The vMatrix: Server Switching

Amr Awadallah and Mendel Rosenblum
Computer Systems Lab, Stanford University
aaa@cs.stanford.edu, mendel@cs.stanford.edu

Abstract

Today most Internet services are pre-assigned to servers statically, hence preventing us from doing real-time sharing of a pool of servers across as group of services with dynamic load. Fluidly copying services in and out of servers remains a challenge due to the many dependencies that such services have on software, hardware, and most importantly, people. In this paper we present a novel solution, which builds on top of the classic operating systems concept of a virtual machine monitor (VMM). A VMM allows us to encapsulate the state of the machine in a virtual machine file, which could then be activated on any real machine running the VMM software. This eliminates the software dependencies problem by allowing us to move the whole machine around including the operating system, libraries, and third party modules that the service depends on. It eliminates the hardware dependencies problem by allowing us to mimic the hardware that the service expects regardless of the real hardware of the hosting machine. It also solves the people dependency problem by presenting the developers and system administrators with the same isolation model that they are used too with statically allocated servers. We describe our vMatrix framework in detail and address how to load balance the virtual machine services across the real-machines to maximize utilization efficiency (in terms of machines and people costs) such that total cost of the system is reduced without degrading the service performance and without requiring cost prohibitive code and architectural changes to existing legacy services. Our solution also offers additional side benefits like on-demand replication for absorbing flash crowds (in case of a newsworthy event like a major catastrophe) and faster failure recovery times.

Keywords: server multiplexing, server switching, load balancing, virtual machine monitor.

1 Introduction

In this paper we describe a practical solution for sharing a pool of servers between a number of Internet services with varying load profiles. The idea is very analogous to the well-known computer networking tradeoff between circuit-switching and packet-switching. Today most Internet services are provisioned in a circuit-switching like fashion that is a pre-determined number of servers are fully dedicated to a given service. The advantages of this approach are isolation and guaranteed performance, and the disadvantage is possibly wasting a lot of capacity if the service is not using all the allocated resources all the time, which typically is the case. A packet-switching like approach stipulates that the service does not get a pre-determined number of servers; rather these servers are allocated on demand based on the load of the service. The advantage of a server switching approach is that it permits statistical multiplexing on the shared resource allowing for very efficient use of the available capacity. In other words, a smaller number of servers can be used to accommodate the same number of services, thus reducing total system cost, both in terms of the cost for the servers, but more importantly the recurring cost of hardware administrators since they now need to maintain less hardware. The disadvantage of a switching approach is that the performance is not 100% guaranteed, rather its within some probabilistic bound, since if all of the services peak at the same time, and we had not kept an adequate buffer of idle servers, then there might not be enough servers to accommodate all of them, causing congestion then eventually what we will call server-loss (i.e. we want another server, but their aren't any left).

Today server switching is primarily possible within a few standardized application server frameworks like ATG Dynamo, IBM WebSphere, and BEA WebLogic [10][11][12], and even within those frameworks, library versions and operating system release mismatches can lead to inter-operability problems. Also non-*blessed* usage by the system developers might lead to dependencies external to the application server framework. But it is even more common that legacy services do not use such application server frameworks at the first place. In those cases it becomes even harder to perform server switching, and the cost of re-architecting the service and rewriting all of the code to fit within a standardized application framework is typically extremely prohibitive, since more frequent than not, internet service infrastructures grow in an evolutionary fashion rather than a revolutionary one.

The main reason leading to hardship moving services in and out of servers is the dependencies that the service code has on operating systems, libraries, third party modules, server hardware, and even people. Simply copying the code of the service is not possible since the target machines need to have exactly the same environment for the code to run unchanged, which is not practical. The library versions that work with one service might cause another service to fail when ran on the same server.

We propose a novel backward-compatible solution that builds on top of the classic OS concept of a *Virtual Machine Monitor (VMM)* [7] (refer to Appendix A for a brief review of Virtual Machines). The observation we make is that a VMM virtualizes the *real machine (RM)* at the hardware layer (CPU, Memory, IO), and exports a *virtual machine (VM)*, which exactly *mimics* what a real machine would look like. This allows us to encapsulate the state of the entire machine in a VM file, which could then be instantiated on any RM running the VMM software. This solves the software dependencies problem since the whole service is transferred with the OS, libraries, code, modules, and code that the service depends on. It solves the hardware dependencies problem since the VMM can *lie* to the overlying OS about the hardware resources available to it (e.g. memory size), hence mimicking the same hardware environment for that service regardless of the real hardware of the hosting real machine (though there might be performance degradation). It also solves the people dependency problem by presenting the developers and system administrators with the same isolation model that they are used too with statically allocated servers.

Hence the problem is reduced to delivering large VM files within a network of RMs running the VMM software; we call this network of virtual machines *the vMatrix*[1].

We do not attempt to build a VMM, but rather we reference existing software for the x86 architecture from VMware, Inc. [3]. Note that similar VMM software is also available from Connectix Virtual PC [21] (now owned by Microsoft), but we choose VMware due to their close relationship with Stanford University and also because they provide a server class VMM (Microsoft is scheduled to release a server class VMM in first half of 2004).

In this paper we present our framework in detail and briefly address how to load balance the virtual machine services across the real-machines to maximize utilization efficiency (in terms of machines and people costs) such that total cost of the system is reduced without degrading the service performance (in terms of latency, throughput, and availability). Another challenge that we touch on is how to avoid making any significant architecture or software changes to existing services so that this solution is backward compatible with legacy services.

We claim that the distinguishing advantages of our approach are the combination of:

(1) Server switching allowing for efficient use of machine and human resources, thus leading to reduced total cost of ownership;

(2) Presenting the developers and system administrators with the same machine isolation model that they are used to;

(3) Backward compatibility leading to very low costs of converting an existing Internet service to run within such a framework;

(4) On-demand cloning of servers to absorb sudden surges of incoming requests; an extreme example is the CNN.com meltdown on Sept 11th, 2001 [33].

(5) Quick re-activation of services to reduce mean recovery time in cases of software crashes, thus leading to higher availability.

[1] The name "*The vMatrix*" comes from the analogy to the 1999 sci-fi movie "*The Matrix*". In the movie, machines controlled humans by virtualizing all their external senses; we propose doing the same back to the machines! It is a virtual matrix of real machine hosts running VMM software, which are ready to be *possessed* by guest VMs (*ghosts*) encapsulating Internet services.

In a previous paper [35] we covered additional advantages of the vMatrix platform that leverage the migration aspects on an Internet-scale to achieve Dynamic Content Distribution.

In section 2 we present a motivating example. In section 3 we present the vMatrix framework. In section 4 we discuss how the vMatrix implementation details. In section 5 we discuss our experiences with two Internet services migrated to the vMatrix platform. Finally in sections 6 and 7 we cover related work then conclude.

2 Motivation

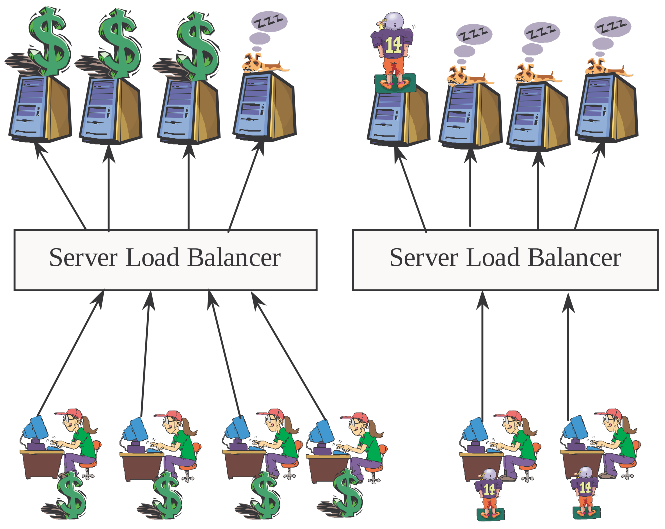


Figure 1: Statically Pre-provisioned services.

Yahoo! Inc. (employer of one of the authors) provides many popular Internet services, lets consider two of them: a financial service providing information about various stocks and mutual funds (finance.yahoo.com), and a sports news service providing information on the latest matches and their scores (sports.yahoo.com). In the current static pre-allocation world, a fixed number of servers will be allocated for each, say 4 for the financial service and 4 for the sports service, as illustrated in Figure 1. It turns out that the load profile for the financial service is such that it's busy during weekday mornings/afternoons, and it's almost idle on weekday evenings/nights and weekends. In contrast, the sports service load profile is such that it's busy on weekends and weekday evenings/nights, and that is almost idle on weekday mornings/afternoons. So if we take a snapshot of these services on a morning of a weekday we will see that the financial service is almost using all the capacity of its servers, while the sports service is using only a small part of its allocated capacity leading to non-efficient use of the available resources (those are the servers illustrated by sleeping dogs in Figure 1. Note that it's not really that a number of servers will be completely idle, but rather all servers will be operating at a portion of their full capacity).

The services are usually statically separated like this because it's typically very hard for two different services to co-exist on the same machine due to the following dependency issues:

(1) *Software dependencies*: the OS release/patch, or the library/module version that makes one service work, might break the other.
(2) *People dependencies*: that is the developers and system administrators responsible for one service would not like to deal with the consequences of actions done by the developers and administrators for the other service. There also might be some security constraints requiring total isolation so that programmers cannot access each other's servers.
(3) *Hardware dependencies*: the service might make some assumptions on memory size, hard-disk space or other hardware resources that might be violated when another service shares with it the same server or when we move the service code to a server that does not satisfy these requirements.

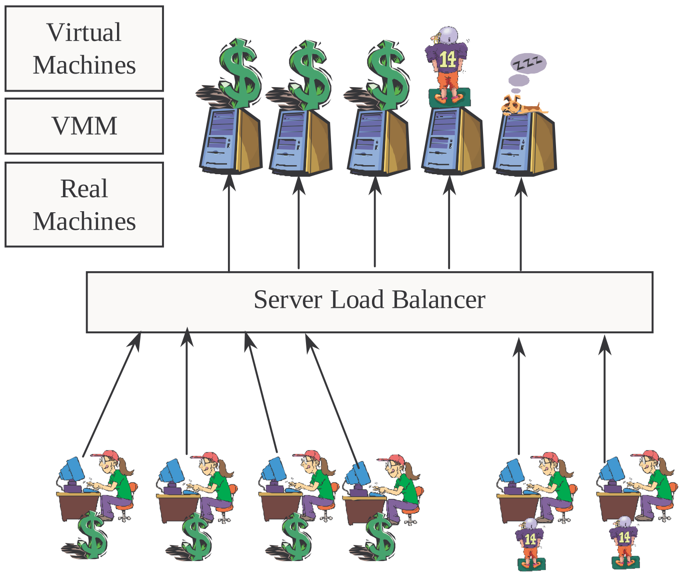


Figure 2: Server Switching (Dynamic Allocation)

In conclusion, the advantage of static pre-allocation is that it solves the software, administration, and hardware dependency problems by providing strict isolation at the hardware level; however, it leads to non-efficient use of the available resources. Note that this non-efficiency is not just a matter of more servers, but also the hardware administration personnel needed to maintain these extra servers (which is a recurring cost that is most probably more expensive than the servers themselves).

Another disadvantage is no fluidity in assigning new servers to a given service if demand unexpectedly surges for it, e.g. a sudden major financial crisis or a catastrophic event. In today's static world it takes from a couple of hours to a few days until enough additional servers are re-allocated from other services to the surging service, which is usually too late!

Within the vMatrix we add a VMM separation layer between the VMs (carrying the OS and code for the services) and RMs (which are the shared resource). Now the RMs can be shared between both services; hence reducing the total number of needed RMs and leading to efficient resource utilization. Looking at the same snapshot we represented in Figure 1, which required 8 RMs, we now only need 4 RMs as is illustrated in Figure 2, however, a 5th RM is still kept as an idle reserve to serve as a buffer in cases of congestion where both services might spike together causing higher than expected demand.

3 The vMatrix Framework

The vMatrix is a network of real machines (RMs) running virtual machine monitor software (VMM) such that virtual machine files (VMs) encapsulating a machine for a given service can be activated on any RM very quickly (on the order of seconds to minutes depending on the underlying infrastructure, e.g. local hard-disks versus a fiber-optic Storage Area Network).

3.1 Main Components

The basic framework for the vMatrix is illustrated in Figure 3. There are 3 main clusters:

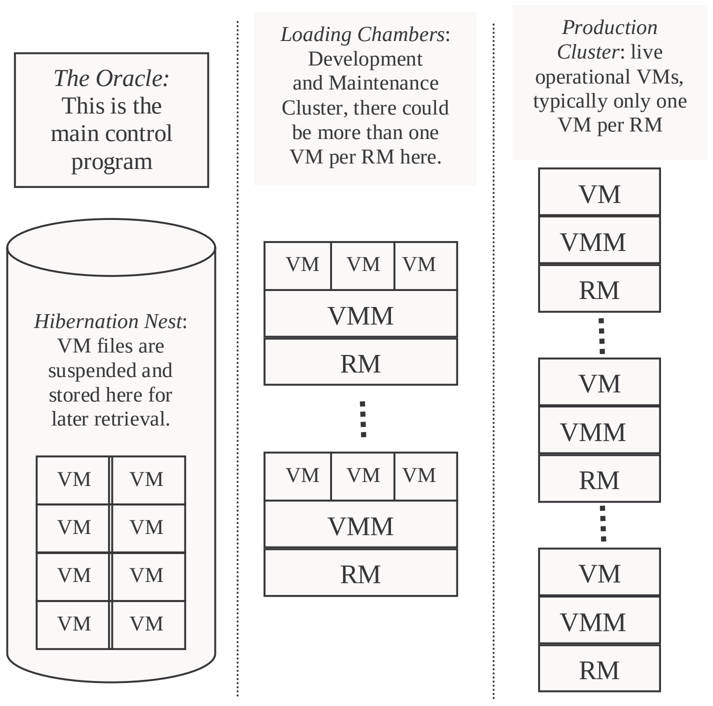
1. The *Production Cluster*: this is where the VMs are instantiated on dedicated RMs to serve live operational load. Note that these operational VMs can be any of the machines in a multi-tier architecture; they could be the front-end web servers, the middle application servers, or the back-end databases. The important distinction is that in this state the VMs are exposed to operational load.
2. The *Loading Chambers:* this is where the VMs are instantiated for maintenance and development purposes. The system administrators and software

developers can get access to the VMs for the purpose of updating code, applying patches, upgrading libraries, etc. In this state we can have more than one VM sharing the same RM, since the VMs are not really exposed to live load.

3. The *Hibernation Nest*: this is simply the backend storage for keeping all the VM files in dormant suspended state. The VMs are not accessible in this mode.

The *Oracle* is the program responsible for maintaining the state of all VMs and RMs and it supervises the vMatrix network. As new RMs are added to the network and loaded with the VMM software, they are subscribed with the Oracle. Similarly, whenever a new VM is created it is registered with the Oracle. The Oracle is also responsible for the matching of VMs to RMs and copying the VM file to that specific RM then activating it.

In our first simple prototype, the Oracle is a Perl script which reads configuration files listing all available RMs and VMs. The Oracle communicates with the RMs to copy VM files from the storage to them (using `scp`), and communicates with the VMM server software on each RM to boot or suspend VMs (this is done using the VMware Perl API which is covered in Appendix B).

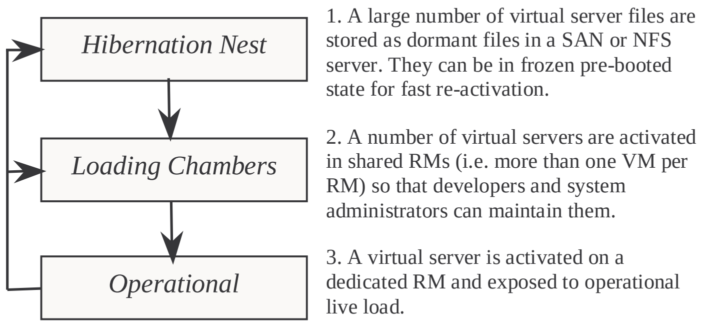


The Oracle: This is the main control program
Hibernation Nest: VM files are suspended and stored here for later retrieval.
VM
Loading Chambers: Development and Maintenance Cluster, there could be more than one VM per RM here.
VM
VMM
RM
Production Cluster: live operational VMs, typically only one VM per RM
VM
VMM
RM

Figure 3: The vMatrix Framework

3.2 VM Server Lifecycle

The simple state diagram shown in Figure 4 describes the lifecycle of a VM Server:



Hibernation Nest
Loading Chambers
Operational
1. A large number of virtual server files are stored as dormant files in a SAN or NFS server. They can be in frozen pre-booted state for fast re-activation.
2. A number of virtual servers are activated in shared RMs (i.e. more than one VM per RM) so that developers and system administrators can maintain them.
3. A virtual server is activated on a dedicated RM and exposed to operational live load.

Figure 4: Lifecycle of a Virtual Server

4 Implementation Details

As contrasted to previous work (which we cover in section 6), we claim that this solution presents the smallest switching cost for porting an existing Internet service into such a dynamic allocation network (i.e. backward compatibility) and at the same time it keeps most of the key advantages of static pre-allocation listed in section 2 (mainly isolating software, people and hardware dependencies). In section 5 we illustrate this ease of conversion through a couple of real-life examples.

4.1 Backward Compatibility

Most services could be ported to this framework with minimal to no code or infrastructure changes; the system administrators and developers would simply need to install the OS and service software inside of a VM, same way they install it inside a RM today. Once that is done the VM is ready to be instantiated on any RM running the VMM software. The VM preparation and software installation is done in the Loading Chambers, where the RMs main purpose is to host many idle VMs so that system administrators and programmers can prepare them for operational deployment. The VMs are not exposed to any operational load while waiting in the Loading Chambers.

Note that VMM software allows for more than one VM to share the same RM, however they are fully isolated and each one can have its own IP address. As far as developers are concerned when they `ssh` to a given VM in the Loading Chambers, they truly believe it's their own fully assigned isolated real machine. However, if these machines are exposed to heavy load, like decompressing a large tar-ball, then neighboring programmers will sense a sudden slow down and can start to notice that they are sharing the machine with somebody else. It must be noted though that server-class VMM software provides a resources quota system that prevents VMs from cannibalizing all of the RM resources (i.e. CPU, Memory, Disk space, IO, Network, etc).

4.2 Load Balancing

The load-balancing of VMs between RMs (which we refer to as server switching to avoid confusion with traditional server load-balancers) is done by building a time-based profile for how much resources each service consumes on all of its allocated RMs. This profile can be built by either polling the OS of the service directly (in a SNMP/MRTG-RRD like fashion), or by asking the VMM to report the resources consumed by the VM, which is very handy in cases where the service OS is not instrumented to report all needed load metrics (primarily CPU utilization, Memory active-working-set, Disk space, Disk IO and Network utilization). Appendix B illustrates the VMware Perl function to return the resources consumed by a given VM averaged over the last 5 minutes.

In this paper we focus on the case of one-2-one matching for operational VMs, i.e. only one VM per RM. This simplification criterion is for the operational front-end VMs only, but we can still have many VMs per RM in the Loading Chambers. This restriction reduces the problem to a simple bottleneck detection and greedy matching algorithm. In a nutshell the Oracle daemon loops over all operational VMs for a given service and detects the ones with a persistent bottleneck (e.g. 100% CPU utilization over last 10 minutes). It then fetches another VM for that service from the Loading Chambers and activates it on an idle RM (the network administrator is then alerted to add this new RM to load balancer rotation, though that can be automated as well). Conversely, if there are no bottlenecks detected for any of the operational VMs for a given service, then its time to move one of the VMs for that service back to the Loading Chambers and free the RM allocated to it. Once good historical load profiles are established, the addition of an operational VM can take place ahead of the bottleneck occurrence, except for sudden demand spikes, which we would still need the switching algorithm to detect and respond to fairly quickly. Finally, the server-switching algorithm needs to take into account the VM sizes to minimize VM switching so that the data center LAN is not overloaded with VM transfers, though this is now less of an issue with the ever increasing LAN network speeds.

An issue that needs to be considered is de-activating VMs with active connections. The graceful solution is to take such servers out of load-balancer rotation, then after detecting that all existing connections are closed, it can be safely suspended and removed from the RM (this is actually very similar to how hardware servers are added and removed in a static solution, just much faster and does not require humans touching the physical machines).

It has to be noted that VMware is now beta testing VMotion [34] technology that can move VMs while maintaining the active connections. However, this solution requires that the source and target RMs mount the same disk volume from a SAN, and that they have CPUs from the same processor family (e.g. PIII and P4 wont work). However, the advantage of VMotion is that it can migrate live servers in less than 2 seconds by doing clever memory deltas using bitmaps.

4.3 Absorbing flash crowds

A flash crowd is an unpredicted increase of web requests, such as an unforeseen surge of stock market activity or a catastrophic newsworthy event. To absorb flash crowds we need on-demand replication, which can be achieved by one of two methods:

(1) *VM Cloning*: by this we mean that a copy of the VM file is done in real-time and instantiated on a new RM. The disadvantage of cloning is that it's not always achievable without some changes to the service architecture or code (e.g. need to change the IP address for the clone, though NAT can be used for that). Also for multi-tier architectures, the back-end tiers typically make assumptions about the IP address or some logical name for the front-ends, which makes it harder to clone the front-ends without making some code changes to the existing back-ends to accept these real-time created front-ends.

(2) *VM Pre-creation*: To avoid having to do any code changes to the existing service code, we can pre-create all the server VMs needed for the worst case scenario in the Loading Chambers, and then shutdown and store all those VMs in the Hibernation Nest. When demand surges we now have a large pool of VMs that we can pull from and activate for this service. The downside of that solution is the extra hard disk space required to store all those VMs, but that is not such a large penalty with the ever-decreasing storage costs. Also smart differential compression techniques (e.g. chain coding [23]) can be used between the VM image files to reduce the total actual hard disk space required, though this might add some decompression overhead in pulling the VM files back from storage. Another downside for this solution is that the system administrators now have to manage all of these VMs (e.g. if there is a new service patch then it will have to be applied to all of them by activating them in the Loading Chambers).

We chose the second approach due to its nice backward compatibility characteristics. Another side advantage is that VMM software enables the suspension of VMs in a live state, such that all CPU registers, memory, and IO buffers are dumped to disk, then the machine could be resumed later at the same checkpoint that it was suspended at (this is similar to suspending/hibernating a laptop). This means that the suspended VMs can be activated in a very short time, typically around 10 to 30 seconds, instead of booting up the machine from idle state which tends to take a long time and consumes more resources. Hence, by pre-booting the service VMs, before suspending and storing them in the Hibernation Nest, then when a flash crowd arrives we can activate them on front-end machines fairly quickly and there is no need to wait for a full boot to take place. Once the flash crowd flood is over then the VMs can be suspended back to dormant state, moved to the Loading Chambers (for software maintenance) or Hibernation Nest (for storage), and the front-end host RMs are now freed for some other service.

4.4 Faster recovery time and higher availability

Another advantage for the quick resumption of VM files from suspended state is improved availability, as a new VM can be instantiated fairly quickly to take over from a VM that failed due to a software crash hence significantly reducing recovery time. In [22] Armando Fox and David Patterson argue that improving MTTR (mean time to recovery) is in many cases more beneficial to improving availability than improving MTTF (mean time to failure, i.e. more reliable hardware). In Appendix B we list the VMware Perl function to check for heartbeats from VMs to make sure they did not crash.

5 Experiences

It is the goal of this work to show that it is possible to encapsulate legacy Internet services via VMMs, to achieve a standardized solution for improving the scalability, interactivity, availability, and efficiency of internet services without requiring cost prohibitive changes to existing system architectures. We illustrate that this is a practical solution by building out a vMatrix prototype, and porting into it a number of existing Internet services ranging from open source services (e.g. PHPnuke [24] and osCommerce [25]) to proprietary services in collaboration with Yahoo, Inc. These services represent the spectrum of practical Internet service architectures (e.g. single tier, two-tier, write-once/read-many, write-many/read-many, single-user, and one-user to many-users).

Our experience confirms that the migration cost is minimal for both the developers of the service and the system administrators, i.e. quick migration, short learning curve, and support for traditional system administration tasks like troubleshooting, rebooting, monitoring, code updates, etc.

5.1 The Experimental Setup

The lab in which we performed the experiments consists of three Pentium III servers at 550MHz, 640MB ram and 9GB hard disks each. The first machine serves as the Production Cluster, the second machine serves as the Loading Chambers, and the third machine serves as the Hibernation Nest and also runs the Oracle software. We used the VMware ESX server, which is a server-class virtual machine monitor. The ESX server consumes about 3.5GB of disk space and 184MB of memory The CPU overhead is typically less than 5%.

5.2 A Web Portal: PHP-Nuke and osCommerce

PHP-Nuke is one of the most popular web content publishing open-source platforms written using the popular PHP web scripting language. It provides many functionalities for a full fledged portal like news, polls, message boards, etc. osCommerce is another popular PHP application that provides an ecommerce store website. It took us less than a couple of hours to support a server running both PHP-Nuke and osCommerce within the vMatrix. We used the Oracle command line interface to create a VM in the Loading Chambers. We then installed on it the software components illustrated in Figure 5. The time it took us to do this is not significantly more than it would take to just install on a real machine. We did not change a single line of source code from those applications, and they became fully supported within the Vmatrix framework as is.

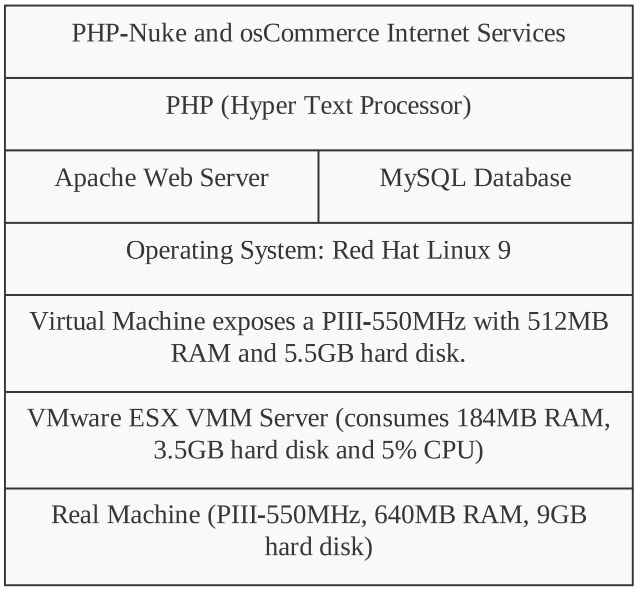

PHP-Nuke and osCommerce Internet Services
PHP (Hyper Text Processor)
Apache Web Server
MySQL Database
Operating System: Red Hat Linux 9
Virtual Machine exposes a PIII-550MHz with 512MB RAM and 5.5GB hard disk.
VMware ESX VMM Server (consumes 184MB RAM, 3.5GB hard disk and 5% CPU)
Real Machine (PIII-550MHz, 640MB RAM, 9GB hard disk)

Figure 5: VM for PHP-Nuke and osCommerce

Once we configured the VM for this web portal in the Loading Chambers, we next instructed the Oracle to activate the VM, which caused the VM to be suspended and then copied over to the operational cluster then resumed. Note that when a VM is suspended in pre-booted state, only 3 files need to be copied, the first is the configuration file for the VM describing its memory size, Ethernet address, etc. The second file represents the hard disk of the VM, and the third file contains the frozen state of the VM (memory, CPU registers, frame buffer, etc). Once the 3 files are copied over to a dedicated RM in the operational cluster, it is resumed and exposed to live load. The Oracle periodically polls all active VMs to check whether they are still on or if they crashed; however this is simply a redundant check since most websites already have more sophisticated pings in place using monitoring tools like Nagios [26].

An expansion of this service that we could not do in our small lab setting is to convert the application into a two-tier architecture, specifically having the MySQL server run in a separate VM. In this case both the front-end PHP server and the backend MySQL server will be hosted in different VMs and there can be more than one front-end PHP server frozen in the Hibernation Nest and ready to be activated to absorb any flash crowds if they occur.

5.3 Yahoo! Autos Search

Yahoo! Autos allows users to search for cars being sold from a number of sources. In this part we took the Yahoo! Autos Search functionality and installed it within the Vmatrix framework as illustrated in Figure 6. This is a typical Yahoo! Autos Search backend server, which provides the front-ends with the ability to call into it with certain search criteria (e.g. car manufacturer, model, year, color, price range, etc), then it performs this search using a custom Yahoo! Search indexing service (known as YSS, short for Yahoo Structured Search). The YSS code is built on top of YLIB, which is custom Yahoo C/C++ libraries. Most of the Yahoo servers use FreeBSD (instead of Linux), so this was a good exercise to show that operating systems other than Linux can work within this platform.

Again, as we demonstrated in previous section, the Yahoo! Autos Search service was installed within the Vmatrix framework in about a few hours and no coding changes what so ever were required to get it up and running. We were then able to perform the migration and cloning functions that otherwise would have required extensive code rewriting on other frameworks.

Another trick that we did in this setup was to lie to the underlying VM as to how much physical memory is really present (so that we can match the memory requirements it needs). Even though the real machine only had around 456MB of available free physical memory, we used the VMM virtualization functions to virtualize the remaining 568MB on disk. The result was that the FreeBSD VM really thought it had 1024MB of physical memory available. Of course it will run a bit slower due to this, so it's not an optimal situation, but it demonstrates how the services can be moved even between non-heterogeneous hardware servers.

Note that performance analysis of VMware virtualization overheads is not the goal of these experiments; rather it's the illustration of the ease of converting an existing service into this framework without requiring any coding or architectural changes. However, we attempted to give brief estimates in Figure 5 and Figure 6, which illustrate that the CPU performance overhead is usually about 5%, and the memory overhead is about 184MB. Also the resulting VM file size was about 4GB, which takes about 10 minutes to transfer on 100MBit Ethernet, and takes under a minute on Gigabit Ethernet. So the activation time to add servers, in case of flash crowds, can be very reasonable and on the order of a few minutes as opposed to a few hours that a manual provisioning would imply.

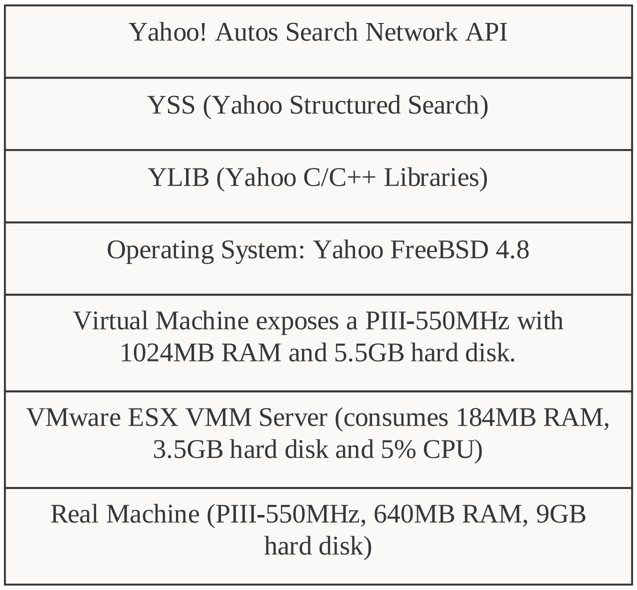

Yahoo! Autos Search Network API
YSS (Yahoo Structured Search)
YLIB (Yahoo C/C++ Libraries)
Operating System: Yahoo FreeBSD 4.8
Virtual Machine exposes a PIII-550MHz with 1024MB RAM and 5.5GB hard disk.
VMware ESX VMM Server (consumes 184MB RAM, 3.5GB hard disk and 5% CPU)
Real Machine (PIII-550MHz, 640MB RAM, 9GB hard disk)

Figure 6: VM for Yahoo! Autos Search

6 Related Work

Previous work in the area of server switching suffers from a common disadvantage, which is requiring the Internet service developers to recode their applications within a new framework or adhere to a set of strict guidelines. In other words, they are not backward compatible. This represents a huge impediment for developers, since it requires them not only to learn how to use a new framework, but also to port all their existing code to this new framework. This is not cost effective since the salary of system programmers is typically much higher than any of the network or server costs to justify such a migration.

Application Servers live ATG Dynamo [10], IBM WebSphere [11], BEA WebLogic [12], and JBoss [27] provide a strict API for system services, and hence it is feasible to move the application between different servers running the same application server. However, programmers do not strictly adhere to these APIs, which prevents application mobility. Also Application servers fail to provide the strict isolation model that developers expect from a dedicated machine. Java servlets [5] can be moved between servers running the Java virtual machine, but this approach suffers from performance degradation due to the real-time byte-code translation that Java requires. Also it requires that existing applications be rewritten within the Java environment, which again presents a high switching cost. Similarly, both Xenoservers [18] and Denali [20] require developers to write their code under a specialized OS optimized for encapsulation and migration.

The Portable Channel Representation [4] is an XML/RDF data model that encapsulates OS and library dependencies to facilitate the copying of a service across different systems. Again it requires the programmers to learn a new framework and port their existing work in to it, and it also does not provide isolation between software belonging to different services. Package manger tools like Debian's APT (Advanced Packaging Tool [28]) or Red Hat's RPM [29] can facilitate the movement of internet service code between servers, however they do not provide any kind of isolation and the aforementioned library version collisions can happen between services installed on same machine. Computing on the Edge [36] also falls in this category and suffers from the same disadvantages.

Disk imaging (aka ghosting) and diskless workstations booting from SANs has been used for years to quickly repurpose hardware to new services. However that approach suffers from inability to concurrently run more than one VM per RM, which is needed in the Loading Chambers so that software developers can maintain their packages and continue to be presented with the same dedicated machine isolation model that they are used to.

OS virtualization, e.g. Ejasent [17] and Ensim [19], traps all OS calls, hence allowing applications to be moved across virtual operating systems. The downside of this solution is that it is OS dependent and imposes strict guidelines on what the applications can and can't do. Zap [37] sits some where between OS virtualization and application virtualization, but does share the same downside of being tied to the OS.

Finally it has to be noted that a number of computer system manufacturers are addressing the server switching space with their own implementations, e.g. IBM is offering OnDemand [30], SUN provides the N1 system [31], and HP has the Utility Data Center [32].

7 Conclusion

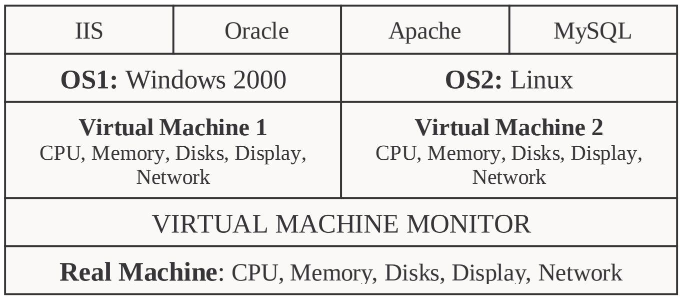
In this paper we presented a novel solution for server switching. The solution is a network of real machines running virtual machine monitor software, hence allowing server virtual machines to be switched between the real machines. We described our approach in detail and provided real-life examples. The advantages of our approach are efficient resource utilization, backward compatibility, flash crowd real-time absorption, and faster recovery times.

Acknowledgments

We would like to acknowledge the help of Abe Taha from Yahoo! Search Technologies for installing the Yahoo! Autos search service within the Vmatrix.

Appendix A

This brief section provided for the benefit of our readers who are not very familiar with VMM technology. A VMM is a thin layer of software that runs on top of a real machine and exports an abstraction of the real machine [7]. This abstraction is a virtualized (mimicked) view of all hardware in the machine (e.g. CPU, Memory, IO) as shown in Figure 7. VMMs allow multiple guest virtual machines with a full OS and applications to run in separate isolated virtual machine spaces, such that they cannot affect each other. Note that unlike a Java Virtual Machine [9], binary code translation, and machine emulation, the instructions in the VM run natively on the processor of the host RM with almost no change, and hence the performance of code running inside of a VM is almost as fast as the code running directly in a RM.



IIS
Oracle
Apache
MySQL
OS1: Windows 2000
OS2: Linux
Virtual Machine 1
CPU, Memory, Disks, Display, Network
Virtual Machine 2
CPU, Memory, Disks, Display, Network
VIRTUAL MACHINE MONITOR
Real Machine: CPU, Memory, Disks, Display, Network

Figure 7: Virtual Machine Monitor

VMMs were introduced in the 1970s by IBM [8] to arbitrate access to hardware of an expensive mainframe machine between a number of client operating systems, and to provide their customers with a forward migration path to newer mainframes. VMMs faded in the 1980s, as the PC became mainstream and computer hardware prices dropped, but were resurrected recently for the x86 architecture by VMware, Inc. [3]. In a well-designed VMM, the code is entirely fooled into believing its mimicked environment such that it cannot detect whether it is running inside a virtual machine or a real machine.

VMware VMM software also provides remote control over the keyboard, monitor, mouse, floppy-drive and CDROM drive of the virtualized machine. This allows owners of the VM to remotely install new software or power cycle the VM without worrying where the machine is physically instantiated, in a sense replacing the popular keyboard/video/mouse (KVM) remote switches (also known as boot boxes).

Appendix B

This section is intended to briefly describe VMware's Perl API to communicate with a VMM server.

```
use VMware::Control;
use VMware::Control::Server;
use VMware::Control::VM;

my $VMMserver = VMware::Control::Server
::new($hostname, $port, $user,
$password);

$VMMserver->connect();

my $VM = VMware::Control::VM
::new($server, $vmconfig);

$VM->connect();

# To get a list of all VMs on a server
my @vmlist = $VMMserver->enumerate();

# To register a new VM on a server
$VMMserver->register($vmconfig);

# To start and stop a VM
$VM->start();
$VM->stop();

# To suspend and resume a VM
$VM->suspend();
$VM->resume();

# To get CPU, Memory, Net, IO stats
$VM->get("Status.Stats.vm.cpuUsage",
5*60);

# Check if VM is running (heart-beat)
$VM->get("Status.Power");
```

References

[1] "TIBCO Rendezvous Messaging System", TIBCO Software Inc. http://www.tibco.com/products/rv/index.html

[2] "Vignette Advanced Deployment Server", Vignette Inc. http://www.vignette.com/CDA/Site/0,2097,1-1-1329-2067-1345-2198,00.html

[3] "VMware Secure Transportable Virtual Machines", VMware Inc. http://www.vmware.com

[4] Beck, Moore, Abrahamsson, Achouiantz and Johansson. "Enabling Full Service Surrogates Using the Portable Channel Representation", 10th Intl. WWW Conference.

[5] "Java Servlet Specification v2.3", Javasoft, Sun Microsystems, 2000.

[6] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. "A Dynamic Object Replication and Migration Protocol for Internet Hosting Service", 19th International Conference on Distributed Computing Systems, Austin, Texas, June 1999, IEEE.

[7] R. P. Goldberg. "Survey of Virtual Machine Research", IEEE Computer, June 1974.

[8] IBM Virtual Machine 370, 1972. IBM Corporation http://www-1.ibm.com/ibm/history/history/year_1970.html

[9] Tim Lindholm and Frank Yellin. "Java Virtual Machine Specification, 2nd Edition", 1999, Addison-Wesley.

[10] ATG Dynamo Application Server, ATG. http://www.atg.com/en/products/das.jhtml

[11] IBM WebSphere Application Server. IBM Corporation. http://www.ibm.com/software/webservers/appserv

[12] BEA Systems WebLogic Application Server , BEA Systems. http://www.beasys.com/products/weblogic

[13] "RFC1918: Address Allocation for Private Intranets"

[14] "RFC1631: The IP Network Address Translator"

[15] "RFC2764: A Framework For IP Based Virtual Private Networks"

[16] "RFC2709: Security Model with Tunnel-mode IPSec for NAT Domains"

[17] "Ejasent: Making the Net Compute", Ejasent Inc. http://www.ejasent.com

[18] Steven Hand, Tim Harris, Evangelos Kotsovinos, and Ian Pratt. "Controlling the XenoServer Open Platform", IEEE OPENARCH'03.

[19] "Ensim: Hosting Automation Solutions", Ensim Inc. http://www.ensim.com

[20] Andrew Whitaker, Marianne Shaw, Steven D. Gribble. "Scale and Performance in the Denali Isolation Kernel", USENIX OSDI 2002.

[21] Connectix Virtual PC. Microsoft Corporation. http://www.microsoft.com/windowsserver2003/evaluation/trial/virtualserver.mspx

[22] Armando Fox and David Patterson. "When Does Fast Recovery Trump High Reliability?", 2nd Workshop on Evaluating and Architecting System Dependability, 2002.

[23] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. "Optimizing the Migration of Virtual Computers", USENIX OSDI 2002.

[24] PHP-Nuke: Open Source Advanced Content Management System: http://www.phpnuke.org

[25] OsCommerce: Open Source eCommerce platform: http://www.oscommerce.com

[26] Nagios: Open Source Host, Service and Network monitoring program: http://www.nagios.org

[27] JBoss: Open Source Java Application Server (J2EE). http://www.jboss.org

[28] APT: Debian's Advanced Package Tool: http://www.debian.org/doc/manuals/apt-howto

[29] RPM: Red Hat's Package Manager: http://www.rpm.org

[30] IBM OnDemand Operating Environment: http://www-3.ibm.com/software/info/openenvironment/

[31] Sun N1: http://www.sun.com/n1

[32] HP Utility Data Center: http://ww.hp.com/solutions1/infrastructure/solutions/utilitydata/

[33] William LeFebvre. "CNN.com: Facing A World of Crisis", Invited talk at USENIX LISA, San Diego, CA, Dec 2001.

[34] "Building Virtual Infrastructure with VMware VirtualCenter", white paper, VMware Inc. http://www.vmware.com/pdf/vc_wp.pdf

[35] Amr Awadallah and Mendel Rosenblum. "The vMatrix: A network of virtual machine monitors for dynamic content distribution", Seventh International Workshop on Web Content Caching and Distribution, August 2002.

[36] Michael Rabinovich, Zhen Xiao, and Amit Aggarwal. "Computing on the Edge: A Platform for Replicating Internet Applications", Eighth International Workshop on Web Content Caching and Distribution, Sept 2003.

[37] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. "The Design and Implementation of Zap: A System for Migrating Computing Environments", USENIX OSDI 2002.