Sun Grid Engine, Kernel Virtualization Machine (KVM), and QEMU, we are able to test our hypothesis without the need for expensive commercial software.

Our method allows for design principles to be put in place at any institution that wishes to harness the power of idle workstations and servers for computational research. Our grid overlay technique allows for dynamic, secure, and flexible allocation of resources for computational research. With our model we are able to construct large grid systems with maximum software abstraction.

In the following sections we will discuss previous research in grid computing with or without virtualization, our own formal problem definition, our proposed methodology, results, and conclusions. In our final section we will go over possible future work that can be based on this thesis.

## PREVIOUS WORK

Grid computing in today's environment is handled by customized middleware that is complicated to run and requires a large group to administrate [12]. With the advent of virtualization as a tool for grid system administrators, we are now able to deploy grids that harvest resources in a virtualized hardware environment, thus increasing the complexity of grids, also known as "The Grid" [13]. By using virtualization, virtual networks, and virtual data storage, we are now able to build large autonomous systems that grow and shrink dynamically to meet the resource demands required by a specific group or job. This method, also known as "The Cloud," is built of a collection of dynamic and statically allocated grids that increase administration and implementation complexity to an interesting new level [8]. With this automatic resource allocation method, one is able to add new services and techniques that were not available before.

A great example of grid use today is the customized systems from companies and researchers around the globe: Google File System, Bio Informatics Research Network (BIRN), CERN, and many others [11]. This grid oriented architecture has allowed companies such as Google to have large data farms with minimal overhead and simplicity in administration [14]. Another company's commercial software on the forefront of "The Cloud", is Amazon.com's Elastic Compute Cloud (EC2). This grid based network of resources allows Amazon.com to resell virtual server resources at the cost of use, such as bandwidth, CPU cycles, and memory usage. This revolution in system service providers has allowed many companies to cut expenses in server hosting costs and IT overhead.

With virtualization we are now able to emulate, and in some cases directly execute, various different operating systems simultaneously on a single computer. Hardware virtualization support has been added into many new CPU architectures such as Intel x86 and

IA-64, AMD VT , and Power PC [6, 4, 5]. This new technology allows many institutions to run multiple servers from one single resource, therefore saving money by consolidating hardware. This helps in cutting cost and administration overhead as well as minimizing down time and securing operating environments. With a minimal amount of memory and CPU overhead, one is now able to run multiple services from these devices. By consolidating resources, virtualization has also allowed many companies to move to a more relaxed and stable upgrade path for software. Since the addition of virtualization features to many popular operating systems, many in the information technology area have viewed this move as beneficial for upgrade paths and minimal down time of services. One is able to change system services with minimal down time through live migration of virtual machines [3].

With the addition of hardware virtualization to architectures such as Intel's Virtual Technology for IA32, x86-64, and IA64, we are now able to have jobs that require high performance computing contained in virtualized environments. By adding virtualization features to the already famous x86 architecture, Intel has created a hardware solution for institutions desiring to make use of virtual machines. This now allows institutions the ability to run performance software in static virtualized containers. This has allowed for the harvesting of many idle CPU cycles in workstations and servers through grid networks. By integrating the design techniques of grid computing with virtualization, one now has the ability to queue on demand services based on resources needed by the applications at any typically idle workstation or server [13].

Resource centric grid computing is a method of distributed computing that has been researched for many years and is finally finding its place in consumer products, such as Amazon EC2. Middleware for grid systems, such as the Condor project at The University of Wisconsin-Madison, Sun Microsystems Grid Engine, BOINC, and the Globus Toolkit, are solutions that many researchers have used in the past to build grid systems [1]. There are also a variety of custom research applications such as SETI@Home and Folding@Home

that use a screen saver, or a low priority background application, that runs on resources in a pool [10]. All of these middleware control applications allow for harvesting of resources by simply adding a customized tool set for grid administrators. Some of the basic tools included are queue management, job preemption, resource management, and data management. With these services one is able to implement high performance computing on computational systems ranging from a simple workstation to complicated large system clusters.

Combining virtual machines with grid management is a technique that is relatively new to computer science, emerging in the past two decades. Other systems, such as the In-VIGO system from the University of Florida, have integrated this technique into what they call a "middleware solution" [13]. By adding this middleware solution, they are able to easily administer virtual machines that are dynamically allocated from a pool of resources. By keeping the individual resources as virtual machines they are able to start and stop a system with minimal notice and migrate the system to a new resource by way of "snap shooting," a novel technique that involves freezing the state of memory, CPU registers, interrupt vectors, and the rest of the current system state by saving them to an instantaneous machine image. By snap shooting, one can now easily migrate a virtual machine from one resource to another with minimal down time. One can see the obvious values of this technique. For example, if a workstation is not being used because the user is on a lunch break or home for the evening, that workstation can be used for other purposes within the grid without affecting the user's experience on the resource. By developing rules for recognizing when a workstation is available, such as keyboard and CPU idle time, one is able to schedule jobs on resources securely in their own environment without adversely affecting users. Once a user returns to their workstation, a snapshot of the virtual machine is taken and the virtual machine is migrated to another free workstation or is put on hold until another workstation becomes available.

Although the combination of grid computing with virtual machines is powerful, many have overlooked the simplicity in the abstraction it allows for resource allocation. The current middleware systems are centralized for a single grid computing environment on a large variable network. Until recently, virtualization in the grid environment has been minimal and only being used in research and large corporations such as Google and Amazon.com [14, 9]. In the next section we describe our approach to harvesting the unused CPU cycles on the many public workstations in the Montana State University Global Student Labs (MSU-GSL).

## METHODOLOGY

In this section we describe our problem by providing a formal problem definition and describing our solution to this problem. Although our model is unique, the methods described here include some that have been used in software engineering for the past decade. The term *cloud computing* refers to a unique method specifically targeted at high performance research computing. Although this model was designed for high performance research computing, it's possible to also apply this model in other areas that are outside the scope of this thesis.

In the following we use the term *resource* to represent a single workstation. This resource may contain multiple processors or processing cores and large amounts of memory.

### Problem Definition

One unique requirement of the Montana State University (MSU) Global Student Labs (GSL) is that students must be given the highest priority for the resources they are using. The GSL also has its own integrated network that spans across campus. This includes specific network security restrictions, infrastructure equipment (routers, switches, servers, etc), and physical network separation via its own fiber and wired infrastructure. Thus, if we are to harvest unused computational cycles and memory from the GSL, we must be able to queue jobs dynamically according to whether a resource is currently in use by a student or not. We can determine whether a resource is in use or not based on the idle state of the resource or other attributes of the resource. One possible solution is to queue jobs while the student is at the resource, but restrict the amount of resources that can be used by a specific job depending on specific resource attributes.

By using virtualization with grid computing techniques on the GSL we can harvest

computing cycles that would otherwise be wasted. Use of the GSL for computational research on campus is in increasing demand by the research staff of MSU.

To accomplish our goals we need to specifically define our problem. With the increased network security, group and user policies, and unreliable system up-time or dedication of each resource, we must be able to:

1. Suspend jobs according to the allocated resource state. (e.g. if a student logs onto the resource while an external job is running).

- (a) Can we base resource usage determination on human activity at the local side of the resource?

2. Resume jobs without losing valuable data.

- (a) Can we resume jobs depending on how long the resource has been idle?

3. Run any combination of selected hardware architectures and operating systems on a GSL workstation according to application requirements.

- (a) Are we able to queue jobs that can run on multiple hardware and software architectures?

The above listed requirements and problem definition lead to our conclusion: there is an underlying design principle that is missing that needs to be represented correctly. Cur-

rent software engineering designs in grid computing and virtualization do not account for the abstraction of multilayer grid systems that can run dynamically and be constructed recursively while providing the most flexibility for an application's required hardware and software environments. There have been successful attempts at such systems, but their underlying infrastructure is confusing and requires specific system administration knowledge [13, 3, 9]. Dynamic and recursive grid layering is a concept that we propose will lead to a solution. Our solution allows for the allocation of resources dynamically and recursively according to the required application environment.

### The Grid Overlay System Model

Our idea is to introduce the concept of a dynamically and recursively allocated grid overlay system by connecting multiple middleware grid applications to form a dynamic grid that is able to run according to all the requirements listed in the problem definition section. With our solution, we are able to make the MSU-GSL available to researchers on and off the MSU campus. We do this by using a combination of overlay layers that we describe in the overlay layers section below. By using these techniques we are able to complete the construction of another grid, which we will call the overlay grid, that is built of virtual machines. The original grid is used to construct a dynamic allocation of virtual machines which then connect to the overlay grid. This allows for dynamically allocated virtual machines from the initial grid that are viewed as physical resources on the overlay grid.

First we must define our layering technique. We borrow our basic concept of service layering from the OSI networking model [15]. Our model (see figure 3.1) is similar to the OSI model in that movement only occurs between adjacent layers. For example, skipping the operating system (O/S) layer and going directly from the resource to the network layer

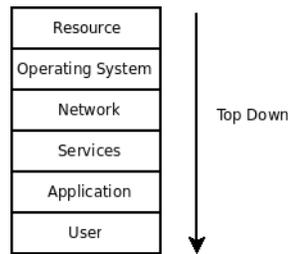


Figure 3.1: Grid Overlay System Model

in the model does not make any sense. This is because we must have an operating system present to be able to access network services and vice versa. Although we use the basic layering concept inside each resource, there are also modifications to the OSI networking model that need to be mentioned. The first difference between the OSI model and the Grid Overlay System (GOS) model is that each collection of layers is not a system, but a resource. The second that some of the layers in our model are recursive. These layers are further referred to as *overlay layers*. By this we mean that specific layers in the model can themselves be composed of other GOS models. These specific layers can be very detailed in how they are implemented, but they still abide by the basic concept of moving from one layer to another without skipping other layers.

By following each layer we are then able to consolidate complex processes into a single layer for overall system simplification. For example, in the case of virtualization we can view a single virtual machine as an overlay layer at the operating system layer (i.e. the virtual machine is a service running on its host; this virtual machine has users, applications, a network, an operating system, and a virtual resource). By having virtualization applications such as KVM or QEMU we are thus able to include a full GOS model at the operating system layer via virtualization. By following this simplified systems engineering process we are able to constrain individual problems to their respective layers. For example, we may decide that we want to run a virtual machine that contains another emulated system. This would mean that the original resource would contain 3 overlay operating system lay-

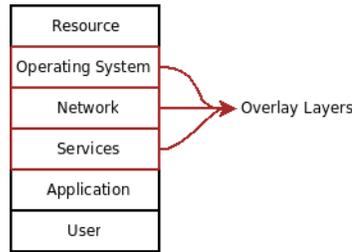


Figure 3.2: GOS Overlay Layers

ers. Therefore, we see that we can have  $n$  operating system overlay layers may exist on the host. One can now see that the depth of an overlay approach is *unbounded* and is limited only to the physical resource's assets, such as RAM, processor cycles, and disk space.

### Overlay Layers

Overlay layers in the GOS model are defined to be abstraction points for possibly complex systems that can be contained in a single layer. A good example of an overlay layer is one in which a virtual machine is serving resources to the application layer by way of dynamically allocated resources from the services layer. For example, a web server may be dynamically allocated on a grid from an idle resource to serve a web application to its users via a virtual machine. This would be considered a overlay layer because the web server itself is a service that requires previous levels of network, operating system, and resource layers. So by our definition of overlay layers we are logically able to conclude that the operating system layer (through virtualization), the network layer (through Network Address Translation(NAT) or Ethernet bridging and many other techniques), and the services layer (by way of virtual machine spawning through grid services), are all layers that are contained on a single resource.

The OS layer may contain multiple virtual machines through direct virtualization. In modern systems, a single resource may have multiple operating systems installed and managed by a hypervisor. The hypervisor acts as a bottom level operating system. It contains a

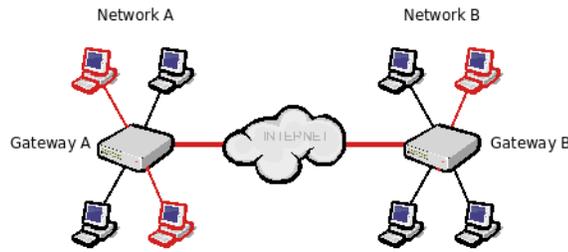


Figure 3.3: Example VLAN Topology (Red Nodes)

scheduler for virtual machines that can be hosted at the operating system layer. The hypervisor is the master control mechanism for the resource that allows for containment of each individual virtual machine and schedules their computational time, memory usage, and disk usage on the single physical resource according to predefined configurations. Through this method we view the operating system layer as an overlay layer.

We can view the network layer as an overlay layer through which many techniques allow for virtual network systems. One example of virtual networks is the virtual local area network topology. This topology allows for the creation of a virtual network that is physically contained inside a larger physical network topology. One can imagine a physical network that connects multiple resources together. If we wanted to create a subset of resources from that physical network, that are virtually connected through specific routing techniques, this would be considered a VLAN (see figure 3.3) .

Another technique used commonly in network topologies is network address translation (NAT). NAT is a topology in which one physical network is connected to another physical network through a gateway. This means that all addresses on the two separate networks are hidden from each other. In other words, the only way for two resources that are separated in each of the physical networks to communicate is through their own gateways (see figure 3.4) .

Bridging is another example that is commonly used to assign multiple addresses in a physical network to a single resource. This is very useful for virtual machines in that

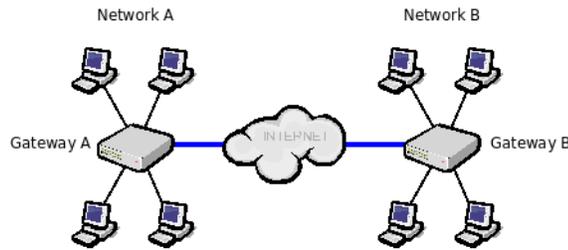


Figure 3.4: Example NAT Topology

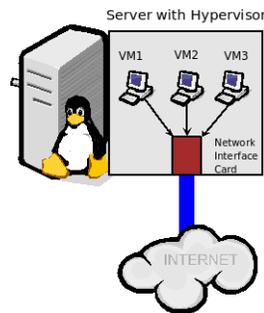


Figure 3.5: Example Bridged Topology

a hypervisor can bridge network interfaces (through I/O redirection) directly to a virtual machine from the physical network. This allows a virtual machine to appear as a unique physical resource to the rest of the network (see figure 3.5) .

We are able to view the network layer as an overlay layer because of the capacity for creating virtual machines that are viewed as individual physical resources to the network. As one can see, the underlying network structure can be very complicated in the network layer. We hide this complex structure by using the network layer as an overlay layer as well.

Viewing the service layer as an overlay layer allows for a unique way of building structures commonly referred to as *the cloud* or *cloud computing*. Using grid computing models one can harvest idle resources from a group of resources, such as a cloud, and in return use these resources according to their idle time, specific processor usage, memory usage, disk usage, and other resource specific statistics. One can see how it would be beneficial

to queue virtual machines on these idle resources for various uses by way of grid models. Once a resource reaches an idle level that makes it available for grid computing use, a virtual machine could be queued to it. Through this technique we view the service layer as an overlay layer.

### Resource Layer

Now that we have explained the basics of the layering technique, we can now provide detail explaining each layer's specific abstraction technique. In the case of the resource layer we are considering a resource to be a physical, or virtual (in the case of a overlay layer), computer. The resource layer therefore contains specific details about a resource, such as its RAM size, RAM usage, CPU frequency, CPU usage, disk size, disk usage, and many other resource statistics. By defining these statistics in the resource layer we are able to abstract a full physical or virtual system with *unbounded* details as a single *resource*.

Each resource is treated as a separate entity, or in other words a single computer, which may have multiple processors or cores. For example, in the GSL there will be more than one resource present in each lab. In other words every physical computer is considered a single entity, and therefore a single resource. In the MSU-GSL, one lab may have up to 70 or even more resources in a single location, each with 1, 2, or more processors. This focus on each individual resource allows for separation of resources at the physical hardware level and will allow us to set specific rules for allowing external jobs to run on a workstation based on individual workstation statistics.

We use the resource layer to define specifics about each workstation or server and to identify not only the hardware statistics, but also the capabilities of the system and its day-to-day use. A simple question and answer session, like the one described here, can be used to describe each resource and therefore show specific characteristics about it.

## 1. Statistical Questions

- (a) How fast is the processor?
- (b) How much Random Access Memory(RAM) does the resource have?
- (c) How much disk space is available for use?

## 2. Use-case Questions

- (a) What times of day is the resource available?
- (b) Can the local user reboot the resource at will?
- (c) Can the resource be used for computational research while a local user is using the resource?

From these listed questions one is able to determine specifics about the individual resource and make informed decisions about the remaining layers of the model based on this information.

Another issue at hand is whether the resource is able to run a virtualized environment with minimal overhead. For this to be possible the resource must have specific hardware that is able to run another operating system in a virtualized environment with minimal resource overhead. If the hardware supports VT, described earlier in the previous work section, then minimal overhead will result by running multiple operating systems in virtual environments on the resource. If the resource does not have VT it is treated as one that may require an emulated environment and thus could result in a large amount of overhead

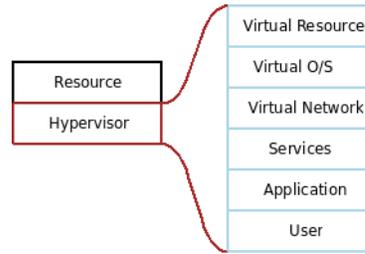


Figure 3.6: Operating System Overlay Layer

when running a virtual machine. This overhead comes from the requirement to emulate a hardware system in software. In such cases registers, I/O, and memory addresses that are implemented in hardware a physical CPU need to be emulated in user space for the virtual machine running on a system. An example of required emulation would be running precompiled PowerPC software on a Intel x86 architecture. Since the PowerPC machine code is different than the Intel x86 machine code, a program (emulator) must be available (in Intel x86 machine code) that interprets the PowerPC machine code. Execution of the interpreter program introduces substantial overhead.

### Operating System Layer

In the operating system layer we specify what operating system is installed that has direct access to the resource. This layer can be considered an overlay layer through which access to virtual resources through virtualization or emulation may occur. Hence, with the advent of virtualization, via a virtualized-based operating system (also known as the hypervisor), we are now able to host multiple operating system environments on one physical resource.

The operating system layer is intended to provide a simplified means for virtualization by the hypervisor. Using a hypervisor to control the hosting of multiple virtualized operating systems allows for many systems concurrently. This means that the resource's hardware registers, I/O, and memory can now be used by multiple virtual machines on a

single hypervisor. This scheme allows a virtualized operating system to dynamically allocate processor cycles, RAM, and disk from the physical resource transparently via the hypervisor Int [6], AMD [4].

A great example of service virtualization, controlled by a hypervisor, is a web server. Most web hosting computers today host multiple websites on one physical resource at the service layer using virtual hosts. Without virtualization, hosting multiple websites on a single computer was in the past prone to security issues. For example, if an attacker hijacked a single website because of a security flaw, it is also likely that the attacker could affect other hosted websites because the entire server is now vulnerable. In the case of virtualized operating systems, an attacker can now only affect a single web server because the actions of the attacker are restricted to the single virtualized operating system.

### Network Layer

The network layer can include many different details about a resource's physical connection and communication with other resources. This layer gives specifics on how data is transferred between different resources, including virtual machines. In the case of non-virtualized resources, one could envision a network that physically connects multiple resources for communication. For example, in modern day networks the task of transferring data on a network is handled by Ethernet and TCP/IP protocols. By using the network layer, we are able to connect multiple resources. This collection of resources can be considered as a pool of resources.

Through the common standard protocols TCP and IP we are able to interconnect virtual machines and non-virtualized machines through various techniques. By using bridging, NAT, VLANs, TCP hole punching, VPNs or other well-known techniques such as semantic overlay networks, we are able to blur the distinction between virtual resources on a network and physical resources. With these and other network techniques we can even break apart

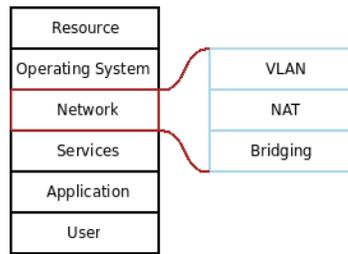


Figure 3.7: Network Overlay

the basic conception that a LAN must be contained inside a distinct physical space. With NAT we can hide the fact that a single resource is running multiple virtual machines or even multiple physical machines as a gateway or router. In return, this allows the network to be available to not just one physical machine but an *unbounded* number of virtual machines hosted on the same resource.

These techniques thus allows us to connect virtual machines directly to a network as if they were individual physical resources, when in essence they are just a virtualized environment running on a single physical resource. In recognizing that we are able to manage data on multiple machines, whether they be physical or virtual, shows this is an overlay layer in the GOS model. By incorporating virtual networks into today’s modern network infrastructures we are now able to view virtual machines as equivalent to physical machines when it comes to network topology. Because of techniques such as bridging, NAT, and installation of multiple physical network interface cards (NICs), we are now able to create networks that involve not only physical machines but virtual machines as well. By keeping our network layer as an overlay layer we can start using techniques such as self organizing peer-to-peer overlay networks.

### Services Layer

The services layer is a key layer in the GOS model and is where the “Grid” distinction of GOS is initiated. This layer contains services such as grid control middleware, web ser-

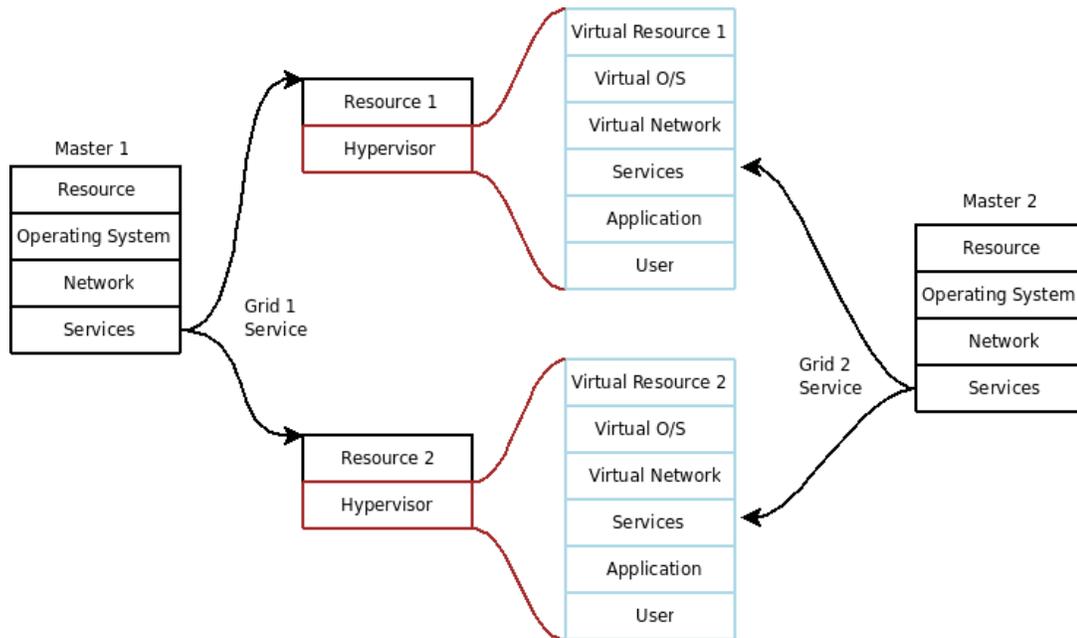


Figure 3.8: Service Overlay Layer

vices, file transfer services, and other required services to complete the task of organizing the connection to previous layers to pass on to the application and user layers. This layer is key to adding the overlay capabilities to the GOS model and is the last overlay layer in the GOS model.

At the service layer we integrate simple tasks such as resource allocation, data handling, and resource scheduling. This layer is where all middleware grid applications, mentioned in the previous work section, exist. Here resources are pooled and services are extended to the application layer for use by the user. This last level of recursive abstraction can be used to queue virtual machines to specific resources based upon resource details outlined in the resource layer section. This layer's abstraction is used to create a large complex system that appear simple on the surface.

By using grid middleware applications we are able to connect or reconnect a virtual machine to the same grid network or a different network entirely through the overlay nature of the network layer. This makes grid overlay possible. By dynamically allocating virtual

machines we can use the service layer to run applications on virtual machines selected from a dynamically allocated pool of virtual resources (see figure 3.8 for a visual representation). This provides an advantage for systems that must dynamically change to match varying resource loads and resource availability. With this technique a resource pool is directly scalable to many changing factors in the physical computing environment in general. This technique is referred to as *Cloud Computing* [8].

### Application Layer

The application layer is targeted to a specific user or group of users of a specified service that was defined in the service layer. This is where custom or prebuilt applications are set to interact with the user directly. Examples of such applications in this layer include web applications similar to G-Mail, Google Docs, or other control applications. All such applications hide the underlying layered support structure from the user. Obviously, abstraction allows the underlying system to be hidden from the user through the cloak of the front-end application. This layer is very important in that it hides specific details of the underlying implementation from the user and simplifies the design of the overall system.

### User Layer

The user layer is specifically tuned for a user's interaction with the GOS model. This can be specified for human interface devices that allow the user to interact with the actual applications. Examples of user layer interaction devices are cell phones, personal computers, workstations, and even dumb terminals. These "end of the line" devices allow users to interact with grid resources through the proper layers of abstraction in the GOS model. Through proper software development in the application layer, one could run any application they would like on the grid without having to know the underlying infrastructure and complicated grid overlay system.

## RESULTS

### The Montana State University Grid

The GOS model was first put into use when the dynamic grid for the Montana State University Global Student Labs (MSU-GSL) was constructed. Our system model was used to build a dynamic computational environment for use in both student and faculty research on campus. By using idle time on workstations in student labs we were able to harness the power of over 360 computers with multiple cores that have an overall floating point operations per second (FLOPS) of 1.4 teraflops. This massive amount of computing power allows for a range of valuable uses not only in research computing, such as is required by the Colleges of Letters and Sciences and Engineering, but also as needed for render farms that are in use by the College of Art & Architecture. We are able to demonstrate that the GOS model works well in these environments.

### Resources

The MSU-GSL contains many computers from Dell Inc. that are cutting edge workstations. The labs are also on an upgrade schedule that ensures that the computers are updated every four years. Through this replacement strategy, we are able to keep our systems up to date. The GSL has 13 labs, as well as randomly placed computers in the Student Union Building (SUB). The labs contain a range of 15 to 70 workstations that can be used by any enrolled student on campus. The labs resources are available both for students using them for individual homework and assignments and for formal instruction.

### Operating System Layer

MSU's GSL provides a single operating system for student use. Given that Microsoft's Windows XP operating system is widely used in work environments around the globe,

MSU-GSL has chosen this operating system as its main choice for students. This is a common choice in academic institutions. Because of this choice, we are constricted to the Windows XP operating system with KQEMU as our only possible open source solution for running virtual machines in the GSL.

To take full advantage of the benefits offered by hardware virtualization, we have to enable hardware virtualization through a plug-in technique, using Windows XP as the hypervisor. To accomplish this task we use QEMU, which is an open source emulator that has been modified to use kernel hardware virtualization in Windows XP. The hardware virtualization method provided by QEMU is called KQEMU, where the K stands for kernel level virtualization. By simply installing a dynamic linked library (DLL) file in Windows, we now able to take full advantage of kernel based hardware virtualization. As explained in the previous work section we are now able to use virtualization with minimal software overhead. QEMU is used as the application interface for the virtualized container making it the primary hypervisor interface.

### Network Layer

The MSU-GSL network is a self contained network that connects multiple labs across the MSU campus. Labs are distributed across campus housed in many different buildings. This placement is intended for the convenience of faculty and students.

Since the GSL network is housed in separate buildings, an underlying network infrastructure must be present to allow for connection between resources. In the GSL case, all labs are connected through a fiber network that returns to a central location on campus. This network is a high-speed TCP/UDP/IP network with public IP addresses. This public address space has advantages and disadvantages. One disadvantage is that with global IP addresses there is a limit as to how many IPs are available. To handle our issue of installing multiple virtual machines on each physical resource in the GSL we decided to use simple

port redirection via our hypervisor's control mechanism and scripting. In other words, if a specific virtual machine needs access to specific TCP and/or UDP ports, then the hypervisor is set to redirect these ports to the VM at VM boot time through command line options. This technique requires that we redirect these ports ahead of the service layer integration to allow for access to our VM from the public IP network. Our first proof of concept was scheduling a Linux VM to open Secure SHell (SSH) access through TCP port 22 and attempt to gain access to this virtual machine through the SSH client. This was a success and showed that port forwarding was working properly in KQEMU. Since we are able to redirect ports from the hypervisor to a VM, we are also able to add specific network services to the service layer through this redirection technique.

### Service Layer

The MSU-GSL service layer is completely handled by the grid application middleware Condor from the University of Wisconsin Madison. This software package allows for dynamic queuing of jobs through the simple grid architecture mentioned in the previous work section. The software requirements for Condor include a master system that keeps track of resources, queues, and jobs. With this software we are able to centrally locate all applications, data files, and results for jobs, and allow access to every workstation on the GSL network for computational needs. Using Condor we are able to queue multiple jobs across our GSL network using simple grid queuing and management.

From this layer we recursively queue another resource layer using the techniques described in the service layer section. By this method we are now able to queue multiple VMs that have multiple emulated/hardware and software architectures. By utilizing the overlay method at the service layer we are able to build another overlay grid by using Sun's Grid Engine (SGE). This allows us to create an entirely separate set of queue-able VMs connected by another middle-ware grid application for dynamic allocation of virtual resources

on multiple operating systems, including Ubuntu Linux, Windows 2003, and Windows XP. In return this allows us to queue computational tasks via a second (virtualized) service layer. By adding this second grid we are then able to queue multiple instances of applications, such as WU-BLAST, with different inputs for each job via SGE. Using the overlay nature of the model, we found this option to be a very powerful one for running jobs that require dynamic input. Using this queuing approach allows for a simplified user experience that is mentioned in the user layer section. In our approach we use scripts by way of two features built into Windows XP: the DOS command line interface and the Visual Basic Script engine. We are thus able to build a virtualized system at the services layer.

#### Application Layer

For the MSU-GSL grid we require the capability to queue multiple VMs that have specific software requirements for each application. Because of the previous layers in our model, we are able to meet most applications' requirements by building specific application environments through virtualization. This allows for maximum application compatibility and simplified administration by abstracting the previous layers to the user through an application's natural interface. One is thus now able to build a virtualized container, and using the service layer, queue multiple instances of the same VM. At the application layer, scripting is used to tie each individual container to a larger system by connecting the service layer to the user layer transparently.

#### User Layer

The user layer is specific to each application as described in the user layer section. For example, our current implementation of 3D Studio Max's Backburner render farm suite uses specific application features to allow for dynamic rendering of data across the render farm. This allows for simple integration into our computational grid without retraining the

	Nodes	Frames	Resolution	Format	DPI	Time
Building	1	1	1920x1280	TIFF	300	3:32:16
Building	25	1	1920x1280	TIFF	300	10:03:10
Robot	1	600	1920x1280 (1080p)	AVI	75	2:34:12
Robot	25	600	1920x1280 (1080p)	AVI	75	8:14:56
Simple	1	200	1920x1280 (1080p)	AVI	75	0:21:32
Simple	25	200	1920x1280 (1080p)	AVI	75	0:45:48

Table 4.1: 3D Studio Max Render Times

user about how their application works. Through the Backburner Queue manager we are able to have complete user control over the virtual cluster. This included all of the queuing features that are associated with Backburner from Autodesk.

#### Run-time Statistics

After the creation of our overlay network we were able to run some basic tests using the aforementioned application. The application that we tested with our overlay grid was 3D Studio Max with Backburner Manager from Autodesk. This application was installed in its own virtual machine that was used to contain the execution environment. By using the steps provided in the GOS model we were able to construct a render farm using non-modified applications that were initially meant for cluster computing. For our tests we used a very detailed still frame and a collection of two animations with different lengths. The results for these renderings is shown in table 4.1 and are viewed in figure 4.1, figure 4.2, and figure 4.3.



Figure 4.1: Sample Building

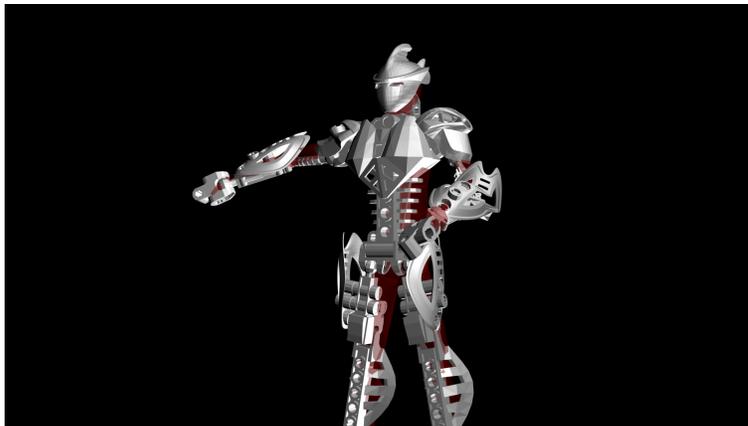


Figure 4.2: Moving Robot

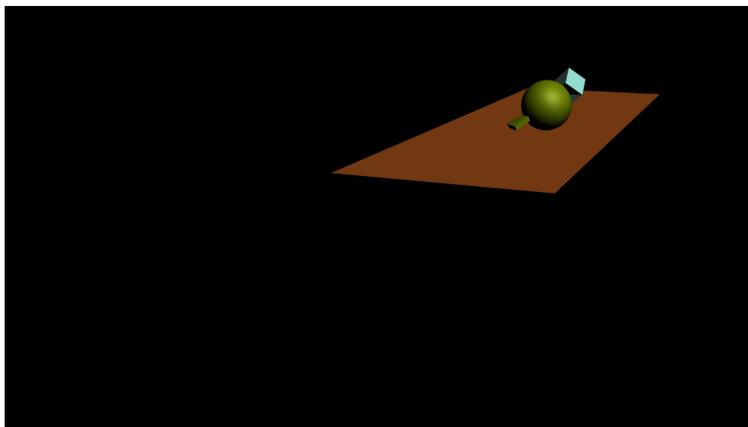


Figure 4.3: Simple Quick Scene

## CONCLUSIONS

### Future Work

We can envision many features that would improve MSU-GSL. One would increase the distribution speed of the virtual machine deployment by implanting new techniques in grid overlay systems, such as P2P data transfer. This data transfer mechanism would minimize the initialization time required for deploying multiple virtual machines across the network all at once. This would be helpful, allowing the user to query virtual machines with minimal wait time for the transfer process.

Another possible extension that could be made to the GOS model is that of virtual resource population on previously constructed systems such as Amazon's EC2. The question would be quite simple. Is one able to implement the GOS model on top of an existing elastic architecture such as EC2?

As processors get smaller, they can be used in increasingly smaller devices, such as cell phones. Wireless communications are now a part of our everyday lives. The GOS model is a unique model that can be used in environments other than research computing. For example, one could imagine using a P2P overlay network for data propagation on wireless networks. Is it possible to implement the GOS model in a cellular network for data transfer through the network layer's recursive nature? It might be possible to identify many techniques that are static in other models that could be dynamic in the GOS model.

A collection of specific applications could be examined further to see how they might fit into the GOS model. For example, Message Passing Interface (MPI) applications were not considered in our model. It might be possible to utilize these applications in the application layer for multilayer communication techniques. For example, one could possibly use MPI to handle communication between multiple resources at the application layer to create a

virtual network layer through message passing.

## REFERENCES

- [1] *Condor - A Hunter of Idle Workstations*, IEEE Press, Washington, D.C., 1988.
- [2] *A comparison of software and hardware techniques for x86 virtualization*, ACM, New York, NY, USA, doi:<http://doi.acm.org/10.1145/1168857.1168860>, 2006.
- [3] *A Three-phase Algorithm for Whole-System Live Migration of Virtual Machines*, ACM/IEEE, Reno, NV, peking University, 2007.
- [4] *AMD I/O Virtualization Technology (IOMMU) Specification*, 2009.
- [5] *POWER5 virtualization: How to set up the IBM Virtual I/O Server*, 2009.
- [6] *Intel 64 and IA-32 Architectures Software Developer's Manuals*, 631-651 pp., 2009.
- [7] Barker, K. J., K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, Entering the petaflop era: the architecture and performance of roadrunner, *Conference on High Performance Networking and Computing*, 1–11, 2008.
- [8] Buyya, R., C. S. Yeo, and S. Venugopal, Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities, in *HPCC*, pp. 5–13, IEEE, 2008.
- [9] Deelman, E., G. Singh, M. Livny, J. Good, and B. Berriman, The cost of doing science in the cloud: The montage example, in *SC'08 USB Key*, ACM/IEEE, Austin, TX, 2008.
- [10] Engelbrecht, H.-J., Internet-based 'social sharing' as a new form of global production: The case of SETI@home, *Telematics and Informatics*, 25(3), 156–168, 2008.
- [11] Holtman, K., Cms data grid system - overview and requirements, 2001.
- [12] Huedo, E., R. S. Montero, and I. M. Llorente, *Software: Practice and Experience*, 34(7), 631–651, 2004.
- [13] Matsunaga, A. M., M. O. Tsugawa, S. Adabala, R. J. Figueiredo, H. Lam, and J. A. B. Fortes, Science gateways made easy: the in-vigo approach: Research articles, *Concurr. Comput. : Pract. Exper.*, 19(6), 905–919, doi:<http://dx.doi.org/10.1002/cpe.v19:6>, 2007.
- [14] Quinlan, S., The google file system, in *The Conference on High Speed Computing*, p. 24, LANL/LLNL/SNL, Salishan Lodge, Gleneden Beach, Oregon, IA-UR-06-?, 2006.

- [15] Zimmermann, H., OSI reference modell - the ISO model of architecture for open systems interconnection, *IEEE Transactions on communication*, 28, 425–432, 1980.